IvorySQL

Version v5.4, 2026-06-17

About IvorySQL 1
Welcome 3
Release 3
Get Started 6
.1. Quick Start 6
Deploy 9
.1. Installation 9
.2. Cluster 14
Oracle Compatibility 20
.1. 1、Case conversion 20
.2. 2、LIKE operator 26
.3. 3、anonymous block 28
.4. 4、functions and stored procedures 29
.5. 5、Built-in data types and built-in functions 41
.6. 6、ports and IP 64
.7. 7、XML Function 64
.8. 8、sequence 70
.9. 9、Package 71
.10. 10、Invisible Columns 75
.11. 11、RowID Column 76
.12. 12、OUT Parameter 78
.13. 13、%Type & %Rowtype 81
.14. 14、NLS Parameters 85
.15. 15、Force View 88
.16. 16、Nested Subfunctions 89
.17. 17、sys_guid Function 91
.18. 18、Empty String to NULL 92
.19. 19、CALL INTO 93
Containerization and Cloud Service 98
Containerization 98
Cloud Service Platform 169
IvorySQL Ecosystem 191
.1. CPU Architecture Adaption 191
.2. Operating System Adaption 191
Eco Component Adaption 194
Monitor and O&M 216
.1. Monitoring 216

.2. Maintenance . . . 272
.3. Operation Management . . . 298
Data Migration . . . 341
.1. Migration . . . 341
IvorySQL Developers . . . 358
.1. Developer . . . 358
.2. Community contribution . . . 437
IvorySQL Architecture Design . . . 460
Query Processing . . . 460
Compatibility Framework . . . 462
1. Objective . . . 463
Compatibility Features . . . 469
Built-in Functions . . . 499
1.4. GB18030 Character Set . . . 502
Reference . . . 506
.1. Tool Reference . . . 506
2. FAQ . . . 615

About IvorySQL

Introduction to IvorySQL

Overview

IvorySQL is an advanced, full-featured, Oracle open source compatible PostgreSQL with a firm commitment to always remain 100% compatible and a direct replacement for the latest PostgreSQL. IvorySQL adds a GUC parameter called 'ivorysql.compatible_mode' to control the compatibility mode of IvorySQL, which has two values: 'oracle' and 'pg'.When initializing the data directory, specify the compatibility mode of the data directory by specifying the '-m' parameter, and '-m pg' then the data directory is PostgreSQL mode.In this mode, the 'ivorysql.compatible_mode' parameter will be invalidated, and the '-m oracle' or if the '-m' parameter is not specified, the data directory will be compatible with Oracle mode.In this mode, the initial value of the 'ivorysql.compatible_mode' parameter is 'oracle' and does not support some PostgreSQL syntax, and the database can support 100% of PostgreSQL syntax and functions by 'set ivorysql.compatible_mode to pg'.

One of the highlights of IvorySQL is the PL/iSQL procedural language, which supports Oracle’ s PL/SQL syntax. At the same time, IvorySQL implements Oracle-compatible functions by adding plug-ins ivorysql_ora bound to the kernel, and the functions currently implemented include built-in functions, data types, system views, merges, and the addition of GUC parameters, and will continue to implement new compatible functions in the form of plug-ins bound to the kernel in the future.

Product Goals and Scope

We are committed to the principles of the open source approach and we strongly believe in building a healthy and inclusive community. We insist that good ideas can come from anywhere. Only by including different perspectives can we make the best decisions. While the first release of IvorySQL focuses on Oracle-compatible features, the future roadmap and feature set will be defined by the community in an open source manner.

Core Features

IvorySQL is developed based on PostgreSQL database and is compatible with Oracle database for strong compatibility. Suitable for PostgreSQL database and Oracle database scenarios.

Competitive Advantages

- Core Open Source: IvorySQL’ s core code including compatible features are all open under the open source protocol, with no vendor restrictions. It is also used in Hankook database company instances and has an active developer community.
- Oracle compatible: Oracle databases can be migrated to IvorySQL.
- Customizable: Simply download the code and customize it the way you want.
- Easy to use: For system administrators, IvorySQL dramatically reduces the cost of administration and maintenance. For developers, IvorySQL provides a simple interface, a minimalist solution, and seamless integration with third-party tools. For data analysis professionals, IvorySQL provides easy access to data.
- Hemco Support: Powered by the leading PostgreSQL database provider, Hemco.

Technical Ecology

IvorySQL is based on PostgreSQL, with complete SQL, rock-solid reliability and a huge ecosystem.

Core Application Scenarios

Ivory database’ s main application scenarios.

- Enterprise database

```
For example, ERP, transaction system, financial system involves funds, customers
and other information, data cannot be lost and business logic is complex. Choosing
IvorySQL as the underlying data storage can help you provide high availability
under the premise of data consistency, and you can implement complex business logic
with simple programming.
```

- Applications with LBS

```
Large-scale games, O2O and other applications need to support world map, nearby
businesses, distance between two points and other capabilities. PostGIS adds
support for geographic objects, allowing you to run location queries in SQL without
complex coding, helping you to rationalize your logic more easily, implement LBS
more conveniently, and improve user stickiness.
```

- Data Warehousing and Big Data

```
With more data types and powerful computing power, IvorySQL makes it easier for you
to build a database warehouse or big data analytics platform to enhance your
business operations.
```

- Website or App Building

```
IvorySQL's good performance and powerful features can effectively improve website
performance and reduce development difficulty.
```

- Database Migration

```
If you need to migrate Oracle database to PostgreSQL database, you can directly use
IvorySQL database for migration.
```

Main, Basic Features

IvorySQL is a powerful open source object-relational database management system (ORDBMS). Used to store data securely, support best practices, and allow them to be retrieved when requests are processed. In addition, it is also compatible with Oracle’ s syntax, which is suitable for scenarios where Oracle is used.

Compatibility with Oracle

- Case conversion
- LIKE operator
- anonymous block
- functions and stored procedures
- Built-in data types and built-in functions
- ports and IP
- XML Function
- sequence

- Package
- Invisible Columns
- RowID Column
- OUT Parameter
- %Type & %Rowtype
- NLS Parameters
- Force View
- Nested Subfunctions
- sys_guid Function
- Empty String to NULL
- CALL INTO

Welcome

IvorySQL is the only Oracle compatible open source PostgreSQL.

Getting Started

Get started by downloading the code from Github.

Releases

Go to IvorySQL Release Page.

About IvorySQL

IvorySQL project is an open source project proposed by Highgo Software to add the Oracle compatibility features into the popular PostgreSQL database.

It is Apache licensed Open Source and always free to use. Any comments please contact support@ivorysql.org

Docs Download

IvorySQL v5.4 pdf documentation

Release

Version Overview

[Release Date: June 17, 2026]

IvorySQL 5.4 is built on PostgreSQL 18.4 and includes a variety of bug fixes. For a comprehensive list of updates, please visit our [documentation site](https://docs.ivorysql.org/).

Enhancements

- PostgreSQL 18.4
    1. Prevent unbounded recursion while processing startup packets.
    2. Fix assorted integer overflows in memory-allocation calculations.
    3. Properly quote subscription names in pg_createsubscriber.
    4. Properly quote object names in logical replication origin checks.

5. Reject over-length options in `ts_headline()`.
6. Detect faulty input when restoring attribute MCV statistics.

For further details, visit PostgreSQL's release notes.

New Features

- Upgrade to PG 18.4 kernel: Feature #1344
  PostgreSQL kernel upgraded to version 18.4.
- [IvorySQL Skills](https://github.com/IvorySQL/skills) Empowering AI assistants to become IvorySQL database experts.
- Online Experience: IvorySQL 5.4: Feature #10
  An interactive, browser-based environment will be launched to allow users to explore and evaluate IvorySQL 5.4 in real time — no installation required.
- Packaging for All Platforms:
  Provides multi-platform media packages for X86, ARM, MIPS, LoongArch architecture.
- Containerized Deployment Support
  Supports one-click deployment of standalone or highly available IvorySQL 5.4 clusters via Docker Compose, Podman, Swarm, Helm, and the IvorySQL 5.4 Operator.
- IvorySQL Cloud 5.4 released (Visual Database Lifecycle Management Platform)
  Offers a fully managed, graphical control plane that handles IvorySQL 5.4 database subscriptions, orchestrates end-to-end lifecycle operations, and integrates surrounding ecosystem services.
- PostgreSQL Extensions
  Support for 10 Additional PostgreSQL Extensions: pg_cron, pgAudit, PostGIS, pgRouting, PGroonga, ddlx, pgsql-http, system_stats, plpgsql_check, pgvector.
- Enhanced the EXTRACT function support to accept additional syntax variations in SQL queries: PR #1222

Source Code

IvorySQL's development is maintained across four main repositories:

- IvorySQL database source code: https://github.com/IvorySQL/IvorySQL
- IvorySQL official website: https://github.com/IvorySQL/Ivory-www
- IvorySQL documentation: https://github.com/IvorySQL/IvorySQL_docs
- IvorySQL Docker: https://github.com/IvorySQL/docker_library

Contributors

The following individuals (in alphabetical order) have contributed to this release as patch authors, committers, reviewers, testers, or reporters of issues.

- 3296442201
- Adria0615
- anapplebeee
- Cédric Villemain
- Fawei Zhao
- Ge Sui
- Grant Zhou
- hanjianqiao

- jerome-peng
- liketoochao
- movead123
- msdnchina
- Pierre Forstmann
- Rophy Tsai
- Shawn Yan
- Shuntian Jiao
- Steven Niu
- tian1220A
- xiaoyu509
- Xiaohui Liu
- Xiangyu Liang
- Xinjie LYU
- Xueyu Gao
- yes7rose
- Yasir Hussain Shah
- z20081120087
- Zhe Zhang
- Zhenhao Pan
- Zheng Tao
- Zhuoyan Shi
- zuinuan123

Get Started

.1. Quick Start

Environmental requirements

- Hardware

Parameter	Minimum	Recommended
CPU	4 cores	16 cores
RAM	4GB	64GB
Storage	800MB,HDD	5GB+,SSD or NvMe
Network	Gigabit network	10G network

- Software

  Currently, IvorySQL supports but is not limited to linux(CentOS 8.X/CentOS Stream 9/Ubuntu).

Quick installation

```
The operating system used for the quick start is CentOS Stream 9.
```

yum installation

- Pre-requirements

  Before getting started, please create an user and grant it root privileges. All the installation steps will be performed by this user. Here we just name it 'ivorysql'. How to create a sudo user

- installation

  Create or edit IvorySQL yum repository configuration /etc/yum.repos.d/ivorysql.repo

```
vim /etc/yum.repos.d/ivorysql.repo
[ivorysql5]
name=IvorySQL Server 5 $releasever - $basearch
baseurl=https://yum.highgo.com/dists/ivorysql-rpms/5/redhat/rhel-$releasever-$basearch
enabled=1
gpgcheck=0
```

After saving and exiting, you can install IvorySQL 4 with the following steps

```
$ sudo dnf install -y ivorysql5-5.4
```

- Setting environment variables

  Add below contents in ~/.bash_profile file and source to make it effective:

```
PATH=/usr/ivory-5/bin:$PATH
export PATH
PGDATA=/usr/ivory-5/data
export PGDATA
```

```
$ source ~/.bash_profile
```

- Initializing database

```
$ initdb -D /usr/ivory-5/data
```

```
  The -D option specifies the directory where the database cluster should be stored.
This is the only information required by initdb, but you can avoid writing it by
setting the PGDATA environment variable, which can be convenient since the database
server can find the database directory later by the same variable.

  For more options, refer to initdb --help.
```

- Starting IvorySQL service

```
$ pg_ctl -D /usr/ivory-5/data -l ivory.log start
```

```
The -D option specifies the file system location of the database configuration files.
If this option is omitted, the environment variable PGDATA in <<setting-environment-
variables>> is used. -l option appends the server log output to filename. If the file
does not exist, it is created.
```

```
For more options, refer to pg_ctl --help.
```

Confirm it’s successfully started:

```
$ ps -ef | grep postgres
ivorysql  3214     1  0 20:35 ?        00:00:00 /usr/ivory-5/bin/postgres -D
/usr/ivory-5/data
ivorysql  3215  3214  0 20:35 ?        00:00:00 postgres: checkpointer
ivorysql  3216  3214  0 20:35 ?        00:00:00 postgres: background writer
ivorysql  3218  3214  0 20:35 ?        00:00:00 postgres: walwriter
ivorysql  3219  3214  0 20:35 ?        00:00:00 postgres: autovacuum launcher
ivorysql  3220  3214  0 20:35 ?        00:00:00 postgres: logical replication launcher
ivorysql  3238  1551  0 20:35 pts/0    00:00:00 grep --color=auto postgres
```

Running IvorySQL in docker

- Get IvorySQL image from Docker Hub

```
$ docker pull ivorysql/IvorySQL 5.4-ubi8
```

- Running IvorySQL

```
$ docker run --name ivorysql -p 5434:5432 -e IVORYSQL_PASSWORD=your_password -d
ivorysql/IvorySQL 5.4-ubi8
```

- Check if the IvorySQL container is running successfully

```
$ docker ps | grep ivorysql
CONTAINER ID   IMAGE              COMMAND                  CREATED          STATUS
PORTS                              NAMES
6faa2d0ed705   IvorySQL 5.4-ubi8   "docker-entrypoint.s…"   50 seconds ago   Up 49
seconds   5866/tcp, 0.0.0.0:5434->5432/tcp   ivorysql
```

Connecting to IvorySQL

Connect to IovrySQL via psql:

```
$ psql -d YOUR_DATABASE
psql (18.4)
Type "help" for help.

ivorysql=#
```

```
  The -d option specifies the name of the database to connect to. ivorysql is the
default database of IvorySQL. However,IvorySQL of lower versions need the users
themselves to connect to postgres database at the first connection and then create the
ivorysql database.The latest IvorySQL can do all these for users.

  For more options, refer to psql --help.
```



When running IvorySQL in Docker, additional parameters need to be added, like: psql -d ivorysql -U ivorysql -h 127.0.0.1 -p 5434

Now you can start your journey of IvorySQL! Enjoy!

To explore additional installation methods, please refer to the Installation.

Deploy

.1. Installation

Introduction

The installation methods for IvorySQL include the following five:

- Yum installation
- Docker installation
- rpm installation
- Source code installation
- deb installation

This chapter will provide detailed instructions on the installation, execution, and uninstallation processes for each method. For a quicker access to IvorySQL, please refer to Quick installation.

Before getting started, please create an user and grant it root privileges. All the installation steps will be performed by this user. Here we just name it 'ivorysql'.

Yum installation

Create or edit IvorySQL yum repository configuration /etc/yum.repos.d/ivorysql.repo

```
vim /etc/yum.repos.d/ivorysql.repo
[ivorysql5]
name=IvorySQL Server 5 $releasever - $basearch
baseurl=https://yum.highgo.com/dists/ivorysql-rpms/5/redhat/rhel-$releasever-$basearch
enabled=1
gpgcheck=0
```

After saving and exiting, you can install IvorySQL 4 with the following steps

```
$ sudo dnf install -y ivorysql5-5.4
```

Docker installation

- Get IvorySQL image from Docker Hub

```
$ docker pull ivorysql/IvorySQL 5.4-ubi8
```

- Run IvorySQL

```
$ docker run --name ivorysql -p 5434:5432 -e IVORYSQL_PASSWORD=your_password -d
ivorysql/IvorySQL 5.4-ubi8
```

-e Parameter Explanation

Parameter Name	Required	Description
IVORYSQL_USER	No	Database user, default is ivorysql
IVORYSQL_PASSWORD	yes	Database password
IVORYSQL_DB	no	Database name,default is ivorysql
POSTGRES_HOST_AUTH_METHOD	no	Modify host authentication method,reference value:md5
POSTGRES_INITDB_ARGS	no	Add additional parameters to initdb,reference value:"--data-checksums"
PGDATA	no	Define the data directory to be located in another path or folder (e.g., subdirectory), defaulting to /var/lib/ivorysql/data
POSTGRES_INITDB_WALDIR	no	Define the IvorySQL transaction folder path, which defaults to a subdirectory within the data directory (PGDATA)



1. It is not recommended to set the POSTGRES_HOST_AUTH_METHOD parameter to trust, as this will make the IVORYSQL_PASSWORD setting ineffective.
2. If the POSTGRES_HOST_AUTH_METHOD parameter is set to scram-sha-256, it is also necessary to set POSTGRES_INITDB_ARGS to --auth-host=scram-sha-256 to ensure proper initialization of the database.

rpm installation

- Installing dependencies

```
$ sudo dnf install -y lz4 libicu libxslt python3
```

- Getting rpms

```
$ sudo wget
https://github.com/IvorySQL/IvorySQL/releases/download/IvorySQL_5.4/IvorySQL-5.4-
9d890e9-20251120.x86_64.rpm
```

- Installing rpms

  Use the following command to install all the rpms:

```
$ sudo yum --disablerepo=* localinstall *.rpm
```

IvorySQL then will be installed in the /usr/ivory-5/ directory.

Source code installation

- Installing dependencies

```
# CentOS
```

```
$ sudo dnf install -y bison readline-devel zlib-devel openssl-devel uuid-devel
$ sudo dnf groupinstall -y 'Development Tools'

# Ubuntu
$ sudo apt install -y bison libreadline-dev zlib1g-dev libssl-dev
$ sudo apt install -y build-essential
$ sudo apt install -y pkg-config uuid-dev
```

- Getting source code

```
$ git clone https://github.com/IvorySQL/IvorySQL.git
$ cd IvorySQL
$ git checkout -b IVORY_REL_5_STABLE origin/IVORY_REL_5_STABLE
```

- Configuring

  In the IvorySQL directory run the following command with --prefix to specify the directory where you want the database to be installed:

```
$ ./configure --prefix=/usr/local/ivorysql/ivorysql-5
```

- Compiling

  Run the following command to compile the source code:

```
$ make
```



When the compilation is completed, you can test the result with 'make check' or 'make all-check-world' before your installation.

- Installing

  Run the following command to install the database system, IvorySQL then will be installed in the directory specified by --prefix:

```
$ sudo make install
```

deb installation

- Installing dependencies

```
$ sudo apt -y install pkg-config libreadline-dev libicu-dev libldap2-dev uuid-dev tcl-dev libperl-dev python3-dev bison flex openssl libssl-dev libpam-dev libxml2-dev libxslt-dev libossp-uuid-dev libselinux-dev gettext
```

- Getting deb

```
$ sudo wget
https://github.com/IvorySQL/IvorySQL/releases/download/IvorySQL_5.4/IvorySQL-5.4-
a50789d-20250304.amd64.deb
```

- Installing deb

```
$ sudo dpkg -i IvorySQL-5.4-a50789d-20250304.amd64.deb
```

IvorySQL will then be installed in the /usr/ivory-5/ directory.

Start Database

Users following the instructions in Yum installation, rpm installation, Source code installation and deb installation need to manually start the database.

- Granting permissions

Execute the following command to grant permissions to the installation user. The example user is ivorysql, and the installation directory is /usr/ivory-5/:

```
$ sudo chown -R ivorysql:ivorysql /usr/ivory-5/
```

- Setting environment variables

  Add below contents in ~/.bash_profile file and source to make it effective:

```
PATH=/usr/ivory-5/bin:$PATH
export PATH
PGDATA=/usr/ivory-5/data
export PGDATA
```

```
$ source ~/.bash_profile
```

- Initializing database

```
$ mkdir /usr/ivory-5/data
$ initdb -D /usr/ivory-5/data
```

```
  The -D option specifies the directory where the database cluster should be stored.
This is the only information required by initdb, but you can avoid writing it by
setting the PGDATA environment variable, which can be convenient since the database
server can find the database directory later by the same variable.

  For more options, refer to initdb --help.
```

- Starting IvorySQL service

```
$ pg_ctl -D /usr/ivory-5/data -l ivory.log start
```

The -D option specifies the file system location of the database configuration files. If this option is omitted, the environment variable PGDATA in [_4-1:::setting-environment-variables] is used. -l option appends the server log output to filename. If the file does not exist, it is created.

For more options, refer to pg_ctl --help.

Confirm it’ s successfully started:

```
$ ps -ef | grep postgres
ivorysql  130427       1  0 02:45 ?        00:00:00 /usr/ivory-5/bin/postgres -D
/usr/ivory-5/data
ivorysql  130428  130427  0 02:45 ?        00:00:00 postgres: checkpointer
ivorysql  130429  130427  0 02:45 ?        00:00:00 postgres: background writer
ivorysql  130431  130427  0 02:45 ?        00:00:00 postgres: walwriter
ivorysql  130432  130427  0 02:45 ?        00:00:00 postgres: autovacuum launcher
ivorysql  130433  130427  0 02:45 ?        00:00:00 postgres: logical replication
launcher
ivorysql  130445  130274  0 02:45 pts/1    00:00:00 grep --color=auto postgres
```

Connecting to IvorySQL

Connect to IovrySQL via psql:

```
$ psql -d YOUR_DATABASE
psql (18.4)
Type "help" for help.

ivorysql=#
```

```
  The -d option specifies the name of the database to connect to. ivorysql is the
default database of IvorySQL. However,IvorySQL of lower versions need the users
themselves to connect to postgres database at the first connection and then create the
ivorysql database.The latest IvorySQL can do all these for users.

  For more options, refer to psql --help.
```



When running IvorySQL in Docker, additional parameters need to be added, like: psql -d ivorysql -U ivorysql -h 127.0.0.1 -p 5434

Uninstallation



No matter which method is used for the uninstallation, make sure the service has been

> stopped cleanly and your data has been backed up safely.

Uninstallation for yum installation

Run the following commands in turn and clean the residual folders:

```
$ sudo dnf remove -y ivorysql5-5.4
```

Uninstallation for docker installation

Stop IvorySQL container and remove IvorySQL image:

```
$ docker stop ivorysql
$ docker rm ivorysql
$ docker rmi ivorysql/IvorySQL 5.4-ubi8
```

Uninstallation for rpm installation

Uninstall the rpms and clear the residual folders:

```
$ sudo yum remove --disablerepo=* ivorysql5\*
$ sudo rm -rf IvorySQL-5.4
```

Uninstallation for source code installation

Uninstall the database system, then clear the residual folders:

```
$ sudo make uninstall
$ make clean
$ sudo rm -rf IvorySQL-5.4
```

Uninstallation for deb installation

Uninstall the database system, then clear the residual folders:

```
$ sudo dpkg -P IvorySQL-5.4
$ sudo rm -rf IvorySQL-5.4
```

.2. Cluster

Primary node

Installing and start database

For quick database installation by yum, please refer to Quick installation。

For more installation options, please refer to Installation。



The master node database needs to be installed and started.

Stopping firewall

Stop firewall for all the nodes in the cluster to ensure the communication:

```
$ sudo systemctl stop firewalld
```

Setting environment variables

To create the streaming replication, we need configure the postgresql.conf and pg_hba.conf files on the primary node.

- postgresql.conf

  Append the following contents to the end of postgresql.conf:

```
listen_addresses = '*'
max_connections = 100
wal_level = replica
max_wal_senders = 5
hot_standby = on
```

- pg_hba.conf

  Append the following contents to the end of pg_hba.conf:

```
host all all 0.0.0.0/0 trust
host replication all 0.0.0.0/0 trust
```



The configuration of pg_hba in the example is only for demo purposes and testing. This configuration will result in invalidation of the database password. Please configure according to the actual environment.

Restarting IvorySQL sevice

```
$ pg_ctl restart
```

Standby node

Installing database

For quick database installation by yum, please refer to Quick installation。

For more installation options, please refer to Installation。



The standby node database only needs to be installed and not started.

Stopping firewall

Stop firewall for all the nodes in the cluster to ensure the communication:

```
$ sudo systemctl stop firewalld
```

Building streaming replication

Run below command on the standby node to take base backups of the primary, that is, to build a streaming replication:

```
$ sudo pg_basebackup -F p -P -X fetch -R -h <primary_ip> -p <primary_port> -U ivorysql
-D /usr/local/ivorysql/ivorysql-5/data
```

- Specifies the host name of the machine on which the server is running;
- Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections. Defaults is 5432;
- User name to connect as;
- Directory to write the output to. pg_basebackup will create the directory and any parent directories if necessary. The directory may already exist, but it is an error if the directory already exists and is not empty.

For more options, refer to pg_basebackup --help.

Setting environment variables

Add below contents in ~/.bash_profile file:

```
PATH=/usr/local/ivorysql/ivorysql-5/bin:$PATH
export PATH
PGDATA=/usr/local/ivorysql/ivorysql-5/data
export PGDATA
```

Source to make it effective:

```
$ source ~/.bash_profile
```

Starting IvorySQL sevice

```
$ sudo pg_ctl -D /usr/local/ivorysql/ivorysql-5/data start
```

Experience the IvorySQL cluster

Checking cluster status

Run below command on the primary node, you will see walsender:

```
$ ps -ef |grep postgres
```

```
...
ivorysql 11176  8067  0 21:54 ?        00:00:00 postgres: walsender ivorysql
192.168.31.102(53416) streaming 0/7000060...
```

while it is walreceiver on standby:

```
$ ps -ef | grep postgres
...
ivorysql  6567  6139  0 21:54 ?        00:00:00 postgres: walreceiver streaming
0/7000060
...
```

On the primary node, connect to IvorySQL and show the status:

```
$ psql -d ivorysql
psql (18.4)
Type "help" for help.


ivorysql=# select * from pg_stat_replication;
  pid  | usesysid | usename  | application_name |  client_addr   | client_hostname |
client_port |         backend_start         | backend_
xmin |   state   | sent_lsn  | write_lsn | flush_lsn | replay_lsn | write_lag |
flush_lag | replay_lag | sync_priority | sync_state |
    reply_time
-------+----------+----------+------------------+----------------+-----------------+
-------------+--------------------------------+---------
-----+-----------+-----------+-----------+-----------+------------+-----------+
-----------+------------+---------------+------------+------
-------------------------
 11176 |       10 | ivorysql | walreceiver      | 192.168.31.102 |                 |
53416 | 2024-12-18 21:54:52.041847-05 |
     | streaming | 0/7000148 | 0/7000148 | 0/7000148 | 0/7000148  |           |
|            |             0 | async      | 2024-
12-18 22:52:07.325111-05
(1 row)
```

Here 192.168.31.102 is the ip address of the standby node, and async means the data synchronization method is asynchronous.

Using the cluster

All writing operations are performed on the primary node, while reading can be on both primary and standby. The data on primary is synchronized to standby through streaming replication. The writing result can be queried on all the nodes in the cluster.

Below is an example. Create a new database test on primary and query:

```
$ psql -d ivorysql
psql (18.4)
Type "help" for help.

ivorysql=# create database test;
CREATE DATABASE
ivorysql=# \l
                                                        List of databases
   Name    |  Owner   | Encoding | Locale Provider |   Collate   |    Ctype    | ICU
Locale | ICU Rules |   Access privileges
-----------+----------+----------+-----------------+-------------+-------------+
------------+-----------+-----------------------
 ivorysql  | ivorysql | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |
|           |
 template0 | ivorysql | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |
|           | =c/ivorysql          +
          |          |          |                 |             |             |
|           | ivorysql=CTc/ivorysql
 template1 | ivorysql | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |
|           | =c/ivorysql          +
          |          |          |                 |             |             |
|           | ivorysql=CTc/ivorysql
 test      | ivorysql | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |
|           |
(4 rows)
```

Query on the standby node:

```
$ psql -d ivorysql
psql (18.4)
Type "help" for help.

ivorysql=# \l
                                                        List of databases
   Name    |  Owner   | Encoding | Locale Provider |   Collate   |    Ctype    | ICU
Locale | ICU Rules |   Access privileges
-----------+----------+----------+-----------------+-------------+-------------+
------------+-----------+-----------------------
 ivorysql  | ivorysql | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |
|           |
 template0 | ivorysql | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |
|           | =c/ivorysql          +
          |          |          |                 |             |             |
|           | ivorysql=CTc/ivorysql
 template1 | ivorysql | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |
```

```
|           | =c/ivorysql          +
          |          |          |                |             |             |
|           | ivorysql=CTc/ivorysql
 test      | ivorysql | UTF8     | libc           | en_US.UTF-8 | en_US.UTF-8 |
|           |
(4 rows)
```

Oracle Compatibility

.1. 1、Case conversion

Objective

- In order to meet the case compatibility of PG and Oracle’ s reference identifiers, ivorysql has designed three case conversion modes for reference identifiers. Select the conversion mode via the GUC parameter "identifier_case_switch";

Function

Three modes of case conversion (interchange by default)

- If the value of the guc parameter "identifier_case_switch" is "normal":

```
1). The letters in the identifier referenced by the double quotation mark are left unchanged.
```

- If the value of the guc parameter "identifier_case_switch" is "interchange":

```
1). If the letters in the identifier referenced by the double quotation mark are all uppercase, uppercase is converted to lowercase.
```

```
2). If the letters in the identifier referenced by the double quotation mark are all lowercase, lowercase is converted to uppercase.
```

```
3). If the letters in the identifier enclosed in double quotation marks are mixed-case, the identifier is left unchanged.
```

- If the value of the guc parameter "identifier_case_switch" is "lowercase":

```
1). If the letters in the identifier referenced by the double quotation mark are all uppercase, uppercase is converted to lowercase.
```

```
2). If the letters in the identifier enclosed in double quotation marks are mixed-case, the identifier is left unchanged.
```

When the database cluster is initialized

- Add the -C option to the initdb program to set the case conversion mode, and the corresponding value of -C is:

```
"normal"   ------ "0"synonymy
```

```
"interchange" ------ "1"synonymy
```

```
"lowercase"  ------ "2"synonymy
```

During initialization of the database cluster, the case conversion pattern is saved to the global/pg_control file in the data directory;

Use Cases

normal

```
ivorysql=# SET ivorysql.compatible_mode to oracle;
SET

ivorysql=# SET ivorysql.enable_case_switch = true;
SET

ivorysql=# SET ivorysql.identifier_case_switch = normal;
SET

ivorysql=# CREATE TABLE "NORMAL_1"(c1 int, c2 int);
CREATE TABLE

ivorysql=# CREATE TABLE "Normal_2"(c1 int, c2 int);
CREATE TABLE

ivorysql=# CREATE TABLE "normal_3"(c1 int, c2 int);
CREATE TABLE

ivorysql=# select * from "NORMAL_1";
 c1 | c2
----+----
(0 rows)

ivorysql=# select * from "Normal_1";
ERROR:  relation "Normal_1" does not exist
LINE 1: select * from "Normal_1";

ivorysql=# select * from "normal_1";
ERROR:  relation "normal" does not exist
LINE 1: select * from "normal";
```

```
ivorysql=# select * from NORMAL_1;
ERROR:  relation "normal_1" does not exist
LINE 1: select * from NORMAL_1;

ivorysql=# select * from "Normal_2";
 c1 | c2
----+----
(0 rows)

ivorysql=# select * from "NORMAL_2";
ERROR:  relation "NORMAL_2" does not exist
LINE 1: select * from "NORMAL_2";

ivorysql=# select * from "normal_2";
ERROR:  relation "normal_2" does not exist
LINE 1: select * from "normal_2";

ivorysql=# select * from Normal_2;
ERROR:  relation "normal_2" does not exist
LINE 1: select * from Normal_2;

ivorysql=# select * from "normal_3";
 c1 | c2
----+----
(0 rows)

ivorysql=# select * from "NORMAL_3";
ERROR:  relation "NORMAL_3" does not exist
LINE 1: select * from "NORMAL_3";

ivorysql=# select * from "Normal_3";
ERROR:  relation "Normal_3" does not exist
LINE 1: select * from "Normal_3";

ivorysql=# drop table "NORMAL_1";
DROP TABLE
ivorysql=# drop table "Normal_2";
DROP TABLE
ivorysql=# drop table "normal_3";
DROP TABLE
```

interchange

```
ivorysql=# SET ivorysql.compatible_mode to oracle;
SET

ivorysql=# SET ivorysql.enable_case_switch = true;
SET

ivorysql=# SET ivorysql.identifier_case_switch = interchange;
SET

ivorysql=# CREATE TABLE "INTER_CHANGE_1"(c1 int, c2 int);
CREATE TABLE

ivorysql=# CREATE TABLE "Inter_Change_2"(c1 int, c2 int);
CREATE TABLE

ivorysql=# CREATE TABLE "inter_change_3"(c1 int, c2 int);
CREATE TABLE

ivorysql=# select * from "INTER_CHANGE_1";
 c1 | c2
----+----
(0 rows)

ivorysql=# select * from "Inter_Change_1";
ERROR:  relation "Inter_Change_1" does not exist
LINE 1: select * from "Inter_Change_1";

ivorysql=# select * from "inter_change_1";
ERROR:  relation "INTER_CHANGE_1" does not exist
LINE 1: select * from "inter_change_1";

ivorysql=# select * from INTER_CHANGE_1;
 c1 | c2
----+----
(0 rows)

ivorysql=# select * from "Inter_Change_2";
 c1 | c2
----+----
(0 rows)

ivorysql=# select * from "INTER_CHANGE_2";
```

```
ERROR:  relation "inter_change_2" does not exist
LINE 1: select * from "INTER_CHANGE_2";

ivorysql=# select * from "inter_change_2";
ERROR:  relation "INTER_CHANGE_2" does not exist
LINE 1: select * from "inter_change_2";

ivorysql=# select * from Inter_Change_2;
ERROR:  relation "inter_change_2" does not exist
LINE 1: select * from Inter_Change_2;

ivorysql=# select * from "inter_change_3";
 c1 | c2
----+----
(0 rows)

ivorysql=# select * from "INTER_CHANGE_3";
ERROR:  relation "inter_change_3" does not exist
LINE 1: select * from "INTER_CHANGE_3";

ivorysql=# select * from "Inter_Change_3";
ERROR:  relation "Inter_Change_3" does not exist
LINE 1: select * from "Inter_Change_3";

ivorysql=# select * from inter_change_3;
ERROR:  relation "inter_change_3" does not exist
LINE 1: select * from "INTER_CHANGE_3";

ivorysql=# drop table "INTER_CHANGE_1";
DROP TABLE
ivorysql=# drop table "Inter_Change_2";
DROP TABLE
ivorysql=# drop table "inter_change_3";
DROP TABLE
```

lowercase

```
ivorysql=# SET ivorysql.compatible_mode to oracle;
SET

ivorysql=# SET ivorysql.enable_case_switch = true;
SET
```

```
ivorysql=# SET ivorysql.identifier_case_switch = lowercase;
SET

ivorysql=# CREATE TABLE "LOWER_CASE_1"(c1 int, c2 int);
CREATE TABLE

ivorysql=# CREATE TABLE "Lower_Case_2"(c1 int, c2 int);
CREATE TABLE

ivorysql=# CREATE TABLE "lower_case_3"(c1 int, c2 int);
CREATE TABLE

ivorysql=# select * from "LOWER_CASE_1";
 c1 | c2
----+----
(0 rows)

ivorysql=# select * from "Lower_Case_1";
ERROR:  relation "Lower_Case_1" does not exist
LINE 1: select * from "Lower_Case_1";

ivorysql=# select * from "lower_case_1";
 c1 | c2
----+----
(0 行记录)


ivorysql=# select * from LOWER_CASE_1;
 c1 | c2
----+----
(0 行记录)


ivorysql=# select * from "Lower_Case_2";
 c1 | c2
----+----
(0 rows)

ivorysql=# select * from "LOWER_CASE_2";
ERROR:  relation "lower_case_2" does not exist
LINE 1: select * from "LOWER_CASE_2";
```

```
ivorysql=# select * from "lower_case_2";
ERROR:  relation "lower_case_2" does not exist
LINE 1: select * from "lower_case_2";

ivorysql=# select * from Lower_Case_2;
ERROR:  relation "lower_case_2" does not exist
LINE 1: select * from Lower_Case_2;

ivorysql=# select * from "lower_case_3";
 c1 | c2
----+----
(0 rows)

ivorysql=# select * from "LOWER_CASE_3";
 c1 | c2
----+----
(0 rows)

ivorysql=# select * from "Lower_Case_3";
ERROR:  relation "Lower_Case_3" does not exist
LINE 1: select * from "Lower_Case_3";

ivorysql=# select * from LOWER_CASE_3;
 c1 | c2
----+----
(0 行记录)

ivorysql=# drop table "NORMAL_1";
DROP TABLE
ivorysql=# drop table "Normal_2";
DROP TABLE
ivorysql=# drop table "normal_3";
DROP TABLE
```

.2. 2、LIKE operator

Objective

- This document is intended to provide people using like fuzzy queries with an in-depth understanding of Oracle-compatible fuzzy query like implementations.

Function description

Database name	Like fuzzy queries
oracle	oracle’s string type is varchar2, which supports fuzzy queries using the Like keyword with wildcards for columns of number, date, and string field types
IvorySQL	The basic type of IvorySQL’s string is text, so like is based on text, and other IvorySQL types can be implicitly converted to text, so that they can be automatically converted without creating operator

Test cases

```
create table t_ora_like (id int ,str1 varchar(8), date1 timestamp with time zone,
date2 time with time zone, num int, str2 varchar(8));
insert into t_ora_like (id ,str1 ,date1 ,date2) values (123456,'test1','2022-09-26
16:39:20','2022-09-26 16:39:20');
insert into t_ora_like (id ,str1 ,date1 ,date2) values (123457,'test2','2022-09-26
16:40:20','2022-09-26 16:40:20');
insert into t_ora_like (id ,str1 ,date1 ,date2) values (223456,'test3','2022-09-26
16:41:20','2022-09-26 16:41:20');
insert into t_ora_like (id ,str1 ,date1 ,date2) values (123458,'test4','2022-09-26
16:42:20','2022-09-26 16:42:20');

select * from t_ora_like where str1 like 'test%';
   id   | str1  |               date1               |    date2    | num | str2
--------+-------+-----------------------------------+-------------+-----+------
 123456 | test1 | 2022-09-26 16:39:20.000000 +08:00 | 16:39:20+08 |     |
 123457 | test2 | 2022-09-26 16:40:20.000000 +08:00 | 16:40:20+08 |     |
 223456 | test3 | 2022-09-26 16:41:20.000000 +08:00 | 16:41:20+08 |     |
 123458 | test4 | 2022-09-26 16:42:20.000000 +08:00 | 16:42:20+08 |     |
(4 rows)

select * from t_ora_like where date1 like '2022%';
   id   | str1  |               date1               |    date2    | num | str2
--------+-------+-----------------------------------+-------------+-----+------
 123456 | test1 | 2022-09-26 16:39:20.000000 +08:00 | 16:39:20+08 |     |
 123457 | test2 | 2022-09-26 16:40:20.000000 +08:00 | 16:40:20+08 |     |
 223456 | test3 | 2022-09-26 16:41:20.000000 +08:00 | 16:41:20+08 |     |
 123458 | test4 | 2022-09-26 16:42:20.000000 +08:00 | 16:42:20+08 |     |
(4 rows)

select * from t_ora_like where date2 like '16%';
   id   | str1  |               date1               |    date2    | num | str2
--------+-------+-----------------------------------+-------------+-----+------
 123456 | test1 | 2022-09-26 16:39:20.000000 +08:00 | 16:39:20+08 |     |
```

```
 123457 | test2 | 2022-09-26 16:40:20.000000 +08:00 | 16:40:20+08 |     |
 223456 | test3 | 2022-09-26 16:41:20.000000 +08:00 | 16:41:20+08 |     |
 123458 | test4 | 2022-09-26 16:42:20.000000 +08:00 | 16:42:20+08 |     |
(4 rows)

select * from t_ora_like where id like '123%';
   id   | str1  |               date1               |    date2    | num | str2
--------+-------+-----------------------------------+-------------+-----+------
 123456 | test1 | 2022-09-26 16:39:20.000000 +08:00 | 16:39:20+08 |     |
 123457 | test2 | 2022-09-26 16:40:20.000000 +08:00 | 16:40:20+08 |     |
 123458 | test4 | 2022-09-26 16:42:20.000000 +08:00 | 16:42:20+08 |     |
(3 rows)

select * from t_ora_like where id like null;
 id | str1 | date1 | date2 | num | str2
----+------+-------+-------+-----+------
(0 rows)
```

.3. 3、anonymous block

Objective

- This document is a design document for the PLSQL anonymous block compatible Oracle syntax function, in order to be compatible with Oracle’ s anonymous block statements in IvorySQL.

Function description

- Anonymous blocks are PLSQL structures that dynamically create and execute procedural code without the need to persistently store the code as database objects in the system directory. In this implementation, IvorySQL is mainly compatible with the syntax format of PLSQL anonymous blocks, and the parts we mainly deal with include client tool psql, master server and PSQL side support.

Test cases

```
declare
i integer := 10;
begin
 raise notice '%', i;
end;
/
NOTICE:  10
```

```
DECLARE
  grade CHAR(1);
BEGIN
```

```
  grade := 'B';
  CASE grade
    WHEN 'A' THEN raise notice 'Excellent';
    WHEN 'B' THEN raise notice 'Very Good';
  END CASE;
EXCEPTION
  WHEN CASE_NOT_FOUND THEN
    raise notice 'No such grade';
END;
/
NOTICE:  Very Good
```

.4. 4、functions and stored procedures

Purpose

PostgreSQL supports functions and stored procedures, but there are syntax differences between PostgreSQL and Oracle. To enable Oracle’ s PL/SQL statements to run on IvorySQL—i.e., to achieve "syntax compatibility"—IvorySQL adopts the following solution: If an Oracle clause has a counterpart clause with the same function in IvorySQL, it is directly mapped to the corresponding IvorySQL clause; otherwise, only the syntax of the Oracle clause is implemented, while its function is not.

Compatibility support

The specific Oracle compatibility support includes the following aspects:

EDITIONABLE/NONEDITIONABLE is supported in CREATE FUNCTION syntax

```
CREATE or replace EDITIONABLE FUNCTION ora_func RETURN integer AS
BEGIN
    RETURN 1;
END;
/

CREATE or replace NONEDITIONABLE FUNCTION ora_func RETURN integer IS
BEGIN
    RETURN 1;
END;
/
```

Keyword 'RETURN' and 'IS' are supported in CREATE FUNCTION syntax

```
CREATE or replace EDITIONABLE FUNCTION ora_func RETURN integer IS
BEGIN
    RETURN 1;
END;
```

```
/
```

In the CREATE FUNCTION syntax, a function can have no parameters, and no parentheses () after the function name

```
CREATE or replace FUNCTION ora_func RETURN integer
SHARING = METADATA
IS
BEGIN
    RETURN 1;
END;
/
```

Increase the number of parameters in CREATE FUNCTION syntax, the max value can be specified by configure command

```
configure --help
    --with-max-funarg=MAXFUNARG
```

Psql uses slash (/) as terminator of statement in CREATE FUNCTION syntax

```
CREATE or replace EDITIONABLE FUNCTION ora_func RETURN integer IS
BEGIN
    RETURN 1;
END;
/
```

No DECLARE keyword before variable in CREATE FUNCTION syntax

```
CREATE OR REPLACE
FUNCTION ora_func (num1 IN int, num2 IN int)
RETURN int
AS
 num3 int :=10;
 num4 int :=10;
 num5 int;
BEGIN
 num3 := num1 + num2;
 num4 := num1 * num2;
 num5 := num3 * num4;
RETURN num5;
END;
/
```

```
CREATE FUNCTION

select ora_func(5,9)from dual;
 ora_func
----------
      630
(1 row)
```

Support NOCOPY for OUT parameter in CREATE FUNCTION syntax

```
CREATE OR REPLACE FUNCTION test_nocopy(a IN int, b OUT NOCOPY int, c IN OUT NOCOPY
int)
RETURN int
IS
BEGIN
   b := a;
   c := a;
   return 1;
END;
/
```

Support sharing clause in CREATE FUNCTION syntax

```
CREATE or replace FUNCTION ora_func RETURN integer
SHARING = METADATA
IS
BEGIN
    RETURN 1;
END;
/

CREATE or replace FUNCTION ora_func RETURN integer
SHARING = NONE
IS
BEGIN
    RETURN 1;
END;
/
```

Support invoker_rights (AUTHID) in CREATE FUNCTION syntax，and change default permission to be DR（DEFINER）

```
CREATE or replace FUNCTION ora_func RETURN integer
```

```
SHARING = NONE AUTHID CURRENT_USER
IS
BEGIN
    RETURN 1;
END;
/
CREATE or replace FUNCTION ora_func RETURN integer
SHARING = NONE AUTHID DEFINER
IS
BEGIN
    RETURN 1;
END;
/
```

Support ACCESSIBLE BY in CREATE FUNCTION syntax

```
CREATE or replace FUNCTION ora_func RETURN integer
SHARING = NONE AUTHID DEFINER ACCESSIBLE BY ( B )
IS
BEGIN
    RETURN 1;
END;
/
CREATE or replace FUNCTION ora_func RETURN integer
SHARING = NONE AUTHID DEFINER ACCESSIBLE BY ( A.B )
IS
BEGIN
    RETURN 1;
END;
/
CREATE or replace FUNCTION ora_func RETURN integer
SHARING = NONE AUTHID DEFINER ACCESSIBLE BY ( FUNCTION A.B )
IS
BEGIN
    RETURN 1;
END;
/
CREATE or replace FUNCTION ora_func RETURN integer
SHARING = NONE AUTHID DEFINER
ACCESSIBLE BY ( FUNCTION A.B, PROCEDURE C.D )
IS
BEGIN
```

```
    RETURN 1;
END;
/
CREATE or replace FUNCTION ora_func RETURN integer
SHARING = NONE AUTHID DEFINER
ACCESSIBLE BY ( FUNCTION A.B, PROCEDURE C.D, PACKAGE E,
TRIGGER F, TYPE G )
IS
BEGIN
    RETURN 1;
END;
/
```

Support DEFAULT COLLATION in CREATE FUNCTION syntax

```
CREATE or replace FUNCTION ora_func RETURN integer
SHARING = NONE AUTHID DEFINER
ACCESSIBLE BY ( FUNCTION A.B, PROCEDURE C.D )
DEFAULT COLLATION USING_NLS_COMP
IS
BEGIN
    RETURN 1;
END;
/
```

Support deterministic clause in CREATE FUNCTION syntax

```
CREATE or replace FUNCTION ora_func RETURN integer
SHARING = NONE AUTHID DEFINER
ACCESSIBLE BY ( FUNCTION A.B, PROCEDURE C.D )
DEFAULT COLLATION USING_NLS_COMP
DETERMINISTIC
IS
BEGIN
    RETURN 1;
END;
/
```

Support parallel_enable clause in CREATE FUNCTION syntax

```
CREATE or replace FUNCTION ora_func RETURN integer
SHARING = NONE AUTHID DEFINER
```

```
ACCESSIBLE BY ( FUNCTION A.B, PROCEDURE C.D )
DEFAULT COLLATION USING_NLS_COMP
DETERMINISTIC
PARALLEL_ENABLE
IS
BEGIN
    RETURN 1;
END;
/
```

Support result_cache clause in CREATE FUNCTION syntax

```
CREATE or replace FUNCTION ora_func RETURN integer
SHARING = NONE AUTHID DEFINER
ACCESSIBLE BY ( FUNCTION A.B, PROCEDURE C.D )
DEFAULT COLLATION USING_NLS_COMP
DETERMINISTIC
PARALLEL_ENABLE ( PARTITION A BY RANGE ( B, C ) CLUSTER A BY ( E,F ) )
RESULT_CACHE
IS
BEGIN
    RETURN 1;
END;
/
CREATE or replace FUNCTION ora_func RETURN integer
SHARING = NONE AUTHID DEFINER
ACCESSIBLE BY ( FUNCTION A.B, PROCEDURE C.D )
DEFAULT COLLATION USING_NLS_COMP
DETERMINISTIC
PARALLEL_ENABLE ( PARTITION A BY RANGE ( B, C ) CLUSTER A BY ( E,F ) )
RESULT_CACHE RELIES_ON ()
IS
BEGIN
    RETURN 1;
END;
/
CREATE or replace FUNCTION ora_func RETURN integer
SHARING = NONE AUTHID DEFINER
ACCESSIBLE BY ( FUNCTION A.B, PROCEDURE C.D )
DEFAULT COLLATION USING_NLS_COMP
DETERMINISTIC
PARALLEL_ENABLE ( PARTITION A BY RANGE ( B, C ) CLUSTER A BY ( E,F ) )
```

```
RESULT_CACHE RELIES_ON ( data_source1, data_source2)
IS
BEGIN
    RETURN 1;
END;
/
```

Support aggregate clause in CREATE FUNCTION syntax

```
CREATE or replace FUNCTION ora_func RETURN integer
SHARING = NONE AUTHID DEFINER
ACCESSIBLE BY ( FUNCTION A.B, PROCEDURE C.D )
DEFAULT COLLATION USING_NLS_COMP
DETERMINISTIC
PARALLEL_ENABLE ( PARTITION A BY RANGE ( B, C ) CLUSTER A BY ( E,F ) )
RESULT_CACHE RELIES_ON ( data_source1, data_source2)
AGGREGATE USING pg_catalog.int4
IS
BEGIN
    RETURN 1;
END;
/
CREATE or replace FUNCTION ora_func RETURN integer
SHARING = NONE AUTHID DEFINER
ACCESSIBLE BY ( FUNCTION A.B, PROCEDURE C.D )
DEFAULT COLLATION USING_NLS_COMP
DETERMINISTIC
PARALLEL_ENABLE ( PARTITION A BY RANGE ( B, C ) CLUSTER A BY ( E,F ) )
RESULT_CACHE RELIES_ON ( data_source1, data_source2)
AGGREGATE USING int
IS
BEGIN
    RETURN 1;
END;
/
```

Support pipelined clause in CREATE FUNCTION syntax

```
CREATE or replace FUNCTION ora_func RETURN integer
SHARING = NONE AUTHID DEFINER
ACCESSIBLE BY ( FUNCTION A.B, PROCEDURE C.D )
DEFAULT COLLATION USING_NLS_COMP
```

```
DETERMINISTIC
PARALLEL_ENABLE ( PARTITION A BY RANGE ( B, C ) CLUSTER A BY ( E,F ) )
RESULT_CACHE RELIES_ON ( data_source1, data_source2)
AGGREGATE USING int
PIPELINED
IS
BEGIN
    RETURN 1;
END;
/
```

Support sql_macro clause in CREATE FUNCTION syntax

```
CREATE or replace FUNCTION ora_func RETURN integer
SHARING = NONE AUTHID DEFINER
ACCESSIBLE BY ( FUNCTION A.B, PROCEDURE C.D )
DEFAULT COLLATION USING_NLS_COMP
DETERMINISTIC
PARALLEL_ENABLE ( PARTITION A BY RANGE ( B, C ) CLUSTER A BY ( E,F ) )
RESULT_CACHE RELIES_ON ( data_source1, data_source2)
AGGREGATE USING int
PIPELINED TABLE POLYMORPHIC USING pg_catalog.int4
SQL_MACRO
IS
BEGIN
    RETURN 1;
END;
/
```

Compatibility with ALTER FUNCTION syntax

```
alter function public.test_func noneditionable;
alter function test_func compile;
alter function test_func compile debug;
alter function test_func compile debug sd = mv;
alter function test_func compile debug reuse settings;
```

Support EDITIONABLE/NONEDITIONABLE in CREATE PROCEDURE syntax

```
CREATE OR REPLACE EDITIONABLE PROCEDURE ora_procedure
IS
        p integer := 20;
```

```
begin
        raise notice '%', p;
end;
/

CREATE OR REPLACE NONEDITIONABLE PROCEDURE ora_procedure
IS
        p integer := 20;
begin
        raise notice '%', p;
end;
/
```

In the CREATE PROCEDURE syntax, a procedure can have no parameters, and no parentheses () after the procedure name

```
CREATE OR REPLACE EDITIONABLE PROCEDURE ora_procedure
IS
        p integer := 20;
begin
        raise notice '%', p;
end;
/
```

Psql uses slash (/) as terminator of statement in CREATE PROCEDURE syntax

```
CREATE OR REPLACE EDITIONABLE PROCEDURE ora_procedure
IS
        p integer := 20;
begin
        raise notice '%', p;
end;
/
```

Support sharing clause in CREATE PROCEDURE syntax

```
CREATE OR REPLACE PROCEDURE ora_procedure
SHARING = METADATA
IS
        p integer := 20;
begin
        raise notice '%', p;
end;
```

```
/
CREATE OR REPLACE PROCEDURE ora_procedure
SHARING = NONE
IS
        p integer := 20;
begin
        raise notice '%', p;
end;
/
```

Support DEFAULT COLLATION clause in CREATE PROCEDURE syntax

```
CREATE OR REPLACE PROCEDURE ora_procedure
SHARING = METADATA
DEFAULT COLLATION USING_NLS_COMP
IS
        p integer := 20;
begin
        raise notice '%', p;
end;
/
```

Support invoker_rights clause (AUTHID) in CREATE PROCEDURE syntax

```
CREATE OR REPLACE PROCEDURE ora_procedure
SHARING = METADATA
DEFAULT COLLATION USING_NLS_COMP
AUTHID CURRENT_USER
IS
        p integer := 20;
begin
        raise notice '%', p;
end;
/
CREATE OR REPLACE PROCEDURE ora_procedure
SHARING = METADATA
DEFAULT COLLATION USING_NLS_COMP
AUTHID DEFINER
IS
        p integer := 20;
begin
        raise notice '%', p;
```

```
end;
/
```

Support ACCESSIBLE BY clause in CREATE PROCEDURE syntax

```
CREATE OR REPLACE PROCEDURE ora_procedure
SHARING = METADATA
DEFAULT COLLATION USING_NLS_COMP
AUTHID CURRENT_USER
ACCESSIBLE BY ( B )
IS
        p integer := 20;
begin
        raise notice '%', p;
end;
/
CREATE OR REPLACE PROCEDURE ora_procedure
SHARING = METADATA
DEFAULT COLLATION USING_NLS_COMP
AUTHID CURRENT_USER
ACCESSIBLE BY ( A.B )
IS
        p integer := 20;
begin
        raise notice '%', p;
end;
/
CREATE OR REPLACE PROCEDURE ora_procedure
SHARING = METADATA
DEFAULT COLLATION USING_NLS_COMP
AUTHID CURRENT_USER
ACCESSIBLE BY ( FUNCTION A.B )
IS
        p integer := 20;
begin
        raise notice '%', p;
end;
/
CREATE OR REPLACE PROCEDURE ora_procedure
SHARING = METADATA
DEFAULT COLLATION USING_NLS_COMP
AUTHID CURRENT_USER
```

```
ACCESSIBLE BY ( FUNCTION A.B, PROCEDURE C.D )
IS
        p integer := 20;
begin
        raise notice '%', p;
end;
/
CREATE OR REPLACE PROCEDURE ora_procedure
SHARING = METADATA
DEFAULT COLLATION USING_NLS_COMP
AUTHID CURRENT_USER
ACCESSIBLE BY ( FUNCTION A.B, PROCEDURE C.D, PACKAGE E, TRIGGER F, TYPE G )
IS
        p integer := 20;
begin
        raise notice '%', p;
end;
/
```

Compatibility with ALTER PROCEDURE syntax

```
alter procedure test_proc editionable;
alter procedure public.test_proc noneditionable;
alter procedure test_proc compile;
alter procedure test_proc compile debug;
alter procedure test_proc compile debug sd = mv;
alter procedure test_proc compile debug reuse settings;
```

Function and procedure can have no parameter

```
create or replace function f_noparentheses
return int is
begin
return 11;
end;
/
select f_noparentheses from dual;

create or replace procedure protest
as
begin
raise notice 'protest';
```

```
end;
/
CALL protest();
```

Views related with function and stored procedure

```
They can be found in file contrib/ivorysql_ora/src/sysview/sysview--1.0.sql.
Including DBA_PROCEDURES, ALL_PROCEDURES, USER_PROCEDURES, DBA_SOURCE, ALL_SOURCE,
USER_SOURCE, DBA_ARGUMENTS, ALL_ARGUMENTS, USER_ARGUMENTS etc.
```

Support (--) and (/* */)

pg_dump adds one slash (/) at the end of definition of function/procedure when backup SQL file

.5. 5、Built-in data types and built-in functions

Built-in data types

char
varchar
varchar2
number
binary_float
binary_double
date
timestamp
timestamp with time zone
timestamp with local time zone
interval year to month
interval day to second
raw
long

Built-in functions

sysdate
systimestamp
add_months
last_day
next_day
months_between
current_date
current_timestamp
new_time

tz_offset
trunc
instr
instrb
substr
substrb
trim
ltrim
rtrim
length
lengthb
rawtohex
replace
regexp_replace
regexp_substr
regexp_instr
regexp_like
to_number
to_char
to_date
to_timestamp
to_timestamp_tz
to_yminterval
to_dsinterval
numtodsinterval
numtoyminterval
localtimestamp
from_tz
sys_extract_utc
sessiontimezone
hextoraw
uid
USERENV
asciistr
to_multi_byte
to_single_byte
compose
decompose
sys_context

Built-in function descriptions

`sysdate` function

function: view the corresponding date and time, the test cases are as follows: Query the date of the current system:

```
select sysdate() from dual;
  sysdate
------------
 2023-07-06
(1 row)
```

Check the date pushed forward by 1 day:

```
select sysdate()-1 from dual;
  ?column?
------------
 2023-07-05
(1 row)
```

`systimestamp` function

function: return the current system date and time (including microseconds and time zone) on the local database, the test cases are as follows: Date and time to query the current date:

```
select systimestamp() from dual;
  systimestamp
-----------------------------------
 2023-07-06 10:18:31.674322 +08:00
(1 row)
```

`add_months` function

function: the function adds a date to the number of months (n), and returns the same day that is n months apart, supporting parameters: date, number; The test cases are as follows: Check the same day of the following month on the current date (July 6):

```
select add_months(sysdate(),1) from dual;
 add_months
------------
 2023-08-06
(1 row)
```

Query the same day of the previous month for the current date:

```
select add_months(sysdate(),-1) from dual;
 add_months
------------
 2023-06-06
(1 row)
```

last_day function

function: return the last day of the month where the specified date is located, support parameters: date, the test cases are as follows: Check the last day of the month in which the day is located:

```
select last_day(sysdate())from dual;
  last_day
------------
 2023-07-31
(1 row)
```

Query the last day of the month on which a day falls:

```
select last_day(to_date('2019-09-01'))from dual;
  last_day
------------
 2019-09-30
(1 row)
```

next_day function

function: return the next date of the specified date. Supported parameters: date, integer /date, text, Note: When the second parameter in the function passes the number of weeks more hours than the existing week, the date of the next week will be returned; When the date passed by the second parameter in the function is greater than the existing number of weeks, the corresponding day of the week of the week is returned. The test cases are as follows: Query the next day of the current date:

```
select next_day(sysdate(),1) from dual;
 next_day
------------
 2023-07-07
(1 row)
```

Next Friday for the current date:

```
select next_day(sysdate(),'FRIDAY') from dual;
  next_day
------------
 2023-07-07
```

```
(1 row)
```

`months_between` function

function: return the month of difference between date1 and date2 of date type, support parameters: date, date, description: if date1 is later than date2, return a positive number; If date1 is earlier than date2, a negative number is returned; If date1 and date2 are the same day of a month, the return result is an integer; If not the same day, results with decimal parts are returned on a monthly basis of 31 days. The test cases are as follows: To find the month that differs between the same day in different months:

```
select months_between(to_date('2023-07-06'),to_date('2023-08-06')) from dual;
 months_between
----------------
             -1
(1 row)
```

Query the month that differs between different days of different months:

```
select months_between(to_date('2023-07-06'),to_date('2023-08-05')) from dual;
   months_between
--------------------
 -0.967741935483871
(1 row)
```

`current_date` function

function: return the current date of the current time zone, the test cases are as follows: To query the current date in the current time zone:

```
select current_date from dual;
 current_date
--------------
 2023-07-06
(1 row)
```

`current_timestamp` function

function: return the current date and current time of the current time zone, including the current time zone information. Support parameters: integer, Note: The returned time can be adjusted with precision. The test cases are as follows: To query the current date and time in the current time zone:

```
select current_timestamp from dual;
      current_timestamp
-----------------------------------
 2023-07-06 10:27:01.440600 +08:00
(1 row)
```

Query the current date and time in the current time zone (the precision is adjusted to the first three decimal places):

```
select current_timestamp(3) from dual;
      current_timestamp
-----------------------------------
 2023-07-06 10:27:14.182000 +08:00
(1 row)
```

`new_time` function

function: return the date in another time zone corresponding to a certain time zone, support parameters: date, text, text, the test case is as follows: Returns the date for the current date in another time zone:

```
select sysdate() bj_time,new_time(sysdate(),'PDT','GMT')los_angles from dual;
  bj_time   | los_angles
------------+------------
 2023-07-06 | 2023-07-06
(1 row)
```

`tz_offset` function

function: return the offset of the given time zone and the standard time zone, support parameters: text, the test case is as follows: Returns the offset of a given time zone from the standard time zone:

```
select tz_offset('US/Eastern') from dual;
 tz_offset
-----------
 -04:00
(1 row)
```

`trunc` function

function: you can intercept the date to get the desired value, such as year, month, day, hour, minute, support parameters: date/date, text, the test case is as follows: Intercept the current date:

```
select trunc(sysdate()) from dual;
 trunc
------------
 2023-07-06
(1 row)
```

Truncating the year, only the year is correct, and the month and day are not accurate values:

```
select trunc(sysdate(),'yyyy') from dual;
```

```
   trunc
------------
 2023-01-01
(1 row)
```

Intercept the month, the return value only the month is correct, the year and day are not accurate values:

```
select trunc(sysdate(),'mm') from dual;
   trunc
------------
 2023-07-01
(1 row)
```

`instr` function

function: string search that checks whether the source string contains the target string and returns the match position. Supported parameter forms are as follows:

- `instr(string, str)`
- `instr(string, str, start_position, nth_appearance)`

The following are test cases:

Returns the first match by default:

```
SELECT INSTR('database administration', 'data') FROM DUAL;
 instr
-------
     1
(1 row)
```

Specify the starting position and match sequence:

```
SELECT INSTR('database administration', 'i', 1, 2) FROM DUAL;
 instr
-------
    15
(1 row)
```

Supports reverse search from the end of the string:

```
SELECT INSTR('mississippi river', 's', -5, 2) FROM DUAL;
 instr
-------
     6
```

```
(1 row)
```

Returns 0 when no match is found:

```
SELECT INSTR('database administration', 'z') FROM DUAL;
 instr
-------
     0
(1 row)
```

`instr` can also be used for like-style fuzzy matching:

```
select * from tableName where instr(name,'helloworld')>0;
```

Implementation notes:

- Searches for one string inside another with a classic brute-force algorithm.
- Reuses the `text_instring` helper function.
- Uses the `isByte` flag to detect multibyte encodings and branch between single-byte and multibyte handling.
- The sign of `position` decides forward or backward search and sets loop bounds and step.
- Iterates byte by byte through the source string, comparing with the pattern until a match is found.

`instrb` function

function: string lookup function, return the position of the string, support parameters: varchar2, text, number DEFAULT 1, number DEFAULT 1, the following are test cases: RETURNS THE POSITION OF THE STRING IN CORPORATE FLOOR WHEN THE FIRST OR OCCURS BY DEFAULT:

```
SELECT INSTRB('CORPORATE FLOOR','OR') "Instring in bytes" FROM DUAL;
 Instring in bytes
-------------------
                 2
(1 row)
```

Returns the position of the string in the corporate floor where the query starts with the fifth character and the second occurrence of or:

```
SELECT INSTRB('CORPORATE FLOOR','OR',5,2) "Instring in bytes" FROM DUAL;
 Instring in bytes
-------------------
                14
(1 row)
```

`substr` function

function: intercept string function, truncated in characters, support parameters: text, integer, test cases are as follows: Intercept the string from the fifth character in 'It is nice today', followed by:

```
SELECT SUBSTR('It is nice today',5) "Substring with bytes" FROM DUAL;


 Substring with bytes
----------------------
  s nice today
(1 row)
```

`substrb` function

function: intercept string function, intercept in bytes, support parameters: varchar2, number/varchar2, number, number, the test cases are as follows: Intercept the string starting with the fifth byte in 'It’ s nice today' and then onwards:

```
SELECT SUBSTRB('It is nice today',5) "Substring with bytes" FROM DUAL;
 Substring with bytes
----------------------
 s nice today
(1 row)
```

Intercept the string in 'It is nice today' starting with the fifth byte and ending with the eighth byte:

```
SELECT SUBSTRB('It is nice today',5,8) "Substring with bytes" FROM DUAL;
 Substring with bytes
----------------------
 s nice t

(1 row)
```

`trim` function

function: remove the left and right spaces or corresponding data of the specified string, support parameters: varchar2 / varchar2, varchar2, the test cases are as follows: Remove the left and right spaces of ' aaa bbb ccc ':

```
select trim('   aaa bbb ccc   ')trim from dual;
  trim
-------------
 aaa bbb ccc
(1 row)
```

Remove aaa from 'aaa bbb ccc':

```
select trim('aaa bbb ccc','aaa')trim from dual;
  trim
----------
  bbb ccc
(1 row)
```

`ltrim` function

function: remove the left space or corresponding data of the specified string, support parameters: varchar2 / varchar2, varchar2, the test cases are as follows: Remove the space to the left of ' abcdefg ':

```
select ltrim('   abcdefg   ')ltrim from dual;
    ltrim
------------
 abcdefg
(1 row)
```

Traverse from the left side of 'abcdefg', remove it as soon as a character appears in 'fegab', and return the result if it is absent:

```
select ltrim('abcdefg','fegab')ltrim from dual;
 ltrim
-------
 cdefg
(1 row)
```

`rtrim` function

function: remove the space on the right side of the specified string, the test case is as follows: Remove the space to the right of ' abcdefg ':

```
select rtrim('   abcdefg   ')rtrim from dual;
     rtrim
----------------
    abcdefg
(1 row)
```

Traverse from the right side of 'abcdefg', remove it as soon as a character appears in 'fegab', and return the result if it is absent:

```
select rtrim('abcdefg','fegab')rtrim from dual;
 rtrim
-------
 abcd
```

```
(1 row)
```

`length` function

function: find the length of the specified string character, support parameters: char/integer/varchar2 The test cases are as follows: Query the character length of 223:

```
select length(223) from dual;
 length
--------
      3
(1 row)
```

Query the character length of '223':

```
select length('223') from dual;
 length
--------
      3
(1 row)
```

To query the character length of 'ivorysql database' :

```
select length('ivorysql database') from dual;
 length
--------
     17
(1 row)
```

`lengthb` function

function: find the length of the specified string byte, support parameters: char/bytea/varchar2 test cases are as follows: Query the byte lengthb of 'ivorysql':

```
select lengthb('ivorysq'::char) from dual;
 lengthb
---------
       1
(1 row)
```

Query the byte lengthb of '0x2C':

```
select lengthb('0x2C'::bytea) from dual;
 lengthb
```

```
---------
       4
(1 row)
```

Query the byte lengthb of the 'ivorysql database':

```
select lengthb('ivorysql database') from dual;
 lengthb
---------
      17
(1 row)
```

`replace` function

function: replace the character in the specified string or delete the character, support parameters: text, text, text/varchar2, varchar2, varchar2 DEFAULT NULL::varchar2, test for example: Replace 'j' in 'jack and jue' with 'bl' :

```
select replace('jack and jue','j','bl') from dual;
    replace
----------------
 black and blue
(1 row)
```

Remove the 'j' in 'jack and jue' :

```
select replace('jack and jue','j') from dual;
  replace
------------
 ack and ue
(1 row)
```

`regexp_replace` function

which is an extension of the replace function. Function: Used to perform matching and replacement through regular expressions. Supported parameters: text, text, text /text, text, text, integer/varchar2, varchar2/varchar2, varchar2 varchar2, varchar2 varchar2, for example: Replace the matched number with *#:

```
select regexp_replace('01234abcd56789','[0-9]','*#')from  dual;
      regexp_replace
--------------------------
 *#*#*#*#*#abcd*#*#*#*#*#
(1 row)
```

Start with the second number by replacing the matched number with *#:

```
select regexp_replace('01234abcd56789','[0-9]','*#',2)from  dual;
    regexp_replace
-------------------------
 0*#*#*#*#abcd*#*#*#*#*#
```

Delete '01' from '01234abcd56789':

```
select regexp_replace('01234abcd56789','01')from dual;
 regexp_replace
----------------
 234abcd56789
(1 row)
```

Replace 01234abcd56789' with 'xxx':

```
select regexp_replace('01234abcd56789','012','xxx')from dual;
 regexp_replace
----------------
 xxx34abcd56789
(1 row)
```

`regexp_substr` function

function: pick up the character substring described by the regular expression, support parameters: text, text, integer /text, text, integer, integer / text, text, integer, integer, text /varchar2, varchar2, the test cases are as follows: Query the 012 string starting with the first number in '012ab34':

```
select regexp_substr('012ab34', '012',1) from dual;
 regexp_substr
---------------
 012
(1 row)
```

Query the 012 string in '012ab34' starting from the first number of the first group:

```
select regexp_substr('012ab34', '012',1,1) from  dual;
 regexp_substr
---------------
 012
(1 row)
```

Query '012a012Ab34' for case-insensitive 012 strings starting from the first number of the first group:

```
select regexp_substr('012a012Ab34', '012A',1,1,'i') from dual;
 regexp_substr
---------------
 012a
(1 row)
```

Query '012a012Ab34' for case-sensitive 012 strings starting from the first group of numbers:

```
select regexp_substr('012a012Ab34', '012A',1,1,'c') from dual;
 regexp_substr
---------------
 012A
(1 row)
```

Query the 'Database' substring in 'Data':

```
select regexp_substr('Database' , 'Data') from dual;
 regexp_substr
---------------
 Data
(1 row)s
```

`regexp_instr` function

function: used to calibrate the start position of the character substring that conforms to the regular expression, support parameters: text, text, integer /text, text, integer, integer / text, text, integer, integer, text, integer / varchar2, varchar2, the test case is as follows: Query 'abcaBcabc' for the position of the abc substring starting from the first character:

```
SELECT regexp_instr('abcaBcabc', 'abc', 1);
 regexp_instr
--------------
 1
(1 row)
```

Query 'abcaBcabc' starting from the first character, where the abc substring appears for the third time:

```
SELECT regexp_instr('abcaBcabc', 'abc', 1, 3);
 regexp_instr
--------------
 7
(1 row)
```

Query 'abcabcabc' starting from the first character and occurring after the second occurrence of the abc

substring:

```
SELECT regexp_instr('abcaBcabc', 'abc', 1, 2,1);
 regexp_instr
--------------
 7
(1 row)
```

Query 'abcaBcabc' from the first character, where it occurs after the first occurrence of the abc substring (case sensitive):

```
SELECT regexp_instr('abcaBcabc', 'abc',1,2,1,'c');
 regexp_instr
--------------
 7
(1 row)
```

Query the 'Database' substring in 'Data':

```
SELECT regexp_instr('Database', 'Data');
 regexp_instr
--------------
            1
(1 row)
```

`regexp_like` function

function: similar to like, used for fuzzy queries. Supported parameters: varchar2, varchar2 /varchar2, varchar2 varchar2, First create a regexp_like table for the test case query:

```
create table t_regexp_like
(
  id varchar(4),
  value varchar(10)

);
insert into t_regexp_like values ('1','1234560');
insert into t_regexp_like values ('2','1234560');
insert into t_regexp_like values ('3','1b3b560');
insert into t_regexp_like values ('4','abc');
insert into t_regexp_like values ('5','abcde');
insert into t_regexp_like values ('6','ADREasx');
insert into t_regexp_like values ('7','123  45');
insert into t_regexp_like values ('8','adc  de');
```

```
insert into t_regexp_like values ('9','adc,.de');
insert into t_regexp_like values ('10','abcbvbnb');
insert into t_regexp_like values ('11','11114560');
```

The test cases are as follows: Query t_regexp_like columns with abc in the table:

```
select * from t_regexp_like where regexp_like(value,'abc');
 id |  value
----+----------
 4  | abc
 5  | abcde
 10 | abcbvbnb
(3 rows)
```

Query t_regexp_like columns with ABC in the table (not case sensitive):

```
select * from t_regexp_like where regexp_like(value,'ABC','i');
 id |  value
----+----------
 4  | abc
 5  | abcde
 10 | abcbvbnb
(3 rows)
```

`to_number` function

function: is to change some processed strings arranged in a certain format back to a numeric format, support parameters: text/text, text test cases are as follows: Convert the string '-34,338,492' to numeric format:

```
SELECT to_number('-34,338,492', '99,999,999') from dual;
 to_number
-----------
 -34338492
(1 row)
```

Convert the string '5.01-' to numeric format:

```
SELECT to_number('5.01-', '9.99S');

 to_number
-----------
 -5.01
```

```
(1 row)
```

`to_char` function

function: convert numbers or dates to character types, support parameters: date/date, text/timestamp/timestamp, text test cases are as follows: To convert the current system date to character format:

```
select to_char(sysdate()) from dual;
  to_char
------------
 2023-07-10
(1 row)
```

Convert current system date to month/day/year character format:

```
select to_char(sysdate(),'mm/dd/yyyy') from dual;
  to_char
------------
 07/10/2023
(1 row)
```

Converts the timestamp format of the current date to character format

```
SELECT to_char(sysdate()::timestamp);
         to_char
----------------------------
 2023-07-10 09:46:44.000000
```

Convert timestamp format of current date to month/date/year character format:

```
SELECT to_char(sysdate()::timestamp,'MM-YYYY-DD');
  to_char
------------
 07-2023-10
(1 row)
```

`to_date` function

function: convert character type to date type, support parameters: text/text, text test cases are as follows: Convert '2023/07/06' to date type:

```
select to_date('20230706') from dual;
  to_date
------------
```

```
 2023-07-06
(1 row)
```

Convert '-44-02-01' to date type:

```
SELECT to_date('-44,0201','YYYY-MM-DD');
 to_date
------------
 0044-02-01
(1 row)
```

`to_timestamp` function

function: can store year, month, day, hour, minute, second, and can also store fractional parts of seconds. Supported parameters: text/text, text test cases are as follows: Query '2018-11-02 12:34:56.025' output as a date:

```
SELECT to_timestamp('20181102.12.34.56.025');
        to_timestamp
----------------------------
 2018-11-02 12:34:56.025000
(1 row)
```

Query '2011,12,18 11:38' output as a date:

```
SELECT to_timestamp('2011,12,18 11:38 ', 'YYYY-MM-DD HH24:MI:SS');
       to_timestamp
----------------------------
 2011-12-18 11:38:00.000000
(1 row)
```

`to_timestamp_tz` function

function: according to the time query, the time string has T, Z and milliseconds, time zone. The test cases are as follows: Query '2016-10-9 14:10:10.123000' output as a date:

```
 SELECT to_timestamp_tz('2016-10-9 14:10:10.123000') FROM DUAL;
         to_timestamp_tz
---------------------------------
 2016-10-09 14:10:10.123000 +08:00
(1 row)
```

Query '10-9-2016 14:10:10.123000 +8:30' output as a date:

```
 SELECT to_timestamp_tz('10-9-2016 14:10:10.123000 +8:30', 'DD-MM-YYYY HH24:MI:SS.FF
```

```
TZH:TZM') FROM DUAL;
          to_timestamp_tz
-----------------------------------
 2016-09-10 13:40:10.123000 +08:00
(1 row)
```

`to_yminterval` function

function: convert a string type to a year and month time difference type, support parameters: text, The test cases are as follows: Query the date after two years and eight months after '20110101':

```
select to_date('20110101','yyyymmdd')+to_yminterval('02-08') from dual;
  ?column?
------------
 2013-09-01
(1 row)
```

`to_dsinterval` function

function: add a date plus a certain hour or number of days into another date, support parameters: text, test cases are as follows: Query the current system time plus the date in 9 and a half hours (currently 2023-07-06, 18:00):

```
select sysdate()+to_dsinterval('0 09:30:00')as newdate from dual;
 newdate
------------
 2023-07-07
(1 row)
```

`numtodsinterval` function

function: convert numbers into time interval type data. The supporting parameters: double precision, text test cases are as follows: Convert 100.00 hours to interval type data:

```
SELECT NUMTODSINTERVAL(100.00, 'hour');
        numtodsinterval
------------------------------
 +000000004 04:00:00.000000000
(1 row)
```

Convert 100 minutes to interval type data:

```
SELECT NUMTODSINTERVAL(100, 'minute');
      numtodsinterval
------------------------------
```

```
 +000000000 01:40:00.000000000
(1 row)
```

`numtoyminterval` function

function: convert numbers into date interval type data. Convert 1, year to date interval: double precision, text, the test case is as follows:

```
SELECT NUMTOYMINTERVAL(1.00,'year');
 numtoyminterval
-----------------
 +000000001-00
(1 row)
```

Convert 1, mouth to date interval:

```
SELECT NUMTOYMINTERVAL(1,'month');
 numtoyminterval
-----------------
 +000000000-01
(1 row)
```

`localtimestamp` function

function: return the date and time in the session, support parameters: integer, add parameters to the function as precision, the test cases are as follows: To return the date and time in the current session:

```
select localtimestamp from dual;
       localtimestamp
----------------------------
 2023-07-07 09:18:15.896472
(1 row)
```

Returns the date and time in the current session with a precision of 1:

```
select localtimestamp(1) from dual;
     localtimestamp
----------------------------
 2023-07-07 09:18:16.100000
(1 row)
```

`from_tz` function

function: convert time from one time zone to another, support parameters; timestamp, text, the test case is as follows: Convert '2000-03-28 08:00:00', '3:00' to the current time zone:

```
SELECT FROM_TZ(TIMESTAMP '2000-03-28 08:00:00', '3:00') FROM DUAL;
              from_tz
-----------------------------------
 2000-03-28 13:00:00.000000 +08:00
(1 row)
```

`sys_extract_utc` function

function: convert a timestamptz to UTC time zone time. Supported parameters: timestamp with time zone The test cases are as follows: Query conversion timestamp '2000-03-28 11:30:00.00 -8:00' to the time after UTC time zone:

```
select sys_extract_utc(timestamp '2000-03-28 11:30:00.00 -8:00') from dual;
      sys_extract_utc
----------------------------
 2000-03-28 19:30:00.000000
(1 row)
```

`sessiontimezone` function

function: view time zone details, test cases are as follows: To view the details of the current time zone:

```
select sessiontimezone() from dual;
 sessiontimezone
-----------------
 Asia/Shanghai
(1 row)
```

After modifying the timezone, check the time zone belief information:

```
set timezone = 'Asia/Hong_Kong';
SET
select sessiontimezone() from dual;
 sessiontimezone
-----------------
 Asia/Hong_Kong
(1 row)
```

`hextoraw` function

function: convert the binary value represented by the string into a RAW value. Support parameters: text, the test cases are as follows: Convert the string 'abcdef' to a raw value:

```
select hextoraw('abcdef')from dual;
 hextoraw
```

```
----------
 \xabcdef
(1 row)
```

`uid` function

function: get the instance name of the database. The test cases are as follows: Get the instance name of the current database:

```
select uid() from dual;
 uid
-----
  10
(1 row)
```

`USERENV` function

function: return the information of the current user environment, the test cases are as follows: Check whether the current user is DBA, and if so, return true:

```
select userenv('isdba')from dual;
 get_isdba
-----------
 TRUE
(1 row)
```

To view the session flag:

```
select userenv('sessionid')from dual;
 get_sessionid
---------------
 1
(1 row)
```

`ASCIISTR` function

function: input string, return ASCII characters, the test cases are as follows: string with only ascii chars:

```
 select asciistr('Hello, World!') from dual;
   asciistr
---------------
 Hello, World!
(1 row)
```

string with non-ascii chars:

```
 select asciistr('你好') from dual;
  asciistr
------------
 \4F60\597D
```

string with mixed ascii and non-ascii:

```
 select asciistr('ABÄCDE') from dual;
  asciistr
------------
 AB\00C4CDE
(1 row)
```

`TO_MULTI_BYTE` function

function: Convert half-width characters in a string to full-width characters: input half-width characters, Convert to full-width characters:

```
select to_multi_byte('1.2'::text) ;
 to_multi_byte
---------------
 1. 2
```

`TO_SINGLE_BYTE` function

function: Convert full-width characters in a string to half-width characters: input full-width characters, Convert to half-width characters:

```
select to_single_byte('1. 2');
 to_single_byte
----------------
 1.2
```

`COMPOSE` function

function: Combine base characters and combining marks into a composite Unicode character: input base character 'a' with a combining mark '768', return à:

```
select compose('a'||chr(768)) from dual;
 compose
---------
 à
(1 row)
```

`DECOMPOSE` function

function: Decompose composite Unicode characters (like those with accents or special symbols) into their base characters and combining marks. input é, return a base character 'e' with a combining mark '301':

```
select asciistr(decompose('é')) from dual;
 asciistr
----------
 e\0301
```

`SYS_CONTEXT` function

which returns the value of the parameter associated with the given context at the current moment. It can be used in both SQL and PL/SQL languages.

```
ivorysql=# select sys_context('USERENV','DB_NAME');
 sys_context
-------------
 ivorysql
(1 row)
```

.6. 6、ports and IP

Objective

- In order to distinguish the Oracle port, IP and PG port IP. THERE IS NOW A NEED TO INCREASE THE PROCESSING OF ORAPORT AND ORAHOST;

Function

- Add ivoryhost: You need to add the parameter ivoryhost when connecting, and its function is similar to host;
- New ivoryport: Compared to host, the function of port is relatively complex. It involves specifying ports in the configure phase and connection phase;

Test method:

```
./configure --with-oraport=5555
./initdb ....
./pg_ctl -D ../data start

./pg_ctl -o "-p 5433 -o 1522" -D ../data
```

.7. 7、XML Function

Objective

In Oracle, SQL code often contains XML functions. To ensure consistency in data format and structure when

migrating from Oracle to IvorySQL, IvorySQL achieves high compatibility with Oracle XML functions, building upon the foundation of PostgreSQL.

This compatibility means that users do not need to make extensive modifications to their existing XML processing logic, thereby ensuring the integrity and accuracy of the data. Furthermore, IvorySQL's cross-platform compatibility reduces the additional user maintenance and upgrade costs caused by format differences, making data processing and management more efficient, reliable, and flexible.



XML (eXtended Markup Language) is a text-based format language used to structure any document that can be tagged. It is a lightweight, extensible, standard, and easy-to-understand language for storing data.

Implementation Principle

IvorySQL achieves compatibility with commonly used XML SQL functions. It maintains consistency with PostgreSQL by utilizing the same underlying processing functions, which are provided by the libxml2 library interface. These XML functions are provided as a sub-plugin of the ivorysql_ora plugin, ensuring compatibility and consistency with PostgreSQL databases in terms of XML processing.

Due to Oracle's XML functions requiring certain parameter types to be XMLType, such as the existsnode() function below:

Prototype: EXISTSNODE(XMLType_instance, XPath_string [, namespace_string])

Demo: SELECT existsnode(XMLType('<a>d'), '/a') from dual;

Therefore, for compatibility purposes, an XMLType data type has been added. Its function is to convert the string provided by the user into an internal XMLType type, allowing SQL statements to be migrated from Oracle to IvorySQL without modification.

Additionally, to avoid confusion with the existing keyword "extract" in PostgreSQL, IvorySQL has renamed the original keyword to "PGEXTRACT" to ensure clarity and accuracy in function calls.

When implementing these Oracle-compatible XML functions, IvorySQL adopted two different approaches. Among them, besides the UPDATEXML function, the other functions are implemented using SQL functions. Since the number of parameters for the UPDATEXML function is uncertain, an expression-based approach was used for its implementation. This required writing specialized syntax parsing and executor code to ensure the correctness and flexibility of its functionality.

Compatible Function

Num	Function Name	Function Introduction
1	extract(XML)	This function is used to return the corresponding content under the XML node path. The parameter XMLType_instance is used to specify the XMLType instance, while Xpath_string is used to specify the XML node path.
2	extractvalue	This function provides retrieval functionality for XML content. The extractvalue function can only return one value from one node.
3	existsnode	This function is used to check whether the XML content matches the specified path expression.
4	deletexml	This function is used to delete XML nodes at specified paths.

5	appendchildxml	This function is used to insert child nodes into an XML object. It takes an XML object and an XML fragment as parameters, and inserts the XML fragment as a child node into the XML object.
6	updatexml	This function is used to update the content of a specific XML node path.
7	insertxmlbefore	This function is used to insert child nodes before a specified path in XML.
8	insertxmlafter	This function is used to insert child nodes after a specified path in XML.
9	insertchildxml	This function is used to insert child nodes into a specified XML path.
10	insertchildxmlbefore	This function is used to insert child nodes before a specified XML path.
11	insertchildxmlafter	This function is used to insert child nodes after a specified XML path.
12	xmlisvalid	This functions is used to check the xml content is valid or not

XML Function Demo

Prepare table and data

```
ivorysql=# set ivorysql.compatible_mode to oracle;
SET
ivorysql=# create table inaf(a int, b xmltype);
CREATE TABLE
ivorysql=# insert into inaf values(1,xmltype('<a><b>100</b></a>'));
INSERT 0 1
ivorysql=# insert into inaf values(2, '');
INSERT 0 1
ivorysql=# select * from inaf;
 a |         b
---+------------------
 1 | <a><b>100</b></a>
 2 |
(2 rows)
ivorysql=# create table xmltest(id int, data XMLType);
CREATE TABLE
ivorysql=# insert into xmltest values(1, '<value>one</value>');
INSERT 0 1
ivorysql=# insert into xmltest values(2, '<value>two</value>');
INSERT 0 1
```

```
ivorysql=# select * from xmltest;
 id |        data
----+--------------------
  1 | <value>one</value>
  2 | <value>two</value>
(2 rows)
ivorysql=# create table xmlnstest(id int, data xmltype);
CREATE TABLE
ivorysql=# INSERT INTO xmlnstest VALUES(1, xmltype('<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:typ="http://www.def.com"
xmlns:web="http://www.abc.com"><soapenv:Body><web:BBB><typ:EEE>41</typ:EEE><typ:FFF>42
</typ:FFF></web:BBB></soapenv:Body></soapenv:Envelope>'));
INSERT 0 1
```

extract(XML)

```
ivorysql# SELECT extract(XMLType('<AA><ID>1</ID></AA>'), '/AA/ID') from dual;
  extract
------------
 <ID>1</ID>
(1 row)
```

extractvalue

```
ivorysql# SELECT extractvalue(XMLType('<a><b>100</b></a>'),'/a/b') from dual;
 extractvalue
--------------
 100
(1 row)
```

existsnode

```
ivorysql=# SELECT existsnode(XMLType('<a><b>d</b></a>'), '/a/b') from dual;
 existsnode
------------
          1
(1 row)
```

deletexml

```
ivorysql=# SELECT
deletexml(XMLType('<test><value>oldnode</value><value>oldnode</value></test>'),
```

```
'/test/value') from dual;
 deletexml
-----------
 <test/>
(1 row)
```

appendchildxml

```
ivorysql=# SELECT
appendchildxml(XMLType('<test><value></value><value></value></test>'),  '/test/value',
XMLTYPE('<name>newnode</name>')) from dual;
      appendchildxml
--------------------------
 <test>                  +
   <value>               +
     <name>newnode</name>+
   </value>              +
   <value>               +
     <name>newnode</name>+
   </value>              +
 </test>
(1 row)
```

updatexml

```
ivorysql=# SELECT updatexml(xmltype('<value>one</value>'), '/value',
xmltype('<newvalue>1111</newvalue>')) FROM dual;
         updatexml
---------------------------
 <newvalue>1111</newvalue>
(1 row)
```

insertxmlbefore

```
ivorysql=# SELECT insertxmlbefore(XMLType('<a>222<b>100</b><b>200</b></a>'), '/a/b',
XMLTYPE('<c>88</c>')) from dual;
                 insertxmlbefore
-------------------------------------------------
 <a>222<c>88</c><b>100</b><c>88</c><b>200</b></a>
(1 row)
```

insertxmlafter

```
ivorysql=# SELECT
insertxmlafter(XMLType('<a><b>100</b></a>'),'/a/b',XMLType('<c>88</c>')) from dual;
 insertxmlafter
----------------
 <a>           +
   <b>100</b>  +
   <c>88</c>   +
 </a>
(1 row)
```

insertchildxml

```
ivorysql=# SELECT insertchildxml(XMLType('<a>one<b></b>three<b></b></a>'), '//b',
'name', XMLTYPE('<name>newnode</name>')) from dual;
                           insertchildxml
-----------------------------------------------------------------------
 <a>one<b><name>newnode</name></b>three<b><name>newnode</name></b></a>
(1 row)
```

insertchildxmlbefore

```
ivorysql=# SELECT insertchildxmlbefore(XMLType('<a><b>100</b></a>'), '/a', 'b',
XMLType('<c>88</c>')) from dual;
 insertchildxmlbefore
----------------------
 <a>                 +
   <c>88</c>         +
   <b>100</b>        +
 </a>
(1 row)
```

insertchildxmlafter

```
ivorysql=# SELECT insertchildxmlafter(XMLType('<a><b>100</b></a>'), '/a', 'b',
XMLType('<c>88</c>')) from dual;
 insertchildxmlafter
---------------------
 <a>                +
   <b>100</b>       +
   <c>88</c>        +
 </a>
```

```
(1 row)
```

xmlisvalid

```
ivorysql=# SELECT xmlisvalid(XMLTYPE('<a>'));
xmlisvalid
------------
f
(1 row)

ivorysql=# SELECT xmlisvalid(XMLTYPE('<a/>'));
xmlisvalid
------------
t
(1 row)
```

.8. 8、sequence

Objective

- This document is designed for compatibility with Oracle sequence functions, in order to use Oracle’s sequences in IvorySQL.

Function description

- Sequence is a database object, similar to tables and views, that represents an integer sequence that can be used by any table and view in the global database namespace. Sequence values can be accessed using NEXTVAL and CURRVAL. Sequences can be in ascending or descending order.

Test cases

```
ivorysql=# CREATE sequence seq;
CREATE SEQUENCE
ivorysql=# SELECT seq.NEXTVAL FROM DUAL;
 nextval
---------
       1
(1 row)
ivorysql=# ALTER sequence seq restart start with 10;
ALTER SEQUENCE
ivorysql=# SELECT seq.NEXTVAL FROM DUAL;
 nextval
---------
      10
(1 row)
```

```
ivorysql=# DROP SEQUENCE seq;
DROP SEQUENCE
```

.9. 9、Package

Objective

IvorySQL provides compatibility for Oracle packages. A package is an encapsulated collection of related program objects stored together in the database. Program objects are procedures, functions, variables, constants, cursors, and exceptions.

This document aims to provide a comprehensive understanding of the process of implementing custom packages.

Function descriptions

IvorySQL provides compatibility for Oracle custom packages, including creation, alteration, and deletion of packages and package bodies. We also add support for package-related commands in the PostgreSQL interactive terminal (psql) with the new \dk command.

Create the specification of a package

The CREATE OR REPLACE PACKAGE statement is used to create or replace the specification of a package. A package is a collection of related stored procedures, functions, and other program objects, stored as a single unit in the database. The package specification declares these objects, and the package body defines them.

To create or replace the specification of a package in your own schema, you must have the CREATE PROCEDURE system privilege. If you are creating or replacing a package in another user’ s schema, you need the CREATE ANY PROCEDURE system privilege.

Create package body

The CREATE OR REPLACE PACKAGE BODY statement is used to create or replace a package body. To create a package body requires the same privileges as creating the specification of a package, and it is required that the package body and package specification be in the same schema, with the package specification already existing. This statement defines the objects declared in the package specification.

When the package specification contains cursors or subprograms, a package body must be present to define them. Otherwise, the package body is optional.

Alter package

The ALTER PACKAGE statement is used to modify the properties of a package.The privileges required to execute the ALTER PACKAGE statement are: you must be the owner of the package or have the ALTER ANY PROCEDURE privilege to modify packages owned by other users.

Drop package and package body

The DROP PACKAGE statement deletes a package in the database. This statement will remove both the package body and the package specification.The DROP PACKAGE BODY statement only deletes the package body.

It is not possible to delete a single object within the package using this statement. The privileges required:the package must be in the user’ s schema, or the user must have the DROP ANY PROCEDURE system privilege.

DISCARD PACKAGE

The DISCARD PACKAGE functionality is implemented for compatibility with PostgreSQL’s DISCARD feature.

Use \dk[+] in psql for package and package body information

In psql, \dk[+] is used to view the definition information of packages and package bodies.

Command	Descriptions
\dk[+]	List the package information currently visible.
\dk[+] xxx	List the package specification and package body content of the xxx package.

Test cases

Create the specification of a package

```
ivorysql=# create or replace package pkg is
ivorysql-#   var1 integer;
ivorysql-#   var2 integer;
ivorysql-#   function test_f(id integer) return integer;
ivorysql-#   procedure test_p(id integer);
ivorysql-# end;
ivorysql-# /
CREATE PACKAGE
```

Create package body

```
ivorysql=# create or replace package body pkg is
ivorysql-#   var3 integer;
ivorysql-#   function test_f(id integer) return integer is
ivorysql-#   begin
ivorysql-#     dbms_output.put_line('pkg test_f');
ivorysql-#     return id;
ivorysql-#   end;
ivorysql-#   procedure test_p(id integer) is
ivorysql-#   begin
ivorysql-#     dbms_output.put_line('pkg proc');
ivorysql-#   end;
ivorysql-#   --privite function
ivorysql-#   function test_piv1(id integer) return integer is
ivorysql-#   begin
ivorysql-#      return id;
ivorysql-#   end;
ivorysql-#   --privite procedure
ivorysql-#   procedure test_piv2(id integer) is
```

```
ivorysql-#   begin
ivorysql-#     dbms_output.put_line('privite proc');
ivorysql-#   end;
ivorysql-# begin
ivorysql-#    var1 := 1;
ivorysql-#    var2 := 2;
ivorysql-#    var3 := 4;
ivorysql-# end;
ivorysql-# /
CREATE PACKAGE BODY
```

Alter package

```
ivorysql=# alter package pkg noneditionable;
ALTER PACKAGE
```

Drop package and package body

```
ivorysql=# Drop package pkg;
DROP PACKAGE

ivorysql=# Drop package body pkg;
DROP PACKAGE BODY
```

DISCARD PACKAGE

```
ivorysql=# discard package;
DISCARD PACKAGES
```

Use \dk[+] in psql for package and package body information

```
ivorysql=# \dk
       List of packages
 Schema |   Name   |  Owner
--------+----------+----------
 public | pkg      | ivorysql
 public | test_pkg | ivorysql
(2 rows)

ivorysql=# \dk pkg
     List of packages
 Schema | Name |  Owner
```

```
--------+------+----------
 public | pkg  | ivorysql
(1 row)

ivorysql=# \dk pkg1
Did not find any package named "pkg1".

ivorysql=# \dk+

List of packages
 Schema |   Name   |  Owner   | Security | Editionable | Use Collation |
Specification                |                          Package Body

--------+----------+----------+----------+-------------+---------------+
-----------------------------------------------+
---------------------------------------------------
---------
 public | pkg      | ivorysql | definer  | Editionable | default       | var1 integer;
+|
        |          |          |          |             |               | var2 integer;
+|
        |          |          |          |             |               | function
test_f(id integer) return integer;  +|
        |          |          |          |             |               | procedure
test_p(id integer);                +|
        |          |          |          |             |               | end
|
 public | test_pkg | ivorysql | definer  | Editionable | default       | var1 integer;
+| FUNCTION test_f(id integer) RETURN integer IS
        +
        |          |          |          |             |               |   FUNCTION
test_f(id integer) RETURN integer;+|   BEGIN
        +
        |          |          |          |             |               | end
|     dbms_output.put_line('invoke function test_pkg.t
est_f');+
        |          |          |          |             |               |
| RETURN 23;
        +
        |          |          |          |             |               |
|   end;
        +
        |          |          |          |             |               |
| BEGIN
        +
        |          |          |          |             |               |
```

```
|   var1 := 23;
        +
        |         |         |         |           |              |
| end
(2 rows)
```

.10. 10、Invisible Columns

Objective

The introduction of this feature is to accommodate Oracle’s invisible column functionality, making database design simpler and enhancing the flexibility of data management. Users will have better control over the visibility of columns.

Invisible columns are also useful during application migration. Making new columns invisible means that they will not be visible to any existing application, but they can still be referenced by any new applications, thus simplifying the online migration of applications.

Function descriptions

Invisible columns allow users to conceal specific columns during operations like SELECT *. You can use invisible column to make changes to a table without disrupting applications that use the table. Any generic access of a table does not show the invisible columns in the table. Invisible columns must be explicitly referenced by column names in order to be accessed in queries and other operations.

Support for invisible columns in psql extended describe (\d+) is added, in oracle mode only.

Test cases

Create a table with invisible columns

```
ivorysql=# CREATE TABLE mytable (a INT, b INT INVISIBLE, c INT);
CREATE TABLE
```

Insert values into invisible columns

You can insert a value into an invisible column only if you explicitly specify the invisible column in the column list for the INSERT statement. Omitting the column list in the INSERT statement is not allowed. Errors will be reported.

```
ivorysql=# INSERT INTO mytable (a, b, c) VALUES (1,2,3);
INSERT 0 1

ivorysql=# INSERT INTO mytable VALUES (4,5,6);
ERROR:  INSERT has more expressions than target columns
LINE 1: INSERT INTO mytable  VALUES (4,5,6);
```

Display output for invisible columns

You can use a SELECT statement to display output for an invisible column only if you explicitly specify the invisible column in the column list.

```
ivorysql=# select * from mytable;
 a | c
---+---
 1 | 3
(1 row)

ivorysql=# select a,b,c from mytable;
 a | b | c
---+---+---
 1 | 2 | 3
(1 row)
```

Support for invisible columns in psql extended describe (\d+) in Oracle mode

```
ivorysql=# \d+ mytable
                                                  Table "public.mytable"
 Column |      Type       | Collation | Nullable | Default | Invisible | Storage |
Compression | Stats target | Description
--------+-----------------+-----------+----------+---------+-----------+---------+
-------------+--------------+-------------
 a      | pg_catalog.int4 |           |          |         |           | plain   |
|              |
 b      | pg_catalog.int4 |           |          |         | invisible | plain   |
|              |
 c      | pg_catalog.int4 |           |          |         |           | plain   |
|              |
Access method: heap
```

Limitations

- Modify column is not supported currently. Will be enhanced in later version;
- In Oracle, there are special considerations for invisible columns and column ordering. When a table contains one or more invisible columns, the invisible columns are not included in the column order for the table. We have not yet addressed this part.

.11. 11、RowID Column

Objective

IvorySQL provides Oracle-compatible RowID functionality. RowID is a pseudo-column automatically generated by the database when a table is created, returning the address of each row in the database.

RowID should have the following characteristics:

1. Logically identifies each row with a unique value
2. Allows quick querying and modification of other columns in the table via ROWID, but cannot be inserted or modified itself

3. Users can control whether this feature is enabled

Enable the functionality

IvorySQL provides multiple ways to enable the RowID functionality.

Enable the functionality through GUC parameters

In IvorySQL’s Oracle-compatible mode, the RowID functionality can be enabled by setting ivorysql.default_with_rowids to on. The default value for this parameter is off. Once enabled, tables created will automatically include a RowID column, which can be viewed using `\d+ table_name`.

```
ivorysql=# show ivorysql.default_with_rowids;
 ivorysql.default_with_rowids
------------------------------
 off
(1 row)
```

```
ivorysql=# create table t(a int);
CREATE TABLE
ivorysql=# \d+ t
                                                  Table "public.t"
 Column |      Type       | Collation | Nullable | Default | Invisible | Storage |
Compression | Stats target | Description
--------+-----------------+-----------+----------+---------+-----------+---------+
-------------+--------------+-------------
 a      | pg_catalog.int4 |           |          |         |           | plain   |
|              |
Access method: heap
```

Enable the functionality by adding the `WITH ROWID` option in the table creation statement.

Users can choose to include this option for tables that require it; without the `WITH ROWID` option, a regular table will be created.

```
ivorysql=# create table t2(a int) with rowid;
CREATE TABLE
ivorysql=# \d+ t2
                                                 Table "public.t2"
 Column |      Type       | Collation | Nullable | Default | Invisible | Storage |
Compression | Stats target | Description
--------+-----------------+-----------+----------+---------+-----------+---------+
-------------+--------------+-------------
 a      | pg_catalog.int4 |           |          |         |           | plain   |
|              |
Indexes:
    "t2_16432_rowid_idx" btree (rowid)
```

```
Access method: heap
Has ROWID: yes
```

Enable the functionality by executing the command `ALTER TABLE ⋯ SET WITH ROWID` on an existing table.

This approach allows a regular table to add a ROWID column using the `ALTER` command when ROWID functionality is needed. The ROWID can also be removed using the `ALTER TABLE ⋯ SET WITHOUT ROWID` command.

```
ivorysql=# create table t3(a int);
CREATE TABLE
ivorysql=# alter table t3 set with rowid;
ALTER TABLE
ivorysql=# \d+ t3;
                                                  Table "public.t3"
 Column |      Type       | Collation | Nullable | Default | Invisible | Storage |
Compression | Stats target | Description
--------+-----------------+-----------+----------+---------+-----------+---------+
-------------+--------------+-------------
 a      | pg_catalog.int4 |           |          |         |           | plain   |
|              |
Access method: heap
Has ROWID: yes
```

.12. 12、OUT Parameter

Objective

IvorySQL provides Oracle-compatible OUT parameter functionality, including functions and procedures with OUT parameters, support for OUT parameters in anonymous blocks, and libpq support for OUT parameters.

This document aims to introduce the functionality of OUT parameters to users.

Function descriptions

IvorySQL provides Oracle-compatible OUT parameter functionality, including the following features.

functions with OUT parameters

Syntax:

```
CREATE [ OR REPLACE ] FUNCTION
    name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, ...]
] )
    [ RETURNS rettype
      | RETURNS TABLE ( column_name column_type [, ...] ) ]
  { LANGUAGE lang_name
    | TRANSFORM { FOR TYPE type_name } [, ... ]
```

```
    | WINDOW
    | { IMMUTABLE | STABLE | VOLATILE }
    | [ NOT ] LEAKPROOF
    | { CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT }
    | { [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER }
    | PARALLEL { UNSAFE | RESTRICTED | SAFE }
    | COST execution_cost
    | ROWS result_rows
    | SUPPORT support_function
    | SET configuration_parameter { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
    | sql_body
  } ...
```

The argmode can be IN, OUT, INOUT (IN OUT), or VARIADIC. If not specified, the default is IN. The argmode can also be written after the argname.

Unlike native PostgreSQL, Oracle-compatible OUT parameters are not associated with the return value data type. Neither IN OUT nor OUT modes can have default values. If there are OUT parameters and the function’s return type is not void, the function body must include a RETURN statement.

support for OUT parameters in anonymous blocks

Supports binding variables in the form of colon placeholders, such as :1, :name.

Added new DO+USING syntax： DO [LANGUAGE lang_name] code [USING IN | OUT | IN OUT, …]

Supports binding variables in libpq by position and by parameter name, providing the system function get_parameter_description, which returns the relationship between variable names and their positions based on the SQL statement.

calling functions with OUT parameters in libpq

The libpq interface provides functions for preparing, binding, and executing statements, which are similar to the corresponding OCI functions. The usage process is as follows:

Use IvyHandleAlloc to allocate statement and error handles. Call IvyStmtPrepare to prepare the statement. Call IvyBindByPos or IvyBindByName to bind parameters. Call IvyStmtExecute to execute, which can be repeated. Call IvyFreeHandle to release the statement and error handles.

Additionally, a series of interface functions have been implemented, including Ivyconnectdb, Ivystatus, Ivyexec, IvyresultStatus, IvyCreatePreparedStatement, IvybindOutParameterByPos, IvyexecPreparedStatement, IvyexecPreparedStatement2, Ivynfields, Ivyntuples, and Ivyclear.

Example programs can be found in the src/interfaces/libpq/ivytest directory of the source code.

test examples

functions with OUT parameters

1. OUT parameters are not associated with the return value data type

```
ivorysql=# create or replace function test_return_out(id integer,price out
integer,name out varchar) return varchar
ivorysql-# as
ivorysql-# begin
ivorysql-#   price := 20000;
ivorysql-#   name := 'test a char out';
ivorysql-#   return 'welcome to QingDao';
ivorysql-# end;
ivorysql-# /
CREATE FUNCTION
```

1. Neither IN OUT nor OUT modes can have default values

```
ivorysql=# create or replace function test_return_inout(id integer,price in out
integer default 100,name out varchar) return varchar
ivorysql-# as
ivorysql-# begin
ivorysql-#   price := 20000 + price;
ivorysql-#   name := 'this is a test';
ivorysql-# return 'welcome to QingDao';
ivorysql-# end;
ivorysql-# /
ERROR:  IN OUT formal parameters may have no default expressions
```

1. If there are OUT parameters and the function’s return type is not void, the function body must include a RETURN statement.

```
--if function's return type is non-void, the function body must has RETURN statement
--if there is no RETURN statement, the function can be created, but when it is called,
--an error is raised
ivorysql=# create or replace function f2(id integer,price out integer) return varchar
ivorysql-# as
ivorysql-# begin
ivorysql-#   price := 2;
ivorysql-# end;
ivorysql-# /
CREATE FUNCTION
ivorysql=# declare
ivorysql-#   a varchar(20);
ivorysql-#   b int;
ivorysql-# begin
ivorysql-#   a := f2(1, b);
```

```
ivorysql-# end;
ivorysql-# /
ERROR:  Function returned without value
CONTEXT:  PL/iSQL function f2(pg_catalog.int4,pg_catalog.int4) line 0 at RETURN
PL/iSQL function inline_code_block line 5 at assignment
```

support OUT parameters in anonymous block

1. Supports binding variables in the form of colon placeholders and new DO+USING syntax

```
ivorysql=# do $$
ivorysql$# declare
ivorysql$#   a int;
ivorysql$# begin
ivorysql$#   :x := 1;
ivorysql$#   :y := 2;
ivorysql$# end; $$ using out, out;
 $1 | $2
----+----
  1 |  2
(1 row)
```

1. system function get_parameter_descr()

```
ivorysql=# select * from get_parameter_description('insert into t values(:x,:y)');
 name  | position
-------+----------
 false |        0
 :x    |        1
 :y    |        2
(3 rows)
```

.13. 13、%Type & %Rowtype

Purpose

IvorySQL provides Oracle-compatible PL/SQL data type functionality, including %TYPE and %ROWTYPE.

This document aims to introduce users to the functionality of %TYPE and %ROWTYPE.

Function descriptions

IvorySQL provides Oracle-compatible %TYPE and %ROWTYPE functionality, including the following content.

If the reference changes, variables declared with %TYPE or %ROWTYPE will change accordingly.

Create table and function.

```
CREATE TABLE t1(id int, name varchar(20));

--function's parameter datatype is tablename.columnname%TYPE
CREATE OR REPLACE FUNCTION fun1(v t1.id%TYPE) RETURN varchar AS
BEGIN
  RETURN v;
END;
/
```

The state of function is valid and function can be executed successfully.

```
SELECT prostatus FROM pg_proc WHERE proname like 'fun1'; --v
 prostatus
-----------
 v
(1 row)

SELECT fun1(1) FROM dual;
 fun1
------
 1
(1 row)
```

Modifying the previously defined declaration of a reference causes the function’s status to become invalid, but the function can still execute successfully.

```
ALTER TABLE t1 ALTER COLUMN id TYPE varchar(20);

SELECT prostatus FROM pg_proc WHERE proname like 'fun1'; --n
 prostatus
-----------
 n
(1 row)

--after changing the column id type from int to varchar, call the function again
SELECT fun1('a') FROM dual;  --successfully
 fun1
------
 a
```

```
(1 row)
```

re-compile the function, its state become valid.

```
ALTER FUNCTION fun1 COMPILE;
SELECT prostatus FROM pg_proc WHERE proname like 'fun1'; --v
 prostatus
-----------
 v
(1 row)
```

Variables declared with %TYPE inherit the constraints of the referenced variable.

Example:

```
--the following testcase will fail
DECLARE
  name     VARCHAR(25) NOT NULL := 'Niu';
  surname  name%TYPE ;
BEGIN
  raise notice 'name=%' ,name;
  raise notice 'surname=%' ,surname;
END;
/

ERROR:  variable "surname" must have a default value, since it's declared NOT NULL
LINE 3:   surname  name%TYPE ;
          ^
```

Use table_name%ROWTYPE or view_name%ROWTYPE as the parameter type of a function / stored procedure or the return type of a function

Example:

```
CREATE TABLE employees(first_name varchar(20) not null,
last_name varchar(20) not null,
phone_number varchar(50));

INSERT INTO employees VALUES ('Steven','Niu','1-650-555-1234');

CREATE OR REPLACE PROCEDURE p0(v employees%ROWTYPE) AS
BEGIN
  raise notice 'v.first_name = %, v.last_name = %, v.phone_number = %',
    v.first_name, v.last_name, v.phone_number;
```

```
END;
/

DECLARE
  a employees%ROWTYPE;
BEGIN
  select * into a from employees ;
  raise notice 'a=%', a;
  call p0(a);
END;
/

NOTICE:  a=(Steven,Niu,1-650-555-1234)
NOTICE:  v.first_name = Steven, v.last_name = Niu, v.phone_number = 1-650-555-1234

\df p0
                        List of functions
 Schema | Name | Result data type |  Argument data types   | Type
--------+------+------------------+------------------------+------
 public | p0   |                  | IN v employees%ROWTYPE | proc
(1 row)
```

Enhancement to INSERT statement

Support inserting a variables declared with %TYPE or %ROWTYPE into table.

grammar:

```
INSERT INTO table_name VALUES row_variable ;
```

Example:

```
CREATE TABLE t1(id int, name varchar(20));

DECLARE
  v1 t1%ROWTYPE;
BEGIN
  FOR i IN 1 .. 5 LOOP
    v1.id := i;
    v1.name := 'a' || i;
    INSERT INTO t1 VALUES v1;
  END LOOP;
END;
/
```

```
SELECT * FROM t1;
 id | name
----+------
  1 | a1
  2 | a2
  3 | a3
  4 | a4
  5 | a5
(5 rows)
```

Enhancement to UPDATE statement

example:

```
CREATE TABLE t1(id int, name varchar(20));

DELETE FROM t1;

DECLARE
  v1 t1%ROWTYPE;
  v2 t1%ROWTYPE;
BEGIN
  v1.id := 11;
  v1.name := 'abc';
  INSERT INTO t1 VALUES v1;
  v2.id := 22;
  v2.name := 'new';
  UPDATE t1 SET ROW = v2;
END;
/

SELECT * FROM t1;
 id | name
----+------
 22 | new
(1 row)
```

.14. 14、NLS Parameters

Purpose

National Language Support(NLS), which refers to the localization support functionality provided by Oracle. IvorySQL offers Oracle-compatible NLS parameter functionality.

Description

The following parameters are included:

Parameter Name	Description
ivorysql.datetime_ignore_nls_mask	Indicates whether the date format ignores the influence of NLS parameters. Default is 0 .
nls_length_semantics	Oracle-compatible parameter that specifies whether the size unit for CHAR , VARCHAR , and VARCHAR2 type modifiers is bytes or characters.
nls_date_format	Specifies the default date format, which can be viewed using the SHOW command. Default is 'YYYY-MM-DD' .
nls_timestamp_format	Oracle-compatible parameter that controls the format of timestamps.
nls_timestamp_tz_format	Oracle-compatible parameter that controls the format of timestamps with time zones.
nls_territory	Oracle-compatible parameter that specifies the default region for the database.
nls_iso_currency	Oracle-compatible parameter that assigns a unique currency symbol to a specified country or region.
nls_currency	Oracle-compatible parameter that specifies the symbol for the local currency, corresponding to the placeholder L in numeric string formats.

NLS Date Mask Settings

Example:

```
ivorysql=# set ivorysql.datetime_ignore_nls_mask = 0;
SET
ivorysql=# select '2025-10-15 11:00:00.102030 CST'::oratimestamp ;
ERROR:  datetime format picture ends before converting entire input string
LINE 1: select '2025-10-15 11:00:00.102030 CST'::oratimestamp ;
               ^
ivorysql=# set ivorysql.datetime_ignore_nls_mask = 2;
SET
ivorysql=# select '2025-10-15 11:00:00.102030 CST'::oratimestamp ;
        oratimestamp
----------------------------
 2025-10-15 11:00:00.102030
(1 row)
```

Disabling NLS Date or Timestamp Format

Example:

```
ivorysql=# select '2025-10-15 11:00:00.102030 '::oratimestamp ;
```

```
        oratimestamp
----------------------------
 2025-10-15 11:00:00.102030
(1 row)

ivorysql=# set nls_timestamp_format="pg";
SET
ivorysql=# select '2025-10-15 11:00:00.102030 '::oratimestamp ;
ERROR:  date format not recognized
LINE 1: select '2025-10-15 11:00:00.102030 '::oratimestamp ;
               ^
```

Parameter `nls_length_semantics` Settings

IvorySQL uses the value of the `nls_length_semantics` parameter to determine the length type, which can be either BYTE or CHAR (default is BYTE)。

Example:

```
ivorysql=# alter session set nls_length_semantics = char;
SET
ivorysql=# create table character_tb(char_c char(6), char_b varchar2(6 byte), char_v
varchar(6));
CREATE TABLE
ivorysql=#
ivorysql=# insert into character_tb values('abcdef', '123456', 'OK');
INSERT 0 1
ivorysql=# select * from character_tb ;
 char_c | char_b | char_v
--------+--------+--------
 abcdef | 123456 | OK
(1 row)

ivorysql=# select length(char_b), length(char_c), length(char_v) from character_tb;
 length | length | length
--------+--------+--------
      6 |      6 |      2
(1 row)
```

NLS Currency Symbols Settings

Example:

```
ivorysql=# show ivorysql.identifier_case_switch;
 ivorysql.identifier_case_switch
```

```
--------------------------------
 interchange
(1 row)

ivorysql=# set nls_currency to "CHINA";
SET
ivorysql=# show nls_currency;
 nls_currency
--------------
 CHINA
(1 row)

ivorysql=# set nls_currency to "China";
SET
ivorysql=# show nls_currency;
 nls_currency
--------------
 China
(1 row)
```



The behavior of `nls_currency` and `nls_iso_currency` is impacted by the Oracle `ivorysql.identifier_case_switch` compatibility feature.

.15. 15、Force View

Purpose

- This document explains the purpose of Force View in IvorySQL and shows how it enables users to create placeholder views when dependencies are not yet ready while keeping the behavior aligned with Oracle.
- Force View supports migrating views even when dependent tables are not yet ready, and can swiftly revert to regular views through automatic or explicit compilation once dependencies are satisfied.

Feature Description

- `CREATE [OR REPLACE] FORCE VIEW`: Creates a view object even when the query references missing tables or functions. The system retains the SQL definition and returns the message `WARNING: View created with compilation errors`.
- Automatic compilation: When a Force view is accessed, IvorySQL attempts recompilation. If it succeeds, the view becomes a normal view; if it fails, it raises `view "<schema>.<name>" has errors`.
- Dependency fallback: When a normal view is invalidated because dependencies are removed or altered, IvorySQL automatically converts it into a Force view, preserving the latest valid definition so it can be restored quickly once dependencies are rebuilt.

Test Cases

Create a Force View with Missing Dependencies

```
-- The base table does not exist, but the Force view placeholder is created
```

```
successfully.
CREATE FORCE VIEW fv_customer AS
SELECT c_id, c_name FROM missing_customer;
-- Expected output: WARNING: View created with compilation errors
```

Automatic Compilation and Recovery

```
-- Provide the missing dependency.
CREATE TABLE missing_customer(
  c_id   int primary key,
  c_name text
);
INSERT INTO missing_customer VALUES (1, 'Alice');

-- Accessing the Force view triggers automatic compilation.
SELECT * FROM fv_customer;
-- On success, the view becomes a normal view and returns data.
```

Explicit Compilation and Failure Fallback

```
-- Recreate the view definition as a Force view to make it invalid again.
CREATE OR REPLACE FORCE VIEW fv_customer AS
SELECT c_id, upper(c_name) AS c_name FROM missing_customer_v2;

-- Explicit compilation fails because dependencies are still missing.
ALTER VIEW fv_customer COMPILE;
-- Expected output: WARNING: View altered with compilation errors

-- Supply the required dependency and compile again.
CREATE TABLE missing_customer_v2(
  c_id   int,
  c_name text
);
ALTER VIEW fv_customer COMPILE;
-- Expected output: ALTER VIEW succeeds and the view returns to normal.
```

.16. 16、Nested Subfunctions

Objective

- Using Oracle-style nested subfunctions in IvorySQL.

Feature Description

- Allows declaring and invoking subfunctions or subprocedures inside anonymous blocks, functions, or procedures, with scope limited to the parent block.
- Nested subfunctions can read and update variables defined in the parent scope while introducing their own local variables; the parent scope cannot directly access the subfunction's internal state.
- Supports overloading resolution that distinguishes homonymous subfunctions by argument count, data type, or named parameters.

Test Cases

```
DO $$
DECLARE
  v_result integer;
  FUNCTION inner_square(p_value integer) RETURN integer IS
  BEGIN
    RAISE NOTICE 'inner_square called';
    RETURN p_value * p_value;
  END;
BEGIN
  v_result := inner_square(10);
  RAISE NOTICE 'result=%', v_result;
END;
$$ LANGUAGE plisql;
```

```
DO $$
DECLARE
  v_base_multiplier  integer := 20;
  v_audit_counter    integer := 0;
  v_result integer;
  FUNCTION inner_square(p_value integer) RETURN integer IS
  BEGIN
    RAISE NOTICE 'inner_square called';
    v_audit_counter := v_audit_counter + 1;
    RETURN v_base_multiplier * p_value;
  END;
BEGIN
  v_result := inner_square(10);
  RAISE NOTICE 'result=%', v_result;
  RAISE NOTICE 'v_audit_counter=%', v_audit_counter;
END;
$$ LANGUAGE plisql;
```

```
-- Polymorphic nested function specializing on argument type
DO $$
DECLARE
  v_last_notice text := 'none';

  FUNCTION describe_value(p_input anyelement) RETURN text IS
  BEGIN
    v_last_notice := format('polymorphic dispatch with %s', pg_typeof(p_input));
    RETURN v_last_notice;
  END;

  FUNCTION describe_value(p_input anyarray, p_element anyelement) RETURN text IS
  BEGIN
    v_last_notice := format('array dispatch with %s', pg_typeof(p_input)::text);
    RETURN v_last_notice;
  END;
BEGIN
  RAISE NOTICE '%', describe_value(100);
  RAISE NOTICE '%', describe_value('IvorySQL'::text); -- explicit cast avoids
ambiguous literal
  RAISE NOTICE '%', describe_value(ARRAY[1,2,3], NULL::int); -- extra arg guides
anyarray resolution
  RAISE NOTICE 'last notice=%', v_last_notice;
END;
$$ LANGUAGE plisql;
```

.17. 17、sys_guid Function

Purpose

IvorySQL’s sys_guid() is a powerful random number generation function that generates and returns a 16-byte database-level unique identifier (raw value).

Usage example

```
ivorysql=# select sys_guid() from dual;
              sys_guid
-----------------------------------
 \x3ed9426c8a093442a38bea09a74f44a1
(1 row)
```

The sys_guid function can generate default values for primary keys when creating a table

```
create table student
(
student_id raw(16) default sys_guid() primary key,
student_name varchar2(100) not null
);
```

The primary key is automatically populated when new data is added

```
insert into student(student_name) values ('Steven');
```

.18. 18、Empty String to NULL

Purpose

- In IvorySQL’s Oracle compatibility mode, support converting empty strings to NULL for storage, providing behavior consistent with Oracle Database.

Feature Description

- Oracle compatibility mode supports converting empty strings to NULL for storage.
- The feature is controlled by parameter `ivorysql.enable_emptystring_to_null`, with default value `on`.
- When this parameter is enabled, inserting an empty string will automatically convert it to NULL value for storage.
- NULL values after conversion can be queried using the `IS NULL` condition.

Test Cases

```
-- Create test table
ivorysql=# create table abc (id int);
CREATE TABLE

-- Check empty string to NULL parameter status
ivorysql=# show ivorysql.enable_emptystring_to_null;
 ivorysql.enable_emptystring_to_NULL
-------------------------------------
 on
(1 row)

-- Insert empty string
ivorysql=# insert into abc values('');
INSERT 0 1
```

```
-- Query table data, displays as NULL
ivorysql=# select * from abc;
 id
\----

(1 row)

-- Query using IS NULL condition
ivorysql=# select * from abc where id is null;
 id
\----

(1 row)
```

.19. 19、CALL INTO

Objective

- In IvorySQL, the CALL statement supports invoking standalone function stored procedures, as well as functions and stored procedures within packages or object types. The CALL syntax for invoking functions adds an INTO clause, where the insertion target is a host variable.

Feature Description

- CALL supports invoking standalone functions and stored procedures, as well as those defined within packages.
- When using CALL to invoke functions, the INTO clause syntax is added, with the insertion target being a host variable.
- When calling functions/stored procedures with no parameters or all default values, empty parentheses cannot be omitted.
- CALL supports referencing bind variables in parameters or the INTO clause when invoking functions and stored procedures.
- The binding variables corresponding to OUT parameters in the CALL statement support validation of precision and data types.
- Output binding variables are not allowed to be repeatedly bound.

Test Cases

Call INTO invokes a function and inserts the result into a host variable.

```
-- Create function
ivorysql=# create or replace function f_defs(a number default 1314)
ivorysql-# return number
ivorysql-# is
ivorysql-# begin
ivorysql-# raise notice '%', a;
ivorysql-# return a;
```

```
ivorysql-# end;
ivorysql-# /
CREATE FUNCTION
-- Declare bind variable
ivorysql=# variable x number
-- Call the function and retrieve the return value
ivorysql=# call f_defs() into :x;
NOTICE: 1314

Call completed.

ivorysql=# print x
  X
------
 1314
```

Call the stored procedure in the package

```
ivorysql=# create table tb1(c1 int);
CREATE TABLE
-- Create package specification
ivorysql=# create or replace package pkg is
ivorysql-# var1 integer;
ivorysql-# procedure test_p ;
ivorysql-# end;
ivorysql-# /
CREATE PACKAGE
-- Create package body
ivorysql=# create or replace package body pkg is
ivorysql-# procedure test_p is
ivorysql-# begin
ivorysql-# insert into tb1 values(1);
ivorysql-# end;
ivorysql-# begin
ivorysql-# var1 := 2;
ivorysql-# end;
ivorysql-# /
CREATE PACKAGE BODY
-- Call the procedure in the package
ivorysql=# call pkg.test_p();
CALL
ivorysql=# select * from tb1;
```

```
 c1
-----
  1
(1 row)
```

Call with no parameters or default parameters

```
-- Create a function with default parameters
ivorysql=# CREATE OR REPLACE FUNCTION default_arg_func(p_num NUMBER DEFAULT 100)
RETURN NUMBER AS
ivorysql-# BEGIN
ivorysql-#   RETURN p_num + 5;
ivorysql-# END;
ivorysql-# /
CREATE FUNCTION
-- Correctly call a function with default parameters (must include parentheses)
ivorysql=# VARIABLE default_result NUMBER;
ivorysql=# CALL default_arg_func() INTO :default_result; -- Use the default value of
100

Call completed.

ivorysql=# PRINT default_result;
 DEFAULT_RESULT
----------------
 105

-- Call with parameters
ivorysql=# CALL default_arg_func(200) INTO :default_result;

Call completed.

ivorysql=# PRINT default_result;
 DEFAULT_RESULT
----------------
 205
```

Reference bind variables in parameters or INTO clauses

```
-- Create functions stand_alone_func
CREATE OR REPLACE FUNCTION stand_alone_func(p_input NUMBER)
RETURN NUMBER
IS
```

```
BEGIN
    RETURN p_input * 2;
END;
/
-- Set input bind variables
ivorysql=# VARIABLE input_num NUMBER = 7;
-- Call a function using bind variables as parameters
ivorysql=# VARIABLE func_result NUMBER;
ivorysql=# CALL stand_alone_func(:input_num) INTO :func_result;

Call completed.

ivorysql=# PRINT func_result;
 FUNC_RESULT
-------------
 14
```

The OUT parameters in CALL statements support precision and data type validation

```
-- Create a procedure with OUT parameters
ivorysql=# CREATE OR REPLACE PROCEDURE out_param_proc(
ivorysql(#   p_in IN VARCHAR2,
ivorysql(#   p_out OUT VARCHAR2,
ivorysql(#   p_num_out OUT NUMBER
ivorysql(# ) AS
ivorysql-# BEGIN
ivorysql-#   p_out := p_in || ' processed';
ivorysql-#   p_num_out := LENGTH(p_in);
ivorysql-# END;
ivorysql-# /
CREATE PROCEDURE
-- Test OUT parameter type matching
ivorysql=# VARIABLE out_var VARCHAR2(50);
ivorysql=# VARIABLE num_var NUMBER;
ivorysql=# CALL out_param_proc('Test input', :out_var, :num_var);

Call completed.

ivorysql=# PRINT out_var;
       OUT_VAR
----------------------
 Test input processed
```

```
ivorysql=# PRINT num_var;
 NUM_VAR
---------
 10

-- Test insufficient OUT parameter precision (will be truncated)
ivorysql=# VARIABLE short_out VARCHAR2(5);
ivorysql=# CALL out_param_proc('Long input string', :short_out, :num_var);

Call completed.

ivorysql=# PRINT short_out;
 SHORT_OUT
-----------
 Long

-- Test type mismatch (will throw an error)
ivorysql=# VARIABLE wrong_type NUMBER;
ivorysql=# CALL out_param_proc('Test', :wrong_type, :num_var);
ERROR:  invalid input syntax for type numeric: "Test processed"
```

Output binding variables do not allow duplicate binding.

```
-- Prepare binding variables
ivorysql=# VARIABLE dup_var VARCHAR2(100);
-- Attempting duplicate binding (will throw an error)
ivorysql=# CALL out_param_func('Test', :dup_var) INTO :dup_var;
ERROR:  output parameter cannot be a duplicate bind
-- Correct approach: Use different binding variables
ivorysql=# VARIABLE out1 VARCHAR2(100);
ivorysql=# CALL out_param_proc('Correct usage', :out1, :num_var);

Call completed.

ivorysql=# PRINT out1;
          OUT1
-------------------------
 Correct usage processed
```

Containerization and Cloud Service

Containerization

..1. K8S deployment

Single-node container

On the master node of the k8s cluster, create a namespace named ivorysql.

```
[root@k8s-master ~]# kubectl create ns ivorysql
```

Download the latest docker_library code.

```
[root@k8s-master ~]# git clone https://github.com/IvorySQL/docker_library.git
```

Enter the single-node directory

```
[root@k8s-master ~]# cd docker_library/k8s-cluster/single
[root@k8s-master single]# vim statefulset.yaml #Update the PVC information and
database password in the StatefulSet to match your actual environment.
```

Use statefulset.yaml to create a single-node pod.

```
[root@k8s-master single]# kubectl apply -f statefulset.yaml
service/ivorysql-svc created
statefulset.apps/ivorysql created
```

Wait for the single-node pod to be successfully created.

```
[root@k8s-master single]# kubectl get all -n ivorysql
NAME             READY   STATUS              RESTARTS   AGE
pod/ivorysql-0   0/1     ContainerCreating   0          47s

NAME                  TYPE       CLUSTER-IP       EXTERNAL-IP   PORT(S)
AGE
service/ivorysql-svc   NodePort   10.108.178.236   <none>
5432:32106/TCP,1521:31887/TCP   47s

NAME                         READY   AGE
statefulset.apps/ivorysql   0/1     47s
[root@k8s-master single]# kubectl get all -n ivorysql
NAME             READY   STATUS    RESTARTS   AGE
pod/ivorysql-0   1/1     Running   0          2m39s
```

```
NAME                    TYPE       CLUSTER-IP       EXTERNAL-IP   PORT(S)
AGE
service/ivorysql-svc    NodePort   10.108.178.236   <none>
5432:32106/TCP,1521:31887/TCP   2m39s

NAME                          READY   AGE
statefulset.apps/ivorysql     1/1     2m39s
```

Connect to IvorySQL via its PostgreSQL port using the psql

```
[root@k8s-master single]# psql -U ivorysql -p 32106 -h 127.0.0.1 -d ivorysql
Password for user ivorysql:

ivorysql=# select version();
                                                          version
------------------------------------------------------------------------------------------
----------------------------------
 PostgreSQL 18.0 (IvorySQL 5.4) on x86_64-pc-linux-gnu, compiled by gcc (GCC) 8.5.0
20210514 (Red Hat 8.5.0-28), 64-bit
(1 row)

ivorysql=# show ivorysql.compatible_mode;
 ivorysql.compatible_mode
--------------------------
 pg
(1 row)

ivorysql=# exit
```

Connect to IvorySQL’s Oracle-compatible port using psql.

```
[root@k8s-master single]# psql -U ivorysql -p 31887 -h 127.0.0.1 -d ivorysql
Password for user ivorysql:

ivorysql=# select version();
                                                          version
------------------------------------------------------------------------------------------
----------------------------------
 PostgreSQL 18.0 (IvorySQL 5.4) on x86_64-pc-linux-gnu, compiled by gcc (GCC) 8.5.0
20210514 (Red Hat 8.5.0-28), 64-bit
(1 row)

ivorysql=# show ivorysql.compatible_mode;
```

```
 ivorysql.compatible_mode
--------------------------
 oracle
(1 row)
```

Uninstall Single-node container

```
[root@k8s-master single]# kubectl delete -f statefulset.yaml
```

High Availability Cluster

Access the master node of the k8s cluster and create a namespace named ivorysql.

```
[root@k8s-master ~]# kubectl create ns ivorysql
```

Download the latest docker_library code.

```
[root@k8s-master ~]# git clone https://github.com/IvorySQL/docker_library.git
```

Enter the high-availability cluster directory.

```
[root@k8s-master ~]# cd docker_library/k8s-cluster/ha-cluster/helm_charts
[root@k8s-master single]# vim values.yaml #Adjust the PVC settings, cluster size, and
other configurations in values.yaml according to your environment. For the database
password, check templates/secret.yaml and modify it as needed.
```

Deploy the high-availability cluster using Helm commands.

```
[root@k8s-master helm_charts]# helm install ivorysql-ha-cluster -n ivorysql .
NAME: ivorysql-ha-cluster
LAST DEPLOYED: Wed Sep 10 09:45:36 2025
NAMESPACE: ivorysql
STATUS: deployed
REVISION: 1
TEST SUITE: None
[root@k8s-master helm_charts]# kubectl get all -n ivorysql
NAME                          READY   STATUS    RESTARTS   AGE
pod/ivorysql-patroni-hac-0    1/1     Running   0          42s
pod/ivorysql-patroni-hac-1    0/1     Running   0          18s

NAME                                  TYPE        CLUSTER-IP       EXTERNAL-IP
PORT(S)                          AGE
service/ivorysql-patroni-hac          NodePort    10.96.119.203    <none>
5432:32391/TCP,1521:32477/TCP   42s
```

```
service/ivorysql-patroni-hac-config   ClusterIP   None            <none>
<none>                          42s
service/ivorysql-patroni-hac-pods     ClusterIP   None            <none>
<none>                          42s
service/ivorysql-patroni-hac-repl     NodePort    10.100.122.0    <none>
5432:30111/TCP,1521:32654/TCP   42s

NAME                                    READY   AGE
statefulset.apps/ivorysql-patroni-hac   1/3     42s
```

Wait until all pods are running successfully, indicating the cluster deployment is complete.

```
[root@k8s-master helm_charts]# kubectl get all -n ivorysql
NAME                          READY   STATUS    RESTARTS   AGE
pod/ivorysql-patroni-hac-0    1/1     Running   0          88s
pod/ivorysql-patroni-hac-1    1/1     Running   0          64s
pod/ivorysql-patroni-hac-2    1/1     Running   0          41s

NAME                                    TYPE        CLUSTER-IP       EXTERNAL-IP
PORT(S)                         AGE
service/ivorysql-patroni-hac            NodePort    10.96.119.203    <none>
5432:32391/TCP,1521:32477/TCP   88s
service/ivorysql-patroni-hac-config   ClusterIP   None             <none>
<none>                          88s
service/ivorysql-patroni-hac-pods     ClusterIP   None             <none>
<none>                          88s
service/ivorysql-patroni-hac-repl     NodePort    10.100.122.0     <none>
5432:30111/TCP,1521:32654/TCP   88s
NAME                                    READY   AGE
statefulset.apps/ivorysql-patroni-hac   3/3     88s
```

Connect to the PostgreSQL and Oracle ports of the cluster’s primary node using psql.

```
[root@k8s-master helm_charts]# psql -U ivorysql -p 32391 -h 127.0.0.1 -d ivorysql
Password for user ivorysql:

ivorysql=# show ivorysql.compatible_mode;
 ivorysql.compatible_mode
--------------------------
 pg
(1 row)

ivorysql=# SELECT pg_is_in_recovery();
 pg_is_in_recovery
```

```
-------------------
 f
(1 row)

ivorysql=# exit
```

```
[root@k8s-master helm_charts]# psql -U ivorysql -p 32477 -h 127.0.0.1 -d ivorysql
Password for user ivorysql:

ivorysql=# show ivorysql.compatible_mode;
 ivorysql.compatible_mode
--------------------------
 oracle
(1 row)

ivorysql=# SELECT pg_is_in_recovery();
 pg_is_in_recovery
-------------------
 f
(1 row)

ivorysql=#
```

Use psql to connect to the PostgreSQL and Oracle ports of the cluster’s standby node.

```
[root@k8s-master helm_charts]# psql -U ivorysql -p 30111 -h 127.0.0.1 -d ivorysql
Password for user ivorysql:

ivorysql=# show ivorysql.compatible_mode;
 ivorysql.compatible_mode
--------------------------
 pg
(1 row)

ivorysql=# SELECT pg_is_in_recovery();
 pg_is_in_recovery
-------------------
 t
(1 row)

ivorysql=# exit
```

```
[root@k8s-master helm_charts]# psql -U ivorysql -p 32654 -h 127.0.0.1 -d ivorysql
Password for user ivorysql:

ivorysql=#  show ivorysql.compatible_mode;
 ivorysql.compatible_mode
--------------------------
 oracle
(1 row)

ivorysql=# SELECT pg_is_in_recovery();
 pg_is_in_recovery
-------------------
 t
(1 row)

ivorysql=#
```

Uninstall high-availability cluster

```
[root@k8s-master helm_charts]# helm uninstall ivorysql-ha-cluster -n ivorysql
```

Remove PVC

```
[root@k8s-master helm_charts]# kubectl delete pvc ivyhac-config-ivorysql-patroni-hac-0
-n ivorysql
[root@k8s-master helm_charts]# kubectl delete pvc ivyhac-config-ivorysql-patroni-hac-1
-n ivorysql
[root@k8s-master helm_charts]# kubectl delete pvc ivyhac-config-ivorysql-patroni-hac-2
-n ivorysql
[root@k8s-master helm_charts]# kubectl delete pvc pgdata-ivorysql-patroni-hac-0 -n
ivorysql
[root@k8s-master helm_charts]# kubectl delete pvc pgdata-ivorysql-patroni-hac-1 -n
ivorysql
[root@k8s-master helm_charts]# kubectl delete pvc pgdata-ivorysql-patroni-hac-2 -n
ivorysql
```

..2. Operator deployment

Operator Installation

1. Fork ivory-operator repository and clone it to your host machine:

```
YOUR_GITHUB_UN="<your GitHub username>"
git clone --depth 1 "git@github.com:${YOUR_GITHUB_UN}/ivory-operator.git"
```

```
cd ivory-operator
```

2. Run the following commands:

```
kubectl apply -k examples/kustomize/install/namespace
kubectl apply --server-side -k examples/kustomize/install/default
```

Getting Started

Throughout this tutorial, we will be building on the example provided in the `examples/kustomize/ivory`.

When referring to a nested object within a YAML manifest, we will be using the `.` format similar to `kubectl explain`. For example, if we want to refer to the deepest element in this yaml file:

```
spec:
  hippos:
    appetite: huge
```

we would say `spec.hippos.appetite`.

`kubectl explain` is your friend. You can use `kubectl explain ivorycluster` to introspect the `ivorycluster.ivory-operator.ivorysql.org` custom resource definition.

Create an Ivory Cluster

Create

Creating an Ivory cluster is pretty simple. Using the example in the `examples/kustomize/ivory` directory, all we have to do is run:

```
kubectl apply -k examples/kustomize/ivory
```

and IVYO will create a simple Ivory cluster named `hippo` in the `ivory-operator` namespace. You can track the status of your Ivory cluster using `kubectl describe` on the `ivoryclusters.ivory-operator.ivorysql.org` custom resource:

```
kubectl -n ivory-operator describe ivoryclusters.ivory-operator.ivorysql.org hippo
```

and you can track the state of the Ivory Pod using the following command:

```
kubectl -n ivory-operator get pods \
  --selector=ivory-operator.ivorysql.org/cluster=hippo,ivory-
operator.ivorysql.org/instance
```

What Just Happened?

IVYO created an Ivory cluster based on the information provided to it in the Kustomize manifests located in the `examples/kustomize/ivory` directory. Let's better understand what happened by inspecting the `examples/kustomize/ivory/ivory.yaml` file:

```
apiVersion: ivory-operator.ivorysql.org/v1beta1
kind: IvoryCluster
metadata:
  name: hippo
spec:
  image: {{< param imageIvorySQL >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - name: instance1
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      image: {{< param imagePGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
```

When we ran the `kubectl apply` command earlier, what we did was create a `ivorycluster` custom resource in Kubernetes. IVYO detected that we added a new `ivorycluster` resource and started to create all the objects needed to run Ivory in Kubernetes!

What else happened? IVYO read the value from `metadata.name` to provide the Ivory cluster with the name `hippo`. Additionally, IVYO knew which containers to use for Ivory and pgBackRest by looking at the values in `spec.image` and `spec.backups.pgbackrest.image` respectively. The value in `spec.postgresVersion` is important as it will help IVYO track which major version of Ivory you are using.

IVYO knows how many Ivory instances to create through the `spec.instances` section of the manifest. While `name` is optional, we opted to give it the name `instance1`. We could have also created multiple replicas and instances during cluster initialization, but we will cover that more when we discuss how to scale and create a HA Ivory cluster.

A very important piece of your `ivorycluster` custom resource is the `dataVolumeClaimSpec` section. This describes the storage that your Ivory instance will use. It is modeled after the Persistent Volume Claim. If you do not provide a `spec.instances.dataVolumeClaimSpec.storageClassName`, then the default storage class in your Kubernetes environment is used.

As part of creating an Ivory cluster, we also specify information about our backup archive. IVYO uses pgBackRest, an open source backup and restore tool designed to handle terabyte-scale backups. As part of

initializing our cluster, we can specify where we want our backups and archives (write-ahead logs or WAL) stored. We will talk about this portion of the `ivorycluster` spec in greater depth in the disaster recovery section of this tutorial, and also see how we can store backups in Amazon S3, Google GCS, and Azure Blob Storage.

Troubleshooting

IvorySQL / pgBackRest Pods Stuck in `Pending` Phase

The most common occurrence of this is due to PVCs not being bound. Ensure that you have set up your storage options correctly in any `volumeClaimSpec`. You can always update your settings and reapply your changes with `kubectl apply`.

Also ensure that you have enough persistent volumes available: your Kubernetes administrator may need to provision more.

If you are on OpenShift, you may need to set `spec.openshift` to `true`.

Next Steps

We're up and running — now let's connect to our Ivory cluster!

Connect to an Ivory Cluster

It's one thing to create an Ivory cluster; it's another thing to connect to it. Let's explore how IVYO makes it possible to connect to an Ivory cluster!

Background: Services, Secrets, and TLS

IVYO creates a series of Kubernetes Services to provide stable endpoints for connecting to your Ivory databases. These endpoints make it easy to provide a consistent way for your application to maintain connectivity to your data. To inspect what services are available, you can run the following command:

```
kubectl -n ivory-operator get svc --selector=ivory-operator.ivorysql.org/cluster=hippo
```

will yield something similar to:

```
NAME              TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
hippo-ha          ClusterIP   10.103.73.92   <none>        5432/TCP   3h14m
hippo-ha-config   ClusterIP   None           <none>        <none>     3h14m
hippo-pods        ClusterIP   None           <none>        <none>     3h14m
hippo-primary     ClusterIP   None           <none>        5432/TCP   3h14m
hippo-replicas    ClusterIP   10.98.110.215  <none>        5432/TCP   3h14m
```

You do not need to worry about most of these Services, as they are used to help manage the overall health of your Ivory cluster. For the purposes of connecting to your database, the Service of interest is called `hippo-primary`. Thanks to IVYO, you do not need to even worry about that, as that information is captured within a Secret!

When your Ivory cluster is initialized, IVYO will bootstrap a database and Ivory user that your application can access. This information is stored in a Secret named with the pattern `<clusterName>-pguser-<userName>`. For our `hippo` cluster, this Secret is called `hippo-pguser-hippo`. This Secret contains the information you need to connect your application to your Ivory database:

- `user`: The name of the user account.

- `password`: The password for the user account.
- `dbname`: The name of the database that the user has access to by default.
- `host`: The name of the host of the database. This references the Service of the primary Ivory instance.
- `port`: The port that the database is listening on.
- `uri`: A PostgresSQL connection URI that provides all the information for logging into the Ivory database.
- `jdbc-uri`: A PostgresSQL JDBC connection URI that provides all the information for logging into the Ivory database via the JDBC driver.

All connections are over TLS. IVYO provides its own certificate authority (CA) to allow you to securely connect your applications to your Ivory clusters. This allows you to use the `verify-full` "SSL mode" of Ivory, which provides eavesdropping protection and prevents MITM attacks. You can also choose to bring your own CA, which is described later in this tutorial in the Customize Cluster section.

Modifying Service Type, NodePort Value and Metadata

By default, IVYO deploys Services with the `ClusterIP` Service type. Based on how you want to expose your database, you may want to modify the Services to use a different Service type and NodePort value.

You can modify the Services that IVYO manages from the following attributes:

- `spec.service` - this manages the Service for connecting to an Ivory primary.
- `spec.userInterface.pgAdmin.service` - this manages the Service for connecting to the pgAdmin management tool.

For example, say you want to set the Ivory primary to use a `NodePort` service, a specific `nodePort` value, and set a specific annotation and label, you would add the following to your manifest:

```
spec:
  service:
    metadata:
      annotations:
        my-annotation: value1
      labels:
        my-label: value2
    type: NodePort
    nodePort: 32000
```

For our `hippo` cluster, you would see the Service type and nodePort modification as well as the annotation and label. For example:

```
kubectl -n ivory-operator get svc --selector=ivory-operator.ivorysql.org/cluster=hippo
```

will yield something similar to:

```
NAME              TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
hippo-ha          NodePort    10.105.57.191   <none>        5432:32000/TCP   48s
hippo-ha-config   ClusterIP   None            <none>        <none>           48s
hippo-pods        ClusterIP   None            <none>        <none>           48s
hippo-primary     ClusterIP   None            <none>        5432/TCP         48s
```

```
hippo-replicas    ClusterIP   10.106.18.99    <none>        5432/TCP        48s
```

and the top of the output from running

```
kubectl -n ivory-operator describe svc hippo-ha
```

will show our custom annotation and label have been added:

```
Name:              hippo-ha
Namespace:         ivory-operator
Labels:            my-label=value2
                   ivory-operator.ivorysql.org/cluster=hippo
                   ivory-operator.ivorysql.org/patroni=hippo-ha
Annotations:       my-annotation: value1
```

Note that setting the `nodePort` value is not allowed when using the (default) `ClusterIP` type, and it must be in-range and not otherwise in use or the operation will fail. Additionally, be aware that any annotations or labels provided here will win in case of conflicts with any annotations or labels a user configures elsewhere.

Finally, if you are exposing your Services externally and are relying on TLS verification, you will need to use the custom TLS features of IVYO).

Connect an Application

For this tutorial, we are going to connect Keycloak, an open source identity management application. Keycloak can be deployed on Kubernetes and is backed by an Ivory database. We provide an example of deploying Keycloak andan ivorycluster, the manifest below deploys it using our `hippo` cluster that is already running:

```
kubectl apply --filename=- <<EOF
apiVersion: apps/v1
kind: Deployment
metadata:
  name: keycloak
  namespace: ivory-operator
  labels:
    app.kubernetes.io/name: keycloak
spec:
  selector:
    matchLabels:
      app.kubernetes.io/name: keycloak
  template:
    metadata:
      labels:
        app.kubernetes.io/name: keycloak
    spec:
```

```
      containers:
      - image: quay.io/keycloak/keycloak:latest
        args: ["start-dev"]
        name: keycloak
        env:
        - name: DB_VENDOR
          value: "ivory"
        - name: DB_ADDR
          valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: host } }
        - name: DB_PORT
          valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: port } }
        - name: DB_DATABASE
          valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: dbname } }
        - name: DB_USER
          valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: user } }
        - name: DB_PASSWORD
          valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: password } }
        - name: KEYCLOAK_ADMIN
          value: "admin"
        - name: KEYCLOAK_ADMIN_PASSWORD
          value: "admin"
        - name: KC_PROXY
          value: "edge"
        ports:
        - name: http
          containerPort: 8080
        - name: https
          containerPort: 8443
        readinessProbe:
          httpGet:
            path: /realms/master
            port: 8080
      restartPolicy: Always
EOF
```

Notice this part of the manifest:

```
- name: DB_ADDR
  valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: host } }
- name: DB_PORT
  valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: port } }
- name: DB_DATABASE
```

```
    valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: dbname } }
  - name: DB_USER
    valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: user } }
  - name: DB_PASSWORD
    valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: password } }
```

The above manifest shows how all of these values are derived from the `hippo-pguser-hippo` Secret. This means that we do not need to know any of the connection credentials or have to insecurely pass them around — they are made directly available to the application!

Using this method, you can tie application directly into your GitOps pipeline that connect to Ivory without any prior knowledge of how IVYO will deploy Ivory: all of the information your application needs is propagated into the Secret!

Next Steps

Now that we have seen how to connect an application to a cluster, let's learn how to create a high availability Ivory cluster!

High Availability

Ivory is known for its reliability: it is very stable and typically "just works." However, there are many things that can happen in a distributed environment like Kubernetes that can affect Ivory uptime, including:

- The database storage disk fails or some other hardware failure occurs
- The network on which the database resides becomes unreachable
- The host operating system becomes unstable and crashes
- A key database file becomes corrupted
- A data center is lost
- A Kubernetes component (e.g. a Service) is accidentally deleted

There may also be downtime events that are due to the normal case of operations, such as performing a minor upgrade, security patching of operating system, hardware upgrade, or other maintenance.

The good news: IVYO is prepared for this, and your Ivory cluster is protected from many of these scenarios. However, to maximize your high availability (HA), let's first scale up your Ivory cluster.

HA Ivory: Adding Replicas to your Ivory Cluster

IVYO provides several ways to add replicas to make a HA cluster:

- Increase the `spec.instances.replicas` value
- Add an additional entry in `spec.instances`

For the purposes of this tutorial, we will go with the first method and set `spec.instances.replicas` to `2`. Your manifest should look similar to:

```
apiVersion: ivory-operator.ivorysql.org/v1beta1
kind: IvoryCluster
metadata:
  name: hippo
spec:
```

```
  image: {{< param imageIvorySQL >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - name: instance1
      replicas: 2
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      image: {{< param imagePGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
```

Apply these updates to your Ivory cluster with the following command:

```
kubectl apply -k examples/kustomize/ivory
```

Within moment, you should see a new Ivory instance initializing! You can see all of your Ivory Pods for the `hippo` cluster by running the following command:

```
kubectl -n ivory-operator get pods \
  --selector=ivory-operator.ivorysql.org/cluster=hippo,ivory-
operator.ivorysql.org/instance-set
```

Let’s test our high availability set up.

Testing Your HA Cluster

An important part of building a resilient Ivory environment is testing its resiliency, so let’s run a few tests to see how IVYO performs under pressure!

Test #1: Remove a Service

Let’s try removing the primary Service that our application is connected to. This test does not actually require a HA Ivory cluster, but it will demonstrate IVYO’s ability to react to environmental changes and heal things to ensure your applications can stay up.

Recall in the connecting a Ivory cluster that we observed the Services that IVYO creates, e.g.:

```
kubectl -n ivory-operator get svc \
  --selector=ivory-operator.ivorysql.org/cluster=hippo
```

yields something similar to:

```
NAME               TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
hippo-ha           ClusterIP   10.103.73.92   <none>        5432/TCP   4h8m
hippo-ha-config    ClusterIP   None           <none>        <none>     4h8m
hippo-pods         ClusterIP   None           <none>        <none>     4h8m
hippo-primary      ClusterIP   None           <none>        5432/TCP   4h8m
hippo-replicas     ClusterIP   10.98.110.215  <none>        5432/TCP   4h8m
```

We also mentioned that the application is connected to the `hippo-primary` Service. What happens if we were to delete this Service?

```
kubectl -n ivory-operator delete svc hippo-primary
```

This would seem like it could create a downtime scenario, but run the above selector again:

```
kubectl -n ivory-operator get svc \
  --selector=ivory-operator.ivorysql.org/cluster=hippo
```

You should see something similar to:

```
NAME               TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
hippo-ha           ClusterIP   10.103.73.92   <none>        5432/TCP   4h8m
hippo-ha-config    ClusterIP   None           <none>        <none>     4h8m
hippo-pods         ClusterIP   None           <none>        <none>     4h8m
hippo-primary      ClusterIP   None           <none>        5432/TCP   3s
hippo-replicas     ClusterIP   10.98.110.215  <none>        5432/TCP   4h8m
```

Wow — IVYO detected that the primary Service was deleted and it recreated it! Based on how your application connects to Ivory, it may not have even noticed that this event took place!

Now let's try a more extreme downtime event.

Test #2: Remove the Primary StatefulSet

StatefulSets are a Kubernetes object that provide helpful mechanisms for managing Pods that interface with stateful applications, such as databases. They provide a stable mechanism for managing Pods to help ensure data is retrievable in a predictable way.

What happens if we remove the StatefulSet that is pointed to the Pod that represents the Ivory primary? First, let's determine which Pod is the primary. We'll store it in an environmental variable for convenience.

```
PRIMARY_POD=$(kubectl -n ivory-operator get pods \
  --selector=ivory-operator.ivorysql.org/role=master \
  -o jsonpath='{.items[*].metadata.labels.ivory-operator\.ivorysql\.org/instance}')
```

Inspect the environmental variable to see which Pod is the current primary:

```
echo $PRIMARY_POD
```

should yield something similar to:

```
hippo-instance1-zj5s
```

We can use the value above to delete the StatefulSet associated with the current Ivory primary instance:

```
kubectl delete sts -n ivory-operator "${PRIMARY_POD}"
```

Let's see what happens. Try getting all of the StatefulSets for the Ivory instances in the `hippo` cluster:

```
kubectl get sts -n ivory-operator \
  --selector=ivory-operator.ivorysql.org/cluster=hippo,ivory-
operator.ivorysql.org/instance
```

You should see something similar to:

```
NAME                 READY   AGE
hippo-instance1-6kbw   1/1     15m
hippo-instance1-zj5s   0/1     1s
```

IVYO recreated the StatefulSet that was deleted! After this "catastrophic" event, IVYO proceeds to heal the Ivory instance so it can rejoin the cluster. We cover the high availability process in greater depth later in the documentation.

What about the other instance? We can see that it became the new primary though the following command:

```
kubectl -n ivory-operator get pods \
  --selector=ivory-operator.ivorysql.org/role=master \
  -o jsonpath='{.items[*].metadata.labels.ivory-operator\.ivorysql\.org/instance}'
```

which should yield something similar to:

```
hippo-instance1-6kbw
```

You can test that the failover successfully occurred in a few ways. You can connect to the example Keycloak application that we deployed in the previous section. Based on Keycloak's connection retry logic, you may

need to wait a moment for it to reconnect, but you will see it connected and resume being able to read and write data. You can also connect to the Ivory instance directly and execute the following command:

```
SELECT NOT pg_catalog.pg_is_in_recovery() is_primary;
```

If it returns `true` (or `t`), then the Ivory instance is a primary!

What if IVYO was down during the downtime event? Failover would still occur: the Ivory HA system works independently of IVYO and can maintain its own uptime. IVYO will still need to assist with some of the healing aspects, but your application will still maintain read/write connectivity to your Ivory cluster!

Synchronous Replication

IvorySQL supports synchronous replication, which is a replication mode designed to limit the risk of transaction loss. Synchronous replication waits for a transaction to be written to at least one additional server before it considers the transaction to be committed. For more information on synchronous replication, please read about IVYO’ s high availability architecture

To add synchronous replication to your Ivory cluster, you can add the following to your spec:

```
spec:
  patroni:
    dynamicConfiguration:
      synchronous_mode: true
```

While PostgreSQL defaults `synchronous_commit` to `on`, you may also want to explicitly set it, in which case the above block becomes:

```
spec:
  patroni:
    dynamicConfiguration:
      synchronous_mode: true
      postgresql:
        parameters:
          synchronous_commit: "on"
```

Note that Patroni, which manages many aspects of the cluster’ s availability, will favor availability over synchronicity. This means that if a synchronous replica goes down, Patroni will allow for asynchronous replication to continue as well as writes to the primary. However, if you want to disable all writing if there are no synchronous replicas available, you would have to enable `synchronous_mode_strict`, i.e.:

```
spec:
  patroni:
    dynamicConfiguration:
      synchronous_mode: true
      synchronous_mode_strict: true
```

Affinity

Kubernetes affinity rules, which include Pod anti-affinity and Node affinity, can help you to define where you want your workloads to reside. Pod anti-affinity is important for high availability: when used correctly, it ensures that your Ivory instances are distributed amongst different Nodes. Node affinity can be used to assign instances to specific Nodes, e.g. to utilize hardware that's optimized for databases.

Understanding Pod Labels

IVYO sets up several labels for Ivory cluster management that can be used for Pod anti-affinity or affinity rules in general. These include:

- `ivory-operator.ivorysql.org/cluster`: This is assigned to all managed Pods in a Ivory cluster. The value of this label is the name of your Ivory cluster, in this case: `hippo`.
- `ivory-operator.ivorysql.org/instance-set`: This is assigned to all Ivory instances within a group of `spec.instances`. In the example above, the value of this label is `instance1`. If you do not assign a label, the value is automatically set by IVYO using a `NN` format, e.g. `00`.
- `ivory-operator.ivorysql.org/instance`: This is a unique label assigned to each Ivory instance containing the name of the Ivory instance.

Let's look at how we can set up affinity rules for our Ivory cluster to help improve high availability.

Pod Anti-affinity

Kubernetes has two types of Pod anti-affinity:

- Preferred: With preferred (`preferredDuringSchedulingIgnoredDuringExecution`) Pod anti-affinity, Kubernetes will make a best effort to schedule Pods matching the anti-affinity rules to different Nodes. However, if it is not possible to do so, then Kubernetes may schedule one or more Pods to the same Node.
- Required: With required (`requiredDuringSchedulingIgnoredDuringExecution`) Pod anti-affinity, Kubernetes mandates that each Pod matching the anti-affinity rules must be scheduled to different Nodes. However, a Pod may not be scheduled if Kubernetes cannot find a Node that does not contain a Pod matching the rules.

There is a trade-off with these two types of pod anti-affinity: while "required" anti-affinity will ensure that all the matching Pods are scheduled on different Nodes, if Kubernetes cannot find an available Node, your Ivory instance may not be scheduled. Likewise, while "preferred" anti-affinity will make a best effort to scheduled your Pods on different Nodes, Kubernetes may compromise and schedule more than one Ivory instance of the same cluster on the same Node.

By understanding these trade-offs, the makeup of your Kubernetes cluster, and your requirements, you can choose the method that makes the most sense for your Ivory deployment. We'll show examples of both methods below!

Using Preferred Pod Anti-Affinity

First, let's deploy our Ivory cluster with preferred Pod anti-affinity. Note that if you have a single-node Kubernetes cluster, you will not see your Ivory instances deployed to different nodes. However, your Ivory instances will be deployed.

We can set up our HA Ivory cluster with preferred Pod anti-affinity like so:

```
apiVersion: ivory-operator.ivorysql.org/v1beta1
kind: IvoryCluster
metadata:
  name: hippo
spec:
```

```
  image: {{< param imageIvorySQL >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - name: instance1
      replicas: 2
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
      affinity:
        podAntiAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - weight: 1
            podAffinityTerm:
              topologyKey: kubernetes.io/hostname
              labelSelector:
                matchLabels:
                  ivory-operator.ivorysql.org/cluster: hippo
                  ivory-operator.ivorysql.org/instance-set: instance1
  backups:
    pgbackrest:
      image: {{< param imagePGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
```

Apply those changes in your Kubernetes cluster.

Let’s take a closer look at this section:

```
affinity:
  podAntiAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
    - weight: 1
      podAffinityTerm:
```

```
        topologyKey: kubernetes.io/hostname
        labelSelector:
          matchLabels:
            ivory-operator.ivorysql.org/cluster: hippo
            ivory-operator.ivorysql.org/instance-set: instance1
```

`spec.instances.affinity.podAntiAffinity` follows the standard Kubernetes Pod anti-affinity spec. The values for the `matchLabels` are derived from what we described in the previous section: `ivory-operator.ivorysql.org/cluster` is set to our cluster name of `hippo`, and `ivory-operator.ivorysql.org/instance-set` is set to the instance set name of `instance1`. We choose a `topologyKey` of `kubernetes.io/hostname`, which is standard in Kubernetes clusters.

Preferred Pod anti-affinity will perform a best effort to schedule your Ivory Pods to different nodes. Let's see how you can require your Ivory Pods to be scheduled to different nodes.

Using Required Pod Anti-Affinity

Required Pod anti-affinity forces Kubernetes to scheduled your Ivory Pods to different Nodes. Note that if Kubernetes is unable to schedule all Pods to different Nodes, some of your Ivory instances may become unavailable.

Using the previous example, let's indicate to Kubernetes that we want to use required Pod anti-affinity for our Ivory clusters:

```
apiVersion: ivory-operator.ivorysql.org/v1beta1
kind: IvoryCluster
metadata:
  name: hippo
spec:
  image: {{< param imageIvorySQL >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - name: instance1
      replicas: 2
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - topologyKey: kubernetes.io/hostname
            labelSelector:
              matchLabels:
                ivory-operator.ivorysql.org/cluster: hippo
                ivory-operator.ivorysql.org/instance-set: instance1
```

```
  backups:
    pgbackrest:
      image: {{< param imagePGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
```

Apply those changes in your Kubernetes cluster.

If you are in a single Node Kubernetes clusters, you will notice that not all of your Ivory instance Pods will be scheduled. This is due to the `requiredDuringSchedulingIgnoredDuringExecution` preference. However, if you have enough Nodes available, you will see the Ivory instance Pods scheduled to different Nodes:

```
kubectl get pods -n ivory-operator -o wide \
  --selector=ivory-operator.ivorysql.org/cluster=hippo,ivory-
operator.ivorysql.org/instance
```

Node Affinity

Node affinity can be used to assign your Ivory instances to Nodes with specific hardware or to guarantee a Ivory instance resides in a specific zone. Node affinity can be set within the `spec.instances.affinity.nodeAffinity` attribute, following the standard Kubernetes node affinity spec.

Let's see an example with required Node affinity. Let's say we have a set of Nodes that are reserved for database usage that have a label `workload-role=db`. We can create a Ivory cluster with a required Node affinity rule to scheduled all of the databases to those Nodes using the following configuration:

```
apiVersion: ivory-operator.ivorysql.org/v1beta1
kind: IvoryCluster
metadata:
  name: hippo
spec:
  image: {{< param imageIvorySQL >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - name: instance1
      replicas: 2
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
```

```
          requests:
            storage: 1Gi
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: workload-role
                operator: In
                values:
                - db
  backups:
    pgbackrest:
      image: {{< param imagePGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
```

Pod Topology Spread Constraints

In addition to affinity and anti-affinity settings, Kubernetes Pod Topology Spread Constraints can also help you to define where you want your workloads to reside. However, while PodAffinity allows any number of Pods to be added to a qualifying topology domain, and PodAntiAffinity allows only one Pod to be scheduled into a single topology domain, topology spread constraints allow you to distribute Pods across different topology domains with a finer level of control.

API Field Configuration

The spread constraint API fields can be configured for instance, PgBouncer and pgBackRest repo host pods. The basic configuration is as follows:

```
      topologySpreadConstraints:
      - maxSkew: <integer>
        topologyKey: <string>
        whenUnsatisfiable: <string>
        labelSelector: <object>
```

where "maxSkew" describes the maximum degree to which Pods can be unevenly distributed, "topologyKey" is the key that defines a topology in the Nodes' Labels, "whenUnsatisfiable" specifies what action should be taken when "maxSkew" can’ t be satisfied, and "labelSelector" is used to find matching Pods.

Example Spread Constraints

To help illustrate how you might use this with your cluster, we can review examples for configuring spread constraints on our Instance and pgBackRest repo host Pods. For this example, assume we have a three node Kubernetes cluster where the first node is labeled with `my-node-label=one`, the second node is labeled with `my-node-label=two` and the final node is labeled `my-node-label=three`. The label key `my-node-label` will function as our `topologyKey`. Note all three nodes in our examples will be schedulable, so a Pod could live on any of the three Nodes.

Instance Pod Spread Constraints

To begin, we can set our topology spread constraints on our cluster Instance Pods. Given this configuration

```
instances:
  - name: instance1
    replicas: 5
    topologySpreadConstraints:
      - maxSkew: 1
        topologyKey: my-node-label
        whenUnsatisfiable: DoNotSchedule
        labelSelector:
          matchLabels:
            ivory-operator.ivorysql.org/instance-set: instance1
```

we will expect 5 Instance pods to be created. Each of these Pods will have the standard `ivory-operator.ivorysql.org/instance-set: instance1` Label set, so each Pod will be properly counted when determining the `maxSkew`. Since we have 3 nodes with a `maxSkew` of 1 and we've set `whenUnsatisfiable` to `DoNotSchedule`, we should see 2 Pods on 2 of the nodes and 1 Pod on the remaining Node, thus ensuring our Pods are distributed as evenly as possible.

pgBackRest Repo Pod Spread Constraints

We can also set topology spread constraints on our cluster's pgBackRest repo host pod. While we normally will only have a single pod per cluster, we could use a more generic label to add a preference that repo host Pods from different clusters are distributed among our Nodes. For example, by setting our `matchLabel` value to `ivory-operator.ivorysql.org/pgbackrest: ""` and our `whenUnsatisfiable` value to `ScheduleAnyway`, we will allow our repo host Pods to be scheduled no matter what Nodes may be available, but attempt to minimize skew as much as possible.

```
    repoHost:
      topologySpreadConstraints:
      - maxSkew: 1
        topologyKey: my-node-label
        whenUnsatisfiable: ScheduleAnyway
        labelSelector:
          matchLabels:
            ivory-operator.ivorysql.org/pgbackrest: ""
```

Putting it All Together

Now that each of our Pods has our desired Topology Spread Constraints defined, let's put together a complete cluster definition:

```yaml
apiVersion: ivory-operator.ivorysql.org/v1beta1
kind: IvoryCluster
metadata:
  name: hippo
spec:
  image: {{< param imageIvorySQL >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - name: instance1
      replicas: 5
      topologySpreadConstraints:
        - maxSkew: 1
          topologyKey: my-node-label
          whenUnsatisfiable: DoNotSchedule
          labelSelector:
            matchLabels:
              ivory-operator.ivorysql.org/instance-set: instance1
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1G
  backups:
    pgbackrest:
      image: {{< param imagePGBackrest >}}
      repoHost:
        topologySpreadConstraints:
        - maxSkew: 1
          topologyKey: my-node-label
          whenUnsatisfiable: ScheduleAnyway
          labelSelector:
            matchLabels:
              ivory-operator.ivorysql.org/pgbackrest: ""
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
```

```
                storage: 1G
```

You can then apply those changes in your Kubernetes cluster.

Once your cluster finishes deploying, you can check that your Pods are assigned to the correct Nodes:

```
kubectl get pods -n ivory-operator -o wide --selector=ivory-
operator.ivorysql.org/cluster=hippo
```

Next Steps

We've now seen how IVYO helps your application stay "always on" with your Ivory database. Now let's explore how IVYO can minimize or eliminate downtime for operations that would normally cause that, such as resizing your Ivory cluster.

Resize an Ivory Cluster

You did it — the application is a success! Traffic is booming, so much so that you need to add more resources to your Ivory cluster. However, you're worried that any resize operation may cause downtime and create a poor experience for your end users.

This is where IVYO comes in: IVYO will help orchestrate rolling out any potentially disruptive changes to your cluster to minimize or eliminate downtime for your application. To do so, we will assume that you have deployed a high availability Ivory cluster as described in the previous section.

Let's dive in.

Resize Memory and CPU

Memory and CPU resources are an important component for vertically scaling your Ivory cluster. Coupled with tweaks to your Ivory configuration file, allocating more memory and CPU to your cluster can help it to perform better under load.

It's important for instances in the same high availability set to have the same resources. IVYO lets you adjust CPU and memory within the `resources` sections of the `ivoryclusters.ivory-operator.ivorysql.org` custom resource. These include:

- `spec.instances.resources` section, which sets the resource values for the IvorySQL container, as well as any init containers in the associated pod and containers created by the `pgDataVolume` and `pgWALVolume` data migration jobs.
- `spec.instances.sidecars.replicaCertCopy.resources` section, which sets the resources for the `replica-cert-copy` sidecar container.
- `spec.backups.pgbackrest.repoHost.resources` section, which sets the resources for the pgBackRest repo host container, as well as any init containers in the associated pod and containers created by the `pgBackRestVolume` data migration job.
- `spec.backups.pgbackrest.sidecars.pgbackrest.resources` section, which sets the resources for the `pgbackrest` sidecar container.
- `spec.backups.pgbackrest.sidecars.pgbackrestConfig.resources` section, which sets the resources for the `pgbackrest-config` sidecar container.
- `spec.backups.pgbackrest.jobs.resources` section, which sets the resources for any pgBackRest backup job.
- `spec.backups.pgbackrest.restore.resources` section, which sets the resources for manual pgBackRest restore jobs.
- `spec.dataSource.ivorycluster.resources` section, which sets the resources for pgBackRest restore jobs created during the cloning process.

The layout of these `resources` sections should be familiar: they follow the same pattern as the standard Kubernetes structure for setting container resources. Note that these settings also allow for the configuration of QoS classes.

For example, using the `spec.instances.resources` section, let's say we want to update our `hippo` Ivory cluster so that each instance has a limit of `2.0` CPUs and `4Gi` of memory. We can make the following changes to the manifest:

```
apiVersion: ivory-operator.ivorysql.org/v1beta1
kind: IvoryCluster
metadata:
  name: hippo
spec:
  image: {{< param imageIvorySQL >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - name: instance1
      replicas: 2
      resources:
        limits:
          cpu: 2.0
          memory: 4Gi
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      image: {{< param imagePGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
```

In particular, we added the following to `spec.instances`:

```
resources:
```

```
  limits:
    cpu: 2.0
    memory: 4Gi
```

Apply these updates to your Ivory cluster with the following command:

```
kubectl apply -k examples/kustomize/ivory
```

Now, let's watch how the rollout happens:

```
watch "kubectl -n ivory-operator get pods \
  --selector=ivory-operator.ivorysql.org/cluster=hippo,ivory
-operator.ivorysql.org/instance \
  -o=jsonpath='{range .items[*]}{.metadata.name}{\"\t\"}{.metadata.labels.ivory-
operator\.ivorysql\.org/role}{\"\t\"}{.status.phase}{\"\t\"}{.spec.containers[].resour
ces.limits}{\"\n\"}{end}'"
```

Observe how each Pod is terminated one-at-a-time. This is part of a "rolling update". Because updating the resources of a Pod is a destructive action, IVYO first applies the CPU and memory changes to the replicas. IVYO ensures that the changes are successfully applied to a replica instance before moving on to the next replica.

Once all of the changes are applied, IVYO will perform a "controlled switchover": it will promote a replica to become a primary, and apply the changes to the final Ivory instance.

By rolling out the changes in this way, IVYO ensures there is minimal to zero disruption to your application: you are able to successfully roll out updates and your users may not even notice!

Resize PVC

Your application is a success! Your data continues to grow, and it's becoming apparently that you need more disk. That's great: you can resize your PVC directly on your `ivoryclusters.ivory-operator.ivorysql.org` custom resource with minimal to zero downtime.

PVC resizing, also known as volume expansion, is a function of your storage class: it must support volume resizing. Additionally, PVCs can only be sized up: you cannot shrink the size of a PVC.

You can adjust PVC sizes on all of the managed storage instances in a Ivory instance that are using Kubernetes storage. These include:

- `spec.instances.dataVolumeClaimSpec.resources.requests.storage`: The Ivory data directory (aka your database).
- `spec.backups.pgbackrest.repos.volume.volumeClaimSpec.resources.requests.storage`: The pgBackRest repository when using "volume" storage

The above should be familiar: it follows the same pattern as the standard Kubernetes PVC structure.

For example, let's say we want to update our `hippo` Ivory cluster so that each instance now uses a `10Gi` PVC and our backup repository uses a `20Gi` PVC. We can do so with the following markup:

```
apiVersion: ivory-operator.ivorysql.org/v1beta1
kind: IvoryCluster
metadata:
```

```
  name: hippo
spec:
  image: {{< param imageIvorySQL >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - name: instance1
      replicas: 2
      resources:
        limits:
          cpu: 2.0
          memory: 4Gi
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 10Gi
  backups:
    pgbackrest:
      image: {{< param imagePGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 20Gi
```

In particular, we added the following to `spec.instances`:

```
dataVolumeClaimSpec:
  resources:
    requests:
      storage: 10Gi
```

and added the following to `spec.backups.pgbackrest.repos.volume`:

```
volumeClaimSpec:
  accessModes:
  - "ReadWriteOnce"
```

```
  resources:
    requests:
      storage: 20Gi
```

Apply these updates to your Ivory cluster with the following command:

```
kubectl apply -k examples/kustomize/ivory
```

Resize PVCs With StorageClass That Does Not Allow Expansion

Not all Kubernetes Storage Classes allow for volume expansion. However, with IVYO, you can still resize your Ivory cluster data volumes even if your storage class does not allow it!

Let’s go back to the previous example:

```
apiVersion: ivory-operator.ivorysql.org/v1beta1
kind: IvoryCluster
metadata:
  name: hippo
spec:
  image: {{< param imageIvorySQL >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - name: instance1
      replicas: 2
      resources:
        limits:
          cpu: 2.0
          memory: 4Gi
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      image: {{< param imagePGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
```

```
            resources:
              requests:
                storage: 20Gi
```

First, create a new instance that has the larger volume size. Call this instance `instance2`. The manifest would look like this:

```
apiVersion: ivory-operator.ivorysql.org/v1beta1
kind: IvoryCluster
metadata:
  name: hippo
spec:
  image: {{< param imageIvorySQL >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - name: instance1
      replicas: 2
      resources:
        limits:
          cpu: 2.0
          memory: 4Gi
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
    - name: instance2
      replicas: 2
      resources:
        limits:
          cpu: 2.0
          memory: 4Gi
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 10Gi
  backups:
    pgbackrest:
      image: {{< param imagePGBackrest >}}
      repos:
```

```
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 20Gi
```

Take note of the block that contains `instance2`:

```
- name: instance2
  replicas: 2
  resources:
    limits:
      cpu: 2.0
      memory: 4Gi
  dataVolumeClaimSpec:
    accessModes:
    - "ReadWriteOnce"
    resources:
      requests:
        storage: 10Gi
```

This creates a second set of two Ivory instances, both of which come up as replicas, that have a larger PVC.

Once this new instance set is available and they are caught to the primary, you can then apply the following manifest:

```
apiVersion: ivory-operator.ivorysql.org/v1beta1
kind: IvoryCluster
metadata:
  name: hippo
spec:
  image: {{< param imageIvorySQL >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - name: instance2
      replicas: 2
      resources:
        limits:
          cpu: 2.0
          memory: 4Gi
```

```
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 10Gi
  backups:
    pgbackrest:
      image: {{< param imagePGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 20Gi
```

This will promote one of the instances with the larger PVC to be the new primary and remove the instances with the smaller PVCs!

This method can also be used to shrink PVCs to use a smaller amount.

Troubleshooting

Ivory Pod Can't Be Scheduled

There are many reasons why a IvorySQL Pod may not be scheduled:

- Resources are unavailable. Ensure that you have a Kubernetes Node with enough resources to satisfy your memory or CPU Request.
- PVC cannot be provisioned. Ensure that you request a PVC size that is available, or that your PVC storage class is set up correctly.

PVCs Do Not Resize

Ensure that your storage class supports PVC resizing. You can check that by inspecting the `allowVolumeExpansion` attribute:

```
kubectl get sc
```

If the storage class does not support PVC resizing, you can use the technique described above to resize PVCs using a second instance set.

Next Steps

You've now resized your Ivory cluster, but how can you configure Ivory to take advantage of the new resources? Let's look at how we can customize the Ivory cluster configuration.

Part of the trick of managing multiple instances in an Ivory cluster is ensuring all of the configuration changes are propagated to each of them. This is where IVYO helps: when you make an Ivory configuration change for a cluster, IVYO will apply it to all of the Ivory instances.

For example, in our previous step we added CPU and memory limits of `2.0` and `4Gi` respectively. Let's tweak some of the Ivory settings to better use our new resources. We can do this in the `spec.patroni.dynamicConfiguration` section. Here is an example updated manifest that tweaks several settings:

```
apiVersion: ivory-operator.ivorysql.org/v1beta1
kind: IvoryCluster
metadata:
  name: hippo
spec:
  image: {{< param imageIvorySQL >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - name: instance1
      replicas: 2
      resources:
        limits:
          cpu: 2.0
          memory: 4Gi
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      image: {{< param imagePGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
  patroni:
    dynamicConfiguration:
```

```
      postgresql:
        parameters:
          max_parallel_workers: 2
          max_worker_processes: 2
          shared_buffers: 1GB
          work_mem: 2MB
```

In particular, we added the following to `spec`:

```
patroni:
  dynamicConfiguration:
    postgresql:
      parameters:
        max_parallel_workers: 2
        max_worker_processes: 2
        shared_buffers: 1GB
        work_mem: 2MB
```

Apply these updates to your Ivory cluster with the following command:

```
kubectl apply -k examples/kustomize/ivory
```

IVYO will go and apply these settings, restarting each Ivory instance when necessary. You can verify that the changes are present using the Ivory `SHOW` command, e.g.

```
SHOW work_mem;
```

should yield something similar to:

```
 work_mem
----------
 2MB
```

Customize TLS

All connections in IVYO use TLS to encrypt communication between components. IVYO sets up a PKI and certificate authority (CA) that allow you create verifiable endpoints. However, you may want to bring a different TLS infrastructure based upon your organizational requirements. The good news: IVYO lets you do this!

How to Customize TLS

There are a few different TLS endpoints that can be customized for IVYO, including those of the Ivory cluster and controlling how Ivory instances authenticate with each other. Let's look at how we can customize TLS by defining

- a `spec.customTLSSecret`, used to both identify the cluster and encrypt communications; and

- a `spec.customReplicationTLSSecret`, used for replication authentication.

To customize the TLS for an Ivory cluster, you will need to create two Secrets in the Namespace of your Ivory cluster. One of these Secrets will be the `customTLSSecret` and the other will be the `customReplicationTLSSecret`. Both secrets contain a TLS key (`tls.key`), TLS certificate (`tls.crt`) and CA certificate (`ca.crt`) to use.



If `spec.customTLSSecret` is provided you must also provide `spec.customReplicationTLSSecret` and both must contain the same `ca.crt`.

The custom TLS and custom replication TLS Secrets should contain the following fields (though see below for a workaround if you cannot control the field names of the Secret's `data`):

```
data:
  ca.crt: <value>
  tls.crt: <value>
  tls.key: <value>
```

For example, if you have files named `ca.crt`, `hippo.key`, and `hippo.crt` stored on your local machine, you could run the following command to create a Secret from those files:

```
kubectl create secret generic -n ivory-operator hippo-cluster.tls \
  --from-file=ca.crt=ca.crt \
  --from-file=tls.key=hippo.key \
  --from-file=tls.crt=hippo.crt
```

After you create the Secrets, you can specify the custom TLS Secret in your `ivorycluster.ivory-operator.ivorysql.org` custom resource. For example, if you created a `hippo-cluster.tls` Secret and a `hippo-replication.tls` Secret, you would add them to your Ivory cluster:

```
spec:
  customTLSSecret:
    name: hippo-cluster.tls
  customReplicationTLSSecret:
    name: hippo-replication.tls
```

If you're unable to control the key-value pairs in the Secret, you can create a mapping to tell the Ivory Operator what key holds the expected value. That would look similar to this:

```
spec:
  customTLSSecret:
    name: hippo.tls
    items:
      - key: <tls.crt key in the referenced hippo.tls Secret>
        path: tls.crt
      - key: <tls.key key in the referenced hippo.tls Secret>
        path: tls.key
```

```
      - key: <ca.crt key in the referenced hippo.tls Secret>
        path: ca.crt
```

For instance, if the `hippo.tls` Secret had the `tls.crt` in a key named `hippo-tls.crt`, the `tls.key` in a key named `hippo-tls.key`, and the `ca.crt` in a key named `hippo-ca.crt`, then your mapping would look like:

```
spec:
  customTLSSecret:
    name: hippo.tls
    items:
      - key: hippo-tls.crt
        path: tls.crt
      - key: hippo-tls.key
        path: tls.key
      - key: hippo-ca.crt
        path: ca.crt
```



Although the custom TLS and custom replication TLS Secrets share the same `ca.crt`, they do not share the same `tls.crt`:

- Your `spec.customTLSSecret` TLS certificate should have a Common Name (CN) setting that matches the primary Service name. This is the name of the cluster suffixed with `-primary`. For example, for our `hippo` cluster this would be `hippo-primary`.
- Your `spec.customReplicationTLSSecret` TLS certificate should have a Common Name (CN) setting that matches `_ivoryrepl`, which is the preset replication user.

As with the other changes, you can roll out the TLS customizations with `kubectl apply`.

Labels

There are several ways to add your own custom Kubernetes Labels to your Ivory cluster.

- Cluster: You can apply labels to any IVYO managed object in a cluster by editing the `spec.metadata.labels` section of the custom resource.
- Ivory: You can apply labels to an Ivory instance set and its objects by editing `spec.instances.metadata.labels`.
- pgBackRest: You can apply labels to pgBackRest and its objects by editing `ivoryclusters.spec.backups.pgbackrest.metadata.labels`.

Annotations

There are several ways to add your own custom Kubernetes Annotations to your Ivory cluster.

- Cluster: You can apply annotations to any IVYO managed object in a cluster by editing the `spec.metadata.annotations` section of the custom resource.
- Ivory: You can apply annotations to an Ivory instance set and its objects by editing `spec.instances.metadata.annotations`.
- pgBackRest: You can apply annotations to pgBackRest and its objects by editing `spec.backups.pgbackrest.metadata.annotations`.

Pod Priority Classes

IVYO allows you to use pod priority classes to indicate the relative importance of a pod by setting a `priorityClassName` field on your Ivory cluster. This can be done as follows:

- Instances: Priority is defined per instance set and is applied to all Pods in that instance set by editing the `spec.instances.priorityClassName` section of the custom resource.
- Dedicated Repo Host: Priority defined under the repoHost section of the spec is applied to the dedicated repo host by editing the `spec.backups.pgbackrest.repoHost.priorityClassName` section of the custom resource.
- Backup (manual and scheduled): Priority is defined under the `spec.backups.pgbackrest.jobs.priorityClassName` section and applies that priority to all pgBackRest backup Jobs (manual and scheduled).
- Restore (data source or in-place): Priority is defined for either a "data source" restore or an in-place restore by editing the `spec.dataSource.ivorycluster.priorityClassName` section of the custom resource.
- Data Migration: The priority defined for the first instance set in the spec (array position 0) is used for the PGDATA and WAL migration Jobs. The pgBackRest repo migration Job will use the priority class applied to the repoHost.

Separate WAL PVCs

IvorySQL commits transactions by storing changes in its Write-Ahead Log (WAL). Because the way WAL files are accessed and utilized often differs from that of data files, and in high-performance situations, it can desirable to put WAL files on separate storage volume. With IVYO, this can be done by adding the `walVolumeClaimSpec` block to your desired instance in your ivorycluster spec, either when your cluster is created or anytime thereafter:

```
spec:
  instances:
    - name: instance
      walVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
```

This volume can be removed later by removing the `walVolumeClaimSpec` section from the instance. Note that when changing the WAL directory, care is taken so as not to lose any WAL files. IVYO only deletes the PVC once there are no longer any WAL files on the previously configured volume.

Database Initialization SQL

IVYO can run SQL for you as part of the cluster creation and initialization process. IVYO runs the SQL using the psql client so you can use meta-commands to connect to different databases, change error handling, or set and use variables. Its capabilities are described in the psql documentation.

Initialization SQL ConfigMap

The Ivory cluster spec accepts a reference to a ConfigMap containing your init SQL file. Update your cluster spec to include the ConfigMap name, `spec.databaseInitSQL.name`, and the data key, `spec.databaseInitSQL.key`, for your SQL file. For example, if you create your ConfigMap with the following command:

```
kubectl -n ivory-operator create configmap hippo-init-sql --from-file=init.sql
=/path/to/init.sql
```

You would add the following section to your ivorycluster spec:

```
spec:
  databaseInitSQL:
    key: init.sql
    name: hippo-init-sql
```



The ConfigMap must exist in the same namespace as your Ivory cluster.

After you add the ConfigMap reference to your spec, apply the change with `kubectl apply -k examples/kustomize/ivory`. IVYO will create your `hippo` cluster and run your initialization SQL once the cluster has started. You can verify that your SQL has been run by checking the `databaseInitSQL` status on your Ivory cluster. While the status is set, your init SQL will not be run again. You can check cluster status with the `kubectl describe` command:

```
kubectl -n ivory-operator describe ivoryclusters.ivory-operator.ivorysql.org hippo
```



In some cases, due to how Kubernetes treats ivorycluster status, IVYO may run your SQL commands more than once. Please ensure that the commands defined in your init SQL are idempotent.

Now that `databaseInitSQL` is defined in your cluster status, verify database objects have been created as expected. After verifying, we recommend removing the `spec.databaseInitSQL` field from your spec. Removing the field from the spec will also remove `databaseInitSQL` from the cluster status.

PSQL Usage

IVYO uses the psql interactive terminal to execute SQL statements in your database. Statements are passed in using standard input and the filename flag (e.g. `psql -f -`).

SQL statements are executed as superuser in the default maintenance database. This means you have full control to create database objects, extensions, or run any SQL statements that you might need.

Integration with User and Database Management

If you are creating users or databases, please see the User/Database Management documentation. Databases created through the user management section of the spec can be referenced in your initialization sql. For example, if a database `zoo` is defined:

```
spec:
  users:
    - name: hippo
      databases:
       - "zoo"
```

You can connect to `zoo` by adding the following `psql` meta-command to your SQL:

```
\c zoo
create table t_zoo as select s, md5(random()::text) from generate_Series(1,5) s;
```

Transaction support

By default, `psql` commits each SQL command as it completes. To combine multiple commands into a single transaction, use the `BEGIN` and `COMMIT` commands.

```
BEGIN;
create table t_random as select s, md5(random()::text) from generate_Series(1,5) s;
COMMIT;
```

PSQL Exit Code and Database Init SQL Status

The exit code from `psql` will determine when the `databaseInitSQL` status is set. When `psql` returns `0` the status will be set and SQL will not be run again. When `psql` returns with an error exit code the status will not be set. IVYO will continue attempting to execute the SQL as part of its reconcile loop until `psql` returns normally. If `psql` exits with a failure, you will need to edit the file in your ConfigMap to ensure your SQL statements will lead to a successful `psql` return. The easiest way to make live changes to your ConfigMap is to use the following `kubectl edit` command:

```
kubectl -n <cluster-namespace> edit configmap hippo-init-sql
```

Be sure to transfer any changes back over to your local file. Another option is to make changes in your local file and use `kubectl --dry-run` to create a template and pipe the output into `kubectl apply`:

```
kubectl create configmap hippo-init-sql --from-file=init.sql=/path/to/init.sql --dry
-run=client -o yaml | kubectl apply -f -
```



If you edit your ConfigMap and your changes aren't showing up, you may be waiting for IVYO to reconcile your cluster. After some time, IVYO will automatically reconcile the cluster or you can trigger reconciliation by applying any change to your cluster (e.g. with `kubectl apply -k examples/kustomize/ivory`).

To ensure that `psql` returns a failure exit code when your SQL commands fail, set the `ON_ERROR_STOP` variable as part of your SQL file:

```
\set ON_ERROR_STOP
\echo Any error will lead to exit code 3
create table t_random as select s, md5(random()::text) from generate_Series(1,5) s;
```

Troubleshooting

Changes Not Applied

If your Ivory configuration settings are not present, ensure that you are using the syntax that Ivory expects. You can see this in the Ivory configuration documentation.

Next Steps

You've now seen how you can further customize your Ivory cluster, but what about managing users and databases? That's a great question that is answered in the next section.

User / Database Management

IVYO comes with some out-of-the-box conveniences for managing users and databases in your Ivory cluster. However, you may have requirements where you need to create additional users, adjust user privileges or add additional databases to your cluster.

For detailed information for how user and database management works in IVYO, please see the User Management section of the architecture guide.

Creating a New User

You can create a new user with the following snippet in the `ivorycluster` custom resource. Let's add this to our `hippo` database:

```
spec:
  users:
    - name: rhino
```

You can now apply the changes and see that the new user is created. Note the following:

- The user would only be able to connect to the default `ivory` database.
- The user will not have any connection credentials populated into the `hippo-pguser-rhino` Secret.
- The user is unprivileged.

Let's create a new database named `zoo` that we will let the `rhino` user access:

```
spec:
  users:
    - name: rhino
      databases:
        - zoo
```

Inspect the `hippo-pguser-rhino` Secret. You should now see that the `dbname` and `uri` fields are now populated!

We can set role privileges by using the standard role attributes that Ivory provides and adding them to the `spec.users.options`. Let's say we want the rhino to become a superuser (be careful about doling out Ivory superuser privileges!). You can add the following to the spec:

```
spec:
  users:
    - name: rhino
      databases:
        - zoo
      options: "SUPERUSER"
```

There you have it: we have created a Ivory user named `rhino` with superuser privileges that has access to the `rhino` database (though a superuser has access to all databases!).

Adjusting Privileges

Let's say you want to revoke the superuser privilege from `rhino`. You can do so with the following:

```
spec:
  users:
    - name: rhino
      databases:
        - zoo
      options: "NOSUPERUSER"
```

If you want to add multiple privileges, you can add each privilege with a space between them in `options`, e.g.:

```
spec:
  users:
    - name: rhino
      databases:
        - zoo
      options: "CREATEDB CREATEROLE"
```

Managing the `ivory` User

By default, IVYO does not give you access to the `ivory` user. However, you can get access to this account by doing the following:

```
spec:
  users:
    - name: ivory
```

This will create a Secret of the pattern `<clusterName>-pguser-ivory` that contains the credentials of the `ivory` account. For our `hippo` cluster, this would be `hippo-pguser-ivory`.

Deleting a User

IVYO does not delete users automatically: after you remove the user from the spec, it will still exist in your cluster. To remove a user and all of its objects, as a superuser you will need to run `DROP OWNED` in each database the user has objects in, and `DROP ROLE` in your Ivory cluster.

For example, with the above `rhino` user, you would run the following:

```
DROP OWNED BY rhino;
DROP ROLE rhino;
```

Note that you may need to run `DROP OWNED BY rhino CASCADE;` based upon your object ownership structure — be very careful with this command!

Deleting a Database

IVYO does not delete databases automatically: after you remove all instances of the database from the spec, it will still exist in your cluster. To completely remove the database, you must run the `DROP DATABASE` command as a Ivory superuser.

For example, to remove the `zoo` database, you would execute the following:

```
DROP DATABASE zoo;
```

Next Steps

Let's look at how IVYO handles disaster recovery!

Disaster Recovery and Cloning

Perhaps someone accidentally dropped the `users` table. Perhaps you want to clone your production database to a step-down environment. Perhaps you want to exercise your disaster recovery system (and it is important that you do!).

Regardless of scenario, it's important to know how you can perform a "restore" operation with IVYO to be able to recovery your data from a particular point in time, or clone a database for other purposes.

Let's look at how we can perform different types of restore operations. First, let's understand the core restore properties on the custom resource.

Restore Properties



IVYO offers the ability to restore from an existing ivorycluster or a remote cloud-based data source, such as S3, GCS, etc. For more on that, see the Clone From Backups Stored in S3 / GCS / Azure Blob Storage section.

Note that you cannot use both a local ivorycluster data source and a remote cloud-based data source at one time; if both the `dataSource.ivorycluster` and `dataSource.pgbackrest` fields are filled in, the local ivorycluster data source will take precedence.

There are several attributes on the custom resource that are important to understand as part of the restore process. All of these attributes are grouped together in the spec.dataSource.ivorycluster section of the custom resource.

Please review the table below to understand how each of these attributes work in the context of setting up a restore operation.

- `spec.dataSource.ivorycluster.clusterName`: The name of the cluster that you are restoring from. This corresponds to the `metadata.name` attribute on a different `ivorycluster` custom resource.
- `spec.dataSource.ivorycluster.clusterNamespace`: The namespace of the cluster that you are restoring from. Used when the cluster exists in a different namespace.
- `spec.dataSource.ivorycluster.repoName`: The name of the pgBackRest repository from the `spec.dataSource.ivorycluster.clusterName` to use for the restore. Can be one of `repo1`, `repo2`, `repo3`, or `repo4`. The repository must exist in the other cluster.
- `spec.dataSource.ivorycluster.options`: Any additional pgBackRest restore options or general options that IVYO allows. For example, you may want to set `--process-max` to help improve performance on larger databases; but you will not be able to set `--target-action`, since that option is currently disallowed. (IVYO always sets it to `promote` if a `--target` is present, and otherwise leaves it blank.)
- `spec.dataSource.ivorycluster.resources`: Setting resource limits and requests of the restore job can ensure that it runs efficiently.

- `spec.dataSource.ivorycluster.affinity`: Custom Kubernetes affinity rules constrain the restore job so that it only runs on certain nodes.
- `spec.dataSource.ivorycluster.tolerations`: Custom Kubernetes tolerations allow the restore job to run on tainted nodes.

Let's walk through some examples for how we can clone and restore our databases.

Clone a Ivory Cluster

Let's create a clone of our `hippo` cluster that we created previously. We know that our cluster is named `hippo` (based on its `metadata.name`) and that we only have a single backup repository called `repo1`.

Let's call our new cluster `elephant`. We can create a clone of the `hippo` cluster using a manifest like this:

```
apiVersion: ivory-operator.ivorysql.org/v1beta1
kind: IvoryCluster
metadata:
  name: elephant
spec:
  dataSource:
    ivoryCluster:
      clusterName: hippo
      repoName: repo1
  image: {{< param imageIvorySQL >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      image: {{< param imagePGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
```

Note this section of the spec:

```
spec:
  dataSource:
    ivoryCluster:
      clusterName: hippo
      repoName: repo1
```

This is the part that tells IVYO to create the `elephant` cluster as an independent copy of the `hippo` cluster.

The above is all you need to do to clone a Ivory cluster! IVYO will work on creating a copy of your data on a new persistent volume claim (PVC) and work on initializing your cluster to spec. Easy!

Perform a Point-in-time-Recovery (PITR)

Did someone drop the user table? You may want to perform a point-in-time-recovery (PITR) to revert your database back to a state before a change occurred. Fortunately, IVYO can help you do that.

You can set up a PITR using the restore command of pgBackRest, the backup management tool that powers the disaster recovery capabilities of IVYO. You will need to set a few options on `spec.dataSource.ivorycluster.options` to perform a PITR. These options include:

- `--type=time`: This tells pgBackRest to perform a PITR.
- `--target`: Where to perform the PITR to. An example recovery target is `2021-06-09 14:15:11-04`. The timezone specified here as -04 for EDT. Please see the pgBackRest documentation for other timezone options.
- `--set` (optional): Choose which backup to start the PITR from.

A few quick notes before we begin:

- To perform a PITR, you must have a backup that finished before your PITR time. In other words, you can't perform a PITR back to a time where you do not have a backup!
- All relevant WAL files must be successfully pushed for the restore to complete correctly.
- Be sure to select the correct repository name containing the desired backup!

With that in mind, let's use the `elephant` example above. Let's say we want to perform a point-in-time-recovery (PITR) to `2021-06-09 14:15:11-04`, we can use the following manifest:

```
apiVersion: ivory-operator.ivorysql.org/v1beta1
kind: IvoryCluster
metadata:
  name: elephant
spec:
  dataSource:
    ivoryCluster:
      clusterName: hippo
      repoName: repo1
      options:
      - --type=time
      - --target="2021-06-09 14:15:11-04"
  image: {{< param imageIvorySQL >}}
```

```
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      image: {{< param imagePGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
```

The section to pay attention to is this:

```
spec:
  dataSource:
    ivoryCluster:
      clusterName: hippo
      repoName: repo1
      options:
      - --type=time
      - --target="2021-06-09 14:15:11-04"
```

Notice how we put in the options to specify where to make the PITR.

Using the above manifest, IVYO will go ahead and create a new Ivory cluster that recovers its data up until `2021-06-09 14:15:11-04`. At that point, the cluster is promoted and you can start accessing your database from that specific point in time!

Perform an In-Place Point-in-time-Recovery (PITR)

Similar to the PITR restore described above, you may want to perform a similar reversion back to a state before a change occurred, but without creating another IvorySQL cluster. Fortunately, IVYO can help you do this as well.

You can set up a PITR using the restore command of pgBackRest, the backup management tool that powers the disaster recovery capabilities of IVYO. You will need to set a few options on `spec.backups.pgbackrest.restore.options` to perform a PITR. These options include:

- `--type=time`: This tells pgBackRest to perform a PITR.
- `--target`: Where to perform the PITR to. An example recovery target is `2021-06-09 14:15:11-04`.
- `--set` (optional): Choose which backup to start the PITR from.

A few quick notes before we begin:

- To perform a PITR, you must have a backup that finished before your PITR time. In other words, you can't perform a PITR back to a time where you do not have a backup!
- All relevant WAL files must be successfully pushed for the restore to complete correctly.
- Be sure to select the correct repository name containing the desired backup!

To perform an in-place restore, users will first fill out the restore section of the spec as follows:

```
spec:
  backups:
    pgbackrest:
      restore:
        enabled: true
        repoName: repo1
        options:
        - --type=time
        - --target="2021-06-09 14:15:11-04"
```

And to trigger the restore, you will then annotate the ivorycluster as follows:

```
kubectl annotate -n ivory-operator ivorycluster hippo --overwrite \
  ivory-operator.ivorysql.org/pgbackrest-restore=id1
```

And once the restore is complete, in-place restores can be disabled:

```
spec:
  backups:
    pgbackrest:
      restore:
        enabled: false
```

Notice how we put in the options to specify where to make the PITR.

Using the above manifest, IVYO will go ahead and re-create your Ivory cluster to recover its data up until `2021-06-09 14:15:11-04`. At that point, the cluster is promoted and you can start accessing your database from that specific point in time!

Restore Individual Databases

You might need to restore specific databases from a cluster backup, for performance reasons or to move selected databases to a machine that does not have enough space to restore the entire cluster backup.



pgBackRest supports this case, but it is important to make sure this is what you want. Restoring in this manner will restore the requested database from backup and make it

> accessible, but all of the other databases in the backup will NOT be accessible after restore.
>
> For example, if your backup includes databases `test1`, `test2`, and `test3`, and you request that `test2` be restored, the `test1` and `test3` databases will NOT be accessible after restore is completed. Please review the pgBackRest documentation on the limitations on restoring individual databases.

You can restore individual databases from a backup using a spec similar to the following:

```
spec:
  backups:
    pgbackrest:
      restore:
        enabled: true
        repoName: repo1
        options:
        - --db-include=hippo
```

where `--db-include=hippo` would restore only the contents of the `hippo` database.

Standby Cluster

Advanced high-availability and disaster recovery strategies involve spreading your database clusters across data centers to help maximize uptime. IVYO provides ways to deploy ivoryclusters that can span multiple Kubernetes clusters using an external storage system or IvorySQL streaming replication. The disaster recovery architecture documentation provides a high-level overview of standby clusters with IVYO can be found in the disaster recovery architecture documentation.

Creating a standby Cluster

This tutorial section will describe how to create three different types of standby clusters, one using an external storage system, one that is streaming data directly from the primary, and one that takes advantage of both external storage and streaming. These example clusters can be created in the same Kubernetes cluster, using a single IVYO instance, or spread across different Kubernetes clusters and IVYO instances with the correct storage and networking configurations.

Repo-based Standby

A repo-based standby will recover from WAL files a pgBackRest repo stored in external storage. The primary cluster should be created with a cloud-based backup configuration. The following manifest defines a ivorycluster with `standby.enabled` set to true and `repoName` configured to point to the `s3` repo configured in the primary:

```
apiVersion: ivory-operator.ivorysql.org/v1beta1
kind: IvoryCluster
metadata:
  name: hippo-standby
spec:
  image: {{< param imageIvorySQL >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
```

```
    - dataVolumeClaimSpec: { accessModes: [ReadWriteOnce], resources: { requests: {
storage: 1Gi } } }
  backups:
    pgbackrest:
      image: {{< param imagePGBackrest >}}
      repos:
      - name: repo1
        s3:
          bucket: "my-bucket"
          endpoint: "s3.ca-central-1.amazonaws.com"
          region: "ca-central-1"
  standby:
    enabled: true
    repoName: repo1
```

Streaming Standby

A streaming standby relies on an authenticated connection to the primary over the network. The primary cluster should be accessible via the network and allow TLS authentication (TLS is enabled by default). In the following manifest, we have `standby.enabled` set to `true` and have provided both the `host` and `port` that point to the primary cluster. We have also defined `customTLSSecret` and `customReplicationTLSSecret` to provide certs that allow the standby to authenticate to the primary. For this type of standby, you must use custom TLS:

```
apiVersion: ivory-operator.ivorysql.org/v1beta1
kind: IvoryCluster
metadata:
  name: hippo-standby
spec:
  image: {{< param imageIvorySQL >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - dataVolumeClaimSpec: { accessModes: [ReadWriteOnce], resources: { requests: {
storage: 1Gi } } }
  backups:
    pgbackrest:
      repos:
      - name: repo1
        volume:
          volumeClaimSpec: { accessModes: [ReadWriteOnce], resources: { requests: {
storage: 1Gi } } }
  customTLSSecret:
    name: cluster-cert
  customReplicationTLSSecret:
    name: replication-cert
```

```
  standby:
    enabled: true
    host: "192.0.2.2"
    port: 5432
```

Streaming Standby with an External Repo

Another option is to create a standby cluster using an external pgBackRest repo that streams from the primary. With this setup, the standby cluster will continue recovering from the pgBackRest repo if streaming replication falls behind. In this manifest, we have enabled the settings from both previous examples:

```
apiVersion: ivory-operator.ivorysql.org/v1beta1
kind: IvoryCluster
metadata:
  name: hippo-standby
spec:
  image: {{< param imageIvorySQL >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - dataVolumeClaimSpec: { accessModes: [ReadWriteOnce], resources: { requests: {
storage: 1Gi } } }
  backups:
    pgbackrest:
      image: {{< param imagePGBackrest >}}
      repos:
      - name: repo1
        s3:
          bucket: "my-bucket"
          endpoint: "s3.ca-central-1.amazonaws.com"
          region: "ca-central-1"
  customTLSSecret:
    name: cluster-cert
  customReplicationTLSSecret:
    name: replication-cert
  standby:
    enabled: true
    repoName: repo1
    host: "192.0.2.2"
    port: 5432
```

Promoting a Standby Cluster

At some point, you will want to promote the standby to start accepting both reads and writes. This has the net effect of pushing WAL (transaction archives) to the pgBackRest repository, so we need to ensure we don’t accidentally create a split-brain scenario. Split-brain can happen if two primary instances attempt to write to the same repository. If the primary cluster is still active, make sure you shutdown the primary before

trying to promote the standby cluster.

Once the primary is inactive, we can promote the standby cluster by removing or disabling its `spec.standby` section:

```
spec:
  standby:
    enabled: false
```

This change triggers the promotion of the standby leader to a primary IvorySQL instance and the cluster begins accepting writes.

Clone From Backups Stored in S3 / GCS / Azure Blob Storage

You can clone a Ivory cluster from backups that are stored in AWS S3 (or a storage system that uses the S3 protocol), GCS, or Azure Blob Storage without needing an active Ivory cluster! The method to do so is similar to how you clone from an existing ivorycluster. This is useful if you want to have a data set for people to use but keep it compressed on cheaper storage.

For the purposes of this example, let's say that you created a Ivory cluster named `hippo` that has its backups stored in S3 that looks similar to this:

```
apiVersion: ivory-operator.ivorysql.org/v1beta1
kind: IvoryCluster
metadata:
  name: hippo
spec:
  image: {{< param imageIvorySQL >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      image: {{< param imagePGBackrest >}}
      configuration:
      - secret:
          name: ivyo-s3-creds
      global:
        repo1-path: /pgbackrest/ivory-operator/hippo/repo1
      manual:
        repoName: repo1
        options:
```

```
        - --type=full
      repos:
      - name: repo1
        s3:
          bucket: "my-bucket"
          endpoint: "s3.ca-central-1.amazonaws.com"
          region: "ca-central-1"
```

Ensure that the credentials in `ivyo-s3-creds` match your S3 credentials. For more details on deploying a Ivory cluster using S3 for backups, please see the Backups section of the tutorial.

For optimal performance when creating a new cluster from an active cluster, ensure that you take a recent full backup of the previous cluster. The above manifest is set up to take a full backup. Assuming `hippo` is created in the `ivory-operator` namespace, you can trigger a full backup with the following command:

```
kubectl annotate -n ivory-operator ivorycluster hippo --overwrite \
  ivory-operator.ivorysql.org/pgbackrest-backup="$( date '+%F_%H:%M:%S' )"
```

Wait for the backup to complete. Once this is done, you can delete the Ivory cluster.

Now, let's clone the data from the `hippo` backup into a new cluster called `elephant`. You can use a manifest similar to this:

```
apiVersion: ivory-operator.ivorysql.org/v1beta1
kind: IvoryCluster
metadata:
  name: elephant
spec:
  image: {{< param imageIvorySQL >}}
  postgresVersion: {{< param postgresVersion >}}
  dataSource:
    pgbackrest:
      stanza: db
      configuration:
      - secret:
          name: ivyo-s3-creds
      global:
        repo1-path: /pgbackrest/ivory-operator/hippo/repo1
      repo:
        name: repo1
        s3:
          bucket: "my-bucket"
          endpoint: "s3.ca-central-1.amazonaws.com"
          region: "ca-central-1"
  instances:
```

```
    - dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      image: {{< param imagePGBackrest >}}
      configuration:
      - secret:
          name: ivyo-s3-creds
      global:
        repo1-path: /pgbackrest/ivory-operator/elephant/repo1
      repos:
      - name: repo1
        s3:
          bucket: "my-bucket"
          endpoint: "s3.ca-central-1.amazonaws.com"
          region: "ca-central-1"
```

There are a few things to note in this manifest. First, note that the `spec.dataSource.pgbackrest` object in our new ivorycluster is very similar but slightly different from the old ivorycluster's `spec.backups.pgbackrest` object. The key differences are:

1. No image is necessary when restoring from a cloud-based data source
2. `stanza` is a required field when restoring from a cloud-based data source
3. `backups.pgbackrest` has a `repos` field, which is an array
4. `dataSource.pgbackrest` has a `repo` field, which is a single object

Note also the similarities:

1. We are reusing the secret for both (because the new restore pod needs to have the same credentials as the original backup pod)
2. The `repo` object is the same
3. The `global` object is the same

This is because the new restore pod for the `elephant` ivorycluster will need to reuse the configuration and credentials that were originally used in setting up the `hippo` ivorycluster.

In this example, we are creating a new cluster which is also backing up to the same S3 bucket; only the `spec.backups.pgbackrest.global` field has changed to point to a different path. This will ensure that the new `elephant` cluster will be pre-populated with the data from `hippo's backups, but will backup to its own folders, ensuring that the original backup repository is appropriately preserved.

Deploy this manifest to create the `elephant` Ivory cluster. Observe that it comes up and running:

```
kubectl -n ivory-operator describe ivorycluster elephant
```

When it is ready, you will see that the number of expected instances matches the number of ready instances, e.g.:

```
Instances:
  Name:               00
  Ready Replicas:     1
  Replicas:           1
  Updated Replicas:   1
```

The previous example shows how to use an existing S3 repository to pre-populate a ivorycluster while using a new S3 repository for backing up. But ivoryclusters that use cloud-based data sources can also use local repositories.

For example, assuming a ivorycluster called `rhino` that was meant to pre-populate from the original `hippo` ivorycluster, the manifest would look like this:

```
apiVersion: ivory-operator.ivorysql.org/v1beta1
kind: IvoryCluster
metadata:
  name: rhino
spec:
  image: {{< param imageIvorySQL >}}
  postgresVersion: {{< param postgresVersion >}}
  dataSource:
    pgbackrest:
      stanza: db
      configuration:
      - secret:
          name: ivyo-s3-creds
      global:
        repo1-path: /pgbackrest/ivory-operator/hippo/repo1
      repo:
        name: repo1
        s3:
          bucket: "my-bucket"
          endpoint: "s3.ca-central-1.amazonaws.com"
          region: "ca-central-1"
  instances:
    - dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
```

```
  backups:
    pgbackrest:
      image: {{< param imagePGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
```

Next Steps

Now we’ ve seen how to clone a cluster and perform a point-in-time-recovery, let’ s see how we can monitor our Ivory cluster to detect and prevent issues from occurring.

Monitoring

While having high availability and disaster recovery systems in place helps in the event of something going wrong with your IvorySQL cluster, monitoring helps you anticipate problems before they happen. Additionally, monitoring can help you diagnose and resolve issues that may cause degraded performance rather than downtime.

Let’ s look at how IVYO allows you to enable monitoring in your cluster.

Adding the Exporter Sidecar

Let’ s look at how we can add the IvorySQL Exporter sidecar to your cluster using the `kustomize/ivory` example in the Postgres Operator examples repository.

Monitoring tools are added using the `spec.monitoring` section of the custom resource. Currently, the only monitoring tool supported is the IvorySQL Exporter configured with pgMonitor.

In the `kustomize/ivory/ivory.yaml` file, add the following YAML to the spec:

```
monitoring:
  pgmonitor:
    exporter:
      image: {{< param imagePostgresExporter >}}
```

Save your changes and run:

```
kubectl apply -k kustomize/ivory
```

IVYO will detect the change and add the Exporter sidecar to all Ivory Pods that exist in your cluster. IVYO will also do the work to allow the Exporter to connect to the database and gather metrics that can be accessed using the IVYO Monitoring stack.

Configuring TLS Encryption for the Exporter

IVYO allows you to configure the exporter sidecar to use TLS encryption. If you provide a custom TLS Secret via the exporter spec:

```
  monitoring:
    pgmonitor:
      exporter:
        customTLSSecret:
          name: hippo.tls
```

Like other custom TLS Secrets that can be configured with IVYO, the Secret will need to be created in the same Namespace as your PostgresCluster. It should also contain the TLS key (`tls.key`) and TLS certificate (`tls.crt`) needed to enable encryption.

```
data:
  tls.crt: <value>
  tls.key: <value>
```

After you configure TLS for the exporter, you will need to update your Prometheus deployment to use TLS, and your connection to the exporter will be encrypted. Check out the Prometheus documentation for more information on configuring TLS for Prometheus.

Accessing the Metrics

Once the IvorySQL Exporter has been enabled in your cluster, follow the steps outlined in IVYO Monitoring to install the monitoring stack. This will allow you to deploy a pgMonitor configuration of Prometheus, Grafana, and Alertmanager monitoring tools in Kubernetes. These tools will be set up by default to connect to the Exporter containers on your Ivory Pods.

Configurate Monitoring

While the default Kustomize install should work in most Kubernetes environments, it may be necessary to further customize the project according to your specific needs.

For instance, by default `fsGroup` is set to `26` for the `securityContext` defined for the various Deployments comprising the IVYO Monitoring stack:

```
securityContext:
  fsGroup: 26
```

In most Kubernetes environments this setting is needed to ensure processes within the container have the permissions needed to write to any volumes mounted to each of the Pods comprising the IVYO Monitoring stack. However, when installing in an OpenShift environment (and more specifically when using the `restricted` Security Context Constraint), the `fsGroup` setting should be removed since OpenShift will automatically handle setting the proper `fsGroup` within the Pod’ s `securityContext`.

Additionally, within this same section it may also be necessary to modify the `supplmentalGroups` setting according to your specific storage configuration:

```
securityContext:
  supplementalGroups : 65534
```

Therefore, the following files (located under `kustomize/monitoring`) should be modified and/or patched (e.g. using additional overlays) as needed to ensure the `securityContext` is properly defined for your Kubernetes environment:

- `deploy-alertmanager.yaml`
- `deploy-grafana.yaml`
- `deploy-prometheus.yaml`

And to modify the configuration for the various storage resources (i.e. PersistentVolumeClaims) created by the IVYO Monitoring installer, the `kustomize/monitoring/pvcs.yaml` file can also be modified.

Additionally, it is also possible to further customize the configuration for the various components comprising the IVYO Monitoring stack (Grafana, Prometheus and/or AlertManager) by modifying the following configuration resources:

- `alertmanager-config.yaml`
- `alertmanager-rules-config.yaml`
- `grafana-datasources.yaml`
- `prometheus-config.yaml`

Finally, please note that the default username and password for Grafana can be updated by modifying the Grafana Secret in file `kustomize/monitoring/grafana-secret.yaml`.

Install

Once the Kustomize project has been modified according to your specific needs, IVYO Monitoring can then be installed using `kubectl` and Kustomize:

```
kubectl apply -k kustomize/monitoring
```

Uninstall

And similarly, once IVYO Monitoring has been installed, it can uninstalled using `kubectl` and Kustomize:

```
kubectl delete -k kustomize/monitoring
```

Next Steps

Now that we can monitor our cluster, let's explore how connection pooling can be enabled using IVYO and how it is helpful.

Connection Pooling

Connection pooling can be helpful for scaling and maintaining overall availability between your application and the database. IVYO helps facilitate this by supporting the PgBouncer connection pooler and state manager.

Let's look at how we can a connection pooler and connect it to our application!

Adding a Connection Pooler

Let's look at how we can add a connection pooler using the `kustomize/keycloak` example in the Ivory Operator repository examples folder.

Connection poolers are added using the `spec.proxy` section of the custom resource. Currently, the only connection pooler supported is PgBouncer.

The only required attribute for adding a PgBouncer connection pooler is to set the `spec.proxy.pgBouncer.image` attribute. In the `kustomize/keycloak/ivory.yaml` file, add the following YAML to the spec:

```
proxy:
  pgBouncer:
    image: {{< param imageIvoryPGBouncer >}}
```

(You can also find an example of this in the `kustomize/examples/high-availability` example).

Save your changes and run:

```
kubectl apply -k kustomize/keycloak
```

IVYO will detect the change and create a new PgBouncer Deployment!

That was fairly easy to set up, so now let's look at how we can connect our application to the connection pooler.

Connecting to a Connection Pooler

When a connection pooler is deployed to the cluster, IVYO adds additional information to the user Secrets to allow for applications to connect directly to the connection pooler. Recall that in this example, our user Secret is called `keycloakdb-pguser-keycloakdb`. Describe the user Secret:

```
kubectl -n ivory-operator describe secrets keycloakdb-pguser-keycloakdb
```

You should see that there are several new attributes included in this Secret that allow for you to connect to your Ivory instance via the connection pooler:

- `pgbouncer-host`: The name of the host of the PgBouncer connection pooler. This references the Service of the PgBouncer connection pooler.
- `pgbouncer-port`: The port that the PgBouncer connection pooler is listening on.
- `pgbouncer-uri`: A PostgreSQL connection URI that provides all the information for logging into the Ivory database via the PgBouncer connection pooler.
- `pgbouncer-jdbc-uri`: A PostgreSQL JDBC connection URI that provides all the information for logging into the Ivory database via the PgBouncer connection pooler using the JDBC driver. Note that by default, the connection string disable JDBC managing prepared transactions for optimal use with PgBouncer.

Open up the file in `kustomize/keycloak/keycloak.yaml`. Update the `DB_ADDR` and `DB_PORT` values to be the following:

```
- name: DB_ADDR
  valueFrom: { secretKeyRef: { name: keycloakdb-pguser-keycloakdb, key: pgbouncer-host
} }
- name: DB_PORT
  valueFrom: { secretKeyRef: { name: keycloakdb-pguser-keycloakdb, key: pgbouncer-port
} }
```

This changes Keycloak's configuration so that it will now connect through the connection pooler.

Apply the changes:

```
kubectl apply -k kustomize/keycloak
```

Kubernetes will detect the changes and begin to deploy a new Keycloak Pod. When it is completed, Keycloak will now be connected to Ivory via the PgBouncer connection pooler!

TLS

IVYO deploys every cluster and component over TLS. This includes the PgBouncer connection pooler. If you are using your own custom TLS setup, you will need to provide a Secret reference for a TLS key / certificate pair for PgBouncer in `spec.proxy.pgBouncer.customTLSSecret`.

Your TLS certificate for PgBouncer should have a Common Name (CN) setting that matches the PgBouncer Service name. This is the name of the cluster suffixed with `-pgbouncer`. For example, for our `hippo` cluster this would be `hippo-pgbouncer`. For the `keycloakdb` example, it would be `keycloakdb-pgbouncer`.

To customize the TLS for PgBouncer, you will need to create a Secret in the Namespace of your Ivory cluster that contains the TLS key (`tls.key`), TLS certificate (`tls.crt`) and the CA certificate (`ca.crt`) to use. The Secret should contain the following values:

```
data:
  ca.crt: <value>
  tls.crt: <value>
  tls.key: <value>
```

For example, if you have files named `ca.crt`, `keycloakdb-pgbouncer.key`, and `keycloakdb-pgbouncer.crt` stored on your local machine, you could run the following command:

```
kubectl create secret generic -n ivory-operator keycloakdb-pgbouncer.tls \
  --from-file=ca.crt=ca.crt \
  --from-file=tls.key=keycloakdb-pgbouncer.key \
  --from-file=tls.crt=keycloakdb-pgbouncer.crt
```

You can specify the custom TLS Secret in the `spec.proxy.pgBouncer.customTLSSecret.name` field in your `ivorycluster.ivory-operator.ivorysql.org` custom resource, e.g.:

```
spec:
  proxy:
    pgBouncer:
      customTLSSecret:
        name: keycloakdb-pgbouncer.tls
```

Customizing

The PgBouncer connection pooler is highly customizable, both from a configuration and Kubernetes deployment standpoint. Let’ s explore some of the customizations that you can do!

Configuration

PgBouncer configuration can be customized through `spec.proxy.pgBouncer.config`. After making configuration changes, IVYO will roll them out to any PgBouncer instance and automatically issue a "reload".

There are several ways you can customize the configuration:

- `spec.proxy.pgBouncer.config.global`: Accepts key-value pairs that apply changes globally to PgBouncer.
- `spec.proxy.pgBouncer.config.databases`: Accepts key-value pairs that represent PgBouncer database definitions.
- `spec.proxy.pgBouncer.config.users`: Accepts key-value pairs that represent connection settings applied to specific users.
- `spec.proxy.pgBouncer.config.files`: Accepts a list of files that are mounted in the `/etc/pgbouncer` directory and loaded before any other options are considered using PgBouncer's include directive.

For example, to set the connection pool mode to `transaction`, you would set the following configuration:

```
spec:
  proxy:
    pgBouncer:
      config:
        global:
          pool_mode: transaction
```

For a reference on PgBouncer configuration please see:

https://www.pgbouncer.org/config.html

Replicas

IVYO deploys one PgBouncer instance by default. You may want to run multiple PgBouncer instances to have some level of redundancy, though you still want to be mindful of how many connections are going to your Ivory database!

You can manage the number of PgBouncer instances that are deployed through the `spec.proxy.pgBouncer.replicas` attribute.

Resources

You can manage the CPU and memory resources given to a PgBouncer instance through the `spec.proxy.pgBouncer.resources` attribute. The layout of `spec.proxy.pgBouncer.resources` should be familiar: it follows the same pattern as the standard Kubernetes structure for setting container resources.

For example, let's say we want to set some CPU and memory limits on our PgBouncer instances. We could add the following configuration:

```
spec:
  proxy:
    pgBouncer:
      resources:
        limits:
          cpu: 200m
```

```
          memory: 128Mi
```

As IVYO deploys the PgBouncer instances using a Deployment these changes are rolled out using a rolling update to minimize disruption between your application and Ivory instances!

Annotations / Labels

You can apply custom annotations and labels to your PgBouncer instances through the `spec.proxy.pgBouncer.metadata.annotations` and `spec.proxy.pgBouncer.metadata.labels` attributes respectively. Note that any changes to either of these two attributes take precedence over any other custom labels you have added.

Pod Anti-Affinity / Pod Affinity / Node Affinity

You can control the pod anti-affinity, pod affinity, and node affinity through the `spec.proxy.pgBouncer.affinity` attribute, specifically:

- `spec.proxy.pgBouncer.affinity.nodeAffinity`: controls node affinity for the PgBouncer instances.
- `spec.proxy.pgBouncer.affinity.podAffinity`: controls Pod affinity for the PgBouncer instances.
- `spec.proxy.pgBouncer.affinity.podAntiAffinity`: controls Pod anti-affinity for the PgBouncer instances.

Each of the above follows the standard Kubernetes specification for setting affinity.

For example, to set a preferred Pod anti-affinity rule for the `kustomize/keycloak` example, you would want to add the following to your configuration:

```
spec:
  proxy:
    pgBouncer:
      affinity:
        podAntiAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - weight: 1
            podAffinityTerm:
              labelSelector:
                matchLabels:
                  ivory-operator.ivorysql.org/cluster: keycloakdb
                  ivory-operator.ivorysql.org/role: pgbouncer
              topologyKey: kubernetes.io/hostname
```

Tolerations

You can deploy PgBouncer instances to Nodes with Taints by setting Tolerations through `spec.proxy.pgBouncer.tolerations`. This attribute follows the Kubernetes standard tolerations layout.

For example, if there were a set of Nodes with a Taint of `role=connection-poolers:NoSchedule` that you want to schedule your PgBouncer instances to, you could apply the following configuration:

```
spec:
  proxy:
```

```
    pgBouncer:
      tolerations:
      - effect: NoSchedule
        key: role
        operator: Equal
        value: connection-poolers
```

Note that setting a toleration does not necessarily mean that the PgBouncer instances will be assigned to Nodes with those taints. Tolerations act as a key: they allow for you to access Nodes. If you want to ensure that your PgBouncer instances are deployed to specific nodes, you need to combine setting tolerations with node affinity.

Pod Spread Constraints

Besides using affinity, anti-affinity and tolerations, you can also set Topology Spread Constraints through `spec.proxy.pgBouncer.topologySpreadConstraints`. This attribute follows the Kubernetes standard topology spread contraint layout.

For example, since each of of our pgBouncer Pods will have the standard `ivory-operator.ivorysql.org/role: pgbouncer` Label set, we can use this Label when determining the `maxSkew`. In the example below, since we have 3 nodes with a `maxSkew` of 1 and we've set `whenUnsatisfiable` to `ScheduleAnyway`, we should ideally see 1 Pod on each of the nodes, but our Pods can be distributed less evenly if other constraints keep this from happening.

```
  proxy:
    pgBouncer:
      replicas: 3
      topologySpreadConstraints:
        - maxSkew: 1
          topologyKey: my-node-label
          whenUnsatisfiable: ScheduleAnyway
          labelSelector:
            matchLabels:
              ivory-operator.ivorysql.org/role: pgbouncer
```

If you want to ensure that your PgBouncer instances are deployed more evenly (or not deployed at all), you need to update `whenUnsatisfiable` to `DoNotSchedule`.

Next Steps

Now that we can enable connection pooling in a cluster, Let's explore some administrative tasks such as manually restarting IvorySQL using IVYO. How do we do that?

Administrative Tasks

Manually Restarting IvorySQL

There are times when you might need to manually restart IvorySQL. This can be done by adding or updating a custom annotation to the cluster's `spec.metadata.annotations` section. IVYO will notice the change and perform a rolling restart.

For example, if you have a cluster named `hippo` in the namespace `ivory-operator`, all you need to do is patch the hippo ivorycluster with the following:

```
kubectl patch ivorycluster/hippo -n ivory-operator --type merge \
  --patch '{"spec":{"metadata":{"annotations":{"restarted":"'"$(date)"'"}}}}'
```

Watch your hippo cluster: you will see the rolling update has been triggered and the restart has begun.

Shutdown

You can shut down an Ivory cluster by setting the `spec.shutdown` attribute to `true`. You can do this by editing the manifest, or, in the case of the `hippo` cluster, executing a command like the below:

```
kubectl patch ivorycluster/hippo -n ivory-operator --type merge \
  --patch '{"spec":{"shutdown": true}}'
```

The effect of this is that all the Kubernetes workloads for this cluster are scaled to 0. You can verify this with the following command:

```
kubectl get deploy,sts,cronjob --selector=ivory-operator.ivorysql.org/cluster=hippo -n
ivory-operator

NAME                              READY   AGE
statefulset.apps/hippo-00-lwgx    0/0     1h

NAME                              SCHEDULE   SUSPEND   ACTIVE
cronjob.batch/hippo-repo1-full    @daily     True      0
```

To turn an Ivory cluster that is shut down back on, you can set `spec.shutdown` to `false`.

Pausing Reconciliation and Rollout

You can pause the Ivory cluster reconciliation process by setting the `spec.paused` attribute to `true`. You can do this by editing the manifest, or, in the case of the `hippo` cluster, executing a command like the below:

```
kubectl patch ivorycluster/hippo -n ivory-operator --type merge \
  --patch '{"spec":{"paused": true}}'
```

Pausing a cluster will suspend any changes to the cluster' s current state until reconciliation is resumed. This allows you to fully control when changes to the ivorycluster spec are rolled out to the Ivory cluster. While paused, no statuses are updated other than the "Progressing" condition.

To resume reconciliation of an Ivory cluster, you can either set `spec.paused` to `false` or remove the setting from your manifest.

Rotating TLS Certificates

Credentials should be invalidated and replaced (rotated) as often as possible to minimize the risk of their misuse. Unlike passwords, every TLS certificate has an expiration, so replacing them is inevitable.

In fact, IVYO automatically rotates the client certificates that it manages before the expiration date on the certificate. A new client certificate will be generated after 2/3rds of its working duration; so, for instance, a IVYO-created certificate with an expiration date 12 months in the future will be replaced by IVYO around the eight month mark. This is done so that you do not have to worry about running into problems or

interruptions of service with an expired certificate.

Triggering a Certificate Rotation

If you want to rotate a single client certificate, you can regenerate the certificate of an existing cluster by deleting the `tls.key` field from its certificate Secret.

Is it time to rotate your IVYO root certificate? All you need to do is delete the `ivyo-root-cacert` secret. IVYO will regenerate it and roll it out seamlessly, ensuring your apps continue communicating with the Ivory cluster without having to update any configuration or deal with any downtime.

```
kubectl delete secret ivyo-root-cacert
```



IVYO only updates secrets containing the generated root certificate. It does not touch custom certificates.

Rotating Custom TLS Certificates

When you use your own TLS certificates with IVYO, you are responsible for replacing them appropriately. Here's how.

IVYO automatically detects and loads changes to the contents of IvorySQL server and replication Secrets without downtime. You or your certificate manager need only replace the values in the Secret referenced by `spec.customTLSSecret`.

If instead you change `spec.customTLSSecret` to refer to a new Secret or new fields, IVYO will perform a rolling restart.



When changing the IvorySQL certificate authority, make sure to update `customReplicationTLSSecret` as well.

Changing the Primary

There may be times when you want to change the primary in your HA cluster. This can be done using the `patroni.switchover` section of the ivorycluster spec. It allows you to enable switchovers in your ivoryclusters, target a specific instance as the new primary, and run a failover if your ivorycluster has entered a bad state.

Let's go through the process of performing a switchover!

First you need to update your spec to prepare your cluster to change the primary. Edit your spec to have the following fields:

```
spec:
  patroni:
    switchover:
      enabled: true
```

After you apply this change, IVYO will be looking for the trigger to perform a switchover in your cluster. You will trigger the switchover by adding the `ivory-operator.ivorysql.org/trigger-switchover` annotation to your custom resource. The best way to set this annotation is with a timestamp, so you know when you initiated the change.

For example, for our `hippo` cluster, we can run the following command to trigger the switchover:

```
kubectl annotate -n ivory-operator ivorycluster hippo \
  ivory-operator.ivorysql.org/trigger-switchover="$(date)"
```



If you want to perform another switchover you can re-run the annotation command and add the `--overwrite` flag:

```
kubectl annotate -n ivory-operator ivorycluster hippo --overwrite \
  ivory-operator.ivorysql.org/trigger-switchover="$(date)"
```

IVYO will detect this annotation and use the Patroni API to request a change to the current primary!

The roles on your database instance Pods will start changing as Patroni works. The new primary will have the `master` role label, and the old primary will be updated to `replica`.

The status of the switch will be tracked using the `status.patroni.switchover` field. This will be set to the value defined in your trigger annotation. If you use a timestamp as the annotation this is another way to determine when the switchover was requested.

After the instance Pod labels have been updated and `status.patroni.switchover` has been set, the primary has been changed on your cluster!



After changing the primary, we recommend that you disable switchovers by setting `spec.patroni.switchover.enabled` to false or remove the field from your spec entirely. If the field is removed the corresponding status will also be removed from the ivorycluster.

Targeting an instance

Another option you have when switching the primary is providing a target instance as the new primary. This target instance will be used as the candidate when performing the switchover. The `spec.patroni.switchover.targetInstance` field takes the name of the instance that you are switching to.

This name can be found in a couple different places; one is as the name of the StatefulSet and another is on the database Pod as the `ivory-operator.ivorysql.org/instance` label. The following commands can help you determine who is the current primary and what name to use as the `targetInstance`:

```
$ kubectl get pods -l ivory-operator.ivorysql.org/cluster=hippo \
    -L ivory-operator.ivorysql.org/instance \
    -L ivory-operator.ivorysql.org/role -n ivory-operator

NAME                     READY   STATUS      RESTARTS   AGE     INSTANCE
ROLE
hippo-instance1-jdb5-0   3/3     Running     0          2m47s   hippo-instance1-jdb5
master
hippo-instance1-wm5p-0   3/3     Running     0          2m47s   hippo-instance1-wm5p
replica
```

In our example cluster `hippo-instance1-jdb5` is currently the primary meaning we want to target `hippo-instance1-wm5p` in the switchover. Now that you know which instance is currently the primary and how to find your `targetInstance`, let's update your cluster spec:

```
spec:
  patroni:
    switchover:
      enabled: true
      targetInstance: hippo-instance1-wm5p
```

After applying this change you will once again need to trigger the switchover by annotating the ivorycluster (see above commands). You can verify the switchover has completed by checking the Pod role labels and `status.patroni.switchover`.

Failover

Finally, we have the option to failover when your cluster has entered an unhealthy state. The only spec change necessary to accomplish this is updating the `spec.patroni.switchover.type` field to the `Failover` type. One note with this is that a `targetInstance` is required when performing a failover. Based on the example cluster above, assuming `hippo-instance1-wm5p` is still a replica, we can update the spec:

```
spec:
  patroni:
    switchover:
      enabled: true
      targetInstance: hippo-instance1-wm5p
      type: Failover
```

Apply this spec change and your ivorycluster will be prepared to perform the failover. Again you will need to trigger the switchover by annotating the ivorycluster (see above commands) and verify that the Pod role labels and `status.patroni.switchover` are updated accordingly.



Errors encountered in the switchover process can leave your cluster in a bad state. If you encounter issues, found in the operator logs, you can update the spec to fix the issues and apply the change. Once the change has been applied, IVYO will attempt to perform the switchover again.

Next Steps

We've covered a lot in terms of building, maintaining, scaling, customizing, restarting, and expanding our Ivory cluster. However, there may come a time where we need to delete our Ivory cluster. How do we do that?

Delete an Ivory Cluster

There comes a time when it is necessary to delete your cluster. If you have been following along with the example, you can delete your Ivory cluster by simply running:

```
kubectl delete -k examples/kustomize/ivory
```

IVYO will remove all of the objects associated with your cluster.

With data retention, this is subject to the retention policy of your PVC. For more information on how Kubernetes manages data retention, please refer to the Kubernetes docs on volume reclaiming.

..3. Docker & Podman deployment

Running IvorySQL in docker

- Get IvorySQL image from Docker Hub

```
$ docker pull ivorysql/IvorySQL 5.4-ubi8
```

- Running IvorySQL

```
$ docker run --name ivorysql -p 5434:5432 -e IVORYSQL_PASSWORD=your_password -d
ivorysql/IvorySQL 5.4-ubi8
```

- Check if the IvorySQL container is running successfully

```
$ docker ps | grep ivorysql
CONTAINER ID   IMAGE               COMMAND                  CREATED          STATUS
PORTS                              NAMES
6faa2d0ed705   IvorySQL 5.4-ubi8   "docker-entrypoint.s…"   50 seconds ago   Up 49
seconds   5866/tcp, 0.0.0.0:5434->5432/tcp   ivorysql
```

Running with Podman

- Pull IvorySQL Image from Docker Hub

```
[highgo@manager-node1 ~]$ podman pull ivorysql/IvorySQL 5.4-ubi8
⊠ docker.io/ivorysql/IvorySQL 5.4-ubi8
Trying to pull docker.io/ivorysql/IvorySQL 5.4-ubi8...
Getting image source signatures
Copying blob 5885448c5c88 done   |
Copying blob 6c502b378234 done   |
Copying blob 8b4f2b90d6b6 done   |
Copying blob 9b000f2935f6 done   |
Copying blob 806f782da874 done   |
Copying blob e4c51845a9eb done   |
Copying blob dcb1e9a04275 done   |
Copying blob 285a279173f8 done   |
Copying blob 1f6f247b9ae0 done   |
Copying blob 3cc81bed8614 done   |
Copying blob 863c87bf25eb done   |
Copying blob 4f4fb700ef54 done   |
Copying config 88e1bbeda8 done   |
Writing manifest to image destination
88e1bbeda81c51d88e12cbd2b19730498f1343d1c64bb3dddc8ffcb08a1f965f
```

- Run IvorySQL Container

```
$ podman run --name ivorysql -p 5434:5432 -e IVORYSQL_PASSWORD=123456 -d
ivorysql/IvorySQL 5.4-ubi8
```

- Check if IvorySQL Container is Running Successfully

```
[highgo@manager-node1 ~]$ podman ps | grep ivorysql
CONTAINER ID           IMAGE                        COMMAND          CREATED
STATUS          PORTS                                NAMES
368dee58d5ef  docker.io/ivorysql/IvorySQL 5.4-ubi8  postgres    20 seconds ago  Up 20
seconds  0.0.0.0:5434->5432/tcp, 1521/tcp, 5866/tcp  ivorysql

[highgo@manager-node1 ~]$ podman exec -it ivorysql /bin/bash
[root@8cc631eb413d /]#
ivorysql=# select version();
                                                    version
-------------------------------------------------------------------------------------
---------------------------------
 PostgreSQL 18.0 (IvorySQL 5.4) on x86_64-pc-linux-gnu, compiled by gcc (GCC) 8.5.0
20210514 (Red Hat 8.5.0-28), 64-bit
(1 row)

ivorysql=#
```

..4. Docker Swarm & Docker Compose deployment

Prepare three servers with network connectivity and set up a Swarm cluster. The test cluster names and corresponding IP addresses are as follows:

manager-node1: 192.168.21.205

manager-node2: 192.168.21.164

manager-node3: 192.168.21.51

```
[root@manager-node1 docker-swarm]# docker node ls
ID                            HOSTNAME        STATUS    AVAILABILITY   MANAGER STATUS
ENGINE VERSION
y9d9wd9t2ncy4t9bvw6bg9sjs *   manager-node1   Ready     Active         Reachable
26.1.4
iv17o6m9t9e06vd9iu1o6damd     manager-node2   Ready     Active         Leader
25.0.4
vjnax76qj812mlvut6cv4qotl     manager-node3   Ready     Active         Reachable
24.0.6
```

Building IvorySQL HA Cluster using Docker Swarm

Download the source code

```
[root@manager-node1 ~]# git clone https://github.com/IvorySQL/docker_library.git
[root@manager-node1 ~]# cd docker_library/docker-cluster/docker-swarm
```

Deploy a three-node etcd cluster

```
[root@manager-node1 docker-swarm]# docker stack deploy -c docker-swarm-etcd.yml
ivoryhac-etcd
Creating network ivoryhac-etcd_etcd-net
Creating service ivoryhac-etcd_etcd3
Creating service ivoryhac-etcd_etcd1
Creating service ivoryhac-etcd_etcd2
[root@manager-node1 docker-swarm]# docker service ls
ID             NAME                   MODE         REPLICAS   IMAGE
PORTS
1jst0mva8o5n   ivoryhac-etcd_etcd1   replicated   1/1
quay.io/coreos/etcd:v3.5.8   *:2379-2380->2379-2380/tcp
sosag5017cis   ivoryhac-etcd_etcd2   replicated   1/1
quay.io/coreos/etcd:v3.5.8
8twpgkzo2mnx   ivoryhac-etcd_etcd3   replicated   1/1
quay.io/coreos/etcd:v3.5.8
```

You can customize the external database directory by modifying the volumes section in docker-swarm-ivypatroni.yml. After making changes, adjust the directory permissions and ownership accordingly. Example as follows:

```
mkdir -p /home/ivorysql/{data,patroni}
chown -R 1000:1000 /home/ivorysql/{data,patroni}
chmod 700 /home/ivorysql/{data,patroni}
```

Deploy an IvorySQL High Availability Cluster

```
[root@manager-node1 docker-swarm]# docker stack deploy -c docker-swarm-ivypatroni.yml
ivoryhac-patroni
Since --detach=false was not specified, tasks will be created in the background.
In a future release, --detach=false will become the default.
Creating service ivoryhac-patroni_ivypatroni1
Creating service ivoryhac-patroni_ivypatroni2
[root@manager-node1 docker-swarm]# docker service ls
ID             NAME                       MODE         REPLICAS   IMAGE
PORTS
1jst0mva8o5n   ivoryhac-etcd_etcd1       replicated   1/1
quay.io/coreos/etcd:v3.5.8                       *:2379-2380->2379-2380/tcp
```

```
sosag5017cis   ivoryhac-etcd_etcd2        replicated   1/1
quay.io/coreos/etcd:v3.5.8
8twpgkzo2mnx   ivoryhac-etcd_etcd3        replicated   1/1
quay.io/coreos/etcd:v3.5.8
uzdvjq5j2gwt   ivorysql-hac_ivypatroni1   replicated   1/1        ivorysql/docker-
swarm-ha-cluster:5.0-4.0.6-ubi8   *:1521->1521/tcp, *:5866->5866/tcp
fr0m9chu3ce8   ivorysql-hac_ivypatroni2   replicated   1/1        ivorysql/docker-
swarm-ha-cluster:5.0-4.0.6-ubi8   *:1522->1521/tcp, *:5867->5866/tcp
```

Connect to the database using psql via Oracle port and PostgreSQL port

```
[root@manager-node1 docker-swarm]# psql -h127.0.0.1 -p1521 -U ivorysql -d ivorysql
Password for user ivorysql:

ivorysql=# select version();
                                                          version
------------------------------------------------------------------------------------------
----------------------------------
 PostgreSQL 18.0 (IvorySQL 5.4) on x86_64-pc-linux-gnu, compiled by gcc (GCC) 8.5.0
20210514 (Red Hat 8.5.0-28), 64-bit
(1 row)

ivorysql=# show ivorysql.compatible_mode;
 ivorysql.compatible_mode
--------------------------
 oracle
(1 row)

ivorysql=# exit
```

```
[root@manager-node1 docker-swarm]# psql -h127.0.0.1 -p5432 -U ivorysql -d ivorysql
Password for user ivorysql:

ivorysql=# select version();
                                                          version
------------------------------------------------------------------------------------------
----------------------------------
 PostgreSQL 18.0 (IvorySQL 5.4) on x86_64-pc-linux-gnu, compiled by gcc (GCC) 8.5.0
20210514 (Red Hat 8.5.0-28), 64-bit
(1 row)

ivorysql=# show ivorysql.compatible_mode;
 ivorysql.compatible_mode
--------------------------
```

```
 pg
(1 row)
```

Uninstall the IvorySQL cluster

```
[root@manager-node1 ~] docker stack rm ivoryhac-patroni
[root@manager-node1 ~] docker stack rm ivoryhac-etcd
```

Set up an IvorySQL HA Cluster using Docker Compose

Download the source code

```
[root@manager-node1 ~]# git clone https://github.com/IvorySQL/docker_library.git
[root@manager-node1 ~]# cd docker_library/docker-cluster/docker-compose
```

Copy files to another server

Copy the etcd and ivypatroni Docker Compose files to other servers respectively.

For example, to the test server:

192.168.21.205 will host etcd1+ivorypatroni1，

192.168.21.164 will host etcd2+ivorypatroni2，

192.168.21.51 will host etcd3+ivorypatroni3

Deploy a three-node etcd cluster, taking node1 as an example

```
[root@manager-node1 docker-compose]# docker-compose -f ./docker-compose-etcd1.yml up
-d
[+] Running 1/1
 ⠿ Container etcd  Started
0.1s
```

Deploy an IvorySQL high-availability cluster.

Deploy the ivyhac service on each node, using node1 as an example.

```
[root@manager-node1 docker-compose]# docker-compose -f ./docker-compose-
ivypatroni_1.yml up -d
[+] Running 1/1
 ⠿ Container ivyhac1  Started
0.1s
[root@manager-node1 docker-compose]# docker ps
CONTAINER ID   IMAGE                                                COMMAND
CREATED          STATUS          PORTS     NAMES
736c0d188bdd   ivorysql/docker-compose-ha-cluster:5.0-4.0.6-ubi8   "/bin/sh /docker-
ent…"   18 seconds ago   Up 17 seconds             ivyhac1
```

```
9d8e04e4f819   quay.io/coreos/etcd:v3.5.8
"/usr/local/bin/etcd"    24 minutes ago   Up 24 minutes             etcd
```

At this point, the one-primary-two-standby cluster setup is complete. Connect to the database using psql via Oracle-compatible ports and PostgreSQL ports.

```
[root@manager-node1 docker-swarm]# psql -h127.0.0.1 -p1521 -U ivorysql -d ivorysql
Password for user ivorysql:

ivorysql=# select version();
                                                      version
-------------------------------------------------------------------------------------------
---------------------------------
 PostgreSQL 18.0 (IvorySQL 5.4) on x86_64-pc-linux-gnu, compiled by gcc (GCC) 8.5.0
20210514 (Red Hat 8.5.0-28), 64-bit
(1 row)

ivorysql=# show ivorysql.compatible_mode;
 ivorysql.compatible_mode
--------------------------
 oracle
(1 row)

ivorysql=# exit
```

```
[root@manager-node1 docker-swarm]# psql -h127.0.0.1 -p5432 -U ivorysql -d ivorysql
Password for user ivorysql:

ivorysql=# select version();
                                                      version
-------------------------------------------------------------------------------------------
---------------------------------
 PostgreSQL 18.0 (IvorySQL 5.4) on x86_64-pc-linux-gnu, compiled by gcc (GCC) 8.5.0
20210514 (Red Hat 8.5.0-28), 64-bit
(1 row)

ivorysql=# show ivorysql.compatible_mode;
 ivorysql.compatible_mode
--------------------------
 pg
(1 row)
```

Uninstall the IvorySQL cluster, using node1 as an example.

```
[root@manager-node1 ~] docker-compose -f ./docker-compose-ivypatroni_1.yml down
[root@manager-node1 ~] docker-compose -f ./docker-compose-etcd1.yml down
```

Cloud Service Platform

..1. IvorySQL Cloud Installation

The IvorySQL Cloud platform is a comprehensive solution that integrates the IvorySQL database and its surrounding ecosystem to deliver end-to-end database and resource management capabilities. Before starting the installation, compile and install the following projects from GitHub:

Frontend: https://github.com/IvorySQL/ivory-cloud-web

Backend: https://github.com/IvorySQL/ivory-cloud

Prepare a Kubernetes cluster (version 1.23) and install `ivory-operator` on the master node:

https://github.com/IvorySQL/ivory-operator/tree/IVYO_REL_5_STABLE

IvorySQL Cloud Platform Installation

The IvorySQL Cloud platform currently supports installation on Linux systems. The required packages are listed below.

Component	Package
Frontend	dist
Backend	cloudnative-1.0-SNAPSHOT.jar
K8S cluster	1. docker.io/ivorysql/ivory-operator:v5.4 2. docker.io/ivorysql/pgadmin:ubi8-9.9-5.0-1 3. docker.io/ivorysql/pgbackrest:ubi8-2.56.0-5.0-1 4. docker.io/ivorysql/postgres-exporter:ubi8-0.17.0-5.0-1 5. docker.io/ivorysql/ivorysql:ubi8-5.0-5.0-1

In addition, install the following supporting components:

- Backend database: Stores and manages all data related to cloud resources, user information, access control, billing, and more. Use a PostgreSQL-compatible database such as PostgreSQL, HighGo DB, or IvorySQL.
- NGINX: Hosts the web user interface of the cloud platform.

Pre-installation Checklist

Complete the preparation steps on every server before installation. Deploy IvorySQL Cloud on a Kubernetes (1.23) cluster that already has a default storage class.

Disable the firewall

Disable the firewall on every server to ensure full connectivity.

```
systemctl stop firewalld.service
```

```
systemctl disable firewalld.service
```

Backend deployment

Backend database

Install the backend database yourself by following the instructions on the IvorySQL official website.

Backend services

Compile the backend service

```
# Clone the code

git clone https://github.com/IvorySQL/ivory-cloud.git

# Go to the project root

cd ivory-cloud
```

Ensure that every `.sh` file in `ivory-cloud\cloudnative\src\main\resources\monitor` and its subdirectories uses the Unix format. If not, run `dos2unix` to convert them.

```
dos2unix cloudnative\src\main\resources\monitor\*

# Build

mvn clean

mvn package -D maven.test.skip=true

The packaged artifact `cloudnative-1.0-SNAPSHOT.jar` can be found under `ivory-
cloud/cloudnative/target`.
```

Deploy the service

```
Execute the following steps on the Kubernetes server:

# Create a working directory

mkdir -p /home/ivory

# Upload `ivory-cloud/cloudnative/target/cloudnative-1.0-SNAPSHOT.jar` to the
directory created above

# Configuration files
```

```
## Create a configuration directory

mkdir -p /home/ivory/config

## Upload configuration files

Copy the following files from `ivory-cloud/cloudnative/src/main/resources` to
`/home/ivory/config`:

application.yaml

application-native.yaml

spring_pro_logback.xml

## Update the configuration

Replace `url`, `username`, and `password` with the database information configured in
<<Backend database>>.

## /home/ivory/config/application-native.yaml

datasource:

druid:

db-type: com.alibaba.druid.pool.DruidDataSource

driver-class-name: org.postgresql.Driver

url: jdbc:postgresql://127.0.0.1:5432/ivorysql

username: ivorysql

password: "ivory@123"
```

Start the backend service

```
# Install JDK 1.8

yum install -y java-1.8.0-openjdk.x86_64
```

```
[root@cloud ivory]# pwd

/home/ivory/

[root@cloud ivory]# nohup java -jar cloudnative-1.0-SNAPSHOT.jar > log_native 2>&1 &

[root@cloud ivory]# ps -ef | grep java

root 77494 1 0 Oct09 ? 00:03:07 java -jar cloudnative-1.0-SNAPSHOT.jar
```

Frontend deployment

Compile the frontend

```
## Fetch the code

git clone https://github.com/IvorySQL/ivory-cloud-web.git

## Go to the project root

cd ivorysql-cloud-web

## Install dependencies

npm install

## Build for production

npm run build:prod
```

Update directory and file permissions

```
# Create a deployment directory

[root@cloud opt]# mkdir -p /opt/cloud/web

# Copy the generated `dist` folder to /opt/cloud/web

# Grant permissions

[root@cloud web]# chmod 755 /opt/cloud/web/dist

[root@cloud web]# chmod -R 777 /opt/cloud/web/dist
```

Update config.js

Edit the configuration file:

```
[root@cloud dist]# pwd

/home/cloud/web/dist

[root@cloud dist]# vi config.js

var PLATFROM_CONFIG = {};

// Replace with the IP address of the current server

PLATFROM_CONFIG.baseUrl = "http://192.168.31.43:8081/cloudapi/api/v1"

// true: show the  “Register”  button on the login page

// false: hide the  “Register”  button on the login page

globalShowRegister = true

// Hide the cloud-native database? true: hide; false: show

disableNative = false

// Database type

dbtype = "IvorySQL"

dbversion = "5.4"
```

Install and configure NGINX

The IvorySQL Cloud host must have NGINX installed to serve the web interface. Users can select any installation method; the following steps are provided for reference.

Download the NGINX source package

```
[root@cloud web]# wget https://nginx.org/download/nginx-1.20.1.tar.gz

[root@cloud web]# ls -lrt

total 3924
```

```
-rwxrwxr-x. 1 root root 1061461 May 25 2021 nginx-1.20.1.tar.gz

-rwxrwxr-x. 1 root root 2943732 Oct  9 16:43 dist.tar.gz

drwxrwxrwx. 4 root root     103 Oct 21 13:20 dist
```

Install dependencies

```
[root@host30 cloud]# yum -y install pcre-devel

[root@host30 cloud]# yum -y install openssl openssl-devel
```

Build and install NGINX

NGINX is installed under the directory specified by `--prefix` during `configure`. The example below installs it to `/opt/cloud/nginx`.

```
## Extract nginx-1.20.1.tar.gz

[root@cloud web]# tar -zxvf nginx-1.20.1.tar.gz

## Verify that nginx-1.20.1 was created

[root@cloud web]# ls -lrt

total 3924

-rwxrwxr-x. 1 root root 1061461 May 25 2021 nginx-1.20.1.tar.gz

-rwxrwxr-x. 1 root root 2943732 Oct  9 16:43 dist.tar.gz

drwxrwxr-x. 9 1001 1001     186 Oct  9 16:53 nginx-1.20.1

drwxrwxrwx. 4 root root     103 Oct 21 13:20 dist

## Configure

[root@cloud web]# cd nginx-1.20.1

[root@cloud nginx-1.20.1]# ./configure --prefix=/opt/cloud/nginx --with
-http_ssl_module

## Compile and install
```

```
[root@cloud nginx-1.20.1]# make

[root@cloud nginx-1.20.1]# make install
```

Update nginx.conf

The configuration file is stored under `/opt/cloud/nginx`. Adjust it according to the README on GitHub, and replace the IP with the address of the current server.

```
server {

listen 9104;

server_name 192.168.31.43;

location / {

root /opt/cloud/web/dist;

index index.html index.htm;

}

error_page 500 502 503 504 /50x.html;

location = /50x.html {

root html;

}

}
```

Start NGINX

```
[root@cloud sbin]# pwd

/opt/cloud/nginx/sbin

[root@cloud sbin]# ./nginx -c /opt/cloud/nginx/conf/nginx.conf

[root@cloud sbin]# ps -ef | grep nginx
```

```
root  2179 131037 0 09:46 pts/1 00:00:00 grep --color=auto nginx

root 55047      1 0 Oct21 ?     00:00:00 nginx: master process ./nginx -c
/opt/cloud/nginx/conf/nginx.conf

nobody 55048 55047 0 Oct21 ?    00:00:00 nginx: worker process
```

Operator deployment

Set up the Kubernetes cluster yourself. This section describes how to install `ivory-operator` on the cluster and preload container images.

Install ivory-operator

Refer to the README on GitHub:

https://github.com/IvorySQL/ivory-operator/tree/IVYO_REL_5_STABLE

Load container images

If your servers have direct access to Docker Hub, you can skip this step. Otherwise, preload the following images on every node in the Kubernetes cluster.

```
docker.io/ivorysql/pgadmin:ubi8-9.9-5.0-1

docker.io/ivorysql/pgbackrest:ubi8-2.56.0-5.0-1

docker.io/ivorysql/pgbouncer:ubi8-1.23.0-5.0-1

docker.io/ivorysql/postgres-exporter:ubi8-0.17.0-5.0-1

docker.io/ivorysql/ivorysql:ubi8-5.0-5.0-1

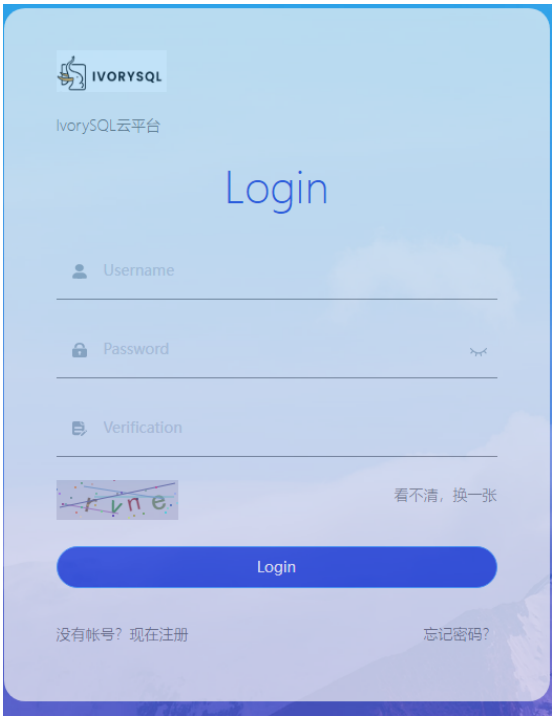
docker.io/prom/prometheus:v2.33.5

docker.io/prom/alertmanager:v0.22.2

docker.io/grafana/grafana:8.5.10
```

..2. IvorySQL Cloud Usage

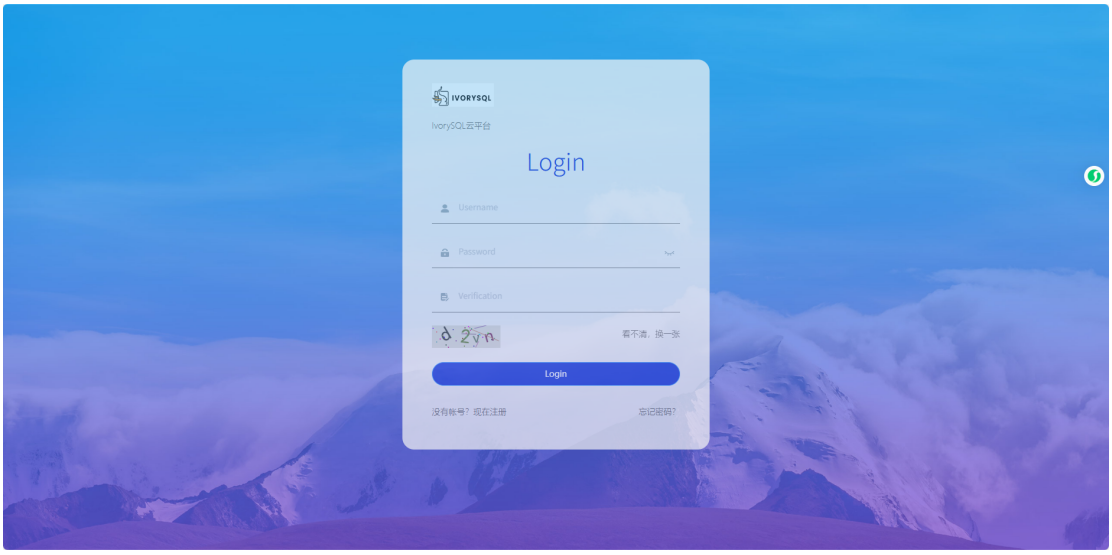
IvorySQL Cloud is a web-based service platform that can be accessed from any computer through a browser. After installing the cloud service platform on the server with IP `192.168.31.43`, open a browser and enter `http://192.168.31.43:9104/` (9104 is the port configured in `nginx.conf.default`) to reach the login page:



Sign In and Sign Out

Sign In

On the login page, enter the prompted information to access the IvorySQL Cloud service platform:



Sign Out

Click the avatar in the upper-right corner to display the current username and the Log Out action. Click Log Out to exit; click the username to stay on the current page:



Administrator Features

Add a Cluster

1. After signing in as the **admin** user, click K8S Cluster Management in the left navigation bar to open the cluster list.



2. Click Add Kubernetes Cluster in the upper-left corner, fill in the cluster information, and submit.



Manage Clusters

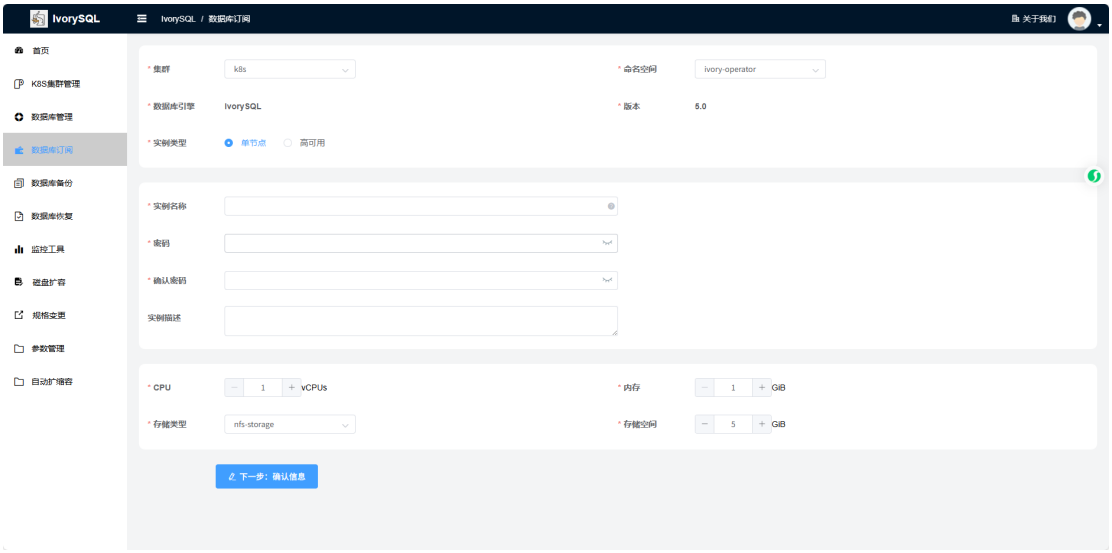
The cluster management page lists details for each cluster and allows you to edit or delete them.



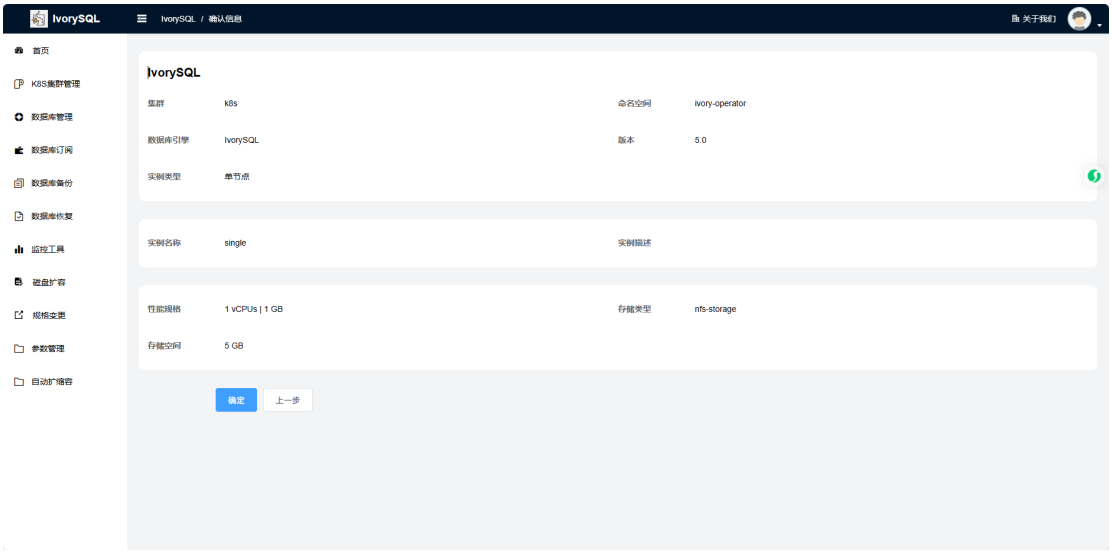
demo User Features

Database Subscription

1. Sign in with the **demo** user.
2. Click Database Subscription in the left navigation, fill in the database parameters, and click Next: Confirm.



3. Review the information and click Confirm.



4. After confirming, the page automatically redirects to Database Management to show the subscription task.



Database Management

Displays all databases managed by the cloud service platform.



Restart a Database

1. Sign in with the demo user.
2. Go to Database Management, select a database, click More in the Actions column, and choose Restart.



3. Review the information and click Confirm.

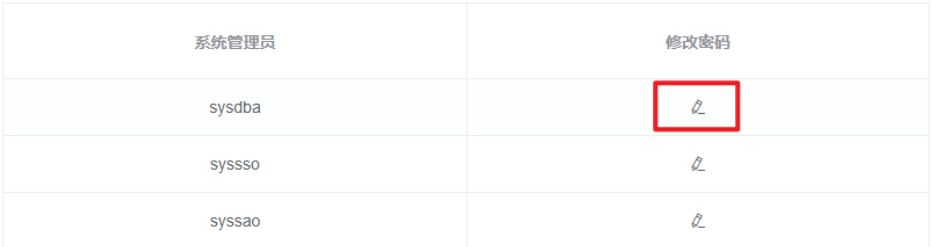


Change the Password

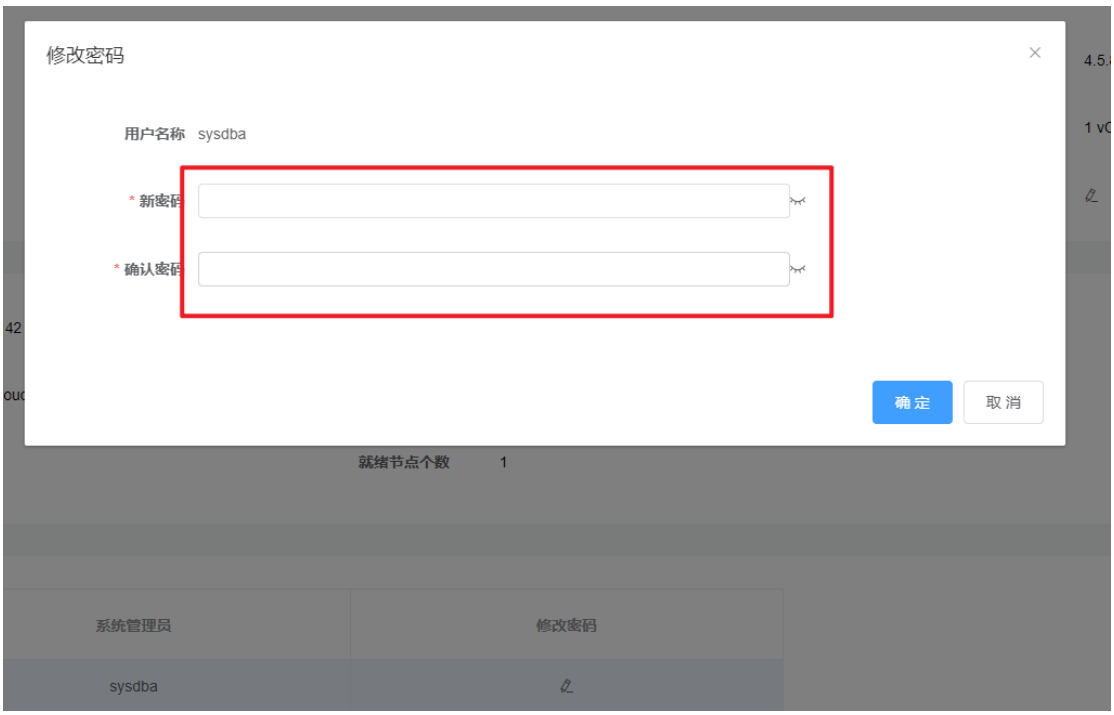
1. Sign in with the demo user.
2. Go to Database Management, select a database, and click its Instance ID.



3. On the database details page, click the password icon.



4. Enter a new password and click Confirm.



Delete an Instance

1. Sign in with the `demo` user.
2. Go to Database Management, select a database, click More in the Actions column, and choose Delete Instance.

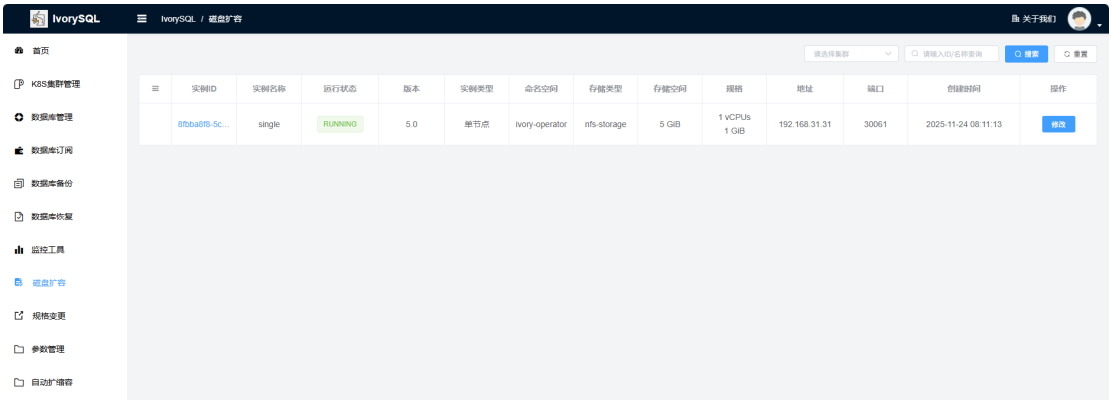


3. Review the confirmation window and click Confirm.



Storage Expansion

1. This feature requires additional plug-ins such as TopoLVM.
2. Sign in with the **demo** user.
3. Click Storage Expansion, select a database, then click Edit in the Actions column; alternatively, go to Database Management, click More, and choose Storage Expansion.



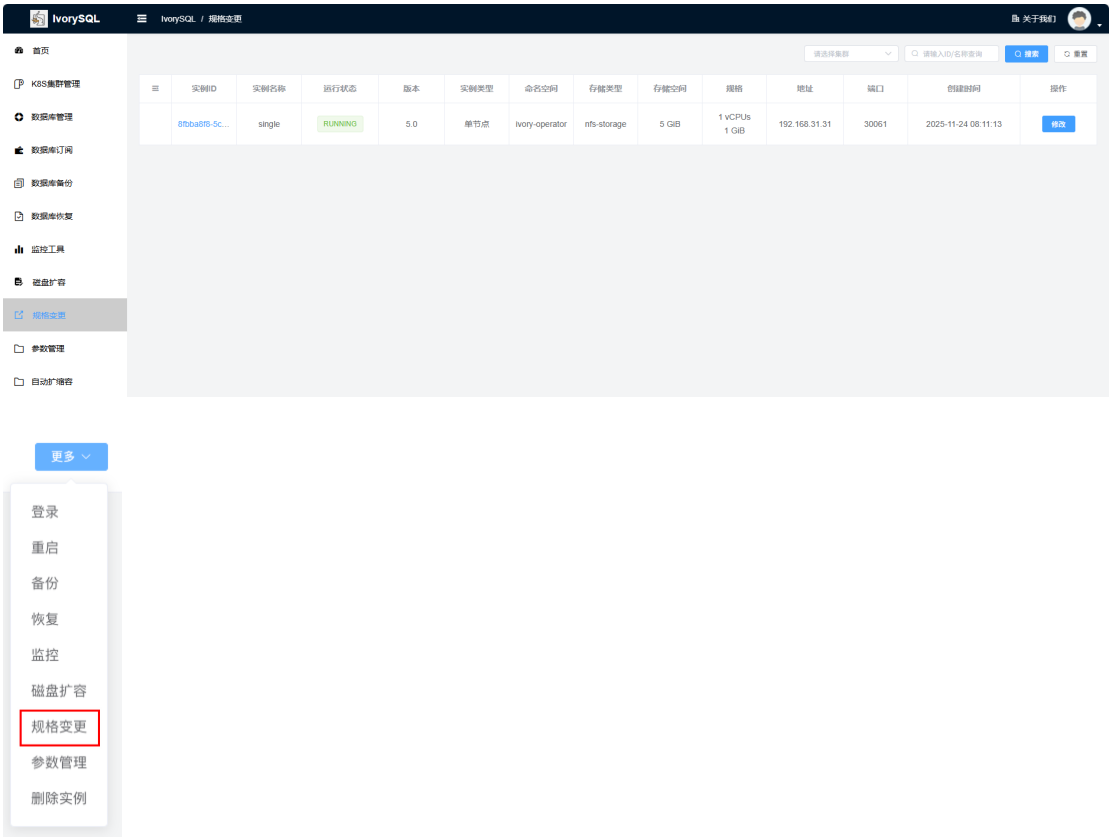


4. Enter the expanded storage size and click Confirm.



Specification Change

1. Sign in with the **demo** user.
2. Click Specification Change, select a database, then click Edit in the Actions column; or go to Database Management, click More, and choose Specification Change.

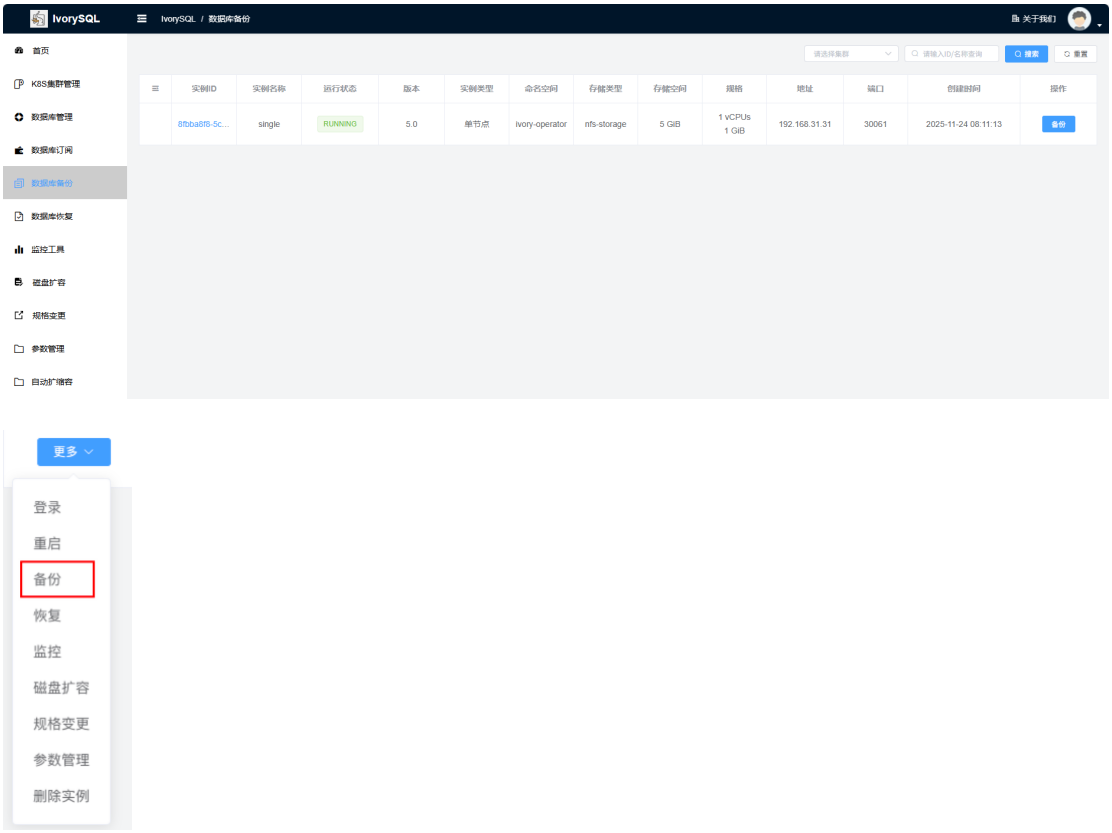


3. Enter the new specification and click Confirm.



Database Backup

1. Sign in with the `demo` user.
2. Go to Database Backup, select a database, and click Backup in the Actions column; or go to Database Management, click More, and choose Backup.

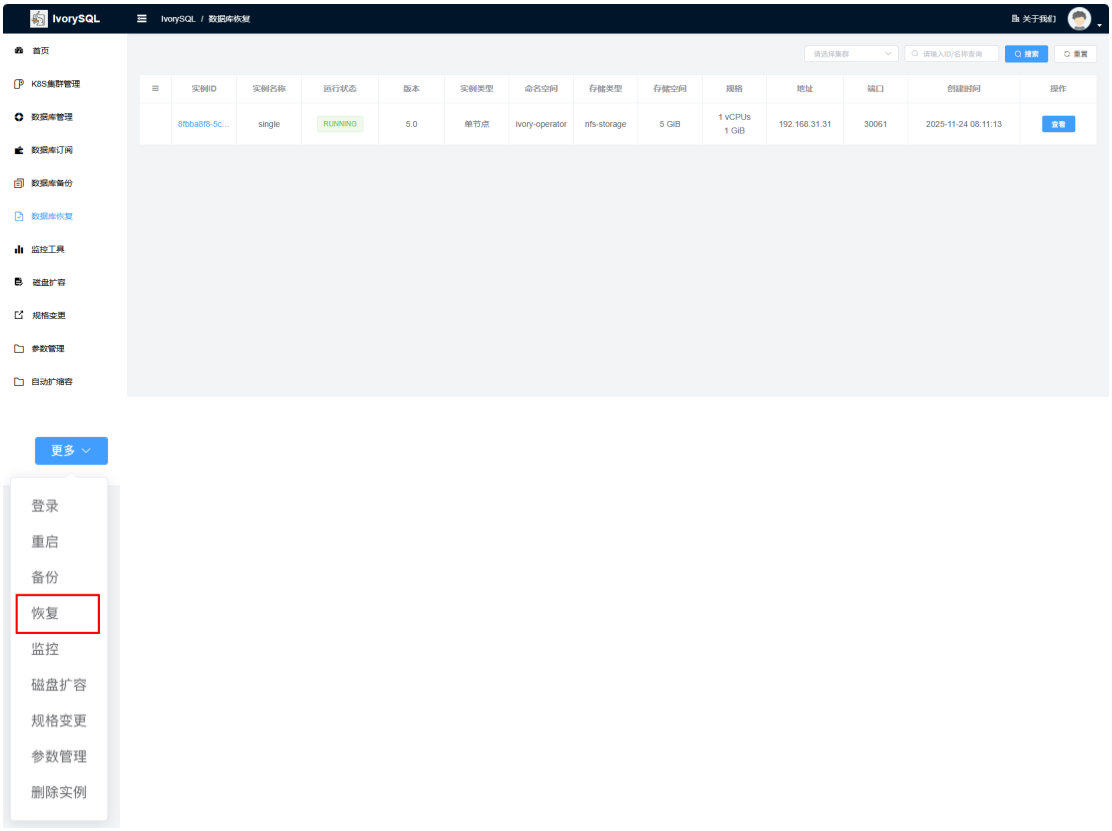


3. Enter a backup name and click Confirm.



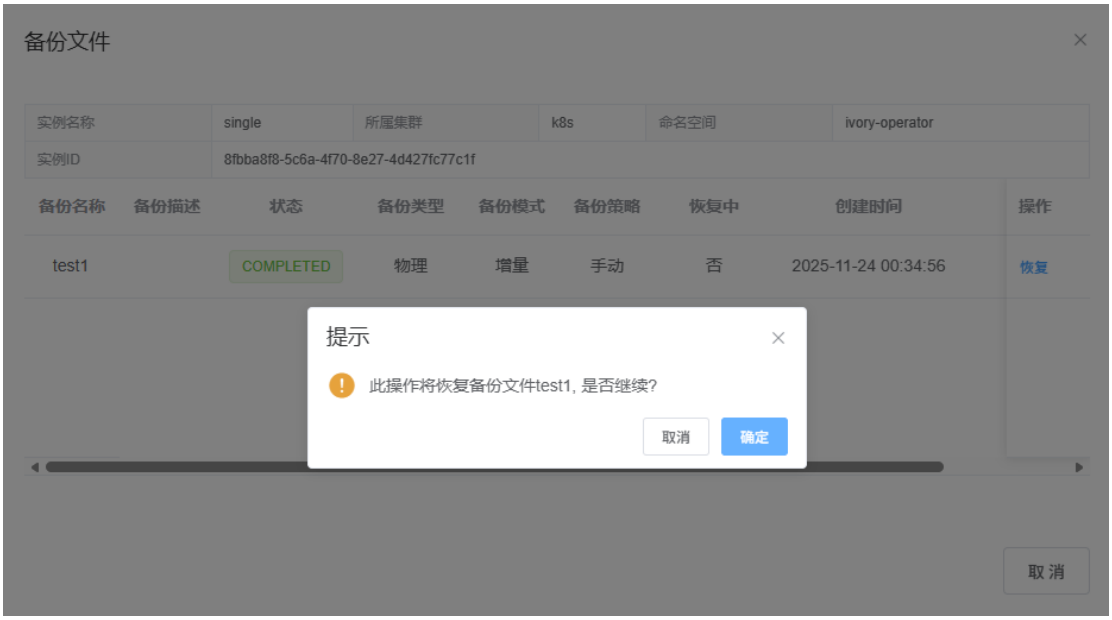
Database Restore

1. Sign in with the `demo` user.
2. Go to Database Restore, select a database, and click View in the Actions column; or go to Database Management, click More, and choose Restore.

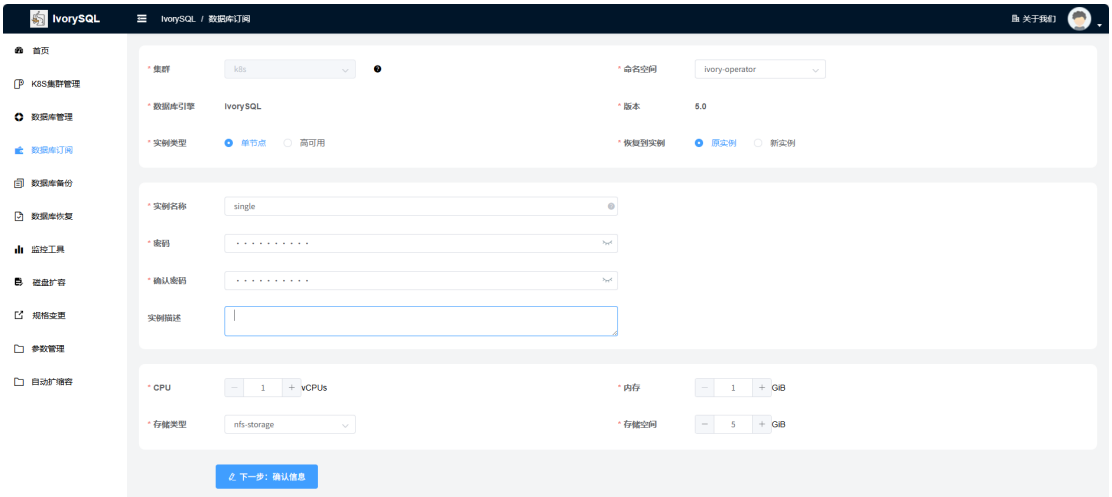


3. Select the backup file and click Restore in the Actions column.





4. Enter the target database information. Use the database password from before the backup.



5. Continue following the workflow described in “4.1 Database Subscription”.

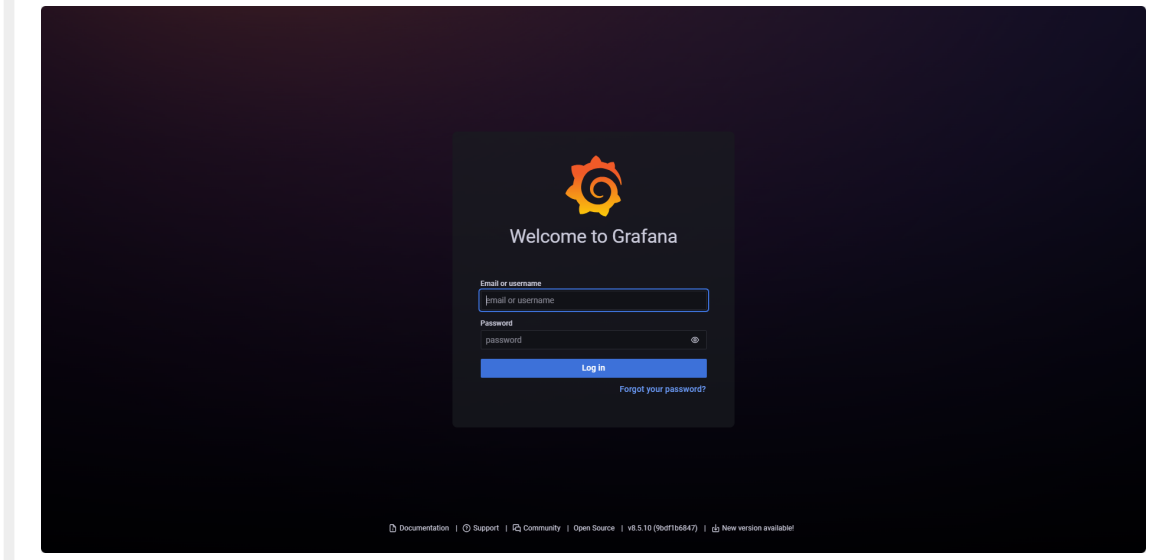
Database Monitoring

1. Sign in with the **demo** user.
2. Click Monitoring Tools in the left navigation and select the cluster where the database resides; or go to Database Management, select a database, click More, and choose Monitoring.

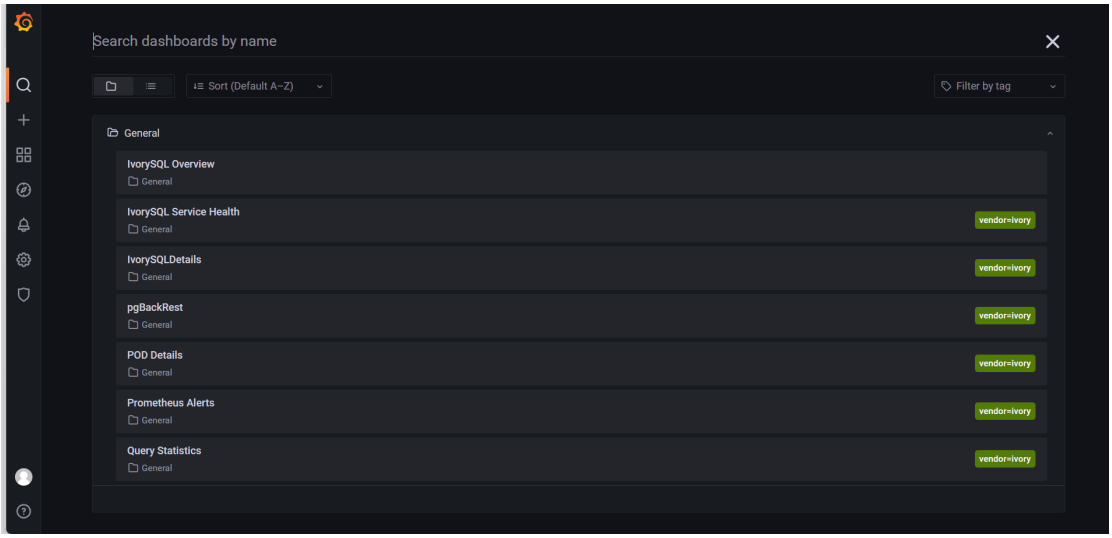


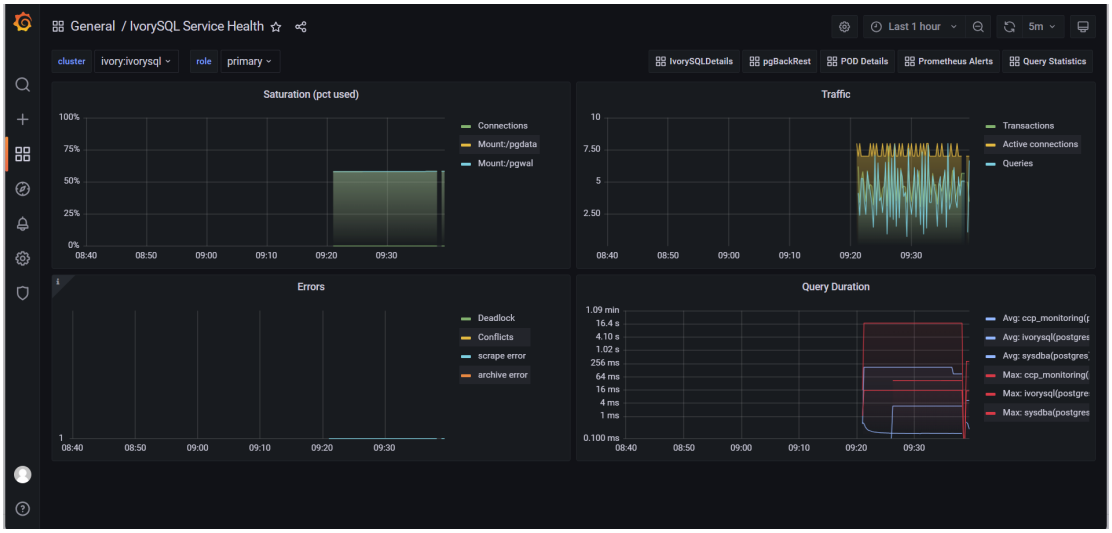


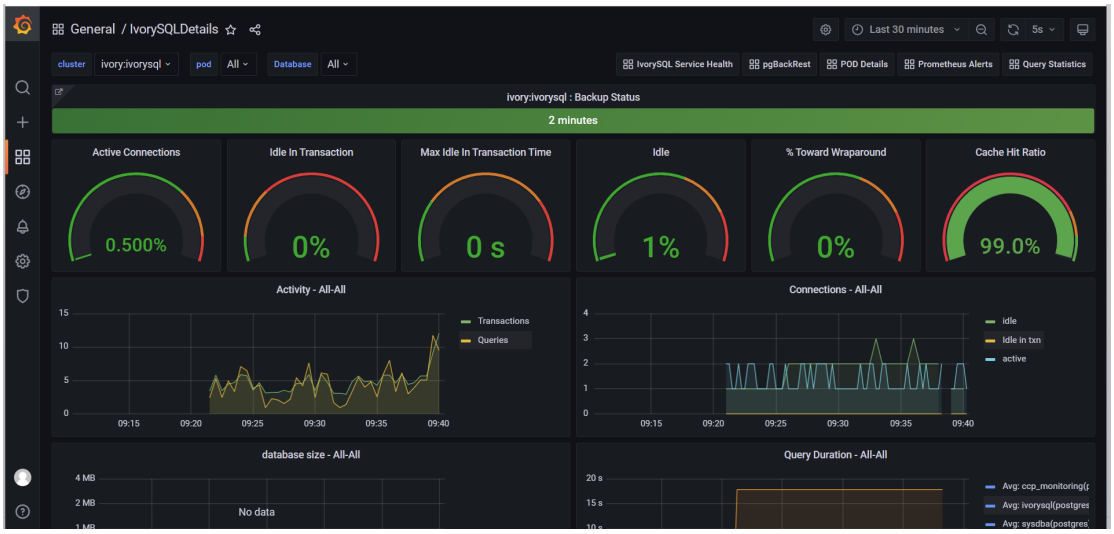
3. After the monitoring stack is created, repeat step (2) to open the monitoring page, then sign in with `admin/admin` and click Login.

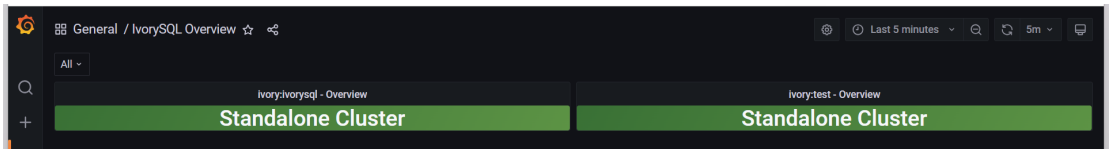


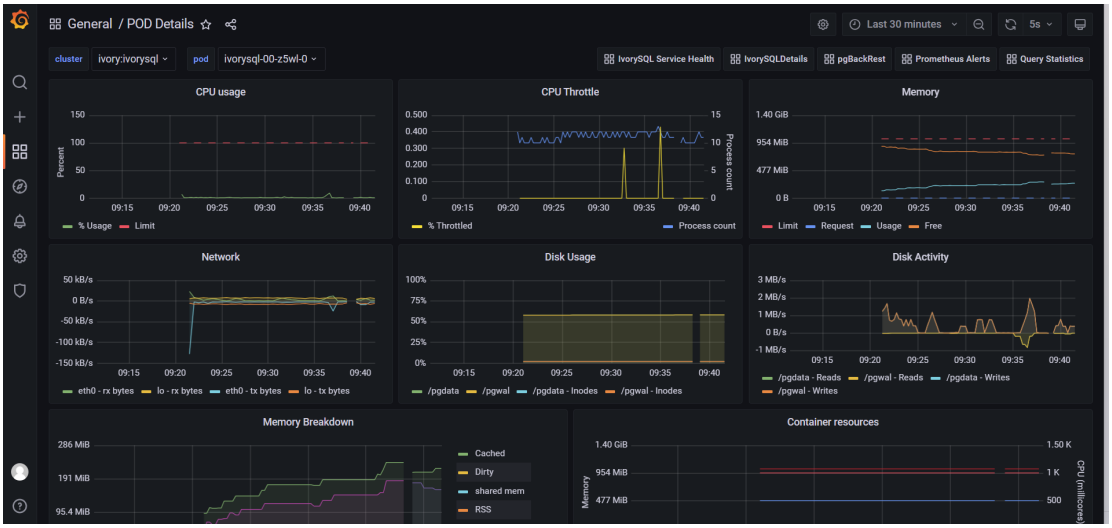
4. Click the magnifying glass icon to view monitoring metrics.

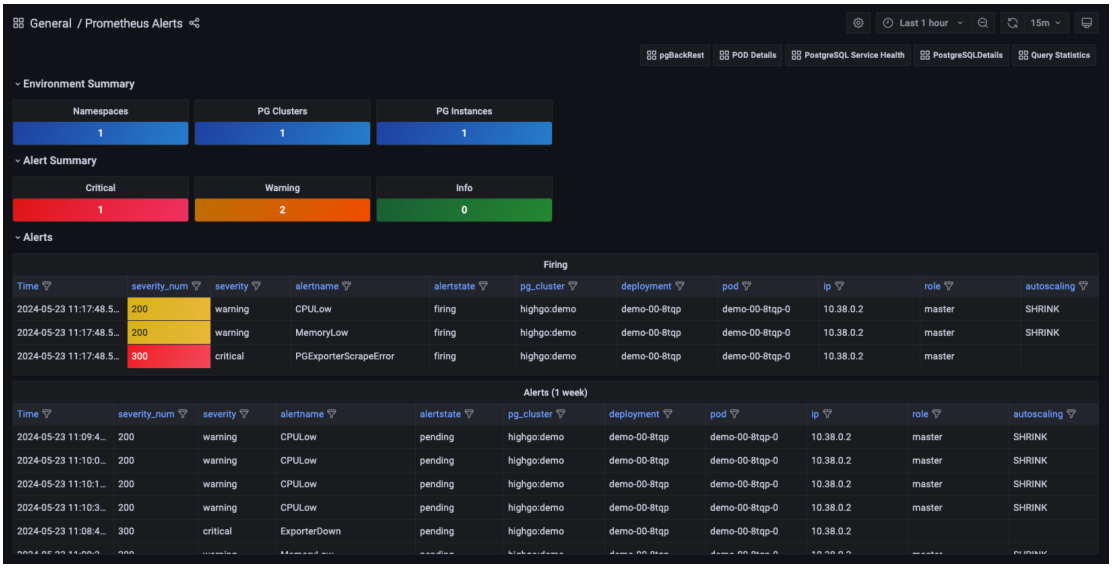

Search dashboards by name
Sort (Default A-Z)
Filter by tag
General
IvorySQL Overview
General
IvorySQL Service Health
General
vendor=ivory
IvorySQLDetails
General
vendor=ivory
pgBackRest
General
vendor=ivory
POD Details
General
vendor=ivory
Prometheus Alerts
General
vendor=ivory
Query Statistics
General
vendor=ivory


General / IvorySQL Service Health
Last 1 hour
5m
cluster
ivory:ivorysql
role
primary
IvorySQLDetails
pgBackRest
POD Details
Prometheus Alerts
Query Statistics
Saturation (pct used)
Connections
Mount:/pgdata
Mount:/pgwal
Traffic
Transactions
Active connections
Queries
Errors
Deadlock
Conflicts
scrape error
archive error
Query Duration


General / IvorySQLDetails
Last 30 minutes
5s
cluster
ivory:ivorysql
pod
All
Database
All
IvorySQL Service Health
pgBackRest
POD Details
Prometheus Alerts
Query Statistics
ivory:ivorysql : Backup Status
2 minutes
Active Connections
0.500%
Idle In Transaction
0%
Max Idle In Transaction Time
0 s
Idle
1%
% Toward Wraparound
0%
Cache Hit Ratio
99.0%
Activity - All-All
Transactions
Queries
Connections - All-All
idle
Idle in txn
active
database size - All-All
No data
Query Duration - All-All


General / IvorySQL Overview
Last 5 minutes
5m
All
ivory:ivorysql - Overview
Standalone Cluster
ivory:test - Overview
Standalone Cluster



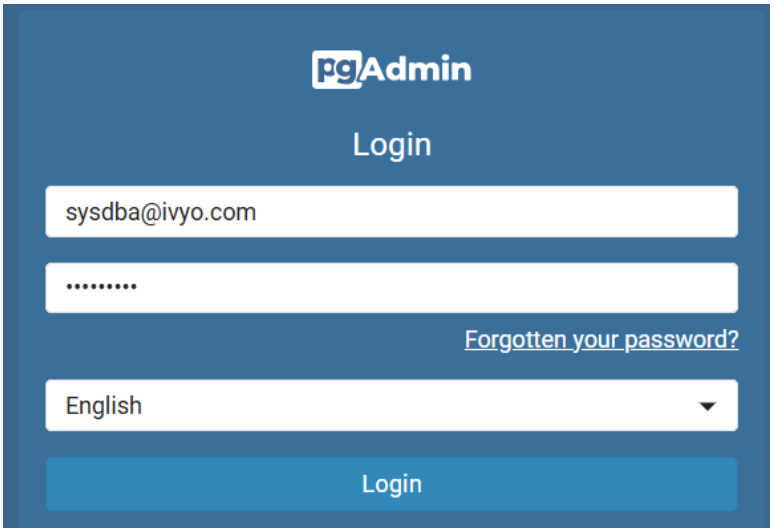


Visual Login Tool

1. Sign in with the `demo` user.
2. Go to Database Management, select a database, click More in the Actions column, and choose Login.



3. On the new page, enter the database account `sysdba@ivyo.com` and the database password, then click Login.



4. Once the connection is established, you can operate on the database.

IvorySQL Ecosystem

.1. CPU Architecture Adaption

IvorySQL adapts the following CPU architectures:

Index	Architecture Name	Adapt to brands	Multi-platform Media Packages
1	x86_64	Intel, AMD, ZHAOXIN, HYGON	deb, rpm
2	aarch64	Phytium, Kunpeng	deb, rpm
3	mips64el	Loongson3000, Loongson4000	deb, rpm
4	loongarch64	Loongson5000	deb, rpm

.2. Operating System Adaption

IvorySQL adapts following operating systems:

Index	OS name	OS introduction	Certification
1	Kylin Advanced Server Operating System V11	Kylin Advanced Server Operating System V11 is a new-generation server operating system tailor-made for enterprise-level critical businesses by Kylin Software, leveraging its years of accumulated technological R&D experience and rich application practice. With high reliability, high availability, high security, high performance, and high scalability as its core advantages, the product, on the basis of in-depth integration of AI technology, has further built a key technology system through independent innovation breakthroughs, including the high-speed network protocol MPTCP, hot patch management, intelligent fault diagnosis, scenario-based optimization, and secure container image repository. It serves as a solid, secure, and reliable supporting foundation for the informatization construction of Party and government organs, the digital transformation of key industries, and the implementation of major national projects.	

2	openKylin 2.0 SP1	OpenAtom openKylin is an open-source project incubated and operated by the OpenAtom Foundation. It was co-founded by basic software and hardware enterprises, non-profit organizations, associations, institutions of higher education, scientific research institutions, and individual developers. With the community vision of "providing the world with an open-source operating system deeply integrated with artificial intelligence technology", the project aims to jointly build a world-leading root community for intelligent desktop open-source operating systems on the basis of openness, voluntariness, equality, and collaboration, and promote the prosperity and development of Linux open-source technology and its software and hardware ecosystem.	

3	OpenAnolis OS 23	Anolis OS 23, the Longxin Operating System, is an operating system developed by the OpenAnolis community based on the Open Source Ecosystem Development and Cooperation Initiative. It independently selects components from upstream native communities, undergoes continuous evolution, and ensures compatibility and stability. As an enterprise-level operating system built on Linux Kernel 6.6 LTS, Anolis OS 23 relies on in-depth optimization of the ANCK 6.6 kernel and fully supports domestic chips such as Haiguang, Phytium, Loongson (LoongArch), and Zhaoxin, as well as general-purpose x86_64/ARM64 architectures. It features specialized enhancements for virtualization, security features, and performance optimization. Through hierarchical architecture design and intelligent tuning tools, it maximizes the performance of hardware-software collaboration. Meanwhile, it natively supports AI ecosystem components and provides secure AI container images, accelerating model development and inference deployment. In terms of development toolchains, it integrates GCC 12.3+/LLVM 17, Python 3.11, Rust, and more, enabling efficient multi-platform development. For the desktop ecosystem, it is compatible with GNOME and DDE desktop environments, and expands the ecosystem to meet diverse scenario needs by integrating the Linglong package manager. Anolis OS 23 supports various common applications and domestic applications, helping enterprises achieve efficient, secure, and reliable digital transformation.	

4	deppin 20	Deepin OS is a Linux distribution dedicated to providing global users with an elegant, user-friendly, secure, and reliable operating system. The official version of Deepin OS 20 (Build 1002) adopts a unified design style, with a redesigned desktop environment and applications, bringing a fresh visual experience. Its underlying repository has been upgraded to Debian 10.5, and the system installation adopts a dual-kernel mechanism (Kernel 5.4, Kernel 5.7), which comprehensively enhances system stability and compatibility. Additionally, it features a newly designed launcher menu, fingerprint recognition, and enhanced system security; some pre-installed applications in the system have been upgraded to the latest versions—all designed to deliver a better experience for you.	

Eco Component Adaption

Overview

IvorySQL Ecosystem Plugin Compatibility List

IvorySQL, as an advanced open-source database compatible with Oracle and based on PostgreSQL, has powerful extension capabilities and supports a rich ecosystem of plugins. These plugins can help users enhance database functionality in different scenarios, including geospatial information processing, vector retrieval, full-text search, data definition extraction, and path planning. The following is a list of major plugins currently officially compatible with and supported by IvorySQL:

+

Index	Plugin Name	Version	Function Description	Use Cases
1	postgis	3.5.4	Provides geospatial data support for IvorySQL, including spatial indexes, spatial functions, and geographic object storage	Geographic Information Systems (GIS), map services, location data analysis
2	pgvector	0.8.1	Supports vector similarity search, can be used to store and retrieve high-dimensional vector data	AI applications, image retrieval, recommendation systems, semantic search
3	pgddl (DDL Extractor)	0.31	Extracts DDL (Data Definition Language) statements from databases, facilitating version management and migration	Database version control, CI/CD integration, structure comparison and synchronization
4	pg_cron	1.6.0	Provides database-internal scheduled task scheduling functionality, supports regular SQL statement execution	Data cleanup, regular statistics, automated maintenance tasks

5	pgsql-http	1.7.0	Allows HTTP requests to be initiated in SQL, interacting with external web services	Data collection, API integration, microservice calls
6	plpgsql_check	2.8	Provides static analysis functionality for PL/pgSQL code, can detect potential errors during development	Stored procedure development, code quality checking, debugging and optimization
7	pgroonga	4.0.4	Provides full-text search functionality for non-English languages, meeting the needs of high-performance applications	Full-text search capabilities for languages like Chinese, Japanese, and Korean
8	pgaudit	18.0	Provides fine-grained auditing, recording database operation logs to support security auditing and compliance checks	Database security auditing, compliance checks, audit report generation
9	pgrouting	3.8.0	Provides routing computation for geospatial data, supporting multiple algorithms and data formats	Geospatial analysis, route planning, logistics optimization
10	system_stats	3.2	Provide functions for accessing system-level statistics.	system monitor

These plugins have all been tested and adapted by the IvorySQL team to ensure stable operation in the IvorySQL environment. Users can select appropriate plugins based on business needs to further enhance the capabilities and flexibility of the database system.

We will continue to expand and enrich the IvorySQL plugin ecosystem. Community developers are welcome to submit new plugin adaptation suggestions or code contributions. For more detailed usage methods and the latest compatible versions of each plugin, please refer to the corresponding documentation chapters for each plugin.

..1. postgis

Overview

IvorySQL is fully compatible with PostgreSQL, allowing for seamless integration with PostGIS.

Installation

Users can select the installation method for PostGIS that best suits their development environment from the PostGIS installation page.

Source Code Installation

In addition to the installation methods provided by the PostGIS community, the IvorySQL community also offers a source code installation method, with Ubuntu 24.04 (x86_64) as the installation environment.



Please ensure that IvorySQL 5.4 or above is installed in the environment.

- Install dependencies

```
sudo apt install \
    docbook-xsl-ns \
    gettext \
```

```
    libgdal-dev \
    libgeos-dev \
    libjson-c-dev \
    libproj-dev \
    libprotobuf-c-dev \
    libsfcgal-dev \
    libxml2-dev \
    libxml2-utils \
    protobuf-c-compiler \
    xsltproc
```

- Install PostGIS

```
$ wget https://download.osgeo.org/postgis/source/postgis-3.5.4.tar.gz
$ tar xvf postgis-3.5.4.tar.gz
$ cd postgis-3.5.4
$ ./configure  --with-pgconfig=/path/to/pg_config   eg: /usr/ivory-5/bin/pg_config,
if ivorysql installation directory is /usr/ivory-5.
$ make
$ sudo make install
```



If configure reports PGXS error, please change --with-pgconfig parameter value and confirm the parameter value based on the installation path of IvorySQL in the environment.

Create Extension and Verify PostGIS Version

Connect to the database with psql and execute the following commands:

```
ivorysql=# CREATE extension postgis;
CREATE EXTENSION

ivorysql=# SELECT * FROM pg_available_extensions WHERE name = 'postgis';
  name   | default_version | installed_version |                          comment
---------+-----------------+-------------------+
------------------------------------------------------------
 postgis | 3.5.4           | 3.5.4             | PostGIS geometry and geography
spatial types and functions
(1 row)
```

Using

For information about using PostGIS, please refer to the PostGIS 3.5 Official Documentation

..2. pgvector

Overview

The vector database is an important component of Generative Artificial Intelligence (GenAI). As a significant extension of PostgreSQL, pgvector not only supports vector calculations of up to 16000 dimensions but also provides powerful vector operations and indexing capabilities, enabling PostgreSQL to directly transform into an efficient vector database. IvorySQL, being developed based on PostgreSQL, inherits the seamless integration capability with pgvector extension, thereby offering users a wider range of data processing and analysis options. Additionally, in Oracle compatibility mode, the pgvector extension is also available, providing great convenience for Oracle users to use vector databases, allowing for easy migration and management of data and achieving more efficient business operations.

Principles

PGVector has two indexing algorithms, IVFFLAT and HNSW.

IVFFLAT

The working principle of IVFFLAT is to cluster similar vectors into regions and build an inverted index mapping each region to its vectors. This allows queries to focus on a subset of the data, enabling fast searches. By adjusting the parameters of lists and probes, IVFFLAT can balance the speed and accuracy of the dataset, enabling PostgreSQL to perform rapid semantic similarity searches on complex data. Through simple queries, applications can find the nearest neighbors to a query vector among millions of high-dimensional vectors. For tasks such as natural language processing and information retrieval, IVFFLAT provides an effective solution.

When building an IVFFLAT index, you need to decide how many lists to include in the index. Each list represents a "center" which are computed using the k-means algorithm. Once all centers are determined, IVFFLAT determines which center each vector is closest to and adds it to the index. When querying vector data, you can decide how many centers to check, which is determined by the ivfflat.probes parameter. This results in a trade-off between ANN performance/recall: the more centers accessed, the more accurate the results, but at the expense of performance.

HNSW

HNSW (Hierarchical Navigating Small World) is a graph-based indexing algorithm consisting of multiple layers of neighborhood graphs, hence the name "hierarchical" NSW method. It constructs multiple layers of navigation graphs for a given graph according to certain rules, with the upper layers of the graph being sparser and the distances between nodes farther apart; and the lower layers of the graph being denser and the distances between nodes closer together. HNSW algorithm is a classic trade-off between space and time, as it achieves high search quality and speed, but at the cost of significant memory overhead. This is because it not only requires storing all vectors in memory but also maintaining the structure of the graph, which also needs to be stored.

Installation



The IvorySQL 5(above version) has been installed in the environment, and the installation path is /usr/local/ivorysql/ivorysql-5

Source Code Installation

- Setting PG_CONFIG

```
export PG_CONFIG=/usr/local/ivorysql/ivorysql-5/bin/pg_config
```

- Pull pg_vector source code

```
git clone --branch v0.8.1 https://github.com/pgvector/pgvector.git
```

- Install pgvector

```
cd pgvector

sudo --preserve-env=PG_CONFIG make
sudo --preserve-env=PG_CONFIG make install
```

- Create pgvector extension

```
[ivorysql@localhost ivorysql-5]$ psql
psql (18.4)
Type "help" for help.

ivorysql=# create extension vector;
CREATE EXTENSION
```

Now, pgvector is installed completely. For more usage cases, please refer to pgvector document

Oracle Compatible

In IvorySQL’s Oracle compatibility mode, the pgvector extension can also work correctly.



We suggest users to test using port 1521, using the command: psql -p 1521.

Data Type

```
ivorysql=# CREATE TABLE items5 (id bigserial PRIMARY KEY, name varchar2(20), num
number(20), embedding bit(3));
CREATE TABLE
ivorysql=# INSERT INTO items5 (name, num, embedding) VALUES ('1st oracle data',0,
'000'), ('2nd oracle data', 111, '111');
INSERT 0 2
ivorysql=# SELECT * FROM items5 ORDER BY bit_count(embedding # '101') LIMIT 5;
 id |      name       | num | embedding
----+-----------------+-----+-----------
  2 | 2nd oracle data | 111 | 111
  1 | 1st oracle data | 0   | 000
```

Anonymous Block

```
ivorysql=# declare
i vector(3) := '[1,2,3]';
```

```
begin
raise notice '%', i;
end;
ivorysql-# /
NOTICE:  [1,2,3]
DO
```

PROCEDURE

```
ivorysql=# CREATE OR REPLACE PROCEDURE ora_procedure()
AS
p vector(3) := '[4,5,6]';
begin
raise notice '%', p;
end;
/
CREATE PROCEDURE
ivorysql=# call ora_procedure();
NOTICE:  [4,5,6]
CALL
```

FUNCTION

```
ivorysql=# CREATE OR REPLACE FUNCTION AddVector(a vector(3), b vector(3))
RETURN vector(3)
IS
BEGIN
RETURN a + b;
END;
/
CREATE FUNCTION
ivorysql=# SELECT AddVector('[1,2,3]','[4,5,6]') FROM DUAL;
 addvector
---------------
 [5,7,9]
(1 row)
```

..3. pgddl(DDL Extractor)

Overview

pgddl is a SQL function extension specifically designed for PostgreSQL databases. It can generate clear, formatted SQL DDL (Data Definition Language) scripts directly from the database system catalog, such as CREATE TABLE or ALTER FUNCTION. It solves the problem that PostgreSQL natively lacks commands like SHOW CREATE TABLE, allowing users to easily obtain object creation statements in a pure SQL environment

without relying on external tools (such as pg_dump).

This extension provides a complete solution through a set of simple SQL functions, with advantages including: requiring only SQL queries to operate, supporting flexible object filtering through WHERE clauses, and intelligently handling dependencies between objects to generate complete scripts including Drop and Create steps. This makes it particularly suitable for scenarios such as database change management, upgrade script writing, and structural auditing.

It should be noted that ddlx is still under development and may not yet cover all PostgreSQL object types and advanced options. Generated scripts should always be checked and tested in non-production environments first to ensure their correctness and safety.

Installation

The IvorySQL installation package already integrates the pgddl plugin. If IvorySQL is installed using the installation package, pgddl can usually be used without manual installation. Other installation methods can refer to the source code installation steps below.



The source installation environment is Ubuntu 24.04 (x86_64). IvorySQL 5 or higher version is already installed in the environment, with the installation path at /usr/local/ivorysql/ivorysql-5

Source Installation

Download pgddl v0.31 code from https://github.com/lacanoid/pgddl.

```
cd pgddl
# Set the PG_CONFIG environment variable to the pg_config path, e.g.:
/usr/local/ivorysql/ivorysql-5/bin/pg_config
make PG_CONFIG=/path/to/pg_config
make PG_CONFIG=/path/to/pg_config install
```

Create Extension and Confirm ddlx Version

Connect to the database with psql and execute the following commands:

```
ivorysql=# CREATE extension ddlx;
CREATE EXTENSION

ivorysql=# SELECT * FROM pg_available_extensions WHERE name = 'ddlx';
 name | default_version | installed_version |         comment
------+-----------------+-------------------+-------------------------
 ddlx | 0.31            | 0.31              | DDL eXtractor functions
(1 row)
```

Usage

For pgddl usage, please refer to the ddlx Official Documentation

..4. pg_cron

Overview

Running periodic tasks in PostgreSQL, such as executing VACUUM or deleting old data, is a common requirement. A simple way to achieve this is to configure cron or other external daemons to periodically connect to the database and run commands. However, as databases increasingly run as managed services or standalone containers, configuring and running a separate daemon often becomes impractical. Additionally, it’ s difficult to make your cron jobs aware of failover or schedule tasks across cluster nodes.

pg_cron is an open-source scheduled task extension for PostgreSQL that allows setting up cron-style task scheduling directly within the database for automating data maintenance tasks (cleanup, aggregation), database health checks, executing stored procedures and custom functions, and other operations. It stores cron jobs in tables, and periodic tasks automatically fail over with the PostgreSQL server. For more details, see pg_cron documentation.

Installation and Configuration



The source installation environment is Ubuntu 24.04 (x86_64). IvorySQL 5 or higher version is already installed in the environment, with the installation path at /usr/local/ivorysql/ivorysql-5

Source Installation

```
# Clone pg_cron source code
git clone https://github.com/citusdata/pg_cron.git
cd pg_cron
# Set pg_config path to PATH environment variable, e.g.:
export PATH=/usr/local/ivorysql/ivorysql-5/bin/:$PATH
make
make install
```

Configuration File (ivorysql.conf)

```
# Shared preload extensions
shared_preload_libraries = 'pg_cron'

# Specify task metadata storage database (default current database)
cron.database_name = 'ivorysql'

# Maximum number of concurrent tasks allowed
cron.max_running_jobs = 5
```

Restart Service

```
pg_ctl restart -D ./data -l logfile
```

Create Extension and Confirm pg_cron Version

Connect to the database with psql and execute the following commands:

```
ivorysql=# CREATE extension pg_cron;
CREATE EXTENSION

ivorysql=# SELECT * FROM pg_available_extensions WHERE name = 'pg_cron';
   name   | default_version | installed_version |         comment
----------+-----------------+-------------------+--------------------------
 pg_cron | 1.6             |                   |Job scheduler for PostgreSQL
(1 row)
```

Core Functionality Usage

Creating Scheduled Tasks

```
SELECT cron.schedule(
    'nightly-data-cleanup',        -- Task name (unique identifier)
    '0 3 * * *',                   -- Cron expression (daily at UTC 3:00)
    $$DELETE FROM logs
      WHERE created_at < now() - interval '30 days'$$  -- SQL to execute
);
```

Cron expression quick reference:

Example	Meaning
'0 * * * *'	Execute every hour on the hour
'*/15 * * * *'	Execute every 15 minutes
'0 9 * * 1-5'	Execute at 9 AM on weekdays
'0 1 1 * *'	Execute at 1 AM on the 1st of every month

pg_cron also allows using '$' to represent the last day of the month.

Task Management

```
# View all tasks
SELECT * FROM cron.job;
```

```
ivorysql=# select jobid, jobname, schedule, command from cron.job order by jobid;
 jobid |        jobname         |  schedule  |                                command
-------+------------------------+------------+-------------------------------------------------------------------
     3 | delete-job-run-details | 0 12 * * * | DELETE FROM cron.job_run_details WHERE end_time < now() - interval '1 hour'
     4 | last-day-of-month-job1 | 0 11 $ * * | vacuum
     5 |                        | 30 seconds | select 1
(3 rows)
```

```
# View task execution history
SELECT * FROM cron.job_run_details ORDER BY start_time DESC LIMIT 10;
```

```
ivorysql=# select jobid, job_pid, database, command, status, start_time, end_time from cron.job_run_details order by start_time desc limit 10;
 jobid | job_pid | database | command  |  status   |          start_time           |           end_time
-------+---------+----------+----------+-----------+-------------------------------+-------------------------------
     5 |  222442 | ivorysql | select 1 | succeeded | 2025-08-21 18:25:38.04872+08  | 2025-08-21 18:25:38.049301+08
     5 |  222436 | ivorysql | select 1 | succeeded | 2025-08-21 18:25:08.043366+08 | 2025-08-21 18:25:08.044139+08
     5 |  222426 | ivorysql | select 1 | succeeded | 2025-08-21 18:24:38.038019+08 | 2025-08-21 18:24:38.038593+08
     5 |  222420 | ivorysql | select 1 | succeeded | 2025-08-21 18:24:08.033158+08 | 2025-08-21 18:24:08.033693+08
     5 |  222411 | ivorysql | select 1 | succeeded | 2025-08-21 18:23:38.028386+08 | 2025-08-21 18:23:38.028805+08
     5 |  222407 | ivorysql | select 1 | succeeded | 2025-08-21 18:23:08.023131+08 | 2025-08-21 18:23:08.023676+08
     5 |  222400 | ivorysql | select 1 | succeeded | 2025-08-21 18:22:38.017147+08 | 2025-08-21 18:22:38.01764+08
(7 rows)
```

```
# Delete task
SELECT cron.unschedule('nightly-data-cleanup');


# Pause task (update status)
UPDATE cron.job SET active = false WHERE jobname = 'delete-job-run-details';
```

..5. pgsql-http

Overview

pgsql-http is an open-source extension designed for PostgreSQL databases that allows users to initiate HTTP requests directly within the database, acting as a built-in web client. The core purpose of this extension is to bridge the gap between databases and external web services, enabling interaction with external web services and API endpoints through simple SQL function calls without relying on external applications or middleware.

With this extension, developers can directly retrieve network data (GET), submit data (POST/PUT), update (PATCH), or delete (DELETE) remote resources in SQL queries, triggers, or stored procedures. It provides rich functionality, including setting request headers, automatic URL encoding, sending JSON data, and parsing response status, headers, and content, greatly simplifying the process of integrating external data into database operations.

Typical application scenarios include: real-time retrieval of external data (such as exchange rates, weather information) and storing it in tables; automatic notification of microservices through triggers when data changes; cleaning data in the database and directly submitting it to external APIs. It provides a powerful and flexible solution for building database-centric integrated applications.

Installation

The pgsql-http plugin has been integrated into the IvorySQL installation package. If IvorySQL is installed using the installation package, pgsql-http can usually be used without manual installation. Other installation methods can refer to the source code installation steps below.



The source installation environment is Ubuntu 24.04 (x86_64). IvorySQL 5 or higher version is already installed in the environment, with the installation path at /usr/local/ivorysql/ivorysql-5

Source Installation

- Install Dependencies

It depends on libcurl, and libcurl development files (such as libcurl4-openssl-dev) need to be installed in advance

```
# Install dependencies
sudo apt install libcurl4-openssl-dev
```

- Compile and Install

Download the 1.7.0 source package pgsql-http-1.7.0.tar.gz from https://github.com/pramsey/pgsql-http/releases/tag/v1.7.0

```
tar xvf pgsql-http-1.7.0.tar.gz
cd pgsql-http-1.7.0
# Ensure pg_config is accessible in PATH, e.g.: /usr/local/ivorysql/ivorysql-5/bin/pg_config
make
sudo make install
```

Create Extension and Confirm http Version

Connect to the database with psql and execute the following commands:

```
ivorysql=# CREATE extension http;
CREATE EXTENSION

ivorysql=# SELECT * FROM pg_available_extensions WHERE name = 'http';
   name    | default_version | installed_version |       comment
-----------+-----------------+-------------------+
------------------------------------------------------------------------
 http      | 1.7             |  1.7              | HTTP client for PostgreSQL, allows
web page retrieval inside the database.
(1 row)
```

Usage

For pgsql-http usage, please refer to pgsql-http official documentation

..6. plpgsql_check

Overview

During PostgreSQL database development, when writing stored procedures and functions, it is often difficult to discover potential issues such as syntax errors, type mismatches, undefined variables, etc., before runtime. Traditional approaches require waiting until the function is actually executed to discover these errors, which not only increases debugging costs but may also cause unexpected failures in production environments.

plpgsql_check is a static code analysis tool (Linter) specifically designed for PostgreSQL’s PL/pgSQL language. It can perform deep checks on the source code of stored procedures and functions without actually executing them. This tool can proactively identify various code quality issues including syntax errors, type mismatches, unused variables, performance problems, security vulnerabilities, etc., helping developers ensure code correctness and robustness during the development phase. For more details, see plpgsql_check official documentation.

Installation



The source installation environment is Ubuntu 24.04 (x86_64). IvorySQL 5 or higher version

> is already installed in the environment, with the installation path at /usr/local/ivorysql/ivorysql-5

Source Installation

```
# Download the 2.8.3 source package plpgsql_check-2.8.3.tar.gz from
https://github.com/okbob/plpgsql_check/releases/tag/v2.8.3
tar xvf plpgsql_check-2.8.3.tar.gz
cd plpgsql_check-2.8.3
# Set pg_config path to PATH environment variable, e.g.:
export PATH=/usr/local/ivorysql/ivorysql-5/bin/:$PATH
make USE_PGXS=1 clean
make USE_PGXS=1 all
sudo make USE_PGXS=1 install
```

Create Extension and Confirm plpgsql_check Version

Connect to the database with psql and execute the following commands:

```
ivorysql=# CREATE EXTENSION plpgsql_check;
CREATE EXTENSION

ivorysql=# SELECT * FROM pg_available_extensions WHERE name = 'plpgsql_check';
     name      | default_version | installed_version |                      comment
---------------+-----------------+-------------------+
---------------------------------------------------
 plpgsql_check | 2.8             | 2.8               | extended check for plpgsql
functions
(1 row)
```

Usage

Check Single Function

```
-- Create a sample function
CREATE OR REPLACE FUNCTION test_function(p_id integer)
RETURNS text AS $$
DECLARE
    v_name text;
    v_unused integer;  -- Unused variable
BEGIN
    SELECT name INTO v_name FROM users WHERE id = p_id;
    RETURN v_naem;  -- Spelling error
END;
$$ LANGUAGE plpgsql;
```

```
-- Use plpgsql_check to check the function
SELECT * FROM plpgsql_check_function('test_function(integer)');
```

Example check result:

```
                  plpgsql_check_function
------------------------------------------------------------
 error:42601:7:assignment:target variable "v_naem" is undefined
 warning:00000:4:DECLARE:unused variable "v_unused"
(2 rows)
```

For more detailed usage methods and advanced features, please refer to plpgsql_check official documentation.

..7. pgroonga

Overview

PostgreSQL has built-in full-text search functionality, but when dealing with large-scale data, complex queries, and non-English languages (especially CJK languages like Chinese, Japanese, and Korean), its functionality and performance may not meet the requirements of high-performance applications.

PGroonga was created to address this need. It is a PostgreSQL extension that deeply integrates Groonga, a high-performance full-featured full-text search engine, with the PostgreSQL database. Groonga itself is an excellent open-source search engine, renowned for its extreme speed and rich functionality, particularly excelling at handling multilingual text. PGroonga's mission is to seamlessly bring Groonga's powerful capabilities into the PostgreSQL world, providing users with an experience that far exceeds native full-text search.

Installation



The PGroonga plugin is already included in the IvorySQL installation package. If you installed IvorySQL using the official released package, you typically do not need to manually install PGroonga and can skip the installation steps.

You can choose their preferred installation method for PGroonga from the PGroonga package installation page.

The IvorySQL community provides source code installation steps, demonstrated below using PGroonga v4.0.4 as an example.

Dependencies

Setup Environment

```
Operating System: Ubuntu 24.04
CPU Architecture: x86_64
IvorySQL: v5.4
```

Install msgpack-c

When compile PGroonga, there is an option: `HAVE_MSGPACK=1`, which is used to support WAL. Enabling this option requires installing msgpack-c 1.4.1 or newer version.

```
sudo apt install libmsgpack-dev
```

Install Groonga

Ensure Groonga >= 14.0.0 is installed.

```
sudo apt install groonga libgroonga-dev
```

Verify Groonga installation:

```
highgo@ubuntu:~/work/IvorySQL/inst$ groonga --version
  Groonga 15.1.7 [Linux,x86_64,utf8,match-escalation-threshold=0,nfkc,mecab,message-
pack,mruby,onigmo,zlib,lz4,zstandard,epoll,apache-
arrow,xxhash,blosc,h3,simdjson,llama.cpp]
```

Compile and Install PGroonga

Download and Extract the PGroonga Source Code

```
wget https://packages.groonga.org/source/pgroonga/pgroonga-4.0.4.tar.gz
tar xvf pgroonga-4.0.4.tar.gz
cd pgroonga-4.0.4
```

Compile

Before running make , ensure that the `pg_config` command is in the `PATH` environment variable. For example, if IvorySQL is installed at `~/work/IvorySQL/inst` , set the environment variables as follows:

```
export PGHOME=~/work/IvorySQL/inst
export PGDATA=$PGHOME/data
export PATH=$PGHOME/bin:$PATH
```

Then execute the following commands to compile and install:

```
make HAVE_MSGPACK=1
make install
```

Create Extension PGroonga and Confirm the version

Connect to the database with `psql` in `pg` mode and execute the following commands:

```
postgres=# CREATE extension pgroonga;
```

```
CREATE EXTENSION
postgres=# SELECT * FROM pg_available_extensions WHERE name = 'pgroonga';
   name   | default_version | installed_version |
comment
----------+-----------------+-------------------+
------------------------------------------------------------------------------
 pgroonga | 4.0.4           | 4.0.4             | Super fast and all languages
supported full text search index based on Groonga
(1 row)

postgres=# select version();
                                  version
------------------------------------------------------------------------------
 PostgreSQL (IvorySQL 5.4) 18.0 on x86_64-linux, compiled by gcc-13.3.0, 64-bit
(1 row)
```

Usage

For PGroonga usage, please refer to the PGroonga Official Documentation

..8. pgaudit

Overview

PgAudit is an auditing extension for IvorySQL that produces traceable log records for critical operations such as DDL, DML, and DCL. With the audit trail, database administrators can meet compliance requirements, quickly detect abnormal behavior, and identify accountability and impact scope when incidents occur.

Key Features

- Comprehensive auditing: Captures `SELECT`, `INSERT`, `UPDATE`, `DELETE`, DDL commands, privilege changes, and more to build a complete activity timeline.
- Flexible scope control: Supports global, role-based, and object-level auditing, allowing fine-grained configuration by user, role, schema, or operation type.
- Seamless integration: Reuses PostgreSQL’s standard logging subsystem and works with tools like `syslog` and `logrotate`, aligning with existing log ingestion and analysis pipelines.
- Compliance ready: Generates structured audit logs suitable for meeting regulatory requirements in finance, government, and other regulated industries.
- Security enhancement: Records and inspects database activity to surface unauthorized access, anomalous DML, or potential data leakage risks in time.
- Operations insight: Helps replay operational actions, locate performance bottlenecks, and support SQL tuning and incident troubleshooting.

Installation and Deployment

Prerequisites

- A IvorySQL installation (recommended version aligned with the targeted PgAudit release).
- Build toolchain: `gcc`, `make`, `tar`, etc.
- Database superuser privileges to modify `ivorysql.conf` and restart the instance.

Compile and Install PgAudit

Taking PgAudit 18.0 as an example:

```
wget https://github.com/pgaudit/pgaudit/archive/refs/tags/18.0.tar.gz
tar -xf 18.0.tar.gz
cd pgaudit-18.0
make install USE_PGXS=1 PG_CONFIG=$PGHOME/bin/pg_config
```

The commands above expect the environment variable `PGHOME` to point to the installed IvorySQL home directory. After installation, `pgaudit.so` will be placed in IvorySQL's extension directory.

Baseline Configuration Before Registering the Extension

1. Modify `ivorysql.conf` to load the plugin and configure common parameters: ---- shared_preload_libraries = 'pgaudit' # Requires an instance restart pgaudit.log = 'read, write, ddl' # Sample audit scope; adjust as needed ----
2. Restart or reload the database instance so the shared library configuration takes effect.

Create the Extension and Verify

```
CREATE EXTENSION IF NOT EXISTS pgaudit;
SELECT name,
       default_version,
       installed_version,
       comment
  FROM pg_available_extensions
 WHERE name = 'pgaudit';
```

If the returned `installed_version` matches the expected release, the extension has been installed successfully.

Usage

Execute the following SQL sample:

```
[source,sql]
----
CREATE TABLE audit_demo(id serial PRIMARY KEY, info text);
INSERT INTO audit_demo(info) VALUES ('pgaudit test');
SELECT * FROM audit_demo;
UPDATE audit_demo SET info = 'pgaudit update' WHERE id = 1;
DELETE FROM audit_demo WHERE id = 1;
----
```

Check the audit logs on the database server:

```
tail -f $PGDATA/log/*.log | grep 'AUDIT:'
```

```
2025-10-31 15:56:32.113 CST [11451] LOG:  AUDIT: SESSION,1,1,DDL,CREATE
SEQUENCE,SEQUENCE,public.audit_demo_id_seq,"CREATE TABLE audit_demo(id serial PRIMARY
KEY, info text)",<not logged>
2025-10-31 15:56:32.113 CST [11451] LOG:  AUDIT: SESSION,1,1,DDL,CREATE
TABLE,TABLE,public.audit_demo,"CREATE TABLE audit_demo(id serial PRIMARY KEY, info
text)",<not logged>
2025-10-31 15:56:32.113 CST [11451] LOG:  AUDIT: SESSION,1,1,DDL,CREATE
INDEX,INDEX,public.audit_demo_pkey,"CREATE TABLE audit_demo(id serial PRIMARY KEY,
info text)",<not logged>
2025-10-31 15:56:32.113 CST [11451] LOG:  AUDIT: SESSION,1,1,DDL,ALTER
SEQUENCE,SEQUENCE,public.audit_demo_id_seq,"CREATE TABLE audit_demo(id serial PRIMARY
KEY, info text)",<not logged>
2025-10-31 15:56:32.117 CST [11451] LOG:  AUDIT: SESSION,2,1,WRITE,INSERT,,,INSERT
INTO audit_demo(info) VALUES ('pgaudit test'),<not logged>
2025-10-31 15:56:32.121 CST [11451] LOG:  AUDIT: SESSION,3,1,READ,SELECT,,,SELECT *
FROM audit_demo,<not logged>
2025-10-31 15:56:32.122 CST [11451] LOG:  AUDIT: SESSION,4,1,WRITE,UPDATE,,,UPDATE
audit_demo SET info = 'pgaudit update' WHERE id = 1,<not logged>
2025-10-31 15:56:32.127 CST [11451] LOG:  AUDIT: SESSION,5,1,WRITE,DELETE,,,DELETE
FROM audit_demo WHERE id = 1,<not logged>
```

To record parameter values as well, enable `pgaudit.log_parameter = 'on'`:

```
ivorysql=# SHOW pgaudit.log_parameter;
pgaudit.log_parameter
----------------------
on
(1 row)
```

..9. pgrouting

Overview

pgRouting is an open-source geospatial routing extension library built on PostgreSQL/PostGIS databases. It endows databases with powerful network analysis capabilities, enabling them to handle complex path planning and graph theory computation problems, such as calculating the shortest path between two points, performing Traveling Salesman Problem (TSP) analysis, or computing service area coverage. It embeds routing algorithms directly into the database, thereby avoiding complex data transfer and computation at the application layer.

The core advantage of this extension lies in its ability to leverage PostgreSQL’s powerful data management capabilities and PostGIS’s rich spatial functions to perform efficient computation on spatial network data directly within the database. This not only simplifies application development processes but also significantly improves the performance of large-scale network analysis by reducing data movement.

pgRouting is widely used in logistics and distribution, traffic navigation, network analysis, urban planning, and supply chain management, among other fields. Its open-source nature has attracted continuous

contributions and improvements from developers worldwide, making it one of the preferred tools for path analysis and network solving in the spatial database domain.

Installation



IvorySQL 5.4 or higher version is already installed in the environment, with the installation path at /usr/local/ivorysql/ivorysql-5

Source Installation

- Install dependencies

It depends on perl, which is generally already installed when installing IvorySQL, so no need to install it here. CMake version requirement >= 3.12, Boost version >= 1.56

```
# Install dependencies
sudo apt install cmake libboost-all-dev
```

- Compile and install

```
wget https://github.com/pgRouting/pgrouting/releases/download/v3.8.0/pgrouting-
3.8.0.tar.gz
tar xvf pgrouting-3.8.0.tar.gz
cd pgrouting-3.8.0
mkdir build
cd build
cmake .. -DPOSTGRESQL_PG_CONFIG=/path/to/pg_config  # eg:
/usr/local/ivorysql/ivorysql-5/bin/pg_config
make
sudo make install
```

Create Extension and Confirm pgRouting Version

Connect to the database with psql and execute the following commands:

```
ivorysql=# CREATE extension pgrouting;
CREATE EXTENSION

ivorysql=# SELECT * FROM pg_available_extensions WHERE name = 'pgrouting';
   name    | default_version | installed_version |       comment
-----------+-----------------+-------------------+---------------------
 pgrouting | 3.8.0           |                   | pgRouting Extension
(1 row)
```

Usage

For pgRouting usage, please refer to the pgRouting Official Documentation

..10. system_stats

Overview

system_stats is a Postgres extension that provides functions to access system level statistics that can be used for monitoring.

Installation



The source code installation environment is Ubuntu 24.04 (x86_64), in which IvorySQL 5 or a later version has been installed. The installation path is /usr/local/ivorysql/ivorysql-5.

Source Code Installation

```
# download source code package from:
https://github.com/EnterpriseDB/system_stats/releases/tag/v3.2
unzip v3.2.zip
cd system_stats-3.2

# compile and install the extension
make PG_CONFIG=/usr/local/ivorysql/ivorysql-5/bin/pg_config
make PG_CONFIG=/usr/local/ivorysql/ivorysql-5/bin/pg_config install
```

Create extension and confirm the version

connect the database with psql in Oracle compatible mode and execute the following commands:

```
ivorysql=# CREATE EXTENSION system_stats;
CREATE EXTENSION

ivorysql=# SELECT * FROM pg_available_extensions WHERE name = 'system_stats';
     name     | default_version | installed_version |                    comment
--------------+-----------------+-------------------+
-----------------------------------------------
 system_stats | 3.0             | 3.0               | EnterpriseDB system statistics
for PostgreSQL
(1 row)
```

使用

```
ivorysql=> select pg_sys_os_info() from dual;
                             pg_sys_os_info
------------------------------------------------------------------------
 """Ubuntu 24.04.1 LTS""                                                +
 ","Linux 6.14.0-29-generic",Ubuntu,"(none)",5568,292,532,x86_64,,864122)
(1 row)
```

```
ivorysql=> select pg_sys_cpu_info() from dual;
                                                                                pg_sys_cpu_info
-------------------------------------------------------------------------------------------------------------------
 (GenuineIntel,"GenuineIntel model 154 family 6","12th Gen Intel(R) Core(TM) i7-
1260P",,0,2,1,x86_64,2496000000,,,,48,32,1280,18432)
(1 row)


ivorysql=> select pg_sys_cpu_usage_info() from dual;
 pg_sys_cpu_usage_info
-----------------------
 (0,0,0,100,0,0,0,,,,)
(1 row)


ivorysql=> select pg_sys_memory_info() from dual;
                                pg_sys_memory_info
-----------------------------------------------------------------------------------
 (4055482368,3911159808,144322560,4055887872,233725952,3822161920,2638602240,,,,,)
(1 row)


ivorysql=> select pg_sys_io_analysis_info() from dual;
                    pg_sys_io_analysis_info
-----------------------------------------------------------------
 (loop0,15,0,21504,0,1,0)
 (loop1,1362,0,27916288,0,680,0)
 (loop2,175,0,3354624,0,131,0)
 (loop3,141,0,1827840,0,83,0)
 (loop4,918,0,32752640,0,212,0)
 (loop5,156,0,2617344,0,67,0)
 (loop6,3707,0,131992576,0,828,0)
 (loop7,243,0,4321280,0,95,0)
 (fd0,0,0,0,0,0,0)
 (sda,639286,1580583,26547332096,240138948608,402702,2744653)
 (sda1,561,0,22966272,0,61,0)
 (sda2,636820,1580583,26491053056,240138948608,402316,2744653)
 (sr0,87,0,3219456,0,45,0)
 (sr1,0,0,0,0,0,0)
 (loop8,228,0,2491392,0,120,0)
```

```
 (loop10,2333,0,36571136,0,1628,0)
 (loop9,3730,0,66532352,0,3973,0)
 (loop11,87,0,709632,0,50,0)
 (loop12,156,0,1948672,0,52,0)
 (loop14,2579,0,97446912,0,2083,0)
 (loop13,30,0,102400,0,14,0)
 (loop15,82,0,1189888,0,117,0)
 (loop16,1357,0,28335104,0,500,0)
 (loop17,110,0,2187264,0,93,0)
 (loop18,129,0,2355200,0,28,0)
 (loop19,110,0,2202624,0,55,0)
(26 rows)


ivorysql=> select pg_sys_disk_info() from dual;
                                pg_sys_disk_info
--------------------------------------------------------------------------------
 (/,/dev/sda2,,,ext4,105086115840,50903396352,48797392896,6553600,619735,5933865)
(1 row)


ivorysql=> select pg_sys_load_avg_info() from dual;
 pg_sys_load_avg_info
----------------------
 (0,0,0,)
(1 row)


ivorysql=> select pg_sys_process_info() from dual;
 pg_sys_process_info
---------------------
 (294,1,201,0,0)
(1 row)


ivorysql=> select pg_sys_network_info() from dual;
                            pg_sys_network_info
--------------------------------------------------------------------------
 (lo,127.0.0.1,1071575,5643,0,0,1071575,5643,0,0,0)
 (ens33,192.168.198.128,619877515,1490961,0,161,5540250106,4345449,0,0,1000)
(2 rows)
```

```
ivorysql=> select pg_sys_cpu_memory_by_process() from dual;
                     pg_sys_cpu_memory_by_process
-------------------------------------------------------------------------
 (1,"(systemd)",864793,0,0.34,13840384)
 (2,"(kthreadd)",864793,0,0,0)
 (3,"(pool_workqueue_release)",864793,0,0,0)
 (4,"(kworker/R-rcu_gp)",864793,0,0,0)
......
```

Monitor and O&M

.1. Monitoring

Monitoring Database Activity

Standard Unix Tools

On most Unix platforms, IvorySQL modifies its command title as reported by `ps`, so that individual server processes can readily be identified. A sample display is

```
$ ps auxww | grep ^postgres
postgres  15551  0.0  0.1  57536  7132 pts/0    S    18:02   0:00 postgres -i
postgres  15554  0.0  0.0  57536  1184 ?        Ss   18:02   0:00 postgres: background
writer
postgres  15555  0.0  0.0  57536   916 ?        Ss   18:02   0:00 postgres:
checkpointer
postgres  15556  0.0  0.0  57536   916 ?        Ss   18:02   0:00 postgres: walwriter
postgres  15557  0.0  0.0  58504  2244 ?        Ss   18:02   0:00 postgres: autovacuum
launcher
postgres  15558  0.0  0.0  17512  1068 ?        Ss   18:02   0:00 postgres: stats
collector
postgres  15582  0.0  0.0  58772  3080 ?        Ss   18:04   0:00 postgres: joe runbug
127.0.0.1 idle
postgres  15606  0.0  0.0  58772  3052 ?        Ss   18:07   0:00 postgres: tgl
regression [local] SELECT waiting
postgres  15610  0.0  0.0  58772  3056 ?        Ss   18:07   0:00 postgres: tgl
regression [local] idle in transaction
```

(The appropriate invocation of `ps` varies across different platforms, as do the details of what is shown. This example is from a recent Linux system.) The first process listed here is the primary server process. The command arguments shown for it are the same ones used when it was launched. The next four processes are background worker processes automatically launched by the primary process. (The "autovacuum launcher" process will not be present if you have set the system not to run autovacuum.) Each of the remaining processes is a server process handling one client connection. Each such process sets its command line display in the form

```
postgres: user database host activity
```

The user, database, and (client) host items remain the same for the life of the client connection, but the activity indicator changes. The activity can be `idle` (i.e., waiting for a client command), `idle in transaction` (waiting for client inside a `BEGIN` block), or a command type name such as `SELECT`. Also, `waiting` is appended if the server process is presently waiting on a lock held by another session. In the above example we can infer that process 15606 is waiting for process 15610 to complete its transaction and thereby release some lock. (Process 15610 must be the blocker, because there is no other active session. In more complicated cases it would be necessary to look into the pg_locks system view to determine who is blocking whom.)

If cluster_name has been configured the cluster name will also be shown in `ps` output:

```
$ psql -c 'SHOW cluster_name'
```

```
 cluster_name
--------------
 server1
(1 row)

$ ps aux|grep server1
postgres   27093  0.0  0.0  30096  2752 ?        Ss   11:34   0:00 postgres: server1:
background writer
...
```

If you have turned off update_process_title then the activity indicator is not updated; the process title is set only once when a new process is launched. On some platforms this saves a measurable amount of per-command overhead; on others it’s insignificant.

> **Tip**
>
> Solaris requires special handling. You must use /usr/ucb/ps, rather than /bin/ps. You also must use two w flags, not just one. In addition, your original invocation of the postgres command must have a shorter ps status display than that provided by each server process. If you fail to do all three things, the ps output for each server process will be the original postgres command line.

The Cumulative Statistics System

IvorySQL’s cumulative statistics system supports collection and reporting of information about server activity. Presently, accesses to tables and indexes in both disk-block and individual-row terms are counted. The total number of rows in each table, and information about vacuum and analyze actions for each table are also counted. If enabled, calls to user-defined functions and the total time spent in each one are counted as well.

IvorySQL also supports reporting dynamic information about exactly what is going on in the system right now, such as the exact command currently being executed by other server processes, and which other connections exist in the system. This facility is independent of the cumulative statistics system.

Statistics Collection Configuration

Since collection of statistics adds some overhead to query execution, the system can be configured to collect or not collect information. This is controlled by configuration parameters that are normally set in **postgresql.conf**.

The parameter track_activities enables monitoring of the current command being executed by any server process.

The parameter track_counts controls whether cumulative statistics are collected about table and index accesses.

The parameter track_functions enables tracking of usage of user-defined functions.

The parameter track_io_timing enables monitoring of block read and write times.

The parameter track_wal_io_timing enables monitoring of WAL write times.

Normally these parameters are set in **postgresql.conf** so that they apply to all server processes, but it is possible to turn them on or off in individual sessions using the SET command. (To prevent ordinary users from hiding their activity from the administrator, only superusers are allowed to change these parameters with **SET**.)

Cumulative statistics are collected in shared memory. Every IvorySQL process collects statistics locally, then updates the shared data at appropriate intervals. When a server, including a physical replica, shuts down cleanly, a permanent copy of the statistics data is stored in the `pg_stat` subdirectory, so that statistics can be retained across server restarts. In contrast, when starting from an unclean shutdown (e.g., after an immediate shutdown, a server crash, starting from a base backup, and point-in-time recovery), all statistics counters are reset.

Viewing Statistics

Several predefined views, listed in Table 1 , are available to show the current state of the system. There are also several other views, listed in Table 2 , available to show the accumulated statistics. Alternatively, one can build custom views using the underlying cumulative statistics functions.

When using the cumulative statistics views and functions to monitor collected data, it is important to realize that the information does not update instantaneously. Each individual server process flushes out accumulated statistics to shared memory just before going idle, but not more frequently than once per `PGSTAT_MIN_INTERVAL` milliseconds (1 second unless altered while building the server); so a query or transaction still in progress does not affect the displayed totals and the displayed information lags behind actual activity. However, current-query information collected by `track_activities` is always up-to-date.

Another important point is that when a server process is asked to display any of the accumulated statistics, accessed values are cached until the end of its current transaction in the default configuration. So the statistics will show static information as long as you continue the current transaction. Similarly, information about the current queries of all sessions is collected when any such information is first requested within a transaction, and the same information will be displayed throughout the transaction. This is a feature, not a bug, because it allows you to perform several queries on the statistics and correlate the results without worrying that the numbers are changing underneath you. When analyzing statistics interactively, or with expensive queries, the time delta between accesses to individual statistics can lead to significant skew in the cached statistics. To minimize skew, `stats_fetch_consistency` can be set to `snapshot`, at the price of increased memory usage for caching not-needed statistics data. Conversely, if it's known that statistics are only accessed once, caching accessed statistics is unnecessary and can be avoided by setting `stats_fetch_consistency` to `none`. You can invoke `pg_stat_clear_snapshot`() to discard the current transaction's statistics snapshot or cached values (if any). The next use of statistical information will (when in snapshot mode) cause a new snapshot to be built or (when in cache mode) accessed statistics to be cached.

A transaction can also see its own statistics (not yet flushed out to the shared memory statistics) in the views `pg_stat_xact_all_tables`, `pg_stat_xact_sys_tables`, `pg_stat_xact_user_tables`, and `pg_stat_xact_user_functions`. These numbers do not act as stated above; instead they update continuously throughout the transaction.

Some of the information in the dynamic statistics views shown in Table 1 is security restricted. Ordinary users can only see all the information about their own sessions (sessions belonging to a role that they are a member of). In rows about other sessions, many columns will be null. Note, however, that the existence of a session and its general properties such as its sessions user and database are visible to all users. Superusers and roles with privileges of built-in role `pg_read_all_stats` can see all the information about all sessions.

Dynamic Statics Views

View Name	Description
`pg_stat_activity`	One row per server process, showing information related to the current activity of that process, such as state and current query.
`pg_stat_replication`	One row per WAL sender process, showing statistics about replication to that sender's connected standby server.
`pg_stat_wal_receiver`	Only one row, showing statistics about the WAL receiver from that receiver's connected server.
`pg_stat_recovery_prefetch`	Only one row, showing statistics about blocks prefetched during recovery.

pg_stat_subscription	At least one row per subscription, showing information about the subscription workers.
pg_stat_ssl	One row per connection (regular and replication), showing information about SSL used on this connection.
pg_stat_gssapi	One row per connection (regular and replication), showing information about GSSAPI authentication and encryption used on this connection.
pg_stat_progress_analyze	One row for each backend (including autovacuum worker processes) running ANALYZE, showing current progress.
pg_stat_progress_create_index	One row for each backend running CREATE INDEX or REINDEX, showing current progress.
pg_stat_progress_vacuum	One row for each backend (including autovacuum worker processes) running VACUUM, showing current progress.
pg_stat_progress_cluster	One row for each backend running CLUSTER or VACUUM FULL, showing current progress.
pg_stat_progress_basebackup	One row for each WAL sender process streaming a base backup, showing current progress.
pg_stat_progress_copy	One row for each backend running COPY, showing current progress.

Collected Statistics Views

View Name	Description
pg_stat_archiver	One row only, showing statistics about the WAL archiver process’ s activity. See pg_stat_archiver for details.
pg_stat_bgwriter	One row only, showing statistics about the background writer process’ s activity. See pg_stat_bgwriter for details.
pg_stat_wal	One row only, showing statistics about WAL activity. See pg_stat_wal for details.
pg_stat_database	One row per database, showing database-wide statistics. See pg_stat_database for details.
pg_stat_database_conflicts	One row per database, showing database-wide statistics about query cancels due to conflict with recovery on standby servers. See pg_stat_database_conflicts for details.
pg_stat_all_tables	One row for each table in the current database, showing statistics about accesses to that specific table. See pg_stat_all_tables for details.
pg_stat_sys_tables	Same as pg_stat_all_tables, except that only system tables are shown.
pg_stat_user_tables	Same as pg_stat_all_tables, except that only user tables are shown.
pg_stat_xact_all_tables	Similar to pg_stat_all_tables, but counts actions taken so far within the current transaction (which are not yet included in pg_stat_all_tables and related views). The columns for numbers of live and dead rows and vacuum and analyze actions are not present in this view.

pg_stat_xact_sys_tables	Same as pg_stat_xact_all_tables, except that only system tables are shown.
pg_stat_xact_user_tables	Same as pg_stat_xact_all_tables, except that only user tables are shown.
pg_stat_all_indexes	One row for each index in the current database, showing statistics about accesses to that specific index. See pg_stat_all_indexes for details.
pg_stat_sys_indexes	Same as pg_stat_all_indexes, except that only indexes on system tables are shown.
pg_stat_user_indexes	Same as pg_stat_all_indexes, except that only indexes on user tables are shown.
pg_statio_all_tables	One row for each table in the current database, showing statistics about I/O on that specific table. See pg_statio_all_tables for details.
pg_statio_sys_tables	Same as pg_statio_all_tables, except that only system tables are shown.
pg_statio_user_tables	Same as pg_statio_all_tables, except that only user tables are shown.
pg_statio_all_indexes	One row for each index in the current database, showing statistics about I/O on that specific index. See pg_statio_all_indexes for details.
pg_statio_sys_indexes	Same as pg_statio_all_indexes, except that only indexes on system tables are shown.
pg_statio_user_indexes	Same as pg_statio_all_indexes, except that only indexes on user tables are shown.
pg_statio_all_sequences	One row for each sequence in the current database, showing statistics about I/O on that specific sequence. See pg_statio_all_sequences for details.
pg_statio_sys_sequences	Same as pg_statio_all_sequences, except that only system sequences are shown. (Presently, no system sequences are defined, so this view is always empty.)
pg_statio_user_sequences	Same as pg_statio_all_sequences, except that only user sequences are shown.
pg_stat_user_functions	One row for each tracked function, showing statistics about executions of that function. See pg_stat_user_functions for details.
pg_stat_xact_user_functions	Similar to pg_stat_user_functions, but counts only calls during the current transaction (which are not yet included in pg_stat_user_functions).
pg_stat_slru	One row per SLRU, showing statistics of operations. See pg_stat_slru for details.
pg_stat_replication_slots	One row per replication slot, showing statistics about the replication slot' s usage. See pg_stat_replication_slots for details.
pg_stat_subscription_stats	One row per subscription, showing statistics about errors. See pg_stat_subscription_stats for details.

The per-index statistics are particularly useful to determine which indexes are being used and how effective they are.

The pg_statio_ views are primarily useful to determine the effectiveness of the buffer cache. When the number of actual disk reads is much smaller than the number of buffer hits, then the cache is satisfying most read requests without invoking a kernel call. However, these statistics do not give the entire story: due to the

way in which IvorySQL handles disk I/O, data that is not in the IvorySQL buffer cache might still reside in the kernel’ s I/O cache, and might therefore still be fetched without requiring a physical read. Users interested in obtaining more detailed information on IvorySQL I/O behavior are advised to use the IvorySQL statistics views in combination with operating system utilities that allow insight into the kernel’ s handling of I/O.

pg_stat_activity

The **pg_stat_activity** view will have one row per server process, showing information related to the current activity of that process.

pg_stat_activity View

Column	Type	Description	
datid	oid	OID of the database this backend is connected to	
datname	name	Name of the database this backend is connected to	
pid	integer	Process ID of this backend	
leader_pid	integer	Process ID of the parallel group leader, if this process is a parallel query worker. NULL if this process is a parallel group leader or does not participate in parallel query.	
usesysid	oid	OID of the user logged into this backend	
usename	name	Name of the user logged into this backend	
application_name	text	Name of the application that is connected to this backend	
client_addr	inet	IP address of the client connected to this backend. If this field is null, it indicates either that the client is connected via a Unix socket on the server machine or that this is an internal process such as autovacuum.	
client_hostname	text	Host name of the connected client, as reported by a reverse DNS lookup of client_addr. This field will only be non-null for IP connections, and only when log_hostname is enabled.	

client_port	integer	TCP port number that the client is using for communication with this backend, or -1 if a Unix socket is used. If this field is null, it indicates that this is an internal server process.	
backend_start	timestamp with time zone	Time when this process was started. For client backends, this is the time the client connected to the server.	
xact_start	timestamp with time zone	Time when this process' current transaction was started, or null if no transaction is active. If the current query is the first of its transaction, this column is equal to the query_start column.	
query_start	timestamp with time zone	Time when the currently active query was started, or if state is not active, when the last query was started	
state_change	timestamp with time zone	Time when the state was last changed	
wait_event_type	text	The type of event for which the backend is waiting, if any; otherwise NULL.	
wait_event	text	Wait event name if backend is currently waiting, otherwise NULL.	

state	text	Current overall state of this backend. Possible values are: active: The backend is executing a query. idle: The backend is waiting for a new client command. idle in transaction: The backend is in a transaction, but is not currently executing a query. idle in transaction (aborted): This state is similar to idle in transaction, except one of the statements in the transaction caused an error. fastpath function call: The backend is executing a fast-path function. disabled: This state is reported if track_activities is disabled in this backend.	
backend_xid	xid	Top-level transaction identifier of this backend, if any.	
backend_xmin	xid	The current backend's xmin horizon.	
query_id	bigint	Identifier of this backend's most recent query. If state is active this field shows the identifier of the currently executing query. In all other states, it shows the identifier of last query that was executed. Query identifiers are not computed by default so this field will be null unless compute_query_id parameter is enabled or a third-party module that computes query identifiers is configured.	

query	text	Text of this backend's most recent query. If state is active this field shows the currently executing query. In all other states, it shows the last query that was executed. By default the query text is truncated at 1024 bytes; this value can be changed via the parameter track_activity_query_size.	
backend_type	text	Type of current backend. Possible types are autovacuum launcher, autovacuum worker, logical replication launcher, logical replication worker, parallel worker, background writer, client backend, checkpointer, archiver, startup, walreceiver, walsender and walwriter. In addition, background workers registered by extensions may have additional types.	

Note

The wait_event and state columns are independent. If a backend is in the active state, it may or may not be waiting on some event. If the state is active and wait_event is non-null, it means that a query is being executed, but is being blocked somewhere in the system.

Wait Event Types

Wait Event Type	Description
Activity	The server process is idle. This event type indicates a process waiting for activity in its main processing loop. wait_event will identify the specific wait point
BufferPin	The server process is waiting for exclusive access to a data buffer. Buffer pin waits can be protracted if another process holds an open cursor that last read data from the buffer in question.
Client	The server process is waiting for activity on a socket connected to a user application. Thus, the server expects something to happen that is independent of its internal processes. wait_event will identify the specific wait point.
Extension	The server process is waiting for some condition defined by an extension module.
IO	The server process is waiting for an I/O operation to complete. wait_event will identify the specific wait point.

`IPC`	The server process is waiting for some interaction with another server process. `wait_event` will identify the specific wait point.
`Lock`	The server process is waiting for a heavyweight lock. Heavyweight locks, also known as lock manager locks or simply locks, primarily protect SQL-visible objects such as tables. However, they are also used to ensure mutual exclusion for certain internal operations such as relation extension. `wait_event` will identify the type of lock awaited.
`LWLock`	The server process is waiting for a lightweight lock. Most such locks protect a particular data structure in shared memory. `wait_event` will contain a name identifying the purpose of the lightweight lock. (Some locks have specific names; others are part of a group of locks each with a similar purpose.) .
`Timeout`	The server process is waiting for a timeout to expire. `wait_event` will identify the specific wait point.

Wait Events of Type `Activity`

`Activity` Wait Event	Description
`ArchiverMain`	Waiting in main loop of archiver process.
`AutoVacuumMain`	Waiting in main loop of autovacuum launcher process.
`BgWriterHibernate`	Waiting in background writer process, hibernating.
`BgWriterMain`	Waiting in main loop of background writer process.
`CheckpointerMain`	Waiting in main loop of checkpointer process.
`LogicalApplyMain`	Waiting in main loop of logical replication apply process.
`LogicalLauncherMain`	Waiting in main loop of logical replication launcher process.
`RecoveryWalStream`	Waiting in main loop of startup process for WAL to arrive, during streaming recovery.
`SysLoggerMain`	Waiting in main loop of syslogger process.
`WalReceiverMain`	Waiting in main loop of WAL receiver process.
`WalSenderMain`	Waiting in main loop of WAL sender process.
`WalWriterMain`	Waiting in main loop of WAL writer process.

Wait Events of Type `BufferPin`

`BufferPin` Wait Event	Description
`BufferPin`	Waiting to acquire an exclusive pin on a buffer.

Wait Events of Type `Client`

`Client` Wait Event	Description
`ClientRead`	Waiting to read data from the client.
`ClientWrite`	Waiting to write data to the client.
`GSSOpenServer`	Waiting to read data from the client while establishing a GSSAPI session.
`LibPQWalReceiverConnect`	Waiting in WAL receiver to establish connection to remote server.

`LibPQWalReceiverReceive`	Waiting in WAL receiver to receive data from remote server.
`SSLOpenServer`	Waiting for SSL while attempting connection.
`WalSenderWaitForWAL`	Waiting for WAL to be flushed in WAL sender process.
`WalSenderWriteData`	Waiting for any activity when processing replies from WAL receiver in WAL sender process.

Wait Events of Type `Extension`

`Extension` Wait Event	Description
`Extension`	Waiting in an extension.

Wait Events of Type `IO`

`IO` Wait Event	Description
`BaseBackupRead`	Waiting for base backup to read from a file.
`BufFileRead`	Waiting for a read from a buffered file.
`BufFileWrite`	Waiting for a write to a buffered file.
`BufFileTruncate`	Waiting for a buffered file to be truncated.
`ControlFileRead`	Waiting for a read from the `pg_control` file.
`ControlFileSync`	Waiting for the `pg_control` file to reach durable storage.
`ControlFileSyncUpdate`	Waiting for an update to the `pg_control` file to reach durable storage.
`ControlFileWrite`	Waiting for a write to the `pg_control` file.
`ControlFileWriteUpdate`	Waiting for a write to update the `pg_control` file.
`CopyFileRead`	Waiting for a read during a file copy operation.
`CopyFileWrite`	Waiting for a write during a file copy operation.
`DSMFillZeroWrite`	Waiting to fill a dynamic shared memory backing file with zeroes.
`DataFileExtend`	Waiting for a relation data file to be extended.
`DataFileFlush`	Waiting for a relation data file to reach durable storage.
`DataFileImmediateSync`	Waiting for an immediate synchronization of a relation data file to durable storage.
`DataFilePrefetch`	Waiting for an asynchronous prefetch from a relation data file.
`DataFileRead`	Waiting for a read from a relation data file.
`DataFileSync`	Waiting for changes to a relation data file to reach durable storage.
`DataFileTruncate`	Waiting for a relation data file to be truncated.
`DataFileWrite`	Waiting for a write to a relation data file.
`LockFileAddToDataDirRead`	Waiting for a read while adding a line to the data directory lock file.
`LockFileAddToDataDirSync`	Waiting for data to reach durable storage while adding a line to the data directory lock file.
`LockFileAddToDataDirWrite`	Waiting for a write while adding a line to the data directory lock file.

LockFileCreateRead	Waiting to read while creating the data directory lock file.
LockFileCreateSync	Waiting for data to reach durable storage while creating the data directory lock file.
LockFileCreateWrite	Waiting for a write while creating the data directory lock file.
LockFileReCheckDataDirRead	Waiting for a read during recheck of the data directory lock file.
LogicalRewriteCheckpointSync	Waiting for logical rewrite mappings to reach durable storage during a checkpoint.
LogicalRewriteMappingSync	Waiting for mapping data to reach durable storage during a logical rewrite.
LogicalRewriteMappingWrite	Waiting for a write of mapping data during a logical rewrite.
LogicalRewriteSync	Waiting for logical rewrite mappings to reach durable storage.
LogicalRewriteTruncate	Waiting for truncate of mapping data during a logical rewrite.
LogicalRewriteWrite	Waiting for a write of logical rewrite mappings.
RelationMapRead	Waiting for a read of the relation map file.
RelationMapSync	Waiting for the relation map file to reach durable storage.
RelationMapWrite	Waiting for a write to the relation map file.
ReorderBufferRead	Waiting for a read during reorder buffer management.
ReorderBufferWrite	Waiting for a write during reorder buffer management.
ReorderLogicalMappingRead	Waiting for a read of a logical mapping during reorder buffer management.
ReplicationSlotRead	Waiting for a read from a replication slot control file.
ReplicationSlotRestoreSync	Waiting for a replication slot control file to reach durable storage while restoring it to memory.
ReplicationSlotSync	Waiting for a replication slot control file to reach durable storage.
ReplicationSlotWrite	Waiting for a write to a replication slot control file.
SLRUFlushSync	Waiting for SLRU data to reach durable storage during a checkpoint or database shutdown.
SLRURead	Waiting for a read of an SLRU page.
SLRUSync	Waiting for SLRU data to reach durable storage following a page write.
SLRUWrite	Waiting for a write of an SLRU page.
SnapbuildRead	Waiting for a read of a serialized historical catalog snapshot.
SnapbuildSync	Waiting for a serialized historical catalog snapshot to reach durable storage.
SnapbuildWrite	Waiting for a write of a serialized historical catalog snapshot.
TimelineHistoryFileSync	Waiting for a timeline history file received via streaming replication to reach durable storage.

TimelineHistoryFileWrite	Waiting for a write of a timeline history file received via streaming replication.
TimelineHistoryRead	Waiting for a read of a timeline history file.
TimelineHistorySync	Waiting for a newly created timeline history file to reach durable storage.
TimelineHistoryWrite	Waiting for a write of a newly created timeline history file.
TwophaseFileRead	Waiting for a read of a two phase state file.
TwophaseFileSync	Waiting for a two phase state file to reach durable storage.
TwophaseFileWrite	Waiting for a write of a two phase state file.
VersionFileWrite	Waiting for the version file to be written while creating a database.
WALBootstrapSync	Waiting for WAL to reach durable storage during bootstrapping.
WALBootstrapWrite	Waiting for a write of a WAL page during bootstrapping.
WALCopyRead	Waiting for a read when creating a new WAL segment by copying an existing one.
WALCopySync	Waiting for a new WAL segment created by copying an existing one to reach durable storage.
WALCopyWrite	Waiting for a write when creating a new WAL segment by copying an existing one.
WALInitSync	Waiting for a newly initialized WAL file to reach durable storage.
WALInitWrite	Waiting for a write while initializing a new WAL file.
WALRead	Waiting for a read from a WAL file.
WALSenderTimelineHistoryRead	Waiting for a read from a timeline history file during a walsender timeline command.
WALSync	Waiting for a WAL file to reach durable storage.
WALSyncMethodAssign	Waiting for data to reach durable storage while assigning a new WAL sync method.
WALWrite	Waiting for a write to a WAL file.

Wait Events of Type IPC

IPC Wait Event	Description
AppendReady	Waiting for subplan nodes of an Append plan node to be ready.
ArchiveCleanupCommand	Waiting for archive_cleanup_command to complete.
ArchiveCommand	Waiting for archive_command to complete.
BackendTermination	Waiting for the termination of another backend.
BackupWaitWalArchive	Waiting for WAL files required for a backup to be successfully archived.
BgWorkerShutdown	Waiting for background worker to shut down.
BgWorkerStartup	Waiting for background worker to start up.
BtreePage	Waiting for the page number needed to continue a parallel B-tree scan to become available.

`BufferIO`	Waiting for buffer I/O to complete.
`CheckpointDone`	Waiting for a checkpoint to complete.
`CheckpointStart`	Waiting for a checkpoint to start.
`ExecuteGather`	Waiting for activity from a child process while executing a `Gather` plan node.
`HashBatchAllocate`	Waiting for an elected Parallel Hash participant to allocate a hash table.
`HashBatchElect`	Waiting to elect a Parallel Hash participant to allocate a hash table.
`HashBatchLoad`	Waiting for other Parallel Hash participants to finish loading a hash table.
`HashBuildAllocate`	Waiting for an elected Parallel Hash participant to allocate the initial hash table.
`HashBuildElect`	Waiting to elect a Parallel Hash participant to allocate the initial hash table.
`HashBuildHashInner`	Waiting for other Parallel Hash participants to finish hashing the inner relation.
`HashBuildHashOuter`	Waiting for other Parallel Hash participants to finish partitioning the outer relation.
`HashGrowBatchesAllocate`	Waiting for an elected Parallel Hash participant to allocate more batches.
`HashGrowBatchesDecide`	Waiting to elect a Parallel Hash participant to decide on future batch growth.
`HashGrowBatchesElect`	Waiting to elect a Parallel Hash participant to allocate more batches.
`HashGrowBatchesFinish`	Waiting for an elected Parallel Hash participant to decide on future batch growth.
`HashGrowBatchesRepartition`	Waiting for other Parallel Hash participants to finish repartitioning.
`HashGrowBucketsAllocate`	Waiting for an elected Parallel Hash participant to finish allocating more buckets.
`HashGrowBucketsElect`	Waiting to elect a Parallel Hash participant to allocate more buckets.
`HashGrowBucketsReinsert`	Waiting for other Parallel Hash participants to finish inserting tuples into new buckets.
`LogicalSyncData`	Waiting for a logical replication remote server to send data for initial table synchronization.
`LogicalSyncStateChange`	Waiting for a logical replication remote server to change state.
`MessageQueueInternal`	Waiting for another process to be attached to a shared message queue.
`MessageQueuePutMessage`	Waiting to write a protocol message to a shared message queue.
`MessageQueueReceive`	Waiting to receive bytes from a shared message queue.
`MessageQueueSend`	Waiting to send bytes to a shared message queue.
`ParallelBitmapScan`	Waiting for parallel bitmap scan to become initialized.

`ParallelCreateIndexScan`	Waiting for parallel `CREATE INDEX` workers to finish heap scan.
`ParallelFinish`	Waiting for parallel workers to finish computing.
`ProcArrayGroupUpdate`	Waiting for the group leader to clear the transaction ID at end of a parallel operation.
`ProcSignalBarrier`	Waiting for a barrier event to be processed by all backends.
`Promote`	Waiting for standby promotion.
`RecoveryConflictSnapshot`	Waiting for recovery conflict resolution for a vacuum cleanup.
`RecoveryConflictTablespace`	Waiting for recovery conflict resolution for dropping a tablespace.
`RecoveryEndCommand`	Waiting for recovery_end_command to complete.
`RecoveryPause`	Waiting for recovery to be resumed.
`ReplicationOriginDrop`	Waiting for a replication origin to become inactive so it can be dropped.
`ReplicationSlotDrop`	Waiting for a replication slot to become inactive so it can be dropped.
`RestoreCommand`	Waiting for restore_command to complete.
`SafeSnapshot`	Waiting to obtain a valid snapshot for a `READ ONLY DEFERRABLE` transaction.
`SyncRep`	Waiting for confirmation from a remote server during synchronous replication.
`WalReceiverExit`	Waiting for the WAL receiver to exit.
`WalReceiverWaitStart`	Waiting for startup process to send initial data for streaming replication.
`XactGroupUpdate`	Waiting for the group leader to update transaction status at end of a parallel operation.

Wait Events of Type `Lock`

`Lock` Wait Event	Description
`advisory`	Waiting to acquire an advisory user lock.
`extend`	Waiting to extend a relation.
`frozenid`	Waiting to update `pg_database.datfrozenxid` and `pg_database.datminmxid`.
`object`	Waiting to acquire a lock on a non-relation database object.
`page`	Waiting to acquire a lock on a page of a relation.
`relation`	Waiting to acquire a lock on a relation.
`spectoken`	Waiting to acquire a speculative insertion lock.
`transactionid`	Waiting for a transaction to finish.
`tuple`	Waiting to acquire a lock on a tuple.
`userlock`	Waiting to acquire a user lock.
`virtualxid`	Waiting to acquire a virtual transaction ID lock.

Wait Events of Type `LWLock`

`LWLock` Wait Event	Description

AddinShmemInit	Waiting to manage an extension’ s space allocation in shared memory.
AutoFile	Waiting to update the postgresql.auto.conf file.
Autovacuum	Waiting to read or update the current state of autovacuum workers.
AutovacuumSchedule	Waiting to ensure that a table selected for autovacuum still needs vacuuming.
BackgroundWorker	Waiting to read or update background worker state.
BtreeVacuum	Waiting to read or update vacuum-related information for a B-tree index.
BufferContent	Waiting to access a data page in memory.
BufferMapping	Waiting to associate a data block with a buffer in the buffer pool.
CheckpointerComm	Waiting to manage fsync requests.
CommitTs	Waiting to read or update the last value set for a transaction commit timestamp.
CommitTsBuffer	Waiting for I/O on a commit timestamp SLRU buffer.
CommitTsSLRU	Waiting to access the commit timestamp SLRU cache.
ControlFile	Waiting to read or update the pg_control file or create a new WAL file.
DynamicSharedMemoryControl	Waiting to read or update dynamic shared memory allocation information.
LockFastPath	Waiting to read or update a process' fast-path lock information.
LockManager	Waiting to read or update information about “heavyweight” locks.
LogicalRepWorker	Waiting to read or update the state of logical replication workers.
MultiXactGen	Waiting to read or update shared multixact state.
MultiXactMemberBuffer	Waiting for I/O on a multixact member SLRU buffer.
MultiXactMemberSLRU	Waiting to access the multixact member SLRU cache.
MultiXactOffsetBuffer	Waiting for I/O on a multixact offset SLRU buffer.
MultiXactOffsetSLRU	Waiting to access the multixact offset SLRU cache.
MultiXactTruncation	Waiting to read or truncate multixact information.
NotifyBuffer	Waiting for I/O on a NOTIFY message SLRU buffer.
NotifyQueue	Waiting to read or update NOTIFY messages.
NotifyQueueTail	Waiting to update limit on NOTIFY message storage.
NotifySLRU	Waiting to access the NOTIFY message SLRU cache.
OidGen	Waiting to allocate a new OID.
OldSnapshotTimeMap	Waiting to read or update old snapshot control information.
ParallelAppend	Waiting to choose the next subplan during Parallel Append plan execution.
ParallelHashJoin	Waiting to synchronize workers during Parallel Hash Join plan execution.

ParallelQueryDSA	Waiting for parallel query dynamic shared memory allocation.
PerSessionDSA	Waiting for parallel query dynamic shared memory allocation.
PerSessionRecordType	Waiting to access a parallel query's information about composite types.
PerSessionRecordTypmod	Waiting to access a parallel query's information about type modifiers that identify anonymous record types.
PerXactPredicateList	Waiting to access the list of predicate locks held by the current serializable transaction during a parallel query.
PredicateLockManager	Waiting to access predicate lock information used by serializable transactions.
ProcArray	Waiting to access the shared per-process data structures (typically, to get a snapshot or report a session's transaction ID).
RelationMapping	Waiting to read or update a `pg_filenode.map` file (used to track the filenode assignments of certain system catalogs).
RelCacheInit	Waiting to read or update a `pg_internal.init` relation cache initialization file.
ReplicationOrigin	Waiting to create, drop or use a replication origin.
ReplicationOriginState	Waiting to read or update the progress of one replication origin.
ReplicationSlotAllocation	Waiting to allocate or free a replication slot.
ReplicationSlotControl	Waiting to read or update replication slot state.
ReplicationSlotIO	Waiting for I/O on a replication slot.
SerialBuffer	Waiting for I/O on a serializable transaction conflict SLRU buffer.
SerializableFinishedList	Waiting to access the list of finished serializable transactions.
SerializablePredicateList	Waiting to access the list of predicate locks held by serializable transactions.
PgStatsDSA	Waiting for stats dynamic shared memory allocator access
PgStatsHash	Waiting for stats shared memory hash table access
PgStatsData	Waiting for shared memory stats data access
SerializableXactHash	Waiting to read or update information about serializable transactions.
SerialSLRU	Waiting to access the serializable transaction conflict SLRU cache.
SharedTidBitmap	Waiting to access a shared TID bitmap during a parallel bitmap index scan.
SharedTupleStore	Waiting to access a shared tuple store during parallel query.
ShmemIndex	Waiting to find or allocate space in shared memory.
SInvalRead	Waiting to retrieve messages from the shared catalog invalidation queue.

SInvalWrite	Waiting to add a message to the shared catalog invalidation queue.
SubtransBuffer	Waiting for I/O on a sub-transaction SLRU buffer.
SubtransSLRU	Waiting to access the sub-transaction SLRU cache.
SyncRep	Waiting to read or update information about the state of synchronous replication.
SyncScan	Waiting to select the starting location of a synchronized table scan.
TablespaceCreate	Waiting to create or drop a tablespace.
TwoPhaseState	Waiting to read or update the state of prepared transactions.
WALBufMapping	Waiting to replace a page in WAL buffers.
WALInsert	Waiting to insert WAL data into a memory buffer.
WALWrite	Waiting for WAL buffers to be written to disk.
WrapLimitsVacuum	Waiting to update limits on transaction id and multixact consumption.
XactBuffer	Waiting for I/O on a transaction status SLRU buffer.
XactSLRU	Waiting to access the transaction status SLRU cache.
XactTruncation	Waiting to execute pg_xact_status or update the oldest transaction ID available to it.
XidGen	Waiting to allocate a new transaction ID.

Note

Extensions can add LWLock types to the list shown in Table 12. In some cases, the name assigned by an extension will not be available in all server processes; so an LWLock wait event might be reported as just "extension" rather than the extension-assigned name.

Wait Events of Type Timeout

Timeout Wait Event	Description
BaseBackupThrottle	Waiting during base backup when throttling activity.
CheckpointWriteDelay	Waiting between writes while performing a checkpoint.
PgSleep	Waiting due to a call to pg_sleep or a sibling function.
RecoveryApplyDelay	Waiting to apply WAL during recovery because of a delay setting.
RecoveryRetrieveRetryInterval	Waiting during recovery when WAL data is not available from any source (pg_wal, archive or stream).
RegisterSyncRequest	Waiting while sending synchronization requests to the checkpointer, because the request queue is full.
VacuumDelay	Waiting in a cost-based vacuum delay point.
VacuumTruncate	Waiting to acquire an exclusive lock to truncate off any empty pages at the end of a table vacuumed.

Here is an example of how wait events can be viewed:

```
SELECT pid, wait_event_type, wait_event FROM pg_stat_activity WHERE wait_event is NOT
NULL;
 pid  | wait_event_type | wait_event
------+-----------------+------------
 2540 | Lock            | relation
 6644 | LWLock          | ProcArray
(2 rows)
```

pg_stat_replication

The pg_stat_replication view will contain one row per WAL sender process, showing statistics about replication to that sender's connected standby server. Only directly connected standbys are listed; no information is available about downstream standby servers.

pg_stat_replication View

Column	Type	Description
pid	integer	Process ID of a WAL sender process
usesysid	oid	OID of the user logged into this WAL sender process
usename	name	Name of the user logged into this WAL sender process
application_name	text	Name of the application that is connected to this WAL sender
client_addr	inet	IP address of the client connected to this WAL sender. If this field is null, it indicates that the client is connected via a Unix socket on the server machine.
client_hostname	text	Host name of the connected client, as reported by a reverse DNS lookup of client_addr. This field will only be non-null for IP connections, and only when log_hostname is enabled.
client_port	integer	TCP port number that the client is using for communication with this WAL sender, or -1 if a Unix socket is used
backend_start	timestamp with time zone	Time when this process was started, i.e., when the client connected to this WAL sender
backend_xmin	xid	This standby's xmin horizon reported by hot_standby_feedback.

state	text	Current WAL sender state. Possible values are: startup: This WAL sender is starting up. catchup: This WAL sender's connected standby is catching up with the primary. streaming: This WAL sender is streaming changes after its connected standby server has caught up with the primary. backup: This WAL sender is sending a backup. stopping: This WAL sender is stopping.
sent_lsn	pg_lsn	Last write-ahead log location sent on this connection
write_lsn	pg_lsn	Last write-ahead log location written to disk by this standby server
flush_lsn	pg_lsn	Last write-ahead log location flushed to disk by this standby server
replay_lsn	pg_lsn	Last write-ahead log location replayed into the database on this standby server
write_lag	interval	Time elapsed between flushing recent WAL locally and receiving notification that this standby server has written it (but not yet flushed it or applied it). This can be used to gauge the delay that synchronous_commit level remote_write incurred while committing if this server was configured as a synchronous standby.
flush_lag	interval	Time elapsed between flushing recent WAL locally and receiving notification that this standby server has written and flushed it (but not yet applied it). This can be used to gauge the delay that synchronous_commit level on incurred while committing if this server was configured as a synchronous standby.
replay_lag	interval	Time elapsed between flushing recent WAL locally and receiving notification that this standby server has written, flushed and applied it. This can be used to gauge the delay that synchronous_commit level remote_apply incurred while committing if this server was configured as a synchronous standby.

sync_priority	integer	Priority of this standby server for being chosen as the synchronous standby in a priority-based synchronous replication. This has no effect in a quorum-based synchronous replication.
sync_state	text	Synchronous state of this standby server. Possible values are: async: This standby server is asynchronous. potential: This standby server is now asynchronous, but can potentially become synchronous if one of current synchronous ones fails. sync: This standby server is synchronous. quorum: This standby server is considered as a candidate for quorum standbys.
reply_time	timestamp with time zone	Send time of last reply message received from standby server

The lag times reported in the pg_stat_replication view are measurements of the time taken for recent WAL to be written, flushed and replayed and for the sender to know about it. These times represent the commit delay that was (or would have been) introduced by each synchronous commit level, if the remote server was configured as a synchronous standby. For an asynchronous standby, the replay_lag column approximates the delay before recent transactions became visible to queries. If the standby server has entirely caught up with the sending server and there is no more WAL activity, the most recently measured lag times will continue to be displayed for a short time and then show NULL.

Lag times work automatically for physical replication. Logical decoding plugins may optionally emit tracking messages; if they do not, the tracking mechanism will simply display NULL lag.

> **Note**
>
> The reported lag times are not predictions of how long it will take for the standby to catch up with the sending server assuming the current rate of replay. Such a system would show similar times while new WAL is being generated, but would differ when the sender becomes idle. In particular, when the standby has caught up completely, pg_stat_replication shows the time taken to write, flush and replay the most recent reported WAL location rather than zero as some users might expect. This is consistent with the goal of measuring synchronous commit and transaction visibility delays for recent write transactions. To reduce confusion for users expecting a different model of lag, the lag columns revert to NULL after a short time on a fully replayed idle system. Monitoring systems should choose whether to represent this as missing data, zero or continue to display the last known value.

pg_stat_replication_slots

The pg_stat_replication_slots view will contain one row per logical replication slot, showing statistics about its usage.

pg_stat_replication_slots View

Column	Type	Description
slot_name	text	A unique, cluster-wide identifier for the replication slot

spill_txns	bigint	Number of transactions spilled to disk once the memory used by logical decoding to decode changes from WAL has exceeded logical_decoding_work_mem. The counter gets incremented for both top-level transactions and subtransactions.
spill_count	bigint	Number of times transactions were spilled to disk while decoding changes from WAL for this slot. This counter is incremented each time a transaction is spilled, and the same transaction may be spilled multiple times.
spill_bytes	bigint	Amount of decoded transaction data spilled to disk while performing decoding of changes from WAL for this slot. This and other spill counters can be used to gauge the I/O which occurred during logical decoding and allow tuning logical_decoding_work_mem.
stream_txns	bigint	Number of in-progress transactions streamed to the decoding output plugin after the memory used by logical decoding to decode changes from WAL for this slot has exceeded logical_decoding_work_mem. Streaming only works with top-level transactions (subtransactions can't be streamed independently), so the counter is not incremented for subtransactions.
stream_count	bigint	Number of times in-progress transactions were streamed to the decoding output plugin while decoding changes from WAL for this slot. This counter is incremented each time a transaction is streamed, and the same transaction may be streamed multiple times.
stream_bytes	bigint	Amount of transaction data decoded for streaming in-progress transactions to the decoding output plugin while decoding changes from WAL for this slot. This and other streaming counters for this slot can be used to tune logical_decoding_work_mem.

total_txns	bigint	Number of decoded transactions sent to the decoding output plugin for this slot. This counts top-level transactions only, and is not incremented for subtransactions. Note that this includes the transactions that are streamed and/or spilled.
total_bytes	bigint	Amount of transaction data decoded for sending transactions to the decoding output plugin while decoding changes from WAL for this slot. Note that this includes data that is streamed and/or spilled.
stats_reset	timestamp with time zone	Time at which these statistics were last reset

pg_stat_wal_receiver

The pg_stat_wal_receiver view will contain only one row, showing statistics about the WAL receiver from that receiver's connected server.

pg_stat_wal_receiver View

Column	Type	Description
pid	integer	Process ID of the WAL receiver process
status	text	Activity status of the WAL receiver process
receive_start_lsn	pg_lsn	First write-ahead log location used when WAL receiver is started
receive_start_tli	integer	First timeline number used when WAL receiver is started
written_lsn	pg_lsn	Last write-ahead log location already received and written to disk, but not flushed. This should not be used for data integrity checks.
flushed_lsn	pg_lsn	Last write-ahead log location already received and flushed to disk, the initial value of this field being the first log location used when WAL receiver is started
received_tli	integer	Timeline number of last write-ahead log location received and flushed to disk, the initial value of this field being the timeline number of the first log location used when WAL receiver is started
last_msg_send_time	timestamp with time zone	Send time of last message received from origin WAL sender
last_msg_receipt_time	timestamp with time zone	Receipt time of last message received from origin WAL sender
latest_end_lsn	pg_lsn	Last write-ahead log location reported to origin WAL sender

latest_end_time	timestamp with time zone	Time of last write-ahead log location reported to origin WAL sender
slot_name	text	Replication slot name used by this WAL receiver
sender_host	text	Host of the IvorySQL instance this WAL receiver is connected to. This can be a host name, an IP address, or a directory path if the connection is via Unix socket. (The path case can be distinguished because it will always be an absolute path, beginning with /.)
sender_port	integer	Port number of the IvorySQL instance this WAL receiver is connected to.
conninfo	text	Connection string used by this WAL receiver, with security-sensitive fields obfuscated.

pg_stat_recovery_prefetch

The **pg_stat_recovery_prefetch** view will contain only one row. The columns **wal_distance**, **block_distance** and **io_depth** show current values, and the other columns show cumulative counters that can be reset with the **pg_stat_reset_shared** function.

pg_stat_recovery_prefetch View

Column	Type	Description
stats_reset	timestamp with time zone	Time at which these statistics were last reset
prefetch	bigint	Number of blocks prefetched because they were not in the buffer pool
hit	bigint	Number of blocks not prefetched because they were already in the buffer pool
skip_init	bigint	Number of blocks not prefetched because they would be zero-initialized
skip_new	bigint	Number of blocks not prefetched because they didn’ t exist yet
skip_fpw	bigint	Number of blocks not prefetched because a full page image was included in the WAL
skip_rep	bigint	Number of blocks not prefetched because they were already recently prefetched
wal_distance	int	How many bytes ahead the prefetcher is looking
block_distance	int	How many blocks ahead the prefetcher is looking
io_depth	int	How many prefetches have been initiated but are not yet known to have completed

pg_stat_subscription

pg_stat_subscription View

Column	Type	Description
subid	oid	OID of the subscription
subname	name	Name of the subscription
pid	integer	Process ID of the subscription worker process
relid	oid	OID of the relation that the worker is synchronizing; null for the main apply worker
received_lsn	pg_lsn	Last write-ahead log location received, the initial value of this field being 0
last_msg_send_time	timestamp with time zone	Send time of last message received from origin WAL sender
last_msg_receipt_time	timestamp with time zone	Receipt time of last message received from origin WAL sender
latest_end_lsn	pg_lsn	Last write-ahead log location reported to origin WAL sender
latest_end_time	timestamp with time zone	Time of last write-ahead log location reported to origin WAL sender

pg_stat_subscription_stats

The **pg_stat_subscription_stats** view will contain one row per subscription.

pg_stat_subscription_stats View

Column	Type	Description
subid	oid	OID of the subscription
subname	name	Name of the subscription
apply_error_count	bigint	Number of times an error occurred while applying changes
sync_error_count	bigint	Number of times an error occurred during the initial table synchronization
stats_reset	timestamp with time zone	Time at which these statistics were last reset

pg_stat_ssl

The **pg_stat_ssl** view will contain one row per backend or WAL sender process, showing statistics about SSL usage on this connection. It can be joined to **pg_stat_activity** or **pg_stat_replication** on the **pid** column to get more details about the connection.

pg_stat_ssl View

Column	Type	Description
pid	integer	Process ID of a backend or WAL sender process
ssl	boolean	True if SSL is used on this connection

version	text	Version of SSL in use, or NULL if SSL is not in use on this connection
cipher	text	Name of SSL cipher in use, or NULL if SSL is not in use on this connection
bits	integer	Number of bits in the encryption algorithm used, or NULL if SSL is not used on this connection
client_dn	text	Distinguished Name (DN) field from the client certificate used, or NULL if no client certificate was supplied or if SSL is not in use on this connection. This field is truncated if the DN field is longer than NAMEDATALEN (64 characters in a standard build)
client_serial	numeric	Serial number of the client certificate, or NULL if no client certificate was supplied or if SSL is not in use on this connection. The combination of certificate serial number and certificate issuer uniquely identifies a certificate (unless the issuer erroneously reuses serial numbers)
issuer_dn	text	DN of the issuer of the client certificate, or NULL if no client certificate was supplied or if SSL is not in use on this connection. This field is truncated like client_dn

pg_stat_gssapi

The pg_stat_gssapi view will contain one row per backend, showing information about GSSAPI usage on this connection. It can be joined to pg_stat_activity or pg_stat_replication on the pid column to get more details about the connection.

pg_stat_gssapi View

Column	Type	Description
pid	integer	Process ID of a backend
gss_authenticated	boolean	True if GSSAPI authentication was used for this connection
principal	text	Principal used to authenticate this connection, or NULL if GSSAPI was not used. Truncated to NAMEDATALEN (64 characters in a standard build) if longer.
encrypted	boolean	True if GSSAPI encryption is in use on this connection

pg_stat_archiver

The pg_stat_archiver view will always have a single row, containing data about the archiver process of the cluster.

pg_stat_archiver View

Column	Type	Description
archived_count	bigint	Number of WAL files that have been successfully archived
last_archived_wal	text	Name of the WAL file most recently successfully archived
last_archived_time	timestamp with time zone	Time of the most recent successful archive operation
failed_count	bigint	Number of failed attempts for archiving WAL files
last_failed_wal	text	Name of the WAL file of the most recent failed archival operation
last_failed_time	timestamp with time zone	Time of the most recent failed archival operation
stats_reset	timestamp with time zone	Time at which these statistics were last reset

Normally, WAL files are archived in order, oldest to newest, but that is not guaranteed, and does not hold under special circumstances like when promoting a standby or after crash recovery. Therefore it is not safe to assume that all files older than last_archived_wal have also been successfully archived.

pg_stat_bgwriter

The pg_stat_bgwriter view will always have a single row, containing global data for the cluster.

pg_stat_bgwriter View

Column	Type	Description
checkpoints_timed	bigint	Number of scheduled checkpoints that have been performed
checkpoints_req	bigint	Number of requested checkpoints that have been performed
checkpoint_write_time	double precision	Total amount of time that has been spent in the portion of checkpoint processing where files are written to disk, in milliseconds
checkpoint_sync_time	double precision	Total amount of time that has been spent in the portion of checkpoint processing where files are synchronized to disk, in milliseconds
buffers_checkpoint	bigint	Number of buffers written during checkpoints
buffers_clean	bigint	Number of buffers written by the background writer
maxwritten_clean	bigint	Number of times the background writer stopped a cleaning scan because it had written too many buffers
buffers_backend	bigint	Number of buffers written directly by a backend

buffers_backend_fsync	bigint	Number of times a backend had to execute its own fsync call (normally the background writer handles those even when the backend does its own write)
buffers_alloc	bigint	Number of buffers allocated
stats_reset	timestamp with time zone	Time at which these statistics were last reset

pg_stat_wal

The pg_stat_wal view will always have a single row, containing data about WAL activity of the cluster.

pg_stat_wal View

Column	Type	Description
pid	integer	Process ID of backend.
datid	oid	OID of the database to which this backend is connected.
datname	name	Name of the database to which this backend is connected.
relid	oid	OID of the table on which the index is being created.
index_relid	oid	OID of the index being created or reindexed. During a non-concurrent CREATE INDEX, this is 0.
command	text	The command that is running: CREATE INDEX, CREATE INDEX CONCURRENTLY, REINDEX, or REINDEX CONCURRENTLY.
phase	text	Current processing phase of index creation. See [Table 1.39](https://www.postgresql.org/docs/current/progress-reporting.html#CREATE-INDEX-PHASES).
lockers_total	bigint	Total number of lockers to wait for, when applicable.
lockers_done	bigint	Number of lockers already waited for.
current_locker_pid	bigint	Process ID of the locker currently being waited for.
blocks_total	bigint	Total number of blocks to be processed in the current phase.
blocks_done	bigint	Number of blocks already processed in the current phase.
tuples_total	bigint	Total number of tuples to be processed in the current phase.
tuples_done	bigint	Number of tuples already processed in the current phase.

partitions_total	bigint	When creating an index on a partitioned table, this column is set to the total number of partitions on which the index is to be created. This field is 0 during a REINDEX.
partitions_done	bigint	When creating an index on a partitioned table, this column is set to the number of partitions on which the index has been created. This field is 0 during a REINDEX.

pg_stat_database

The pg_stat_database view will contain one row for each database in the cluster, plus one for shared objects, showing database-wide statistics.

pg_stat_database View

Column	Type	Description
datid	oid	OID of this database, or 0 for objects belonging to a shared relation
datname	name	Name of this database, or NULL for shared objects.
numbackends	integer	Number of backends currently connected to this database, or NULL for shared objects. This is the only column in this view that returns a value reflecting current stateall other columns return the accumulated values since the last reset.
xact_commit	bigint	Number of transactions in this database that have been committed
xact_rollback	bigint	Number of transactions in this database that have been rolled back
blks_read	bigint	Number of disk blocks read in this database
blks_hit	bigint	Number of times disk blocks were found already in the buffer cache, so that a read was not necessary (this only includes hits in the IvorySQL buffer cache, not the operatinsystem’ s file system cache)
tup_returned	bigint	Number of live rows fetched by sequential scans and index entries returned by index scans in this database
tup_fetched	bigint	Number of live rows fetched by index scans in this database
tup_inserted	bigint	Number of rows inserted by queries in this database

tup_updated	bigint	Number of rows updated by queries in this database
tup_deleted	bigint	Number of rows deleted by queries in this database
conflicts	bigint	Number of queries canceled due to conflicts with recovery in this database. (Conflicts occur only on standby servers; seepg_stat_database_conflicts](https://www.postgresql.org/docs/current/monitoring-stats.html#MONITORING-PG-STAT-DATABASE-CONFLICTS-VIEW) for details.)
temp_files	bigint	Number of temporary files created by queries in this database. All temporary files are counted, regardless of why the temporary file was created (e.g., sorting or hashing)and regardless of the [log_temp_files](https://www.postgresql.org/docs/current/runtime-config-logging.html#GUC-LOG-TEMP-FILES) setting.
temp_bytes	bigint	Total amount of data written to temporary files by queries in this database. All temporary files are counted, regardless of why the temporary file was created, and regardlesof the [log_temp_files](https://www.postgresql.org/docs/current/runtime-config-logging.html#GUC-LOG-TEMP-FILES) setting.
deadlocks	bigint	Number of deadlocks detected in this database
checksum_failures	bigint	Number of data page checksum failures detected in this database (or on a shared object), or NULL if data checksums are not enabled.
checksum_last_failure	timestamp with time zone	Time at which the last data page checksum failure was detected in this database (or on a shared object), or NULL if data checksums are noenabled.
blk_read_time	double precision	Time spent reading data file blocks by backends in this database, in milliseconds (if [track_io_timing](https://www.postgresql.org/docs/currenruntime-config-statistics.html#GUC-TRACK-IO-TIMING) is enabled, otherwise zero)

blk_write_time	double precision	Time spent writing data file blocks by backends in this database, in milliseconds (if [track_io_timing](https://www.postgresql.org/docs/currenruntime-config-statistics.html#GUC-TRACK-IO-TIMING) is enabled, otherwise zero)
session_time	double precision	Time spent by database sessions in this database, in milliseconds (note that statistics are only updated when the state of a session changes, so if sessions havbeen idle for a long time, this idle time won't be included)
active_time	double precision	Time spent executing SQL statements in this database, in milliseconds (this corresponds to the states active and fastpath function call inpg_stat_activity](https://www.postgresql.org/docs/current/monitoring-stats.html#MONITORING-PG-STAT-ACTIVITY-VIEW))
idle_in_transaction_time	double precision	Time spent idling while in a transaction in this database, in milliseconds (this corresponds to the states idle in transaction and idle itransaction (aborted) in [pg_stat_activity](https://www.postgresql.org/docs/current/monitoring-stats.html#MONITORING-PG-STAT-ACTIVITY-VIEW))
sessions	bigint	Total number of sessions established to this database
sessions_abandoned	bigint	Number of database sessions to this database that were terminated because connection to the client was lost
sessions_fatal	bigint	Number of database sessions to this database that were terminated by fatal errors
sessions_killed	bigint	Number of database sessions to this database that were terminated by operator intervention
stats_reset	timestamp with time zone	Time at which these statistics were last reset

pg_stat_database_conflicts

The pg_stat_database_conflicts view will contain one row per database, showing database-wide statistics about query cancels occurring due to conflicts with recovery on standby servers. This view will only contain

information on standby servers, since conflicts do not occur on primary servers.

`pg_stat_database_conflicts` View

Column	Type	Description
`datid`	`oid`	OID of a database
`datname`	`name`	Name of this database
`confl_tablespace`	`bigint`	Number of queries in this database that have been canceled due to dropped tablespaces
`confl_lock`	`bigint`	Number of queries in this database that have been canceled due to lock timeouts
`confl_snapshot`	`bigint`	Number of queries in this database that have been canceled due to old snapshots
`confl_bufferpin`	`bigint`	Number of queries in this database that have been canceled due to pinned buffers
`confl_deadlock`	`bigint`	Number of queries in this database that have been canceled due to deadlocks

`pg_stat_all_tables`

The `pg_stat_all_tables` view will contain one row for each table in the current database (including TOAST tables), showing statistics about accesses to that specific table. The `pg_stat_user_tables` and `pg_stat_sys_tables` views contain the same information, but filtered to only show user and system tables respectively.

`pg_stat_all_tables` View

Column	Type	Description
`relid`	`oid`	OID of a table
`schemaname`	`name`	Name of the schema that this table is in
`relname`	`name`	Name of this table
`seq_scan`	`bigint`	Number of sequential scans initiated on this table
`seq_tup_read`	`bigint`	Number of live rows fetched by sequential scans
`idx_scan`	`bigint`	Number of index scans initiated on this table
`idx_tup_fetch`	`bigint`	Number of live rows fetched by index scans
`n_tup_ins`	`bigint`	Number of rows inserted
`n_tup_upd`	`bigint`	Number of rows updated (includes HOT updated rows)
`n_tup_del`	`bigint`	Number of rows deleted
`n_tup_hot_upd`	`bigint`	Number of rows HOT updated (i.e., with no separate index update required)
`n_live_tup`	`bigint`	Estimated number of live rows

n_dead_tup	bigint	Estimated number of dead rows
n_mod_since_analyze	bigint	Estimated number of rows modified since this table was last analyzed
n_ins_since_vacuum	bigint	Estimated number of rows inserted since this table was last vacuumed
last_vacuum	timestamp with time zone	Last time at which this table was manually vacuumed (not counting VACUUM FULL)
last_autovacuum	timestamp with time zone	Last time at which this table was vacuumed by the autovacuum daemon
last_analyze	timestamp with time zone	Last time at which this table was manually analyzed
last_autoanalyze	timestamp with time zone	Last time at which this table was analyzed by the autovacuum daemon
vacuum_count	bigint	Number of times this table has been manually vacuumed (not counting VACUUM FULL)
autovacuum_count	bigint	Number of times this table has been vacuumed by the autovacuum daemon
analyze_count	bigint	Number of times this table has been manually analyzed
autoanalyze_count	bigint	Number of times this table has been analyzed by the autovacuum daemon

pg_stat_all_indexes

The pg_stat_all_indexes view will contain one row for each index in the current database, showing statistics about accesses to that specific index. The pg_stat_user_indexes and pg_stat_sys_indexes views contain the same information, but filtered to only show user and system indexes respectively.

pg_stat_all_indexes View

Column	Type	Description
relid	oid	OID of the table for this index
indexrelid	oid	OID of this index
schemaname	name	Name of the schema this index is in
relname	name	Name of the table for this index
indexrelname	name	Name of this index
idx_scan	bigint	Number of index scans initiated on this index
idx_tup_read	bigint	Number of index entries returned by scans on this index
idx_tup_fetch	bigint	Number of live table rows fetched by simple index scans using this index

Indexes can be used by simple index scans, “bitmap” index scans, and the optimizer. In a bitmap scan the output of several indexes can be combined via AND or OR rules, so it is difficult to associate individual heap row fetches with specific indexes when a bitmap scan is used. Therefore, a bitmap scan increments the `pg_stat_all_indexes.idx_tup_read` count(s) for the index(es) it uses, and it increments the `pg_stat_all_tables.idx_tup_fetch` count for the table, but it does not affect `pg_stat_all_indexes.idx_tup_fetch`. The optimizer also accesses indexes to check for supplied constants whose values are outside the recorded range of the optimizer statistics because the optimizer statistics might be stale.

> **Note**
>
> The `idx_tup_read` and `idx_tup_fetch` counts can be different even without any use of bitmap scans, because `idx_tup_read` counts index entries retrieved from the index while `idx_tup_fetch` counts live rows fetched from the table. The latter will be less if any dead or not-yet-committed rows are fetched using the index, or if any heap fetches are avoided by means of an index-only scan.

pg_statio_all_tables

The `pg_statio_all_tables` view will contain one row for each table in the current database (including TOAST tables), showing statistics about I/O on that specific table. The `pg_statio_user_tables` and `pg_statio_sys_tables` views contain the same information, but filtered to only show user and system tables respectively.

`pg_statio_all_tables` View

Column	Type	Description
`relid`	`oid`	OID of a table
`schemaname`	`name`	Name of the schema that this table is in
`relname`	`name`	Name of this table
`heap_blks_read`	`bigint`	Number of disk blocks read from this table
`heap_blks_hit`	`bigint`	Number of buffer hits in this table
`idx_blks_read`	`bigint`	Number of disk blocks read from all indexes on this table
`idx_blks_hit`	`bigint`	Number of buffer hits in all indexes on this table
`toast_blks_read`	`bigint`	Number of disk blocks read from this table’s TOAST table (if any)
`toast_blks_hit`	`bigint`	Number of buffer hits in this table’s TOAST table (if any)
`tidx_blks_read`	`bigint`	Number of disk blocks read from this table’s TOAST table indexes (if any)
`tidx_blks_hit`	`bigint`	Number of buffer hits in this table’s TOAST table indexes (if any)

pg_statio_all_indexes

The `pg_statio_all_indexes` view will contain one row for each index in the current database, showing statistics about I/O on that specific index. The `pg_statio_user_indexes` and `pg_statio_sys_indexes` views contain the same information, but filtered to only show user and system indexes respectively.

`pg_statio_all_indexes` View

Column	Type	Description
relid	oid	OID of the table for this index
indexrelid	oid	OID of this index
schemaname	name	Name of the schema this index is in
relname	name	Name of the table for this index
indexrelname	name	Name of this index
idx_blks_read	bigint	Number of disk blocks read from this index
idx_blks_hit	bigint	Number of buffer hits in this index

pg_statio_all_sequences

The pg_statio_all_sequences view will contain one row for each sequence in the current database, showing statistics about I/O on that specific sequence.

pg_statio_all_sequences View

Column	Type	Description
relid	oid	OID of a sequence
schemaname	name	Name of the schema this sequence is in
relname	name	Name of this sequence
blks_read	bigint	Number of disk blocks read from this sequence
blks_hit	bigint	Number of buffer hits in this sequence

pg_stat_user_functions

The pg_stat_user_functions view will contain one row for each tracked function, showing statistics about executions of that function. The track_functions parameter controls exactly which functions are tracked.

pg_stat_user_functions View

Column	Type	Description
funcid	oid	OID of a function
schemaname	name	Name of the schema this function is in
funcname	name	Name of this function
calls	bigint	Number of times this function has been called
total_time	double precision	Total time spent in this function and all other functions called by it, in milliseconds
self_time	double precision	Total time spent in this function itself, not including other functions called by it, in milliseconds

pg_stat_slru

IvorySQL accesses certain on-disk information via SLRU (simple least-recently-used) caches. The pg_stat_slru view will contain one row for each tracked SLRU cache, showing statistics about access to

cached pages.

`pg_stat_slru` View

Column	Type	Description
`name`	`text`	Name of the SLRU
`blks_zeroed`	`bigint`	Number of blocks zeroed during initializations
`blks_hit`	`bigint`	Number of times disk blocks were found already in the SLRU, so that a read was not necessary (this only includes hits in the SLRU, not the operating system's file system cache)
`blks_read`	`bigint`	Number of disk blocks read for this SLRU
`blks_written`	`bigint`	Number of disk blocks written for this SLRU
`blks_exists`	`bigint`	Number of blocks checked for existence for this SLRU
`flushes`	`bigint`	Number of flushes of dirty data for this SLRU
`truncates`	`bigint`	Number of truncates for this SLRU
`stats_reset`	`timestamp with time zone`	Time at which these statistics were last reset

Statistics Functions

Other ways of looking at the statistics can be set up by writing queries that use the same underlying statistics access functions used by the standard views shown above. For details such as the functions' names, consult the definitions of the standard views. (For example, in psql you could issue `\d+ pg_stat_activity`.) The access functions for per-database statistics take a database OID as an argument to identify which database to report on. The per-table and per-index functions take a table or index OID. The functions for per-function statistics take a function OID. Note that only tables, indexes, and functions in the current database can be seen with these functions.

Additional Statistics Functions

Function	Description
`pg_backend_pid` () → `integer`	Returns the process ID of the server process attached to the current session.
`pg_stat_get_activity` (`integer`) → `setof record`	Returns a record of information about the backend with the specified process ID, or one record for each active backend in the system if `NULL` is specified. The fields returned are a subset of those in the `pg_stat_activity` view.
`pg_stat_get_snapshot_timestamp` () → `timestamp with time zone`	Returns the timestamp of the current statistics snapshot, or NULL if no statistics snapshot has been taken. A snapshot is taken the first time cumulative statistics are accessed in a transaction if `stats_fetch_consistency` is set to `snapshot`.
`pg_stat_clear_snapshot` () → `void`	Discards the current statistics snapshot or cached information.

`pg_stat_reset` () → `void`	Resets all statistics counters for the current database to zero. This function is restricted to superusers by default, but other users can be granted EXECUTE to run the function.
`pg_stat_reset_shared` (`text`) → `void`	Resets some cluster-wide statistics counters to zero, depending on the argument. The argument can be `bgwriter`, `archiver`, `wal`, or `recovery_prefetch` to reset the counters shown in the respective views. This function is restricted to superusers by default, but other users can be granted EXECUTE to run the function.
`pg_stat_reset_single_table_counters` (`oid`) → `void`	Resets statistics for a single table or index in the current database or shared across all databases in the cluster to zero. This function is restricted to superusers by default, but other users can be granted EXECUTE to run the function.
`pg_stat_reset_single_function_counters` (`oid`) → `void`	Resets statistics for a single function in the current database to zero. This function is restricted to superusers by default, but other users can be granted EXECUTE to run the function.
`pg_stat_reset_slru` (`text`) → `void`	Resets statistics to zero for a single SLRU cache, or for all SLRUs in the cluster. If the argument is NULL, all counters shown in the `pg_stat_slru` view for all SLRU caches are reset. The argument can be one of `CommitTs`, `MultiXactMember`, `MultiXactOffset`, `Notify`, `Serial`, `Subtrans`, or `Xact`. If the argument is `other` (or any unrecognized name), then the counters for all other SLRU caches, such as extension-defined caches, are reset. This function is restricted to superusers by default, but other users can be granted EXECUTE to run the function.
`pg_stat_reset_replication_slot` (`text`) → `void`	Resets statistics of the replication slot defined by the argument. If the argument is `NULL`, resets statistics for all the replication slots. This function is restricted to superusers by default, but other users can be granted EXECUTE to run the function.
`pg_stat_reset_subscription_stats` (`oid`) → `void`	Resets statistics for a single subscription shown in the `pg_stat_subscription_stats` view to zero. If the argument is `NULL`, reset statistics for all subscriptions. This function is restricted to superusers by default, but other users can be granted EXECUTE to run the function.

Warning

Using `pg_stat_reset()` also resets counters that autovacuum uses to determine when to trigger a vacuum or an analyze. Resetting these counters can cause autovacuum to not perform necessary work, which can cause problems such as table bloat or out-dated table statistics. A database-wide `ANALYZE` is recommended after the statistics have been reset.

`pg_stat_get_activity`, the underlying function of the `pg_stat_activity` view, returns a set of records containing all the available information about each backend process. Sometimes it may be more convenient to obtain just a subset of this information. In such cases, an older set of per-backend statistics access functions can be used; These access functions use a backend ID number, which ranges from one to the number of currently active backends. The function `pg_stat_get_backend_idset` provides a convenient way to generate one row for each active backend for invoking these functions. For example, to show the PIDs and current queries of all backends:

```
SELECT pg_stat_get_backend_pid(s.backendid) AS pid,
       pg_stat_get_backend_activity(s.backendid) AS query
    FROM (SELECT pg_stat_get_backend_idset() AS backendid) AS s;
```

Per-Backend Statistics Functions

Function	Description
`pg_stat_get_backend_idset` () → `setof integer`	Returns the set of currently active backend ID numbers (from 1 to the number of active backends).
`pg_stat_get_backend_activity` (`integer`) → `text`	Returns the text of this backend’s most recent query.
`pg_stat_get_backend_activity_start` (`integer`) → `timestamp with time zone`	Returns the time when the backend’s most recent query was started.
`pg_stat_get_backend_client_addr` (`integer`) → `inet`	Returns the IP address of the client connected to this backend.
`pg_stat_get_backend_client_port` (`integer`) → `integer`	Returns the TCP port number that the client is using for communication.
`pg_stat_get_backend_dbid` (`integer`) → `oid`	Returns the OID of the database this backend is connected to.
`pg_stat_get_backend_pid` (`integer`) → `integer`	Returns the process ID of this backend.
`pg_stat_get_backend_start` (`integer`) → `timestamp with time zone`	Returns the time when this process was started.
`pg_stat_get_backend_userid` (`integer`) → `oid`	Returns the OID of the user logged into this backend.
`pg_stat_get_backend_wait_event_type` (`integer`) → `text`	Returns the wait event type name if this backend is currently waiting, otherwise NULL.
`pg_stat_get_backend_wait_event` (`integer`) → `text`	Returns the wait event name if this backend is currently waiting, otherwise NULL.
`pg_stat_get_backend_xact_start` (`integer`) → `timestamp with time zone`	Returns the time when the backend’s current transaction was started.

View Locks

- Another useful tool for monitoring database activity is the `pg_locks` system table. It allows the database administrator to view information about the outstanding locks in the lock manager. For example, this capability can be used to:
  - View all the locks currently outstanding, all the locks on relations in a particular database, all the locks on a particular relation, or all the locks held by a particular IvorySQL session.
  - Determine the relation in the current database with the most ungranted locks (which might be a source of contention among database clients).
  - Determine the effect of lock contention on overall database performance, as well as the extent to which contention varies with overall database traffic.

Progress Reporting

IvorySQL has the ability to report the progress of certain commands during command execution. Currently, the only commands which support progress reporting are `ANALYZE`, `CLUSTER`, `CREATE INDEX`, `VACUUM`, `COPY`, and BASE_BACKUP (i.e., replication command that pg_basebackup issues to take a base backup). This may be expanded in the future.

ANALYZE Progress Reporting

Whenever `ANALYZE` is running, the `pg_stat_progress_analyze` view will contain a row for each backend that

is currently running that command. The tables below describe the information that will be reported and provide information about how to interpret it.

`pg_stat_progress_analyze` View

Column	Type	Description	
`pid`	`integer`	Process ID of backend.	
`datid`	`oid`	OID of the database to which this backend is connected.	
`datname`	`name`	Name of the database to which this backend is connected.	
`relid`	`oid`	OID of the table being analyzed.	
`phase`	`text`	Current processing phase. See Table 1.37.	
`sample_blks_total`	`bigint`	Total number of heap blocks that will be sampled.	
`sample_blks_scanned`	`bigint`	Number of heap blocks scanned.	
`ext_stats_total`	`bigint`	Number of extended statistics.	
`ext_stats_computed`	`bigint`	Number of extended statistics computed. This counter only advances when the phase is `computing extended statistics`.	
`child_tables_total`	`bigint`	Number of child tables.	
`child_tables_done`	`bigint`	Number of child tables scanned. This counter only advances when the phase is `acquiring inherited sample rows`.	
`current_child_table_relid`	`oid`	OID of the child table currently being scanned. This field is only valid when the phase is `acquiring inherited sample rows`.	

ANALYZE Phases

Phase	Description
`initializing`	The command is preparing to begin scanning the heap. This phase is expected to be very brief.
`acquiring sample rows`	The command is currently scanning the table given by `relid` to obtain sample rows.
`acquiring inherited sample rows`	The command is currently scanning child tables to obtain sample rows. Columns `child_tables_total`, `child_tables_done`, and `current_child_table_relid` contain the progress information for this phase.

`computing statistics`	The command is computing statistics from the sample rows obtained during the table scan.
`computing extended statistics`	The command is computing extended statistics from the sample rows obtained during the table scan.
`finalizing analyze`	The command is updating `pg_class`. When this phase is completed, `ANALYZE` will end.

> **Note**
>
> Note that when `ANALYZE` is run on a partitioned table, all of its partitions are also recursively analyzed. In that case, `ANALYZE` progress is reported first for the parent table, whereby its inheritance statistics are collected, followed by that for each partition.

CREATE INDEX Progress Reporting

Whenever `CREATE INDEX` or `REINDEX` is running, the `pg_stat_progress_create_index` view will contain one row for each backend that is currently creating indexes. The tables below describe the information that will be reported and provide information about how to interpret it.

`pg_stat_progress_create_index` View

Column	Type	Description	
`pid`	`integer`	Process ID of backend.	
`datid`	`oid`	OID of the database to which this backend is connected.	
`datname`	`name`	Name of the database to which this backend is connected.	
`relid`	`oid`	OID of the table on which the index is being created.	
`index_relid`	`oid`	OID of the index being created or reindexed. During a non-concurrent `CREATE INDEX`, this is 0.	
`command`	`text`	The command that is running: `CREATE INDEX`, `CREATE INDEX CONCURRENTLY`, `REINDEX`, or `REINDEX CONCURRENTLY`.	
`phase`	`text`	Current processing phase of index creation. See Table 1.39.	
`lockers_total`	`bigint`	Total number of lockers to wait for, when applicable.	
`lockers_done`	`bigint`	Number of lockers already waited for.	
`current_locker_pid`	`bigint`	Process ID of the locker currently being waited for.	

`blocks_total`	`bigint`	Total number of blocks to be processed in the current phase.	
`blocks_done`	`bigint`	Number of blocks already processed in the current phase.	
`tuples_total`	`bigint`	Total number of tuples to be processed in the current phase.	
`tuples_done`	`bigint`	Number of tuples already processed in the current phase.	
`partitions_total`	`bigint`	When creating an index on a partitioned table, this column is set to the total number of partitions on which the index is to be created. This field is `0` during a `REINDEX`.	
`partitions_done`	`bigint`	When creating an index on a partitioned table, this column is set to the number of partitions on which the index has been created. This field is `0` during a `REINDEX`.	

CREATE INDEX Phases

Phase	Description
`initializing`	`CREATE INDEX` or `REINDEX` is preparing to create the index. This phase is expected to be very brief.
`waiting for writers before build`	`CREATE INDEX CONCURRENTLY` or `REINDEX CONCURRENTLY` is waiting for transactions with write locks that can potentially see the table to finish. This phase is skipped when not in concurrent mode. Columns `lockers_total`, `lockers_done` and `current_locker_pid` contain the progress information for this phase.
`building index`	The index is being built by the access method-specific code. In this phase, access methods that support progress reporting fill in their own progress data, and the subphase is indicated in this column. Typically, `blocks_total` and `blocks_done` will contain progress data, as well as potentially `tuples_total` and `tuples_done`.
`waiting for writers before validation`	`CREATE INDEX CONCURRENTLY` or `REINDEX CONCURRENTLY` is waiting for transactions with write locks that can potentially write into the table to finish. This phase is skipped when not in concurrent mode. Columns `lockers_total`, `lockers_done` and `current_locker_pid` contain the progress information for this phase.

`index validation: scanning index`	`CREATE INDEX CONCURRENTLY` is scanning the index searching for tuples that need to be validated. This phase is skipped when not in concurrent mode. Columns `blocks_total` (set to the total size of the index) and `blocks_done` contain the progress information for this phase.
`index validation: sorting tuples`	`CREATE INDEX CONCURRENTLY` is sorting the output of the index scanning phase.
`index validation: scanning table`	`CREATE INDEX CONCURRENTLY` is scanning the table to validate the index tuples collected in the previous two phases. This phase is skipped when not in concurrent mode. Columns `blocks_total` (set to the total size of the table) and `blocks_done` contain the progress information for this phase.
`waiting for old snapshots`	`CREATE INDEX CONCURRENTLY` or `REINDEX CONCURRENTLY` is waiting for transactions that can potentially see the table to release their snapshots. This phase is skipped when not in concurrent mode. Columns `lockers_total`, `lockers_done` and `current_locker_pid` contain the progress information for this phase.
`waiting for readers before marking dead`	`REINDEX CONCURRENTLY` is waiting for transactions with read locks on the table to finish, before marking the old index dead. This phase is skipped when not in concurrent mode. Columns `lockers_total`, `lockers_done` and `current_locker_pid` contain the progress information for this phase.
`waiting for readers before dropping`	`REINDEX CONCURRENTLY` is waiting for transactions with read locks on the table to finish, before dropping the old index. This phase is skipped when not in concurrent mode. Columns `lockers_total`, `lockers_done` and `current_locker_pid` contain the progress information for this phase.

VACUUM Progress Reporting

Whenever `VACUUM` is running, the `pg_stat_progress_vacuum` view will contain one row for each backend (including autovacuum worker processes) that is currently vacuuming. The tables below describe the information that will be reported and provide information about how to interpret it. Progress for `VACUUM FULL` commands is reported via `pg_stat_progress_cluster` because both `VACUUM FULL` and `CLUSTER` rewrite the table, while regular `VACUUM` only modifies it in place.

`pg_stat_progress_vacuum` View

Column	Type	Description	
`pid`	`integer`	Process ID of backend.	
`datid`	`oid`	OID of the database to which this backend is connected.	
`datname`	`name`	Name of the database to which this backend is connected.	
`relid`	`oid`	OID of the table being vacuumed.	
`phase`	`text`	Current processing phase of vacuum.	

heap_blks_total	bigint	Total number of heap blocks in the table. This number is reported as of the beginning of the scan; blocks added later will not be (and need not be) visited by this VACUUM.	
heap_blks_scanned	bigint	Number of heap blocks scanned. Because the visibility map is used to optimize scans, some blocks will be skipped without inspection; skipped blocks are included in this total, so that this number will eventually become equal to heap_blks_total when the vacuum is complete. This counter only advances when the phase is scanning heap.	
heap_blks_vacuumed	bigint	Number of heap blocks vacuumed. Unless the table has no indexes, this counter only advances when the phase is vacuuming heap. Blocks that contain no dead tuples are skipped, so the counter may sometimes skip forward in large increments.	
index_vacuum_count	bigint	Number of completed index vacuum cycles.	
max_dead_tuples	bigint	Number of dead tuples that we can store before needing to perform an index vacuum cycle, based on maintenance_work_mem.	
num_dead_tuples	bigint	Number of dead tuples collected since the last index vacuum cycle.	

VACUUM Phases

Phase	Description
initializing	VACUUM is preparing to begin scanning the heap. This phase is expected to be very brief.
scanning heap	VACUUM is currently scanning the heap. It will prune and defragment each page if required, and possibly perform freezing activity. The heap_blks_scanned column can be used to monitor the progress of the scan.

`vacuuming indexes`	`VACUUM` is currently vacuuming the indexes. If a table has any indexes, this will happen at least once per vacuum, after the heap has been completely scanned. It may happen multiple times per vacuum if maintenance_work_mem (or, in the case of autovacuum, autovacuum_work_mem if set) is insufficient to store the number of dead tuples found.
`vacuuming heap`	`VACUUM` is currently vacuuming the heap. Vacuuming the heap is distinct from scanning the heap, and occurs after each instance of vacuuming indexes. If `heap_blks_scanned` is less than `heap_blks_total`, the system will return to scanning the heap after this phase is completed; otherwise, it will begin cleaning up indexes after this phase is completed.
`cleaning up indexes`	`VACUUM` is currently cleaning up indexes. This occurs after the heap has been completely scanned and all vacuuming of the indexes and the heap has been completed.
`truncating heap`	`VACUUM` is currently truncating the heap so as to return empty pages at the end of the relation to the operating system. This occurs after cleaning up indexes.
`performing final cleanup`	`VACUUM` is performing final cleanup. During this phase, `VACUUM` will vacuum the free space map, update statistics in `pg_class`, and report statistics to the cumulative statistics system. When this phase is completed, `VACUUM` will end.

CLUSTER Progress Reporting

Whenever `CLUSTER` or `VACUUM FULL` is running, the `pg_stat_progress_cluster` view will contain a row for each backend that is currently running either command. The tables below describe the information that will be reported and provide information about how to interpret it.

`pg_stat_progress_cluster` View

Column	Type	Description	
`pid`	`integer`	Process ID of backend.	
`datid`	`oid`	OID of the database to which this backend is connected.	
`datname`	`name`	Name of the database to which this backend is connected.	
`relid`	`oid`	OID of the table being clustered.	
`command`	`text`	The command that is running. Either `CLUSTER` or `VACUUM FULL`.	
`phase`	`text`	Current processing phase. See Table 1.43.	
`cluster_index_relid`	`oid`	If the table is being scanned using an index, this is the OID of the index being used; otherwise, it is zero.	

heap_tuples_scanned	bigint	Number of heap tuples scanned. This counter only advances when the phase is seq scanning heap, index scanning heap or writing new heap.	
heap_tuples_written	bigint	Number of heap tuples written. This counter only advances when the phase is seq scanning heap, index scanning heap or writing new heap.	
heap_blks_total	bigint	Total number of heap blocks in the table. This number is reported as of the beginning of seq scanning heap.	
heap_blks_scanned	bigint	Number of heap blocks scanned. This counter only advances when the phase is seq scanning heap.	
index_rebuild_count	bigint	Number of indexes rebuilt. This counter only advances when the phase is rebuilding index.	

CLUSTER and VACUUM FULL Phases

Phase	Description
initializing	The command is preparing to begin scanning the heap. This phase is expected to be very brief.
seq scanning heap	The command is currently scanning the table using a sequential scan.
index scanning heap	CLUSTER is currently scanning the table using an index scan.
sorting tuples	CLUSTER is currently sorting tuples.
writing new heap	CLUSTER is currently writing the new heap.
swapping relation files	The command is currently swapping newly-built files into place.
rebuilding index	The command is currently rebuilding an index.
performing final cleanup	The command is performing final cleanup. When this phase is completed, CLUSTER or VACUUM FULL will end.

Base Backup Progress Reporting

Whenever an application like pg_basebackup is taking a base backup, the pg_stat_progress_basebackup view will contain a row for each WAL sender process that is currently running the BASE_BACKUP replication command and streaming the backup. The tables below describe the information that will be reported and provide information about how to interpret it.

pg_stat_progress_basebackup View

Column TypeDescription

pid `integer` Process ID of a WAL sender process.
phase `text` Current processing phase.
backup_total `bigint` **Total amount of data that will be streamed. This is estimated and reported as of the beginning of `streaming database files`** phase. Note that this is only an approximation since the database may change during **streaming database files** phase and WAL log may be included in the backup later. This is always the same value as **backup_streamed** once the amount of data streamed exceeds the estimated total size. If the estimation is disabled in pg_basebackup (i.e., **--no-estimate-size** option is specified), this is **NULL**.
backup_streamed `bigint` **Amount of data streamed. This counter only advances when the phase is `streaming database files`** or **transferring wal files**.
tablespaces_total `bigint` Total number of tablespaces that will be streamed.
tablespaces_streamed `bigint` **Number of tablespaces streamed. This counter only advances when the phase is `streaming database files`**.

Base Backup Phases

Phase	Description
initializing	The WAL sender process is preparing to begin the backup. This phase is expected to be very brief.
waiting for checkpoint to finish	The WAL sender process is currently performing **pg_backup_start** to prepare to take a base backup, and waiting for the start-of-backup checkpoint to finish.
estimating backup size	The WAL sender process is currently estimating the total amount of database files that will be streamed as a base backup.
streaming database files	The WAL sender process is currently streaming database files as a base backup.
waiting for wal archiving to finish	The WAL sender process is currently performing **pg_backup_stop** to finish the backup, and waiting for all the WAL files required for the base backup to be successfully archived. If either **--wal-method=none** or **--wal-method=stream** is specified in pg_basebackup, the backup will end when this phase is completed.
transferring wal files	The WAL sender process is currently transferring all WAL logs generated during the backup. This phase occurs after **waiting for wal archiving to finish** phase if **--wal-method=fetch** is specified in pg_basebackup. The backup will end when this phase is completed.

COPY Progress Reporting

Whenever **COPY** is running, the **pg_stat_progress_copy** view will contain one row for each backend that is currently running a **COPY** command. The table below describes the information that will be reported and provides information about how to interpret it.

pg_stat_progress_copy View

Column TypeDescription
pid `integer` Process ID of backend.
datid `oid` OID of the database to which this backend is connected.
datname `name` Name of the database to which this backend is connected.
relid `oid` **OID of the table on which the `COPY`** command is executed. It is set to **0** if copying from a **SELECT** query.

command text	`The command that is running: `COPY FROM`, or `COPY TO`.
type text	`The io type that the data is read from or written to: `FILE`, `PROGRAM`, `PIPE` (for `COPY FROM STDIN` and `COPY TO STDOUT`), or `CALLBACK` (used for example during the initial table synchronization in logical replication).
bytes_processed bigint	`Number of bytes already processed by `COPY` command.
bytes_total bigint	`Size of source file for `COPY FROM` command in bytes. It is set to `0` if not available.
tuples_processed bigint	`Number of tuples already processed by `COPY` command.
tuples_excluded bigint	`Number of tuples not processed because they were excluded by the `WHERE` clause of the `COPY` command.

Dynamic Tracing

IvorySQL provides facilities to support dynamic tracing of the database server. This allows an external utility to be called at specific points in the code and thereby trace execution.

A number of probes or trace points are already inserted into the source code. These probes are intended to be used by database developers and administrators. By default the probes are not compiled into IvorySQL; the user needs to explicitly tell the configure script to make the probes available.

Currently, the DTrace utility is supported, which, at the time of this writing, is available on Solaris, macOS, FreeBSD, NetBSD, and Oracle Linux. The SystemTap project for Linux provides a DTrace equivalent and can also be used. Supporting other dynamic tracing utilities is theoretically possible by changing the definitions for the macros in `src/include/utils/probes.h`.

Compiling for Dynamic Tracing

By default, probes are not available, so you will need to explicitly tell the configure script to make the probes available in IvorySQL. To include DTrace support specify `--enable-dtrace` to configure.

Built-in Probes

A number of standard probes are provided in the source code, More probes can certainly be added to enhance IvorySQL’ s observability.

Built-in DTrace Probes

Name	Parameters	Description
transaction-start	(LocalTransactionId)	Probe that fires at the start of a new transaction. arg0 is the transaction ID.
transaction-commit	(LocalTransactionId)	Probe that fires when a transaction completes successfully. arg0 is the transaction ID.
transaction-abort	(LocalTransactionId)	Probe that fires when a transaction completes unsuccessfully. arg0 is the transaction ID.
query-start	(const char *)	Probe that fires when the processing of a query is started. arg0 is the query string.
query-done	(const char *)	Probe that fires when the processing of a query is complete. arg0 is the query string.
query-parse-start	(const char *)	Probe that fires when the parsing of a query is started. arg0 is the query string.

query-parse-done	(const char *)	Probe that fires when the parsing of a query is complete. arg0 is the query string.
query-rewrite-start	(const char *)	Probe that fires when the rewriting of a query is started. arg0 is the query string.
query-rewrite-done	(const char *)	Probe that fires when the rewriting of a query is complete. arg0 is the query string.
query-plan-start	()	Probe that fires when the planning of a query is started.
query-plan-done	()	Probe that fires when the planning of a query is complete.
query-execute-start	()	Probe that fires when the execution of a query is started.
query-execute-done	()	Probe that fires when the execution of a query is complete.
statement-status	(const char *)	Probe that fires anytime the server process updates its pg_stat_activity.status. arg0 is the new status string.
checkpoint-start	(int)	Probe that fires when a checkpoint is started. arg0 holds the bitwise flags used to distinguish different checkpoint types, such as shutdown, immediate or force.
checkpoint-done	(int, int, int, int, int)	Probe that fires when a checkpoint is complete. (The probes listed next fire in sequence during checkpoint processing.) arg0 is the number of buffers written. arg1 is the total number of buffers. arg2, arg3 and arg4 contain the number of WAL files added, removed and recycled respectively.
clog-checkpoint-start	(bool)	Probe that fires when the CLOG portion of a checkpoint is started. arg0 is true for normal checkpoint, false for shutdown checkpoint.
clog-checkpoint-done	(bool)	Probe that fires when the CLOG portion of a checkpoint is complete. arg0 has the same meaning as for clog-checkpoint-start.
subtrans-checkpoint-start	(bool)	Probe that fires when the SUBTRANS portion of a checkpoint is started. arg0 is true for normal checkpoint, false for shutdown checkpoint.
subtrans-checkpoint-done	(bool)	Probe that fires when the SUBTRANS portion of a checkpoint is complete. arg0 has the same meaning as for subtrans-checkpoint-start.

multixact-checkpoint-start	(bool)	Probe that fires when the MultiXact portion of a checkpoint is started. arg0 is true for normal checkpoint, false for shutdown checkpoint.
multixact-checkpoint-done	(bool)	Probe that fires when the MultiXact portion of a checkpoint is complete. arg0 has the same meaning as for multixact-checkpoint-start.
buffer-checkpoint-start	(int)	Probe that fires when the buffer-writing portion of a checkpoint is started. arg0 holds the bitwise flags used to distinguish different checkpoint types, such as shutdown, immediate or force.
buffer-sync-start	(int, int)	Probe that fires when we begin to write dirty buffers during checkpoint (after identifying which buffers must be written). arg0 is the total number of buffers. arg1 is the number that are currently dirty and need to be written.
buffer-sync-written	(int)	Probe that fires after each buffer is written during checkpoint. arg0 is the ID number of the buffer.
buffer-sync-done	(int, int, int)	Probe that fires when all dirty buffers have been written. arg0 is the total number of buffers. arg1 is the number of buffers actually written by the checkpoint process. arg2 is the number that were expected to be written (arg1 of buffer-sync-start); any difference reflects other processes flushing buffers during the checkpoint.
buffer-checkpoint-sync-start	()	Probe that fires after dirty buffers have been written to the kernel, and before starting to issue fsync requests.
buffer-checkpoint-done	()	Probe that fires when syncing of buffers to disk is complete.
twophase-checkpoint-start	()	Probe that fires when the two-phase portion of a checkpoint is started.
twophase-checkpoint-done	()	Probe that fires when the two-phase portion of a checkpoint is complete.

buffer-read-start	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, bool)	Probe that fires when a buffer read is started. arg0 and arg1 contain the fork and block numbers of the page (but arg1 will be -1 if this is a relation extension request). arg2, arg3, and arg4 contain the tablespace, database, and relation OIDs identifying the relation. arg5 is the ID of the backend which created the temporary relation for a local buffer, or InvalidBackendId (-1) for a shared buffer. arg6 is true for a relation extension request, false for normal read.
buffer-read-done	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, bool, bool)	Probe that fires when a buffer read is complete. arg0 and arg1 contain the fork and block numbers of the page (if this is a relation extension request, arg1 now contains the block number of the newly added block). arg2, arg3, and arg4 contain the tablespace, database, and relation OIDs identifying the relation. arg5 is the ID of the backend which created the temporary relation for a local buffer, or InvalidBackendId (-1) for a shared buffer. arg6 is true for a relation extension request, false for normal read. arg7 is true if the buffer was found in the pool, false if not.
buffer-flush-start	(ForkNumber, BlockNumber, Oid, Oid, Oid)	Probe that fires before issuing any write request for a shared buffer. arg0 and arg1 contain the fork and block numbers of the page. arg2, arg3, and arg4 contain the tablespace, database, and relation OIDs identifying the relation.
buffer-flush-done	(ForkNumber, BlockNumber, Oid, Oid, Oid)	Probe that fires when a write request is complete. (Note that this just reflects the time to pass the data to the kernel; it' s typically not actually been written to disk yet.) The arguments are the same as for buffer-flush-start.
buffer-write-dirty-start	(ForkNumber, BlockNumber, Oid, Oid, Oid)	Probe that fires when a server process begins to write a dirty buffer. (If this happens often, it implies that shared_buffers is too small or the background writer control parameters need adjustment.) arg0 and arg1 contain the fork and block numbers of the page. arg2, arg3, and arg4 contain the tablespace, database, and relation OIDs identifying the relation.

buffer-write-dirty-done	(ForkNumber, BlockNumber, Oid, Oid, Oid)	Probe that fires when a dirty-buffer write is complete. The arguments are the same as for buffer-write-dirty-start.
wal-buffer-write-dirty-start	()	Probe that fires when a server process begins to write a dirty WAL buffer because no more WAL buffer space is available. (If this happens often, it implies that wal_buffers is too small.)
wal-buffer-write-dirty-done	()	Probe that fires when a dirty WAL buffer write is complete.
wal-insert	(unsigned char, unsigned char)	Probe that fires when a WAL record is inserted. arg0 is the resource manager (rmid) for the record. arg1 contains the info flags.
wal-switch	()	Probe that fires when a WAL segment switch is requested.
smgr-md-read-start	(ForkNumber, BlockNumber, Oid, Oid, Oid, int)	Probe that fires when beginning to read a block from a relation. arg0 and arg1 contain the fork and block numbers of the page. arg2, arg3, and arg4 contain the tablespace, database, and relation OIDs identifying the relation. arg5 is the ID of the backend which created the temporary relation for a local buffer, or InvalidBackendId (-1) for a shared buffer.
smgr-md-read-done	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, int, int)	Probe that fires when a block read is complete. arg0 and arg1 contain the fork and block numbers of the page. arg2, arg3, and arg4 contain the tablespace, database, and relation OIDs identifying the relation. arg5 is the ID of the backend which created the temporary relation for a local buffer, or InvalidBackendId (-1) for a shared buffer. arg6 is the number of bytes actually read, while arg7 is the number requested (if these are different it indicates trouble).
smgr-md-write-start	(ForkNumber, BlockNumber, Oid, Oid, Oid, int)	Probe that fires when beginning to write a block to a relation. arg0 and arg1 contain the fork and block numbers of the page. arg2, arg3, and arg4 contain the tablespace, database, and relation OIDs identifying the relation. arg5 is the ID of the backend which created the temporary relation for a local buffer, or InvalidBackendId (-1) for a shared buffer.

smgr-md-write-done	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, int, int)	Probe that fires when a block write is complete. arg0 and arg1 contain the fork and block numbers of the page. arg2, arg3, and arg4 contain the tablespace, database, and relation OIDs identifying the relation. arg5 is the ID of the backend which created the temporary relation for a local buffer, or InvalidBackendId (-1) for a shared buffer. arg6 is the number of bytes actually written, while arg7 is the number requested (if these are different it indicates trouble).
sort-start	(int, bool, int, int, bool, int)	Probe that fires when a sort operation is started. arg0 indicates heap, index or datum sort. arg1 is true for unique-value enforcement. arg2 is the number of key columns. arg3 is the number of kilobytes of work memory allowed. arg4 is true if random access to the sort result is required. arg5 indicates serial when 0, parallel worker when 1, or parallel leader when 2.
sort-done	(bool, long)	Probe that fires when a sort is complete. arg0 is true for external sort, false for internal sort. arg1 is the number of disk blocks used for an external sort, or kilobytes of memory used for an internal sort.
lwlock-acquire	(char *, LWLockMode)	Probe that fires when an LWLock has been acquired. arg0 is the LWLock's tranche. arg1 is the requested lock mode, either exclusive or shared.
lwlock-release	(char *)	Probe that fires when an LWLock has been released (but note that any released waiters have not yet been awakened). arg0 is the LWLock's tranche.
lwlock-wait-start	(char *, LWLockMode)	Probe that fires when an LWLock was not immediately available and a server process has begun to wait for the lock to become available. arg0 is the LWLock's tranche. arg1 is the requested lock mode, either exclusive or shared.
lwlock-wait-done	(char *, LWLockMode)	Probe that fires when a server process has been released from its wait for an LWLock (it does not actually have the lock yet). arg0 is the LWLock's tranche. arg1 is the requested lock mode, either exclusive or shared.

lwlock-condacquire	(char *, LWLockMode)	Probe that fires when an LWLock was successfully acquired when the caller specified no waiting. arg0 is the LWLock's tranche. arg1 is the requested lock mode, either exclusive or shared.
lwlock-condacquire-fail	(char *, LWLockMode)	Probe that fires when an LWLock was not successfully acquired when the caller specified no waiting. arg0 is the LWLock's tranche. arg1 is the requested lock mode, either exclusive or shared.
lock-wait-start	(unsigned int, unsigned int, unsigned int, unsigned int, unsigned int, LOCKMODE)	Probe that fires when a request for a heavyweight lock (lmgr lock) has begun to wait because the lock is not available. arg0 through arg3 are the tag fields identifying the object being locked. arg4 indicates the type of object being locked. arg5 indicates the lock type being requested.
lock-wait-done	(unsigned int, unsigned int, unsigned int, unsigned int, unsigned int, LOCKMODE)	Probe that fires when a request for a heavyweight lock (lmgr lock) has finished waiting (i.e., has acquired the lock). The arguments are the same as for lock-wait-start.
deadlock-found	()	Probe that fires when a deadlock is found by the deadlock detector.

Defined Types Used in Probe Parameters

Type	Definition
LocalTransactionId	unsigned int
LWLockMode	int
LOCKMODE	int
BlockNumber	unsigned int
Oid	unsigned int
ForkNumber	int
bool	unsigned char

Using Probes

The example below shows a DTrace script for analyzing transaction counts in the system, as an alternative to snapshotting pg_stat_database before and after a performance test:

```
#!/usr/sbin/dtrace -qs

postgresql$1:::transaction-start
{
      @start["Start"] = count();
      self->ts  = timestamp;
}
```

```
postgresql$1:::transaction-abort
{
      @abort["Abort"] = count();
}

postgresql$1:::transaction-commit
/self->ts/
{
      @commit["Commit"] = count();
      @time["Total time (ns)"] = sum(timestamp - self->ts);
      self->ts=0;
}
```

When executed, the example D script gives output such as:

```
# ./txn_count.d `pgrep -n postgres` or ./txn_count.d <PID>
^C

Start                                          71
Commit                                         70
Total time (ns)                          2312105013
```

> **Note**
>
> SystemTap uses a different notation for trace scripts than DTrace does, even though the underlying trace points are compatible. One point worth noting is that at this writing, SystemTap scripts must reference probe names using double underscores in place of hyphens. This is expected to be fixed in future SystemTap releases.

Defining New Probes

New probes can be defined within the code wherever the developer desires, though this will require a recompilation. Below are the steps for inserting new probes:

1. Decide on probe names and data to be made available through the probes
2. Add the probe definitions to `src/backend/utils/probes.d`
3. Include `pg_trace.h` if it is not already present in the module(s) containing the probe points, and insert `TRACE_POSTGRESQL` probe macros at the desired locations in the source code
4. Recompile and verify that the new probes are available

Example: Here is an example of how you would add a probe to trace all new transactions by transaction ID.

1. Decide that the probe will be named `transaction-start` and requires a parameter of type `LocalTransactionId`
2. Add the probe definition to `src/backend/utils/probes.d`:

```
probe transaction__start(LocalTransactionId);
```

```
Note the use of the double underline in the probe name. In a DTrace script using
the probe, the double underline needs to be replaced with a hyphen, so
`transaction-start` is the name to document for users.
```

3. At compile time, `transaction__start` is converted to a macro called `TRACE_POSTGRESQL_TRANSACTION_START` (notice the underscores are single here), which is available by including `pg_trace.h`. Add the macro call to the appropriate location in the source code. In this case, it looks like the following:

```
TRACE_POSTGRESQL_TRANSACTION_START(vxid.localTransactionId);
```

4. After recompiling and running the new binary, check that your newly added probe is available by executing the following DTrace command. You should see similar output:

```
# dtrace -ln transaction-start
   ID    PROVIDER          MODULE           FUNCTION NAME
18705 postgresql49878     postgres     StartTransactionCommand transaction-start
18755 postgresql49877     postgres     StartTransactionCommand transaction-start
18805 postgresql49876     postgres     StartTransactionCommand transaction-start
18855 postgresql49875     postgres     StartTransactionCommand transaction-start
18986 postgresql49873     postgres     StartTransactionCommand transaction-start
```

There are a few things to be careful about when adding trace macros to the C code:

- You should take care that the data types specified for a probe’ s parameters match the data types of the variables used in the macro. Otherwise, you will get compilation errors.
- On most platforms, if IvorySQL is built with `--enable-dtrace`, the arguments to a trace macro will be evaluated whenever control passes through the macro, even if no tracing is being done. This is usually not worth worrying about if you are just reporting the values of a few local variables. But beware of putting expensive function calls into the arguments. If you need to do that, consider protecting the macro with a check to see if the trace is actually enabled:

```
if (TRACE_POSTGRESQL_TRANSACTION_START_ENABLED())
    TRACE_POSTGRESQL_TRANSACTION_START(some_function(...));
```

Each trace macro has a corresponding `ENABLED` macro.

Monitoring Disk Usage

Determining Disk Usage

Each table has a primary heap disk file where most of the data is stored. If the table has any columns with potentially-wide values, there also might be a TOAST file associated with the table, which is used to store values too wide to fit comfortably in the main table . There will be one valid index on the TOAST table, if present. There also might be indexes associated with the base table. Each table and index is stored in a separate disk file — possibly more than one file, if the file would exceed one gigabyte.

You can monitor disk space in three ways: using the SQL functions, using the oid2name module, or using manual inspection of the system catalogs. The SQL functions are the easiest to use and are generally recommended. The remainder of this section shows how to do it by inspection of the system catalogs.

Using psql on a recently vacuumed or analyzed database, you can issue queries to see the disk usage of any table:

```
SELECT pg_relation_filepath(oid), relpages FROM pg_class WHERE relname = 'customer';

 pg_relation_filepath | relpages
----------------------+----------
 base/16384/16806     |       60
(1 row)
```

Each page is typically 8 kilobytes. (Remember, `relpages` is only updated by `VACUUM`, `ANALYZE`, and a few DDL commands such as `CREATE INDEX`.) The file path name is of interest if you want to examine the table's disk file directly.

To show the space used by TOAST tables, use a query like the following:

```
SELECT relname, relpages
FROM pg_class,
     (SELECT reltoastrelid
      FROM pg_class
      WHERE relname = 'customer') AS ss
WHERE oid = ss.reltoastrelid OR
      oid = (SELECT indexrelid
             FROM pg_index
             WHERE indrelid = ss.reltoastrelid)
ORDER BY relname;

       relname        | relpages
----------------------+----------
 pg_toast_16806       |        0
 pg_toast_16806_index |        1
```

You can easily display index sizes, too:

```
SELECT c2.relname, c2.relpages
```

```
FROM pg_class c, pg_class c2, pg_index i
WHERE c.relname = 'customer' AND
      c.oid = i.indrelid AND
      c2.oid = i.indexrelid
ORDER BY c2.relname;

      relname      | relpages
-------------------+----------
 customer_id_index |       26
```

It is easy to find your largest tables and indexes using this information:

```
SELECT relname, relpages
FROM pg_class
ORDER BY relpages DESC;

       relname        | relpages
----------------------+----------
 bigtable             |     3290
 customer             |     3144
```

Disk Full Failure

The most important disk monitoring task of a database administrator is to make sure the disk doesn’t become full. A filled data disk will not result in data corruption, but it might prevent useful activity from occurring. If the disk holding the WAL files grows full, database server panic and consequent shutdown might occur.

If you cannot free up additional space on the disk by deleting other things, you can move some of the database files to other file systems by making use of tablespaces.

Tip

Some file systems perform badly when they are almost full, so do not wait until the disk is completely full to take action.

If your system supports per-user disk quotas, then the database will naturally be subject to whatever quota is placed on the user the server runs as. Exceeding the quota will have the same bad effects as running out of disk space entirely.

.2. Maintenance

Routine Vacuuming

IvorySQL databases require periodic maintenance known as vacuuming. For many installations, it is sufficient to let vacuuming be performed by the autovacuum daemon. You might need to adjust the autovacuuming parameters described there to obtain best results for your situation. Some database administrators will want to supplement or replace the daemon’s activities with manually-managed `VACUUM` commands, which typically are executed according to a schedule by cron or Task Scheduler scripts. To set

up manually-managed vacuuming properly, it is essential to understand the issues discussed in the next few subsections. Administrators who rely on autovacuuming may still wish to skim this material to help them understand and adjust autovacuuming.

Vacuuming Basics

IvorySQL’s command has to process each table on a regular basis for several reasons:

1. To recover or reuse disk space occupied by updated or deleted rows.
2. To update data statistics used by the PostgreSQL query planner.
3. To update the visibility map, which speeds up index-only scans.
4. To protect against loss of very old data due to transaction ID wraparound or multixact ID wraparound.

Each of these reasons dictates performing `VACUUM` operations of varying frequency and scope, as explained in the following subsections.

There are two variants of `VACUUM`: standard `VACUUM` and `VACUUM FULL`. `VACUUM FULL` can reclaim more disk space but runs much more slowly. Also, the standard form of `VACUUM` can run in parallel with production database operations. (Commands such as `SELECT`, `INSERT`, `UPDATE`, and `DELETE` will continue to function normally, though you will not be able to modify the definition of a table with commands such as `ALTER TABLE` while it is being vacuumed.) `VACUUM FULL` requires an `ACCESS EXCLUSIVE` lock on the table it is working on, and therefore cannot be done in parallel with other use of the table. Generally, therefore, administrators should strive to use standard `VACUUM` and avoid `VACUUM FULL`.

`VACUUM` creates a substantial amount of I/O traffic, which can cause poor performance for other active sessions. There are configuration parameters that can be adjusted to reduce the performance impact of background vacuuming.

Recovering Disk Space

In IvorySQL， an `UPDATE` or `DELETE` of a row does not immediately remove the old version of the row. This approach is necessary to gain the benefits of multiversion concurrency control : the row version must not be deleted while it is still potentially visible to other transactions. But eventually, an outdated or deleted row version is no longer of interest to any transaction. The space it occupies must then be reclaimed for reuse by new rows, to avoid unbounded growth of disk space requirements. This is done by running `VACUUM`.

The standard form of `VACUUM` removes dead row versions in tables and indexes and marks the space available for future reuse. However, it will not return the space to the operating system, except in the special case where one or more pages at the end of a table become entirely free and an exclusive table lock can be easily obtained. In contrast, `VACUUM FULL` actively compacts tables by writing a complete new version of the table file with no dead space. This minimizes the size of the table, but can take a long time. It also requires extra disk space for the new copy of the table, until the operation completes.

The usual goal of routine vacuuming is to do standard `VACUUM`s often enough to avoid needing `VACUUM FULL`. The autovacuum daemon attempts to work this way, and in fact will never issue `VACUUM FULL`. In this approach, the idea is not to keep tables at their minimum size, but to maintain steady-state usage of disk space: each table occupies space equivalent to its minimum size plus however much space gets used up between vacuum runs. Although `VACUUM FULL` can be used to shrink a table back to its minimum size and return the disk space to the operating system, there is not much point in this if the table will just grow again in the future. Thus, moderately-frequent standard `VACUUM` runs are a better approach than infrequent `VACUUM FULL` runs for maintaining heavily-updated tables.

Some administrators prefer to schedule vacuuming themselves, for example doing all the work at night when load is low. The difficulty with doing vacuuming according to a fixed schedule is that if a table has an unexpected spike in update activity, it may get bloated to the point that `VACUUM FULL` is really necessary to reclaim space. Using the autovacuum daemon alleviates this problem, since the daemon schedules vacuuming dynamically in response to update activity. It is unwise to disable the daemon completely unless you have an extremely predictable workload. One possible compromise is to set the daemon’s parameters so that it will only react to unusually heavy update activity, thus keeping things from getting out of hand, while scheduled `VACUUM`s are expected to do the bulk of the work when the load is typical.

For those not using autovacuum, a typical approach is to schedule a database-wide `VACUUM` once a day during a low-usage period, supplemented by more frequent vacuuming of heavily-updated tables as necessary. (Some installations with extremely high update rates vacuum their busiest tables as often as once every few minutes.) If you have multiple databases in a cluster, don't forget to `VACUUM` each one; the program vacuumdb might be helpful.

> Tip
>
> Plain `VACUUM` may not be satisfactory when a table contains large numbers of dead row versions as a result of massive update or delete activity. If you have such a table and you need to reclaim the excess disk space it occupies, you will need to use `VACUUM FULL`, or alternatively `CLUSTER` or one of the table-rewriting variants of `ALTER TABLE`. These commands rewrite an entire new copy of the table and build new indexes for it. All these options require an `ACCESS EXCLUSIVE` lock. Note that they also temporarily use extra disk space approximately equal to the size of the table, since the old copies of the table and indexes can't be released until the new ones are complete.

> Tip
>
> If you have a table whose entire contents are deleted on a periodic basis, consider doing it with `TRUNCATE` rather than using `DELETE` followed by `VACUUM`. `TRUNCATE` removes the entire content of the table immediately, without requiring a subsequent `VACUUM` or `VACUUM FULL` to reclaim the now-unused disk space. The disadvantage is that strict MVCC semantics are violated.

Updating Planner Statistics

The IvorySQL query planner relies on statistical information about the contents of tables in order to generate good plans for queries. These statistics are gathered by the `ANALYZE` command, which can be invoked by itself or as an optional step in `VACUUM`. It is important to have reasonably accurate statistics, otherwise poor choices of plans might degrade database performance.

The autovacuum daemon, if enabled, will automatically issue `ANALYZE` commands whenever the content of a table has changed sufficiently. However, administrators might prefer to rely on manually-scheduled `ANALYZE` operations, particularly if it is known that update activity on a table will not affect the statistics of "interesting" columns. The daemon schedules `ANALYZE` strictly as a function of the number of rows inserted or updated; it has no knowledge of whether that will lead to meaningful statistical changes.

Tuples changed in partitions and inheritance children do not trigger analyze on the parent table. If the parent table is empty or rarely changed, it may never be processed by autovacuum, and the statistics for the inheritance tree as a whole won't be collected. It is necessary to run `ANALYZE` on the parent table manually in order to keep the statistics up to date.

As with vacuuming for space recovery, frequent updates of statistics are more useful for heavily-updated tables than for seldom-updated ones. But even for a heavily-updated table, there might be no need for statistics updates if the statistical distribution of the data is not changing much. A simple rule of thumb is to think about how much the minimum and maximum values of the columns in the table change. For example, a `timestamp` column that contains the time of row update will have a constantly-increasing maximum value as rows are added and updated; such a column will probably need more frequent statistics updates than, say, a column containing URLs for pages accessed on a website. The URL column might receive changes just as often, but the statistical distribution of its values probably changes relatively slowly.

It is possible to run `ANALYZE` on specific tables and even just specific columns of a table, so the flexibility exists to update some statistics more frequently than others if your application requires it. In practice, however, it is usually best to just analyze the entire database, because it is a fast operation. `ANALYZE` uses a statistically random sampling of the rows of a table rather than reading every single row.

> Tip
>
> Although per-column tweaking of `ANALYZE` frequency might not be very productive, you might find it

worthwhile to do per-column adjustment of the level of detail of the statistics collected by `ANALYZE`. Columns that are heavily used in `WHERE` clauses and have highly irregular data distributions might require a finer-grain data histogram than other columns. See `ALTER TABLE SET STATISTICS`, or change the database-wide default using the default_statistics_target configuration parameter.

Tip

The autovacuum daemon does not issue `ANALYZE` commands for foreign tables, since it has no means of determining how often that might be useful. If your queries require statistics on foreign tables for proper planning, it's a good idea to run manually-managed `ANALYZE` commands on those tables on a suitable schedule.

Tip

The autovacuum daemon does not issue `ANALYZE` commands for partitioned tables. Inheritance parents will only be analyzed if the parent itself is changed - changes to child tables do not trigger autoanalyze on the parent table. If your queries require statistics on parent tables for proper planning, it is necessary to periodically run a manual `ANALYZE` on those tables to keep the statistics up to date.

Updating the Visibility Map

Vacuum maintains a visibility map for each table to keep track of which pages contain only tuples that are known to be visible to all active transactions (and all future transactions, until the page is again modified). This has two purposes. First, vacuum itself can skip such pages on the next run, since there is nothing to clean up.

Second, it allows IvorySQL to answer some queries using only the index, without reference to the underlying table. Since PostgreSQL indexes don't contain tuple visibility information, a normal index scan fetches the heap tuple for each matching index entry, to check whether it should be seen by the current transaction. An index-only scan, on the other hand, checks the visibility map first. If it's known that all tuples on the page are visible, the heap fetch can be skipped. This is most useful on large data sets where the visibility map can prevent disk accesses. The visibility map is vastly smaller than the heap, so it can easily be cached even when the heap is very large.

Preventing Transaction ID Wraparound Failures

IvorySQL's MVCC transaction semantics depend on being able to compare transaction ID (XID) numbers: a row version with an insertion XID greater than the current transaction's XID is "in the future" and should not be visible to the current transaction. But since transaction IDs have limited size (32 bits) a cluster that runs for a long time (more than 4 billion transactions) would suffer transaction ID wraparound: the XID counter wraps around to zero, and all of a sudden transactions that were in the past appear to be in the future — which means their output become invisible. In short, catastrophic data loss. (Actually the data is still there, but that's cold comfort if you cannot get at it.) To avoid this, it is necessary to vacuum every table in every database at least once every two billion transactions.

The reason that periodic vacuuming solves the problem is that `VACUUM` will mark rows as frozen, indicating that they were inserted by a transaction that committed sufficiently far in the past that the effects of the inserting transaction are certain to be visible to all current and future transactions. Normal XIDs are compared using modulo-232 arithmetic. This means that for every normal XID, there are two billion XIDs that are "older" and two billion that are "newer"; another way to say it is that the normal XID space is circular with no endpoint. Therefore, once a row version has been created with a particular normal XID, the row version will appear to be "in the past" for the next two billion transactions, no matter which normal XID we are talking about. If the row version still exists after more than two billion transactions, it will suddenly appear to be in the future. To prevent this, IvorySQL reserves a special XID, `FrozenTransactionId`, which does not follow the normal XID comparison rules and is always considered older than every normal XID. Frozen row versions are treated as if the inserting XID were `FrozenTransactionId`, so that they will appear to be "in the past" to all normal transactions regardless of wraparound issues, and so such row versions will be valid

until deleted, no matter how long that is.

vacuum_freeze_min_age controls how old an XID value has to be before rows bearing that XID will be frozen. Increasing this setting may avoid unnecessary work if the rows that would otherwise be frozen will soon be modified again, but decreasing this setting increases the number of transactions that can elapse before the table must be vacuumed again.

VACUUM uses the visibility map to determine which pages of a table must be scanned. Normally, it will skip pages that don't have any dead row versions even if those pages might still have row versions with old XID values. Therefore, normal VACUUM`s won't always freeze every old row version in the table. When that happens, `VACUUM will eventually need to perform an aggressive vacuum, which will freeze all eligible unfrozen XID and MXID values, including those from all-visible but not all-frozen pages. In practice most tables require periodic aggressive vacuuming. vacuum_freeze_table_age controls when VACUUM does that: all-visible but not all-frozen pages are scanned if the number of transactions that have passed since the last such scan is greater than vacuum_freeze_table_age minus vacuum_freeze_min_age. Setting vacuum_freeze_table_age to 0 forces VACUUM to always use its aggressive strategy.

The maximum time that a table can go unvacuumed is two billion transactions minus the vacuum_freeze_min_age value at the time of the last aggressive vacuum. If it were to go unvacuumed for longer than that, data loss could result. To ensure that this does not happen, autovacuum is invoked on any table that might contain unfrozen rows with XIDs older than the age specified by the configuration parameter autovacuum_freeze_max_age. (This will happen even if autovacuum is disabled.)

This implies that if a table is not otherwise vacuumed, autovacuum will be invoked on it approximately once every autovacuum_freeze_max_age minus vacuum_freeze_min_age transactions. For tables that are regularly vacuumed for space reclamation purposes, this is of little importance. However, for static tables (including tables that receive inserts, but no updates or deletes), there is no need to vacuum for space reclamation, so it can be useful to try to maximize the interval between forced autovacuums on very large static tables. Obviously one can do this either by increasing autovacuum_freeze_max_age or decreasing vacuum_freeze_min_age.

The effective maximum for vacuum_freeze_table_age is 0.95 * autovacuum_freeze_max_age; a setting higher than that will be capped to the maximum. A value higher than autovacuum_freeze_max_age wouldn't make sense because an anti-wraparound autovacuum would be triggered at that point anyway, and the 0.95 multiplier leaves some breathing room to run a manual VACUUM before that happens. As a rule of thumb, vacuum_freeze_table_age should be set to a value somewhat below autovacuum_freeze_max_age, leaving enough gap so that a regularly scheduled VACUUM or an autovacuum triggered by normal delete and update activity is run in that window. Setting it too close could lead to anti-wraparound autovacuums, even though the table was recently vacuumed to reclaim space, whereas lower values lead to more frequent aggressive vacuuming.

The sole disadvantage of increasing autovacuum_freeze_max_age (and vacuum_freeze_table_age along with it) is that the pg_xact and pg_commit_ts subdirectories of the database cluster will take more space, because it must store the commit status and (if track_commit_timestamp is enabled) timestamp of all transactions back to the autovacuum_freeze_max_age horizon. The commit status uses two bits per transaction, so if autovacuum_freeze_max_age is set to its maximum allowed value of two billion, pg_xact can be expected to grow to about half a gigabyte and pg_commit_ts to about 20GB. If this is trivial compared to your total database size, setting autovacuum_freeze_max_age to its maximum allowed value is recommended. Otherwise, set it depending on what you are willing to allow for pg_xact and pg_commit_ts storage. (The default, 200 million transactions, translates to about 50MB of pg_xact storage and about 2GB of pg_commit_ts storage.)

One disadvantage of decreasing vacuum_freeze_min_age is that it might cause VACUUM to do useless work: freezing a row version is a waste of time if the row is modified soon thereafter (causing it to acquire a new XID). So the setting should be large enough that rows are not frozen until they are unlikely to change any more.

To track the age of the oldest unfrozen XIDs in a database, VACUUM stores XID statistics in the system tables pg_class and pg_database. In particular, the relfrozenxid column of a table's pg_class row contains the oldest remaining unfrozen XID at the end of the most recent VACUUM that successfully advanced relfrozenxid (typically the most recent aggressive VACUUM). Similarly, the datfrozenxid column of a database's pg_database row is a lower bound on the unfrozen XIDs appearing in that database — it is just the minimum of the per-table relfrozenxid values within the database. A convenient way to examine this

information is to execute queries such as:

```
SELECT c.oid::regclass as table_name,
       greatest(age(c.relfrozenxid),age(t.relfrozenxid)) as age
FROM pg_class c
LEFT JOIN pg_class t ON c.reltoastrelid = t.oid
WHERE c.relkind IN ('r', 'm');

SELECT datname, age(datfrozenxid) FROM pg_database;
```

The age column measures the number of transactions from the cutoff XID to the current transaction's XID.

VACUUM normally only scans pages that have been modified since the last vacuum, but relfrozenxid can only be advanced when every page of the table that might contain unfrozen XIDs is scanned. This happens when relfrozenxid is more than vacuum_freeze_table_age transactions old, when VACUUM's `FREEZE option is used, or when all pages that are not already all-frozen happen to require vacuuming to remove dead row versions. When VACUUM scans every page in the table that is not already all-frozen, it should set age(relfrozenxid) to a value just a little more than the vacuum_freeze_min_age setting that was used (more by the number of transactions started since the VACUUM started). VACUUM will set relfrozenxid to the oldest XID that remains in the table, so it's possible that the final value will be much more recent than strictly required. If no relfrozenxid-advancing VACUUM is issued on the table until autovacuum_freeze_max_age is reached, an autovacuum will soon be forced for the table.

If for some reason autovacuum fails to clear old XIDs from a table, the system will begin to emit warning messages like this when the database's oldest XIDs reach forty million transactions from the wraparound point:

```
WARNING:  database "mydb" must be vacuumed within 39985967 transactions
HINT:  To avoid a database shutdown, execute a database-wide VACUUM in that database.
```

(A manual VACUUM should fix the problem, as suggested by the hint; but note that the VACUUM must be performed by a superuser, else it will fail to process system catalogs and thus not be able to advance the database's datfrozenxid.) If these warnings are ignored, the system will shut down and refuse to start any new transactions once there are fewer than three million transactions left until wraparound:

```
ERROR:  database is not accepting commands to avoid wraparound data loss in database
"mydb"
HINT:  Stop the postmaster and vacuum that database in single-user mode.
```

The three-million-transaction safety margin exists to let the administrator recover without data loss, by manually executing the required VACUUM commands. However, since the system will not execute commands once it has gone into the safety shutdown mode, the only way to do this is to stop the server and start the server in single-user mode to execute VACUUM. The shutdown mode is not enforced in single-user mode. See the postgres reference page for details about using single-user mode.

Multixact IDs are used to support row locking by multiple transactions. Since there is only limited space in a tuple header to store lock information, that information is encoded as a "multiple transaction ID", or multixact ID for short, whenever there is more than one transaction concurrently locking a row. Information about which transaction IDs are included in any particular multixact ID is stored separately in the pg_multixact subdirectory, and only the multixact ID appears in the xmax field in the tuple header. Like transaction IDs, multixact IDs are implemented as a 32-bit counter and corresponding storage, all of which requires careful aging management, storage cleanup, and wraparound handling. There is a separate storage area which holds the list of members in each multixact, which also uses a 32-bit counter and which must

also be managed.

Whenever VACUUM scans any part of a table, it will replace any multixact ID it encounters which is older than vacuum_multixact_freeze_min_age by a different value, which can be the zero value, a single transaction ID, or a newer multixact ID. For each table, pg_class.relminmxid stores the oldest possible multixact ID still appearing in any tuple of that table. If this value is older than vacuum_multixact_freeze_table_age, an aggressive vacuum is forced. As discussed in the previous section, an aggressive vacuum means that only those pages which are known to be all-frozen will be skipped. mxid_age() can be used on pg_class .relminmxid to find its age.

Aggressive VACUUM`s, regardless of what causes them, are guaranteed to be able to advance the table' s `relminmxid. Eventually, as all tables in all databases are scanned and their oldest multixact values are advanced, on-disk storage for older multixacts can be removed.

As a safety device, an aggressive vacuum scan will occur for any table whose multixact-age is greater than autovacuum_multixact_freeze_max_age. Also, if the storage occupied by multixacts members exceeds 2GB, aggressive vacuum scans will occur more often for all tables, starting with those that have the oldest multixact-age. Both of these kinds of aggressive scans will occur even if autovacuum is nominally disabled.

The Autovacuum Daemon

IvorySQL has an optional but highly recommended feature called autovacuum, whose purpose is to automate the execution of VACUUM and ANALYZE commands. When enabled, autovacuum checks for tables that have had a large number of inserted, updated or deleted tuples. These checks use the statistics collection facility; therefore, autovacuum cannot be used unless track_counts is set to true. In the default configuration, autovacuuming is enabled and the related configuration parameters are appropriately set.

The "autovacuum daemon" actually consists of multiple processes. There is a persistent daemon process, called the autovacuum launcher, which is in charge of starting autovacuum worker processes for all databases. The launcher will distribute the work across time, attempting to start one worker within each database every autovacuum_naptime seconds. (Therefore, if the installation has N databases, a new worker will be launched every autovacuum_naptime/N seconds.) A maximum of autovacuum_max_workers worker processes are allowed to run at the same time. If there are more than autovacuum_max_workers databases to be processed, the next database will be processed as soon as the first worker finishes. Each worker process will check each table within its database and execute VACUUM and/or ANALYZE as needed. log_autovacuum_min_duration can be set to monitor autovacuum workers' activity.

If several large tables all become eligible for vacuuming in a short amount of time, all autovacuum workers might become occupied with vacuuming those tables for a long period. This would result in other tables and databases not being vacuumed until a worker becomes available. There is no limit on how many workers might be in a single database, but workers do try to avoid repeating work that has already been done by other workers. Note that the number of running workers does not count towards max_connections or superuser_reserved_connections limits.

Tables whose relfrozenxid value is more than autovacuum_freeze_max_age transactions old are always vacuumed (this also applies to those tables whose freeze max age has been modified via storage parameters; see below). Otherwise, if the number of tuples obsoleted since the last VACUUM exceeds the "vacuum threshold", the table is vacuumed. The vacuum threshold is defined as:

```
vacuum threshold = vacuum base threshold + vacuum scale factor * number of tuples
```

where the vacuum base threshold is autovacuum_vacuum_threshold, the vacuum scale factor is autovacuum_vacuum_scale_factor, and the number of tuples is pg_class.reltuples.

The table is also vacuumed if the number of tuples inserted since the last vacuum has exceeded the defined insert threshold, which is defined as:

```
vacuum insert threshold = vacuum base insert threshold + vacuum insert scale factor *
```

```
number of tuples
```

where the vacuum insert base threshold is autovacuum_vacuum_insert_threshold, and vacuum insert scale factor is autovacuum_vacuum_insert_scale_factor. Such vacuums may allow portions of the table to be marked as all visible and also allow tuples to be frozen, which can reduce the work required in subsequent vacuums. For tables which receive **INSERT** operations but no or almost no **UPDATE**/**DELETE** operations, it may be beneficial to lower the table's autovacuum_freeze_min_age as this may allow tuples to be frozen by earlier vacuums. The number of obsolete tuples and the number of inserted tuples are obtained from the cumulative statistics system; it is a semi-accurate count updated by each **UPDATE**, **DELETE** and **INSERT** operation. (It is only semi-accurate because some information might be lost under heavy load.) If the **relfrozenxid** value of the table is more than **vacuum_freeze_table_age** transactions old, an aggressive vacuum is performed to freeze old tuples and advance **relfrozenxid**; otherwise, only pages that have been modified since the last vacuum are scanned.

For analyze, a similar condition is used: the threshold, defined as:

```
analyze threshold = analyze base threshold + analyze scale factor * number of tuples
```

is compared to the total number of tuples inserted, updated, or deleted since the last **ANALYZE**.

Partitioned tables are not processed by autovacuum. Statistics should be collected by running a manual **ANALYZE** when it is first populated, and again whenever the distribution of data in its partitions changes significantly.

Temporary tables cannot be accessed by autovacuum. Therefore, appropriate vacuum and analyze operations should be performed via session SQL commands.

The default thresholds and scale factors are taken from **postgresql.conf**, but it is possible to override them (and many other autovacuum control parameters) on a per-table basis; see Storage Parameters for more information. If a setting has been changed via a table's storage parameters, that value is used when processing that table; otherwise the global settings are used. See Section 20.10 for more details on the global settings.

When multiple workers are running, the autovacuum cost delay parameters (see Section 20.4.4) are "balanced" among all the running workers, so that the total I/O impact on the system is the same regardless of the number of workers actually running. However, any workers processing tables whose per-table **autovacuum_vacuum_cost_delay** or **autovacuum_vacuum_cost_limit** storage parameters have been set are not considered in the balancing algorithm.

Autovacuum workers generally don't block other commands. If a process attempts to acquire a lock that conflicts with the **SHARE UPDATE EXCLUSIVE** lock held by autovacuum, lock acquisition will interrupt the autovacuum. For conflicting lock modes, see Table 13.3. However, if the autovacuum is running to prevent transaction ID wraparound (i.e., the autovacuum query name in the **pg_stat_activity** view ends with **(to prevent wraparound)**), the autovacuum is not automatically interrupted.

> **Warning**
>
> Regularly running commands that acquire locks conflicting with a **SHARE UPDATE EXCLUSIVE** lock (e.g., ANALYZE) can effectively prevent autovacuums from ever completing.

Routine Reindexing

In some situations it is worthwhile to rebuild indexes periodically with the REINDEX command or a series of individual rebuilding steps.

B-tree index pages that have become completely empty are reclaimed for re-use. However, there is still a possibility of inefficient use of space: if all but a few index keys on a page have been deleted, the page

remains allocated. Therefore, a usage pattern in which most, but not all, keys in each range are eventually deleted will see poor use of space. For such usage patterns, periodic reindexing is recommended.

The potential for bloat in non-B-tree indexes has not been well researched. It is a good idea to periodically monitor the index's physical size when using any non-B-tree index type.

Also, for B-tree indexes, a freshly-constructed index is slightly faster to access than one that has been updated many times because logically adjacent pages are usually also physically adjacent in a newly built index. (This consideration does not apply to non-B-tree indexes.) It might be worthwhile to reindex periodically just to improve access speed.

REINDEX can be used safely and easily in all cases. This command requires an **ACCESS EXCLUSIVE** lock by default, hence it is often preferable to execute it with its **CONCURRENTLY** option, which requires only a **SHARE UPDATE EXCLUSIVE** lock.

Log File Maintenance

It is a good idea to save the database server's log output somewhere, rather than just discarding it via **/dev/null**. The log output is invaluable when diagnosing problems.Log output tends to be voluminous (especially at higher debug levels) so you won't want to save it indefinitely. You need to rotate the log files so that new log files are started and old ones removed after a reasonable period of time.

If you simply direct the stderr of **postgres** into a file, you will have log output, but the only way to truncate the log file is to stop and restart the server. This might be acceptable if you are using PostgreSQL in a development environment, but few production servers would find this behavior acceptable.

A better approach is to send the server's stderr output to some type of log rotation program. There is a built-in log rotation facility, which you can use by setting the configuration parameter **logging_collector** to **true** in **postgresql.conf**. You can also use this approach to capture the log data in machine readable CSV (comma-separated values) format.

Alternatively, you might prefer to use an external log rotation program if you have one that you are already using with other server software. For example, the rotatelogs tool included in the Apache distribution can be used with PostgreSQL. One way to do this is to pipe the server's stderr output to the desired program. If you start the server with **pg_ctl**, then stderr is already redirected to stdout, so you just need a pipe command, for example:

```
pg_ctl start | rotatelogs /var/log/pgsql_log 86400
```

You can combine these approaches by setting up logrotate to collect log files produced by PostgreSQL built-in logging collector. In this case, the logging collector defines the names and location of the log files, while logrotate periodically archives these files. When initiating log rotation, logrotate must ensure that the application sends further output to the new file. This is commonly done with a **postrotate** script that sends a **SIGHUP** signal to the application, which then reopens the log file. In PostgreSQL, you can run **pg_ctl** with the **logrotate** option instead. When the server receives this command, the server either switches to a new log file or reopens the existing file, depending on the logging configuration.

> **Note**
>
> When using static log file names, the server might fail to reopen the log file if the max open file limit is reached or a file table overflow occurs. In this case, log messages are sent to the old log file until a successful log rotation. If logrotate is configured to compress the log file and delete it, the server may lose the messages logged in this time frame. To avoid this issue, you can configure the logging collector to dynamically assign log file names and use a **prerotate** script to ignore open log files.

Another production-grade approach to managing log output is to send it to syslog and let syslog deal with file rotation. To do this, set the configuration parameter **log_destination** to **syslog** (to log to syslog only) in **postgresql.conf**. Then you can send a **SIGHUP** signal to the syslog daemon whenever you want to force it to

start writing a new log file. If you want to automate log rotation, the logrotate program can be configured to work with log files from syslog.

On many systems, however, syslog is not very reliable, particularly with large log messages; it might truncate or drop messages just when you need them the most. Also, on Linux, syslog will flush each message to disk, yielding poor performance. (You can use a "-" at the start of the file name in the syslog configuration file to disable syncing.)

Note that all the solutions described above take care of starting new log files at configurable intervals, but they do not handle deletion of old, no-longer-useful log files. You will probably want to set up a batch job to periodically delete old log files. Another possibility is to configure the rotation program so that old log files are overwritten cyclically.

pgBadger is an external project that does sophisticated log file analysis. check_postgres provides Nagios alerts when important messages appear in the log files, as well as detection of many other extraordinary conditions.

High Availability, Load Balancing, and Replication

Comparison of Different Solutions

Shared Disk Failover

Shared disk failover avoids synchronization overhead by having only one copy of the database. It uses a single disk array that is shared by multiple servers. If the main database server fails, the standby server is able to mount and start the database as though it were recovering from a database crash. This allows rapid failover with no data loss.

Shared hardware functionality is common in network storage devices. Using a network file system is also possible, though care must be taken that the file system has full POSIX behavior . One significant limitation of this method is that if the shared disk array fails or becomes corrupt, the primary and standby servers are both nonfunctional. Another issue is that the standby server should never access the shared storage while the primary server is running.

File System (Block Device) Replication

A modified version of shared hardware functionality is file system replication, where all changes to a file system are mirrored to a file system residing on another computer. The only restriction is that the mirroring must be done in a way that ensures the standby server has a consistent copy of the file system — specifically, writes to the standby must be done in the same order as those on the primary. DRBD is a popular file system replication solution for Linux.

Write-Ahead Log Shipping

Warm and hot standby servers can be kept current by reading a stream of write-ahead log (WAL) records. If the main server fails, the standby contains almost all of the data of the main server, and can be quickly made the new primary database server. This can be synchronous or asynchronous and can only be done for the entire database server.

A standby server can be implemented using file-based log shipping or streaming replication, or a combination of both. For information on hot standby

Logical Replication

Logical replication allows a database server to send a stream of data modifications to another server. IvorySQL logical replication constructs a stream of logical data modifications from the WAL. Logical replication allows replication of data changes on a per-table basis. In addition, a server that is publishing its own changes can also subscribe to changes from another server, allowing data to flow in multiple directions. For more information on logical replication. Through the logical decoding interface , third-party extensions can also provide similar functionality.

Trigger-Based Primary-Standby Replication

A trigger-based replication setup typically funnels data modification queries to a designated primary server. Operating on a per-table basis, the primary server sends data changes (typically) asynchronously to the standby servers. Standby servers can answer queries while the primary is running, and may allow some local data changes or write activity. This form of replication is often used for offloading large analytical or data warehouse queries.

Slony-I is an example of this type of replication, with per-table granularity, and support for multiple standby servers. Because it updates the standby server asynchronously (in batches), there is possible data loss during fail over.

SQL-Based Replication Middleware

With SQL-based replication middleware, a program intercepts every SQL query and sends it to one or all servers. Each server operates independently. Read-write queries must be sent to all servers, so that every server receives any changes. But read-only queries can be sent to just one server, allowing the read workload to be distributed among them.

If queries are simply broadcast unmodified, functions like `random()`, `CURRENT_TIMESTAMP`, and sequences can have different values on different servers. This is because each server operates independently, and because SQL queries are broadcast rather than actual data changes. If this is unacceptable, either the middleware or the application must determine such values from a single source and then use those values in write queries. Care must also be taken that all transactions either commit or abort on all servers, perhaps using two-phase commit (PREPARE TRANSACTION and COMMIT PREPARED). Pgpool-II and Continuent Tungsten are examples of this type of replication.

Asynchronous Multimaster Replication

For servers that are not regularly connected or have slow communication links, like laptops or remote servers, keeping data consistent among servers is a challenge. Using asynchronous multimaster replication, each server works independently, and periodically communicates with the other servers to identify conflicting transactions. The conflicts can be resolved by users or conflict resolution rules. Bucardo is an example of this type of replication.

Synchronous Multimaster Replication

In synchronous multimaster replication, each server can accept write requests, and modified data is transmitted from the original server to every other server before each transaction commits. Heavy write activity can cause excessive locking and commit delays, leading to poor performance. Read requests can be sent to any server. Some implementations use shared disk to reduce the communication overhead. Synchronous multimaster replication is best for mostly read workloads, though its big advantage is that any server can accept write requests — there is no need to partition workloads between primary and standby servers, and because the data changes are sent from one server to another, there is no problem with non-deterministic functions like `random()`.

IvorySQL does not offer this type of replication, though PostgreSQL two-phase commit (PREPARE TRANSACTION and COMMIT PREPARED) can be used to implement this in application code or middleware.

The following table summarizes the capabilities of each of these scenarios.

Feature	Shared Disk	File System Repl.	Write-Ahead Log Shipping	Logical Repl.	Trigger-Based Repl.	SQL Repl. Middle-ware	Async. MM Repl.	Sync. MM Repl.
Popular examples	NAS	DRBD	built-in streaming repl.	built-in logical repl., pglogical	Londiste, Slony	pgpool-II	Bucardo	
Comm. method	shared disk	disk blocks	WAL	logical decoding	table rows	SQL	table rows	table rows and row locks

No special hardware required		•	•	•	•	•	•	•
Allows multiple primary servers				•		•	•	•
No overhead on primary	•		•	•		•		
No waiting for multiple servers	•		with sync off	with sync off	•		•	
Primary failure will never lose data	•	•	with sync on	with sync on		•		•
Replicas accept read-only queries			with hot standby	•	•	•	•	•
Per-table granularity				•	•		•	•
No conflict resolution necessary	•	•	•		•	•		•

There are a few solutions that do not fit into the above categories:

- Data Partitioning

  ```
  Data partitioning splits tables into data sets. Each set can be modified by only
  one server. For example, data can be partitioned by offices, e.g., London and
  Paris, with a server in each office. If queries combining London and Paris data are
  necessary, an application can query both servers, or primary/standby replication
  can be used to keep a read-only copy of the other office's data on each server.
  ```

- Multiple-Server Parallel Query Execution

  ```
  Many of the above solutions allow multiple servers to handle multiple queries, but
  none allow a single query to use multiple servers to complete faster. This solution
  allows multiple servers to work concurrently on a single query. It is usually
  accomplished by splitting the data among servers and having each server execute its
  part of the query and return results to a central server where they are combined
  and returned to the user. This can be implemented using the PL/Proxy tool set.
  ```

Log-Shipping Standby Servers

Planning

It is usually wise to create the primary and standby servers so that they are as similar as possible, at least from the perspective of the database server. In particular, the path names associated with tablespaces will

be passed across unmodified, so both primary and standby servers must have the same mount paths for tablespaces if that feature is used. Keep in mind that if CREATE TABLESPACE is executed on the primary, any new mount point needed for it must be created on the primary and all standby servers before the command is executed. Hardware need not be exactly the same, but experience shows that maintaining two identical systems is easier than maintaining two dissimilar ones over the lifetime of the application and system. In any case the hardware architecture must be the same — shipping from, say, a 32-bit to a 64-bit system will not work.

In general, log shipping between servers running different major IvorySQL release levels is not possible. It is the policy of the IvorySQL Global Development Group not to make changes to disk formats during minor release upgrades, so it is likely that running different minor release levels on primary and standby servers will work successfully. However, no formal support for that is offered and you are advised to keep primary and standby servers at the same release level as much as possible. When updating to a new minor release, the safest policy is to update the standby servers first — a new minor release is more likely to be able to read WAL files from a previous minor release than vice versa.

Standby Server Operation

A server enters standby mode if a `standby.signal` file exists in the data directory when the server is started.

In standby mode, the server continuously applies WAL received from the primary server. The standby server can read WAL from a WAL archive (see restore_command) or directly from the primary over a TCP connection (streaming replication). The standby server will also attempt to restore any WAL found in the standby cluster's `pg_wal` directory. That typically happens after a server restart, when the standby replays again WAL that was streamed from the primary before the restart, but you can also manually copy files to `pg_wal` at any time to have them replayed.

At startup, the standby begins by restoring all WAL available in the archive location, calling `restore_command`. Once it reaches the end of WAL available there and `restore_command` fails, it tries to restore any WAL available in the `pg_wal` directory. If that fails, and streaming replication has been configured, the standby tries to connect to the primary server and start streaming WAL from the last valid record found in archive or `pg_wal`. If that fails or streaming replication is not configured, or if the connection is later disconnected, the standby goes back to step 1 and tries to restore the file from the archive again. This loop of retries from the archive, `pg_wal`, and via streaming replication goes on until the server is stopped or failover is triggered by a trigger file.

Standby mode is exited and the server switches to normal operation when `pg_ctl promote` is run, `pg_promote()` is called, or a trigger file is found (`promote_trigger_file`). Before failover, any WAL immediately available in the archive or in `pg_wal` will be restored, but no attempt is made to connect to the primary.

Preparing the Primary for Standby Servers

Set up continuous archiving on the primary to an archive directory accessible from the standby.The archive location should be accessible from the standby even when the primary is down, i.e., it should reside on the standby server itself or another trusted server, not on the primary server.

If you want to use streaming replication, set up authentication on the primary server to allow replication connections from the standby server(s); that is, create a role and provide a suitable entry or entries in `pg_hba.conf` with the database field set to `replication`. Also ensure `max_wal_senders` is set to a sufficiently large value in the configuration file of the primary server. If replication slots will be used, ensure that `max_replication_slots` is set sufficiently high as well.

Setting Up a Standby Server

To set up the standby server, restore the base backup taken from primary server . Create a file `standby.signal` in the standby's cluster data directory. Set restore_command to a simple command to copy files from the WAL archive. If you plan to have multiple standby servers for high availability purposes, make sure that `recovery_target_timeline` is set to `latest` (the default), to make the standby server follow the timeline change that occurs at failover to another standby.

> **Note**
>
> restore_command should return immediately if the file does not exist; the server will retry the command again if necessary.

If you want to use streaming replication, fill in primary_conninfo with a libpq connection string, including the host name (or IP address) and any additional details needed to connect to the primary server. If the primary needs a password for authentication, the password needs to be specified in primary_conninfo as well.

If you're setting up the standby server for high availability purposes, set up WAL archiving, connections and authentication like the primary server, because the standby server will work as a primary server after failover.

If you're using a WAL archive, its size can be minimized using the archive_cleanup_command parameter to remove files that are no longer required by the standby server. The pg_archivecleanup utility is designed specifically to be used with `archive_cleanup_command` in typical single-standby configurations, see pg_archivecleanup. Note however, that if you're using the archive for backup purposes, you need to retain files needed to recover from at least the latest base backup, even if they're no longer needed by the standby.

A simple example of configuration is:

```
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo password=foopass options=''-c
wal_sender_timeout=5000'''
restore_command = 'cp /path/to/archive/%f %p'
archive_cleanup_command = 'pg_archivecleanup /path/to/archive %r'
```

You can have any number of standby servers, but if you use streaming replication, make sure you set `max_wal_senders` high enough in the primary to allow them to be connected simultaneously.

Streaming Replication

Streaming replication allows a standby server to stay more up-to-date than is possible with file-based log shipping. The standby connects to the primary, which streams WAL records to the standby as they're generated, without waiting for the WAL file to be filled.

Streaming replication is asynchronous by default , in which case there is a small delay between committing a transaction in the primary and the changes becoming visible in the standby. This delay is however much smaller than with file-based log shipping, typically under one second assuming the standby is powerful enough to keep up with the load. With streaming replication, `archive_timeout` is not required to reduce the data loss window.

If you use streaming replication without file-based continuous archiving, the server might recycle old WAL segments before the standby has received them. If this occurs, the standby will need to be reinitialized from a new base backup. You can avoid this by setting `wal_keep_size` to a value large enough to ensure that WAL segments are not recycled too early, or by configuring a replication slot for the standby. If you set up a WAL archive that's accessible from the standby, these solutions are not required, since the standby can always use the archive to catch up provided it retains enough segments.

To use streaming replication, set up a file-based log-shipping standby server. The step that turns a file-based log-shipping standby into streaming replication standby is setting the `primary_conninfo` setting to point to the primary server. Set listen_addresses and authentication options (see `pg_hba.conf`) on the primary so that the standby server can connect to the `replication` pseudo-database on the primary server.

On systems that support the keepalive socket option, setting tcp_keepalives_idle, tcp_keepalives_interval and tcp_keepalives_count helps the primary promptly notice a broken connection.

Set the maximum number of concurrent connections from the standby servers (see max_wal_senders for details).

When the standby is started and `primary_conninfo` is set correctly, the standby will connect to the primary after replaying all WAL files available in the archive. If the connection is established successfully, you will see a `walreceiver` in the standby, and a corresponding `walsender` process in the primary.

Authentication

It is very important that the access privileges for replication be set up so that only trusted users can read the WAL stream, because it is easy to extract privileged information from it. Standby servers must authenticate to the primary as an account that has the `REPLICATION` privilege or a superuser. It is recommended to create a dedicated user account with `REPLICATION` and `LOGIN` privileges for replication. While `REPLICATION` privilege gives very high permissions, it does not allow the user to modify any data on the primary system, which the `SUPERUSER` privilege does.

Client authentication for replication is controlled by a `pg_hba.conf` record specifying `replication` in the `database` field. For example, if the standby is running on host IP `192.168.1.100` and the account name for replication is `foo`, the administrator can add the following line to the `pg_hba.conf` file on the primary:

```
# Allow the user "foo" from host 192.168.1.100 to connect to the primary
# as a replication standby if the user's password is correctly supplied.
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    replication     foo             192.168.1.100/32        md5
```

The host name and port number of the primary, connection user name, and password are specified in the primary_conninfo. The password can also be set in the `~/.pgpass` file on the standby (specify `replication` in the `database` field). For example, if the primary is running on host IP `192.168.1.50`, port `5432`, the account name for replication is `foo`, and the password is `foopass`, the administrator can add the following line to the `postgresql.conf` file on the standby:

```
# The standby connects to the primary that is running on host 192.168.1.50
# and port 5432 as the user "foo" whose password is "foopass".
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo password=foopass'
```

Monitoring

An important health indicator of streaming replication is the amount of WAL records generated in the primary, but not yet applied in the standby. You can calculate this lag by comparing the current WAL write location on the primary with the last WAL location received by the standby. These locations can be retrieved using `pg_current_wal_lsn` on the primary and `pg_last_wal_receive_lsn` on the standby, respectively . The last WAL receive location in the standby is also displayed in the process status of the WAL receiver process, displayed using the `ps` command .

You can retrieve a list of WAL sender processes via the `pg_stat_replication` view. Large differences between `pg_current_wal_lsn` and the view’ s `sent_lsn` field might indicate that the primary server is under heavy load, while differences between `sent_lsn` and `pg_last_wal_receive_lsn` on the standby might indicate network delay, or that the standby is under heavy load.

On a hot standby, the status of the WAL receiver process can be retrieved via the `pg_stat_wal_receiver` view. A large difference between `pg_last_wal_replay_lsn` and the view’ s `flushed_lsn` indicates that WAL is being received faster than it can be replayed.

Replication Slots

Replication slots provide an automated way to ensure that the primary does not remove WAL segments until they have been received by all standbys, and that the primary does not remove rows which could cause a recovery conflict even when the standby is disconnected.

In lieu of using replication slots, it is possible to prevent the removal of old WAL segments using wal_keep_size, or by storing the segments in an archive using archive_command or archive_library. However, these methods often result in retaining more WAL segments than required, whereas replication slots retain only the number of segments known to be needed. On the other hand, replication slots can retain so many WAL segments that they fill up the space allocated for `pg_wal`; max_slot_wal_keep_size limits the size of WAL files retained by replication slots.

Similarly, hot_standby_feedback and vacuum_defer_cleanup_age provide protection against relevant rows being removed by vacuum, but the former provides no protection during any time period when the standby is not connected, and the latter often needs to be set to a high value to provide adequate protection. Replication slots overcome these disadvantages.

Querying And Manipulating Replication Slots

Each replication slot has a name, which can contain lower-case letters, numbers, and the underscore character.

Existing replication slots and their state can be seen in the `pg_replication_slots` view.

Slots can be created and dropped either via the streaming replication protocol or via SQL functions .

Configuration Example

You can create a replication slot like this:

```
postgres=# SELECT * FROM pg_create_physical_replication_slot('node_a_slot');
  slot_name  | lsn
-------------+-----
 node_a_slot |

postgres=# SELECT slot_name, slot_type, active FROM pg_replication_slots;
  slot_name  | slot_type | active
-------------+-----------+--------
 node_a_slot | physical  | f
(1 row)
```

To configure the standby to use this slot, `primary_slot_name` should be configured on the standby. Here is a simple example:

```
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo password=foopass'
primary_slot_name = 'node_a_slot'
```

Cascading Replication

The cascading replication feature allows a standby server to accept replication connections and stream WAL records to other standbys, acting as a relay. This can be used to reduce the number of direct connections to the primary and also to minimize inter-site bandwidth overheads.

A standby acting as both a receiver and a sender is known as a cascading standby. Standbys that are more directly connected to the primary are known as upstream servers, while those standby servers further away are downstream servers. Cascading replication does not place limits on the number or arrangement of downstream servers, though each standby connects to only one upstream server which eventually links to a single primary server.

A cascading standby sends not only WAL records received from the primary but also those restored from the archive. So even if the replication connection in some upstream connection is terminated, streaming replication continues downstream for as long as new WAL records are available.

Cascading replication is currently asynchronous. Synchronous replication settings have no effect on cascading replication at present.

Hot standby feedback propagates upstream, whatever the cascaded arrangement.

If an upstream standby server is promoted to become the new primary, downstream servers will continue to stream from the new primary if `recovery_target_timeline` is set to `'latest'` (the default).

To use cascading replication, set up the cascading standby so that it can accept replication connections (that is, set max_wal_senders and hot_standby, and configure host-based authentication). You will also need to set `primary_conninfo` in the downstream standby to point to the cascading standby.

Synchronous Replication

IvorySQL streaming replication is asynchronous by default. If the primary server crashes then some transactions that were committed may not have been replicated to the standby server, causing data loss. The amount of data loss is proportional to the replication delay at the time of failover.

Synchronous replication offers the ability to confirm that all changes made by a transaction have been transferred to one or more synchronous standby servers. This extends that standard level of durability offered by a transaction commit. This level of protection is referred to as 2-safe replication in computer science theory, and group-1-safe (group-safe and 1-safe) when `synchronous_commit` is set to `remote_write`.

When requesting synchronous replication, each commit of a write transaction will wait until confirmation is received that the commit has been written to the write-ahead log on disk of both the primary and standby server. The only possibility that data can be lost is if both the primary and the standby suffer crashes at the same time. This can provide a much higher level of durability, though only if the sysadmin is cautious about the placement and management of the two servers. Waiting for confirmation increases the user’ s confidence that the changes will not be lost in the event of server crashes but it also necessarily increases the response time for the requesting transaction. The minimum wait time is the round-trip time between primary and standby.

Read-only transactions and transaction rollbacks need not wait for replies from standby servers. Subtransaction commits do not wait for responses from standby servers, only top-level commits. Long running actions such as data loading or index building do not wait until the very final commit message. All two-phase commit actions require commit waits, including both prepare and commit.

A synchronous standby can be a physical replication standby or a logical replication subscriber. It can also be any other physical or logical WAL replication stream consumer that knows how to send the appropriate feedback messages. Besides the built-in physical and logical replication systems, this includes special programs such as `pg_receivewal` and `pg_recvlogical` as well as some third-party replication systems and custom programs. Check the respective documentation for details on synchronous replication support.

Basic Configuration

Once streaming replication has been configured, configuring synchronous replication requires only one additional configuration step: synchronous_standby_names must be set to a non-empty value. `synchronous_commit` must also be set to `on`, but since this is the default value, typically no change is required.This configuration will cause each commit to wait for confirmation that the standby has written the commit record to durable storage. `synchronous_commit` can be set by individual users, so it can be configured in the configuration file, for particular users or databases, or dynamically by applications, in order to control the durability guarantee on a per-transaction basis.

After a commit record has been written to disk on the primary, the WAL record is then sent to the standby. The standby sends reply messages each time a new batch of WAL data is written to disk, unless `wal_receiver_status_interval` is set to zero on the standby. In the case that `synchronous_commit` is set to `remote_apply`, the standby sends reply messages when the commit record is replayed, making the transaction visible. If the standby is chosen as a synchronous standby, according to the setting of `synchronous_standby_names` on the primary, the reply messages from that standby will be considered along

with those from other synchronous standbys to decide when to release transactions waiting for confirmation that the commit record has been received. These parameters allow the administrator to specify which standby servers should be synchronous standbys. Note that the configuration of synchronous replication is mainly on the primary. Named standbys must be directly connected to the primary; the primary knows nothing about downstream standby servers using cascaded replication.

Setting `synchronous_commit` to `remote_write` will cause each commit to wait for confirmation that the standby has received the commit record and written it out to its own operating system, but not for the data to be flushed to disk on the standby. This setting provides a weaker guarantee of durability than `on` does: the standby could lose the data in the event of an operating system crash, though not a PostgreSQL crash. However, it' s a useful setting in practice because it can decrease the response time for the transaction. Data loss could only occur if both the primary and the standby crash and the database of the primary gets corrupted at the same time.

Setting `synchronous_commit` to `remote_apply` will cause each commit to wait until the current synchronous standbys report that they have replayed the transaction, making it visible to user queries. In simple cases, this allows for load balancing with causal consistency.

Users will stop waiting if a fast shutdown is requested. However, as when using asynchronous replication, the server will not fully shutdown until all outstanding WAL records are transferred to the currently connected standby servers.

Multiple Synchronous Standbys

Synchronous replication supports one or more synchronous standby servers; transactions will wait until all the standby servers which are considered as synchronous confirm receipt of their data. The number of synchronous standbys that transactions must wait for replies from is specified in `synchronous_standby_names`. This parameter also specifies a list of standby names and the method (`FIRST` and `ANY`) to choose synchronous standbys from the listed ones.

The method `FIRST` specifies a priority-based synchronous replication and makes transaction commits wait until their WAL records are replicated to the requested number of synchronous standbys chosen based on their priorities. The standbys whose names appear earlier in the list are given higher priority and will be considered as synchronous. Other standby servers appearing later in this list represent potential synchronous standbys. If any of the current synchronous standbys disconnects for whatever reason, it will be replaced immediately with the next-highest-priority standby.

An example of `synchronous_standby_names` for a priority-based multiple synchronous standbys is:

```
synchronous_standby_names = 'FIRST 2 (s1, s2, s3)'
```

In this example, if four standby servers `s1`, `s2`, `s3` and `s4` are running, the two standbys `s1` and `s2` will be chosen as synchronous standbys because their names appear early in the list of standby names. `s3` is a potential synchronous standby and will take over the role of synchronous standby when either of `s1` or `s2` fails. `s4` is an asynchronous standby since its name is not in the list.

The method `ANY` specifies a quorum-based synchronous replication and makes transaction commits wait until their WAL records are replicated to at least the requested number of synchronous standbys in the list.

An example of `synchronous_standby_names` for a quorum-based multiple synchronous standbys is:

```
synchronous_standby_names = 'ANY 2 (s1, s2, s3)'
```

In this example, if four standby servers `s1`, `s2`, `s3` and `s4` are running, transaction commits will wait for replies from at least any two standbys of `s1`, `s2` and `s3`. `s4` is an asynchronous standby since its name is not in the list.

The synchronous states of standby servers can be viewed using the `pg_stat_replication` view.

Synchronous replication usually requires carefully planned and placed standby servers to ensure applications perform acceptably. Waiting doesn't utilize system resources, but transaction locks continue to be held until the transfer is confirmed. As a result, incautious use of synchronous replication will reduce performance for database applications because of increased response times and higher contention.

PostgreSQL allows the application developer to specify the durability level required via replication. This can be specified for the system overall, though it can also be specified for specific users or connections, or even individual transactions.

For example, an application workload might consist of: 10% of changes are important customer details, while 90% of changes are less important data that the business can more easily survive if it is lost, such as chat messages between users.

With synchronous replication options specified at the application level (on the primary) we can offer synchronous replication for the most important changes, without slowing down the bulk of the total workload. Application level options are an important and practical tool for allowing the benefits of synchronous replication for high performance applications.

You should consider that the network bandwidth must be higher than the rate of generation of WAL data.

Planning For High Availability

`synchronous_standby_names` specifies the number and names of synchronous standbys that transaction commits made when `synchronous_commit` is set to `on`, `remote_apply` or `remote_write` will wait for responses from. Such transaction commits may never be completed if any one of synchronous standbys should crash.

The best solution for high availability is to ensure you keep as many synchronous standbys as requested. This can be achieved by naming multiple potential synchronous standbys using `synchronous_standby_names`.

In a priority-based synchronous replication, the standbys whose names appear earlier in the list will be used as synchronous standbys. Standbys listed after these will take over the role of synchronous standby if one of current ones should fail.

In a quorum-based synchronous replication, all the standbys appearing in the list will be used as candidates for synchronous standbys. Even if one of them should fail, the other standbys will keep performing the role of candidates of synchronous standby.

When a standby first attaches to the primary, it will not yet be properly synchronized. This is described as `catchup` mode. Once the lag between standby and primary reaches zero for the first time we move to real-time `streaming` state. The catch-up duration may be long immediately after the standby has been created. If the standby is shut down, then the catch-up period will increase according to the length of time the standby has been down. The standby is only able to become a synchronous standby once it has reached `streaming` state. This state can be viewed using the `pg_stat_replication` view.

If primary restarts while commits are waiting for acknowledgment, those waiting transactions will be marked fully committed once the primary database recovers. There is no way to be certain that all standbys have received all outstanding WAL data at time of the crash of the primary. Some transactions may not show as committed on the standby, even though they show as committed on the primary. The guarantee we offer is that the application will not receive explicit acknowledgment of the successful commit of a transaction until the WAL data is known to be safely received by all the synchronous standbys.

If you really cannot keep as many synchronous standbys as requested then you should decrease the number of synchronous standbys that transaction commits must wait for responses from in `synchronous_standby_names` (or disable it) and reload the configuration file on the primary server.

If the primary is isolated from remaining standby servers you should fail over to the best candidate of those other remaining standby servers.

If you need to re-create a standby server while transactions are waiting, make sure that the commands pg_backup_start() and pg_backup_stop() are run in a session with `synchronous_commit` = `off`, otherwise

those requests will wait forever for the standby to appear.

Continuous Archiving in Standby

When continuous WAL archiving is used in a standby, there are two different scenarios: the WAL archive can be shared between the primary and the standby, or the standby can have its own WAL archive. When the standby has its own WAL archive, set `archive_mode` to `always`, and the standby will call the archive command for every WAL segment it receives, whether it's by restoring from the archive or by streaming replication. The shared archive can be handled similarly, but the `archive_command` or `archive_library` must test if the file being archived exists already, and if the existing file has identical contents. This requires more care in the `archive_command` or `archive_library`, as it must be careful to not overwrite an existing file with different contents, but return success if the exactly same file is archived twice. And all that must be done free of race conditions, if two servers attempt to archive the same file at the same time.

If `archive_mode` is set to `on`, the archiver is not enabled during recovery or standby mode. If the standby server is promoted, it will start archiving after the promotion, but will not archive any WAL or timeline history files that it did not generate itself. To get a complete series of WAL files in the archive, you must ensure that all WAL is archived, before it reaches the standby. This is inherently true with file-based log shipping, as the standby can only restore files that are found in the archive, but not if streaming replication is enabled. When a server is not in recovery mode, there is no difference between `on` and `always` modes.

Failover

If the primary server fails then the standby server should begin failover procedures.

If the standby server fails then no failover need take place. If the standby server can be restarted, even some time later, then the recovery process can also be restarted immediately, taking advantage of restartable recovery. If the standby server cannot be restarted, then a full new standby server instance should be created.

If the primary server fails and the standby server becomes the new primary, and then the old primary restarts, you must have a mechanism for informing the old primary that it is no longer the primary. This is sometimes known as STONITH (Shoot The Other Node In The Head), which is necessary to avoid situations where both systems think they are the primary, which will lead to confusion and ultimately data loss.

Many failover systems use just two systems, the primary and the standby, connected by some kind of heartbeat mechanism to continually verify the connectivity between the two and the viability of the primary. It is also possible to use a third system (called a witness server) to prevent some cases of inappropriate failover, but the additional complexity might not be worthwhile unless it is set up with sufficient care and rigorous testing.

PostgreSQL does not provide the system software required to identify a failure on the primary and notify the standby database server. Many such tools exist and are well integrated with the operating system facilities required for successful failover, such as IP address migration.

Once failover to the standby occurs, there is only a single server in operation. This is known as a degenerate state. The former standby is now the primary, but the former primary is down and might stay down. To return to normal operation, a standby server must be recreated, either on the former primary system when it comes up, or on a third, possibly new, system. The pg_rewind utility can be used to speed up this process on large clusters. Once complete, the primary and standby can be considered to have switched roles. Some people choose to use a third server to provide backup for the new primary until the new standby server is recreated, though clearly this complicates the system configuration and operational processes.

So, switching from primary to standby server can be fast but requires some time to re-prepare the failover cluster. Regular switching from primary to standby is useful, since it allows regular downtime on each system for maintenance. This also serves as a test of the failover mechanism to ensure that it will really work when you need it. Written administration procedures are advised.

To trigger failover of a log-shipping standby server, run `pg_ctl promote`, call `pg_promote()`, or create a trigger file with the file name and path specified by the `promote_trigger_file`. If you're planning to use `pg_ctl promote` or to call `pg_promote()` to fail over, `promote_trigger_file` is not required. If you're setting up the reporting servers that are only used to offload read-only queries from the primary, not for high availability

purposes, you don't need to promote it.

Hot Standby

Hot standby is the term used to describe the ability to connect to the server and run read-only queries while the server is in archive recovery or standby mode. This is useful both for replication purposes and for restoring a backup to a desired state with great precision. The term hot standby also refers to the ability of the server to move from recovery through to normal operation while users continue running queries and/or keep their connections open.

Running queries in hot standby mode is similar to normal query operation, though there are several usage and administrative differences explained below.

User's Overview

When the hot_standby parameter is set to true on a standby server, it will begin accepting connections once the recovery has brought the system to a consistent state. All such connections are strictly read-only; not even temporary tables may be written.

The data on the standby takes some time to arrive from the primary server so there will be a measurable delay between primary and standby. Running the same query nearly simultaneously on both primary and standby might therefore return differing results. We say that data on the standby is eventually consistent with the primary. Once the commit record for a transaction is replayed on the standby, the changes made by that transaction will be visible to any new snapshots taken on the standby. Snapshots may be taken at the start of each query or at the start of each transaction, depending on the current transaction isolation level.

Transactions started during hot standby may issue the following commands:

- Query access: `SELECT`, `COPY TO`
- Cursor commands: `DECLARE`, `FETCH`, `CLOSE`
- Settings: `SHOW`, `SET`, `RESET`
- Transaction management commands:
  - `BEGIN`, `END`, `ABORT`, `START TRANSACTION`
  - `SAVEPOINT`, `RELEASE`, `ROLLBACK TO SAVEPOINT`
  - `EXCEPTION` blocks and other internal subtransactions
- `LOCK TABLE`, though only when explicitly in one of these modes: `ACCESS SHARE`, `ROW SHARE` or `ROW EXCLUSIVE`.
- Plans and resources: `PREPARE`, `EXECUTE`, `DEALLOCATE`, `DISCARD`
- Plugins and extensions: `LOAD`
- `UNLISTEN`

Transactions started during hot standby will never be assigned a transaction ID and cannot write to the system write-ahead log. Therefore, the following actions will produce error messages:

- Data Manipulation Language (DML): `INSERT`, `UPDATE`, `DELETE`, `COPY FROM`, `TRUNCATE`. Note that there are no allowed actions that result in a trigger being executed during recovery. This restriction applies even to temporary tables, because table rows cannot be read or written without assigning a transaction ID, which is currently not possible in a hot standby environment.
- Data Definition Language (DDL): `CREATE`, `DROP`, `ALTER`, `COMMENT`. This restriction applies even to temporary tables, because carrying out these operations would require updating the system catalog tables.
- `SELECT ... FOR SHARE | UPDATE`, because row locks cannot be taken without updating the underlying data files.
- Rules on `SELECT` statements that generate DML commands.
- `LOCK` that explicitly requests a mode higher than `ROW EXCLUSIVE MODE`.
- `LOCK` in short default form, since it requests `ACCESS EXCLUSIVE MODE`.

- Transaction management commands that explicitly set non-read-only state:
  - `BEGIN READ WRITE`, `START TRANSACTION READ WRITE`
  - `SET TRANSACTION READ WRITE`, `SET SESSION CHARACTERISTICS AS TRANSACTION READ WRITE`
  - `SET transaction_read_only = off`
- Two-phase commit commands: `PREPARE TRANSACTION`, `COMMIT PREPARED`, `ROLLBACK PREPARED` because even read-only transactions need to write WAL in the prepare phase (the first phase of two phase commit).
- Sequence updates: `nextval()`, `setval()`
- `LISTEN`, `NOTIFY`

In normal operation, "read-only" transactions are allowed to use `LISTEN` and `NOTIFY`, so hot standby sessions operate under slightly tighter restrictions than ordinary read-only sessions. It is possible that some of these restrictions might be loosened in a future release.

During hot standby, the parameter `transaction_read_only` is always true and may not be changed. But as long as no attempt is made to modify the database, connections during hot standby will act much like any other database connection. If failover or switchover occurs, the database will switch to normal processing mode. Sessions will remain connected while the server changes mode. Once hot standby finishes, it will be possible to initiate read-write transactions (even from a session begun during hot standby).

Users can determine whether hot standby is currently active for their session by issuing `SHOW in_hot_standby`. (In server versions before 14, the `in_hot_standby` parameter did not exist; a workable substitute method for older servers is `SHOW transaction_read_only`.) In addition, a set of functions allow users to access information about the standby server. These allow you to write programs that are aware of the current state of the database. These can be used to monitor the progress of recovery, or to allow you to write complex programs that restore the database to particular states.

Handling Query Conflicts

The primary and standby servers are in many ways loosely connected. Actions on the primary will have an effect on the standby. As a result, there is potential for negative interactions or conflicts between them. The easiest conflict to understand is performance: if a huge data load is taking place on the primary then this will generate a similar stream of WAL records on the standby, so standby queries may contend for system resources, such as I/O.

There are also additional types of conflict that can occur with hot standby. These conflicts are hard conflicts in the sense that queries might need to be canceled and, in some cases, sessions disconnected to resolve them. The user is provided with several ways to handle these conflicts. Conflict cases include:

- Access Exclusive locks taken on the primary server, including both explicit `LOCK` commands and various DDL actions, conflict with table accesses in standby queries.
- Dropping a tablespace on the primary conflicts with standby queries using that tablespace for temporary work files.
- Dropping a database on the primary conflicts with sessions connected to that database on the standby.
- Application of a vacuum cleanup record from WAL conflicts with standby transactions whose snapshots can still "see" any of the rows to be removed.
- Application of a vacuum cleanup record from WAL conflicts with queries accessing the target page on the standby, whether or not the data to be removed is visible.

On the primary server, these cases simply result in waiting; and the user might choose to cancel either of the conflicting actions. However, on the standby there is no choice: the WAL-logged action already occurred on the primary so the standby must not fail to apply it. Furthermore, allowing WAL application to wait indefinitely may be very undesirable, because the standby's state will become increasingly far behind the primary's. Therefore, a mechanism is provided to forcibly cancel standby queries that conflict with to-be-applied WAL records.

An example of the problem situation is an administrator on the primary server running `DROP TABLE` on a table that is currently being queried on the standby server. Clearly the standby query cannot continue if the `DROP`

TABLE is applied on the standby. If this situation occurred on the primary, the DROP TABLE would wait until the other query had finished. But when DROP TABLE is run on the primary, the primary doesn't have information about what queries are running on the standby, so it will not wait for any such standby queries. The WAL change records come through to the standby while the standby query is still running, causing a conflict. The standby server must either delay application of the WAL records (and everything after them, too) or else cancel the conflicting query so that the DROP TABLE can be applied.

When a conflicting query is short, it's typically desirable to allow it to complete by delaying WAL application for a little bit; but a long delay in WAL application is usually not desirable. So the cancel mechanism has parameters, max_standby_archive_delay and max_standby_streaming_delay, that define the maximum allowed delay in WAL application. Conflicting queries will be canceled once it has taken longer than the relevant delay setting to apply any newly-received WAL data. There are two parameters so that different delay values can be specified for the case of reading WAL data from an archive (i.e., initial recovery from a base backup or "catching up" a standby server that has fallen far behind) versus reading WAL data via streaming replication.

In a standby server that exists primarily for high availability, it's best to set the delay parameters relatively short, so that the server cannot fall far behind the primary due to delays caused by standby queries. However, if the standby server is meant for executing long-running queries, then a high or even infinite delay value may be preferable. Keep in mind however that a long-running query could cause other sessions on the standby server to not see recent changes on the primary, if it delays application of WAL records.

Once the delay specified by max_standby_archive_delay or max_standby_streaming_delay has been exceeded, conflicting queries will be canceled. This usually results just in a cancellation error, although in the case of replaying a DROP DATABASE the entire conflicting session will be terminated. Also, if the conflict is over a lock held by an idle transaction, the conflicting session is terminated (this behavior might change in the future).

Canceled queries may be retried immediately (after beginning a new transaction, of course). Since query cancellation depends on the nature of the WAL records being replayed, a query that was canceled may well succeed if it is executed again.

Keep in mind that the delay parameters are compared to the elapsed time since the WAL data was received by the standby server. Thus, the grace period allowed to any one query on the standby is never more than the delay parameter, and could be considerably less if the standby has already fallen behind as a result of waiting for previous queries to complete, or as a result of being unable to keep up with a heavy update load.

The most common reason for conflict between standby queries and WAL replay is "early cleanup". Normally, PostgreSQL allows cleanup of old row versions when there are no transactions that need to see them to ensure correct visibility of data according to MVCC rules. However, this rule can only be applied for transactions executing on the primary. So it is possible that cleanup on the primary will remove row versions that are still visible to a transaction on the standby.

Experienced users should note that both row version cleanup and row version freezing will potentially conflict with standby queries. Running a manual VACUUM FREEZE is likely to cause conflicts even on tables with no updated or deleted rows.

Users should be clear that tables that are regularly and heavily updated on the primary server will quickly cause cancellation of longer running queries on the standby. In such cases the setting of a finite value for max_standby_archive_delay or max_standby_streaming_delay can be considered similar to setting statement_timeout.

Remedial possibilities exist if the number of standby-query cancellations is found to be unacceptable. The first option is to set the parameter hot_standby_feedback, which prevents VACUUM from removing recently-dead rows and so cleanup conflicts do not occur. If you do this, you should note that this will delay cleanup of dead rows on the primary, which may result in undesirable table bloat. However, the cleanup situation will be no worse than if the standby queries were running directly on the primary server, and you are still getting the benefit of off-loading execution onto the standby. If standby servers connect and disconnect frequently, you might want to make adjustments to handle the period when hot_standby_feedback feedback is not being provided. For example, consider increasing max_standby_archive_delay so that queries are not rapidly canceled by conflicts in WAL archive files during disconnected periods. You should also consider increasing max_standby_streaming_delay to avoid rapid cancellations by newly-arrived streaming WAL entries after reconnection.

Another option is to increase vacuum_defer_cleanup_age on the primary server, so that dead rows will not be cleaned up as quickly as they normally would be. This will allow more time for queries to execute before they are canceled on the standby, without having to set a high `max_standby_streaming_delay`. However it is difficult to guarantee any specific execution-time window with this approach, since `vacuum_defer_cleanup_age` is measured in transactions executed on the primary server.

The number of query cancels and the reason for them can be viewed using the `pg_stat_database_conflicts` system view on the standby server. The `pg_stat_database` system view also contains summary information.

Users can control whether a log message is produced when WAL replay is waiting longer than `deadlock_timeout` for conflicts. This is controlled by the log_recovery_conflict_waits parameter.

Administrator's Overview

If `hot_standby` is `on` in `postgresql.conf` (the default value) and there is a `standby.signal` file present, the server will run in hot standby mode. However, it may take some time for hot standby connections to be allowed, because the server will not accept connections until it has completed sufficient recovery to provide a consistent state against which queries can run. During this period, clients that attempt to connect will be refused with an error message. To confirm the server has come up, either loop trying to connect from the application, or look for these messages in the server logs:

```
LOG:  entering standby mode

... then some time later ...

LOG:  consistent recovery state reached
LOG:  database system is ready to accept read-only connections
```

Consistency information is recorded once per checkpoint on the primary. It is not possible to enable hot standby when reading WAL written during a period when `wal_level` was not set to `replica` or `logical` on the primary. Reaching a consistent state can also be delayed in the presence of both of these conditions:

- A write transaction has more than 64 subtransactions
- Very long-lived write transactions

If you are running file-based log shipping ("warm standby"), you might need to wait until the next WAL file arrives, which could be as long as the `archive_timeout` setting on the primary.

The settings of some parameters determine the size of shared memory for tracking transaction IDs, locks, and prepared transactions. These shared memory structures must be no smaller on a standby than on the primary in order to ensure that the standby does not run out of shared memory during recovery. For example, if the primary had used a prepared transaction but the standby had not allocated any shared memory for tracking prepared transactions, then recovery could not continue until the standby's configuration is changed. The parameters affected are:

- `max_connections`
- `max_prepared_transactions`
- `max_locks_per_transaction`
- `max_wal_senders`
- `max_worker_processes`

The easiest way to ensure this does not become a problem is to have these parameters set on the standbys to values equal to or greater than on the primary. Therefore, if you want to increase these values, you should do so on all standby servers first, before applying the changes to the primary server. Conversely, if you want to decrease these values, you should do so on the primary server first, before applying the changes to all standby servers. Keep in mind that when a standby is promoted, it becomes the new reference for the

required parameter settings for the standbys that follow it. Therefore, to avoid this becoming a problem during a switchover or failover, it is recommended to keep these settings the same on all standby servers.

The WAL tracks changes to these parameters on the primary. If a hot standby processes WAL that indicates that the current value on the primary is higher than its own value, it will log a warning and pause recovery, for example:

```
WARNING:  hot standby is not possible because of insufficient parameter settings
DETAIL:  max_connections = 80 is a lower setting than on the primary server, where its
value was 100.
LOG:  recovery has paused
DETAIL:  If recovery is unpaused, the server will shut down.
HINT:  You can then restart the server after making the necessary configuration
changes.
```

At that point, the settings on the standby need to be updated and the instance restarted before recovery can continue. If the standby is not a hot standby, then when it encounters the incompatible parameter change, it will shut down immediately without pausing, since there is then no value in keeping it up.

It is important that the administrator select appropriate settings for max_standby_archive_delay and max_standby_streaming_delay. The best choices vary depending on business priorities. For example if the server is primarily tasked as a High Availability server, then you will want low delay settings, perhaps even zero, though that is a very aggressive setting. If the standby server is tasked as an additional server for decision support queries then it might be acceptable to set the maximum delay values to many hours, or even -1 which means wait forever for queries to complete.

Transaction status "hint bits" written on the primary are not WAL-logged, so data on the standby will likely re-write the hints again on the standby. Thus, the standby server will still perform disk writes even though all users are read-only; no changes occur to the data values themselves. Users will still write large sort temporary files and re-generate relcache info files, so no part of the database is truly read-only during hot standby mode. Note also that writes to remote databases using dblink module, and other operations outside the database using PL functions will still be possible, even though the transaction is read-only locally.

The following types of administration commands are not accepted during recovery mode:

- Data Definition Language (DDL): e.g., `CREATE INDEX`
- Privilege and Ownership: `GRANT`, `REVOKE`, `REASSIGN`
- Maintenance commands: `ANALYZE`, `VACUUM`, `CLUSTER`, `REINDEX`

Again, note that some of these commands are actually allowed during "read only" mode transactions on the primary.

As a result, you cannot create additional indexes that exist solely on the standby, nor statistics that exist solely on the standby. If these administration commands are needed, they should be executed on the primary, and eventually those changes will propagate to the standby.

`pg_cancel_backend()` and `pg_terminate_backend()` will work on user backends, but not the startup process, which performs recovery. `pg_stat_activity` does not show recovering transactions as active. As a result, `pg_prepared_xacts` is always empty during recovery. If you wish to resolve in-doubt prepared transactions, view `pg_prepared_xacts` on the primary and issue commands to resolve transactions there or resolve them after the end of recovery.

`pg_locks` will show locks held by backends, as normal. `pg_locks` also shows a virtual transaction managed by the startup process that owns all `AccessExclusiveLocks` held by transactions being replayed by recovery. Note that the startup process does not acquire locks to make database changes, and thus locks other than `AccessExclusiveLocks` do not show in `pg_locks` for the Startup process; they are just presumed to exist.

The Nagios plugin check_pgsql will work, because the simple information it checks for exists. The check_postgres monitoring script will also work, though some reported values could give different or confusing results. For example, last vacuum time will not be maintained, since no vacuum occurs on the standby. Vacuums running on the primary do still send their changes to the standby.

WAL file control commands will not work during recovery, e.g., `pg_backup_start`, `pg_switch_wal` etc.

Dynamically loadable modules work, including `pg_stat_statements`.

Advisory locks work normally in recovery, including deadlock detection. Note that advisory locks are never WAL logged, so it is impossible for an advisory lock on either the primary or the standby to conflict with WAL replay. Nor is it possible to acquire an advisory lock on the primary and have it initiate a similar advisory lock on the standby. Advisory locks relate only to the server on which they are acquired.

Trigger-based replication systems such as Slony, Londiste and Bucardo won't run on the standby at all, though they will run happily on the primary server as long as the changes are not sent to standby servers to be applied. WAL replay is not trigger-based so you cannot relay from the standby to any system that requires additional database writes or relies on the use of triggers.

New OIDs cannot be assigned, though some UUID generators may still work as long as they do not rely on writing new status to the database.

Currently, temporary table creation is not allowed during read-only transactions, so in some cases existing scripts will not run correctly. This restriction might be relaxed in a later release. This is both an SQL standard compliance issue and a technical issue.

`DROP TABLESPACE` can only succeed if the tablespace is empty. Some standby users may be actively using the tablespace via their `temp_tablespaces` parameter. If there are temporary files in the tablespace, all active queries are canceled to ensure that temporary files are removed, so the tablespace can be removed and WAL replay can continue.

Running `DROP DATABASE` or `ALTER DATABASE … SET TABLESPACE` on the primary will generate a WAL entry that will cause all users connected to that database on the standby to be forcibly disconnected. This action occurs immediately, whatever the setting of `max_standby_streaming_delay`. Note that `ALTER DATABASE … RENAME` does not disconnect users, which in most cases will go unnoticed, though might in some cases cause a program confusion if it depends in some way upon database name.

In normal (non-recovery) mode, if you issue `DROP USER` or `DROP ROLE` for a role with login capability while that user is still connected then nothing happens to the connected user — they remain connected. The user cannot reconnect however. This behavior applies in recovery also, so a `DROP USER` on the primary does not disconnect that user on the standby.

The cumulative statistics system is active during recovery. All scans, reads, blocks, index usage, etc., will be recorded normally on the standby. However, WAL replay will not increment relation and database specific counters. I.e. replay will not increment pg_stat_all_tables columns (like n_tup_ins), nor will reads or writes performed by the startup process be tracked in the pg_statio views, nor will associated pg_stat_database columns be incremented.

Autovacuum is not active during recovery. It will start normally at the end of recovery.

The checkpointer process and the background writer process are active during recovery. The checkpointer process will perform restartpoints (similar to checkpoints on the primary) and the background writer process will perform normal block cleaning activities. This can include updates of the hint bit information stored on the standby server. The `CHECKPOINT` command is accepted during recovery, though it performs a restartpoint rather than a new checkpoint.

Hot Standby Parameter Reference

On the primary, parameters wal_level and vacuum_defer_cleanup_age can be used. max_standby_archive_delay and max_standby_streaming_delay have no effect if set on the primary.

On the standby, parameters hot_standby, max_standby_archive_delay and max_standby_streaming_delay

can be used. vacuum_defer_cleanup_age has no effect as long as the server remains in standby mode, though it will become relevant if the standby becomes primary.

Caveats

There are several limitations of hot standby. These can and probably will be fixed in future releases:

- Full knowledge of running transactions is required before snapshots can be taken. Transactions that use large numbers of subtransactions (currently greater than 64) will delay the start of read-only connections until the completion of the longest running write transaction. If this situation occurs, explanatory messages will be sent to the server log.
- Valid starting points for standby queries are generated at each checkpoint on the primary. If the standby is shut down while the primary is in a shutdown state, it might not be possible to re-enter hot standby until the primary is started up, so that it generates further starting points in the WAL logs. This situation isn’ t a problem in the most common situations where it might happen. Generally, if the primary is shut down and not available anymore, that’ s likely due to a serious failure that requires the standby being converted to operate as the new primary anyway. And in situations where the primary is being intentionally taken down, coordinating to make sure the standby becomes the new primary smoothly is also standard procedure.
- At the end of recovery, `AccessExclusiveLocks` held by prepared transactions will require twice the normal number of lock table entries. If you plan on running either a large number of concurrent prepared transactions that normally take `AccessExclusiveLocks`, or you plan on having one large transaction that takes many `AccessExclusiveLocks`, you are advised to select a larger value of `max_locks_per_transaction`, perhaps as much as twice the value of the parameter on the primary server. You need not consider this at all if your setting of `max_prepared_transactions` is 0.
- The Serializable transaction isolation level is not yet available in hot standby. An attempt to set a transaction to the serializable isolation level in hot standby mode will generate an error.

.3. Operation Management

Since IvorySQL is based on PostgreSQL, it is recommended that when reading and understanding this section, O&M staff also refer to doc。

Upgrade IvorySQL version

Overview of upgrade scheme

The IvorySQL version number consists of a major version and a minor version. For example, 3 in IvorySQL 3.2 is the major version and 2 is the minor version.

Releasing a minor version is not going to change the internal storage format, so it is always compatible with the same major version. For example, IvorySQL 3.4 is compatible with Ivory SQL 3.0 and the subsequent Ivory SQL 3.x. Upgrading for these compatible versions is as simple as shutting down the database service, installing a replacement binary executable, and restarting the service.

Next, we focus on cross-version upgrades of IvorySQL, for example, from IvorySQL 2.3 to IvorySQL 3.2. Major version upgrades may modify the internal data storage format and therefore require additional operations to be performed. The common cross-version upgrade methods and applicable scenarios are as follows.

Upgrade method	Applicable scenarios	Shutdown time
pg_dumpall	Small to medium sized databases, e.g. less than 100GB to support cross-platform data migration	Depends on the size of the database
pg_upgrade	Large and medium-sized databases, e.g., >100GB local upgrade	A few minutes

Logical Replication	Large and medium-sized databases, e.g. >100GB cross-platform support	A few seconds

Attention： New major releases usually introduce some user-visible incompatibilities and may therefore require application programming changes. All user-visible changes are listed in description， Please pay special attention to the section labeled "Migration". Although you may upgrade from one major version to another without upgrading an intermediate version, you should read the major release notes for all intermediate versions.

Upgrade data via pg_dumpall

The traditional cross-version upgrade method uses pg_dump/pg_dumpall to logically backup the database and then restore it in the new version via pg_restore. It is recommended to use the new version of pg_dump/pg_dumpall tool when exporting the old version of the database. You can take advantage of its latest parallel export and restore features, while reducing database bloat problems.

Logical backup and restore is very simple but slow. Downtime depends on the size of the database, so it is suitable for small to medium sized database upgrades.

The following describes how this upgrade method works. If the current IvorySQL software installation directory is located in /usr/local/pgsql and the data directory is located in /usr/local/pgsql/data, we do the upgrade on the same server.

1.Stop the application before performing a logical backup and make sure that no data is updated, as updates after the backup has started will not be exported. If necessary, modify the /usr/local/pgsql/data/pg_hba.conf file to disable others from accessing the database. Then backup the database.

```
pg_dumpall > outputfile
```

> If you have installed a new version of IvorySQL, you can use the new version of the pg_dumpall command to back up the old version of the database.

2.Stop the backend services of older versions.

```
pg_ctl stop
```

Or stop the background service by other means.

3.If the installation directory does not contain a specific version identifier, the directory can be renamed and modified back if necessary. Directories can be renamed using a command similar to the following.

```
mv /usr/local/pgsql /usr/local/pgsql.old
```

4.Install the new version of IvorySQL software, if the installation directory is still /usr/local/pgsql.

5.Initializing a new database cluster requires the use of a database specific user (usually postgres; if upgrading the version, this user should already exist) to perform the operation.

```
/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
```

6.Change the old configuration file pg_hba.conf, postgresql.conf, etc. in the corresponding new

configuration file.

7.To start a new version of the backend service using a dedicated database user.

```
/usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data
```

8.Finally, using the new version of the psql command to restore the data.

```
/usr/local/pgsql/bin/psql -d postgres -f outputfile
```

To reduce downtime, you can install the new version of IvorySQL to another directory, while starting the service using a different port. Then perform both the export and import of the database.

```
pg_dumpall -p 5432 | psql -d postgres -p 5433
```

When the above operation is executed, the old and new versions of the backend service run simultaneously, with the new version using port 5433 and the old version using port 5432.

Upgrade with the pg_upgrade utility

The pg_upgrade tool is a built-in cross-version upgrade utility in PostgreSQL that enables in-place database upgrades without requiring export and import operations. Since IvorySQL is derived from PostgreSQL, it can also use the pg_upgrade tool for major version upgrades. Below is a brief introduction on how to use pg_upgrade to upgrade IvorySQL to the latest 5.4 version on the CentOS8 platform.

pg_upgrade provides a pre-upgrade compatibility check (using the -c or --check option), which can identify issues such as plugin or data type incompatibilities. If the --link option is specified, the new version service can directly use the existing database files without copying, typically completing the upgrade in just a few minutes.

Commonly used parameters include:

- -b bindir, --old-bindir=bindir: The directory of the old IvorySQL executable files;
- -B bindir, --new-bindir=bindir: The directory of the new IvorySQL executable files;
- -d configdir, --old-datadir=configdir: The data directory of the old version;
- -D configdir, --new-datadir=configdir: The data directory of the new version;
- -c, --check: Only check upgrade compatibility without modifying any data;
- -k, --link: Upgrade using hard link method;

Upgrade preparation:

First, stop the old version of the IvorySQL 4.6 database:

```
/usr/ivory-4/bin/pg_ctl -D ./data stop
```

Then install the new version of the IvorySQL 5.4 database:

```
dnf install -y ivorysql5-5.4
```

Initialize the new IvorySQL 5.4 data directory:

```
/usr/ivory-5/bin/initdb -D ./data
```

Check version compatibility:

```
/usr/ivory-5/bin/pg_upgrade --old-datadir=/home/ivorysql/test/4.6/data --new
-datadir=/home/ivorysql/test/5.4/data --old-bindir=/usr/ivory-4/bin/ --new
-bindir=/usr/ivory-5/bin/ --check
```

The appearance of "Clusters are compatible" at the end indicates that there are no compatibility issues between the two versions of data, and the upgrade can proceed.

Official upgrade:

```
/usr/ivory-5/bin/pg_upgrade --old-datadir=/home/ivorysql/test/4.6/data --new
-datadir=/home/ivorysql/test/5.4/data --old-bindir=/usr/ivory-4/bin/ --new
-bindir=/usr/ivory-5/bin/
```

Seeing "Upgrade Complete" indicates that the upgrade has been successfully completed.

Update statistics:

pg_upgrade creates new system tables and reuses old data for the upgrade. However, statistics are not migrated during the upgrade process. Therefore, before enabling the new version, you should first recollect statistics to avoid incorrect query plans caused by missing statistics.

Start the new version of the database.

```
/usr/ivory-5/bin/pg_ctl -D ./data -l logfile start
```

Manually run the vacuum command

```
vacuum --all --analyze-in-stage -h 127.0.0.1 -p 1521
```

Cleanup after upgrade

```
rm -rf /home/ivorysql/test/4.6/data
```

pg_upgrade Documentoutlines the steps required above.

Upgrade data by copying

We can also create a fallback server using logical replication of an updated version of IvorySQL, which supports replication between different major versions of IvorySQL. The fallback server can be on the same computer or on a different computer. Once it is synchronized with the primary server (running an older version of IvorySQL), you can switch hosts and use the backup server as the host, and then shut down the older database instance. Such a switchover allows an upgrade with only a few seconds of downtime.

This upgrade method can be used with built-in logical replication tools and external logical replication systems such as pglogical, Slony, Londiste, and Bucardo.

Use pg_upgrade to upgrade PostgreSQL to IvorySQL

`pg_upgrade` supports upgrading native PostgreSQL clusters to IvorySQL. When the source cluster is native PostgreSQL and the target is an IvorySQL cluster, the `-g`（`--using-ora-pg`）parameter must be used.

Upgrading from PostgreSQL to IvorySQL is considered a "cross-product" migration. It is recommended to complete comprehensive testing and validation before performing the operation in a production environment, and to develop a detailed rollback plan.

Parameter description

parameter	description
`-g` / `--using-ora-pg`	Indicates that the source cluster is PostgreSQL and the target is an IvorySQL cluster. When this option is enabled, `pg_upgrade` uniformly uses the PostgreSQL standard port when connecting to both clusters, ensuring proper handling of the upgrade.

Upgrade steps

Step 1: Initialize the new IvorySQL cluster

```
# Use IvorySQL's initdb to initialize the new cluster
/opt/ivorysql/bin/initdb -D /data/ivorysql/data
```

Step 2: Stop the source PostgreSQL cluster

```
/usr/lib/postgresql/16/bin/pg_ctl -D /data/pg/data stop
```

Step 3: Run the upgrade check (optional)

Use `-c` to perform compatibility checks only without modifying any data:

```
/opt/ivorysql/bin/pg_upgrade \
  -b /usr/lib/postgresql/16/bin \   # Source PG executable directory
  -B /opt/ivorysql/bin \            # Target IvorySQL executable directory
  -d /data/pg/data \                # Source PG data directory
  -D /data/ivorysql/data \          # Target IvorySQL data directory
  -g \                              # Source is PostgreSQL, target is IvorySQL
  -c                                # Check only, do not upgrade
```

Step 4: Perform the upgrade

```
/opt/ivorysql/bin/pg_upgrade \
  -b /usr/lib/postgresql/16/bin \
  -B /opt/ivorysql/bin \
  -d /data/pg/data \
  -D /data/ivorysql/data \
  -g
```

Step 5: Start IvorySQL and verify

```
/opt/ivorysql/bin/pg_ctl -D /data/ivorysql/data start


# Verify if the database is functioning properly
/opt/ivorysql/bin/psql -p 5432 -c "SELECT version();"
```

Step 6: Clean up the old cluster

After the upgrade is completed,`pg_upgrade` will generate a cleanup script:

```
./delete_old_cluster.sh
```

Quick reference for common parameters

Parameters	Long options	Description
`-b`	`--old-bindir`	Source cluster executable directory
`-B`	`--new-bindir`	Target cluster executable directory
`-d`	`--old-datadir`	Source cluster data directory
`-D`	`--new-datadir`	Target cluster data directory
`-g`	`--using-ora-pg`	Upgrade from PostgreSQL to IvorySQL
`-p`	`--old-port`	Source cluster port
`-P`	`--new-port`	Target cluster PG port
`-q`	`--old-oraport`	Source cluster Oracle port
`-Q`	`--new-oraport`	Target cluster Oracle port
`-j`	`--jobs`	Number of parallel processes
`-k`	`--link`	Use hard links instead of file copying
`-c`	`--check`	Check compatibility only, do not perform the upgrade
`-v`	`--verbose`	Output detailed logs

Precautions

- The `-g` parameter is only used when the source is a pure PostgreSQL cluster、and the target is IvorySQL . If the source cluster is already IvorySQL, this parameter is not needed.
- Both the source and target clusters must be in a stopped state during the upgrade.
- Be sure to perform a complete backup of the source cluster before upgrading.

Managing IvorySQL Versions

IvorySQL is based on PostgreSQL and is updated at the same frequency as PostgreSQL, with one major release per year and one minor release per quarter. IvorySQL 5.4 is based on PostgreSQL 18.4, and all versions of IvorySQL are backward compatible.The relevant version features can be viewed by looking at Official Website。

Managing IvorySQL database access

IvorySQL uses the concept of Roles to manage database access rights. A role can be thought of as a database user or a group of database users, depending on how the role is set. Roles can own database

objects (for example, tables and functions) and can delegate permissions on those objects to other roles to control who can access which objects. In addition, membership in one role can be granted to another role, allowing member roles to use the permissions granted to another role.

The concept of roles encompasses the concepts of "user" and "group".

The database roles are conceptually completely separate from the operating system users. It may in fact be easier to maintain a correspondence, but this is not necessary. Database roles are global within a database cluster installation (and not within a separate database). To create a role, use the CREATE ROLE SQL command.

```
CREATE ROLE name;
```

Name follows the rules of SQL identifiers: either unadorned with no special characters or surrounded by double quotes (in fact, you will always have to add additional options to the command, such as LOGIN. see below for more details). To remove an existing role, use the similar DROP ROLE command.

```
DROP ROLE name;
```

For convenience, the createuser and dropuser programs are provided as wrappers for these SQL commands, which can be invoked from the shell command line at

```
createuser name
dropuser name
```

To determine the set of existing roles, check the pg_roles system directory, e.g.

```
SELECT rolname FROM pg_roles;
```

The \du meta command of the psql program can also be used to list existing roles.

To bootstrap a database system, a system that has just been initialized always contains a predefined role. This role is always a "superuser" and by default (unless changed when running initdb) it is named the same as the OS user who initialized the database cluster. By convention, this role will be named postgres. In order to create more roles, you must first connect as the initial role.

Each connection to the database server is established using a particular role name, and this role determines the initial access rights to the command that initiates the connection. The role name to use for a particular database connection is indicated by the client, which initiates the connection request in an application-related style. For example, the psql program uses the -U command line option to specify which role to connect under. Many applications assume that this name is the current operating system user by default (including createuser and psql). Therefore it is often convenient to maintain a name correspondence between the role and the OS user.

The set of database roles that a given client connection can use to connect to is determined by the authentication settings of that client, so a client is not limited to connecting with a role that matches its OS user, just as a person’ s login name does not need to match her real name. Because role identity determines the set of permissions available to a connected client, be careful when setting up a multi-user environment to configure permissions.

A database role can have a number of attributes that define the permissions of the role and interact with the client authentication system.

It is often convenient to group users together to facilitate the management of permissions: in this way,

permissions can be granted to or reclaimed from an entire group. This is done in IvorySQL by creating a role that represents a group, and then granting membership in that group role to individual user roles.

Since roles can own database objects and hold privileges to access other objects, deleting a role is often not a one-time DROP ROLE solution. Any objects owned by that user must first be deleted or transferred to another owner, and any privileges that have been granted to that role must be withdrawn.

For more details on database access management, refers to doc.

Defining Data Objects

IvorySQL is based on PostgreSQL and has a full SQL with defined data objects that can be referred to doc.On top of that, IvorySQL has done some Oracle proprietary data object compatibility for Oracle compatibility.

VARCHAR2

Overview

Variable length strings with maximum length bytes or characters. You must specify the size for VARCHAR2. The minimum size is 1 byte or 1 character.

Grammar

```
VARCHAR2(size)
```

Use Cases

```
create table test(a varchar2(5));
CREATE TABLE

SET NLS_LENGTH_SEMANTICS TO CHAR;
SET

SHOW NLS_LENGTH_SEMANTICS;
 nls_length_semantics
----------------------
 char
(1 row)

insert into test values ('Hello,Mr.li');
INSERT 0 1
```

Search Data

IvorySQL is developed based on PostgreSQL, with full SQL, query data specific operations can be referred to doc.

Use of foreign data

IvorySQL implements portions of the SQL/MED specification, allowing you to access data that resides outside IvorySQL using regular SQL queries. Such data is referred to as foreign data. (Note that this usage is

not to be confused with foreign keys, which are a type of constraint within the database.)

Foreign data is accessed with help from a foreign data wrapper. A foreign data wrapper is a library that can communicate with an external data source, hiding the details of connecting to the data source and obtaining data from it. There are some foreign data wrappers available as contrib modules;see Appendix F. Other kinds of foreign data wrappers might be found as third party products. If none of the existing foreign data wrappers suit your needs, you can write your own; see doc.

To access foreign data, you need to create a foreign server object, which defines how to connect to a particular external data source according to the set of options used by its supporting foreign data wrapper. Then you need to create one or more foreign tables, which define the structure of the remote data. A foreign table can be used in queries just like a normal table, but a foreign table has no storage in the IvorySQL server. Whenever it is used, IvorySQL asks the foreign data wrapper to fetch data from the external source, or transmit data to the external source in the case of update commands.

Accessing remote data may require authenticating to the external data source. This information can be provided by a user mapping, which can provide additional data such as user names and passwords based on the current IvorySQL role.

Backup and Restore

As with everything that contains valuable data, IvorySQL databases should be backed up regularly. While the procedure is essentially simple, it is important to have a clear understanding of the underlying techniques and assumptions.

There are three fundamentally different approaches to backing up IvorySQL data:

- SQL dump
- File system level backup
- Continuous archiving

SQL Dump

The idea behind this dump method is to generate a file with SQL commands that, when fed back to the server, will recreate the database in the same state as it was at the time of the dump. IvorySQL provides the utility program pg_dumpfor this purpose. The basic usage of this command is:

```
pg_dump dbname > dumpfile
```

As you see, pg_dump writes its result to the standard output. We will see below how this can be useful. While the above command creates a text file, pg_dump can create files in other formats that allow for parallelism and more fine-grained control of object restoration.

pg_dump is a regular IvorySQL client application (albeit a particularly clever one). This means that you can perform this backup procedure from any remote host that has access to the database. pg_dump must have read access to all tables that you want to back up, so in order to back up the entire database you almost always have to run it as a database superuser. (If you do not have sufficient privileges to back up the entire database, you can still back up portions of the database to which you do have access using options such as `-n \`schema\`` or `-t \`table\``.)

To specify which database server pg_dump should contact, use the command line options `-h \`host\`` and `-p \`port. The default host is the local host or whatever your \`HOST` environment variable specifies. Similarly, the default port is indicated by the `PORT` environment variable or, failing that, by the compiled-in default. (Conveniently, the server will normally have the same compiled-in default.)

pg_dump will by default connect with the database user name that is equal to the current operating system user name. To override this, either specify the `-U` option or set the environment variable `PGUSER`. Remember that pg_dump connections are subject to the normal client authentication mechanisms 。

An important advantage of pg_dump over the other backup methods described later is that pg_dump's output can generally be re-loaded into newer versions of IvorySQL, whereas file-level backups and continuous archiving are both extremely server-version-specific. pg_dump is also the only method that will work when transferring a database to a different machine architecture, such as going from a 32-bit to a 64-bit server.

Dumps created by pg_dump are internally consistent, meaning, the dump represents a snapshot of the database at the time pg_dump began running. pg_dump does not block other operations on the database while it is working. (Exceptions are those operations that need to operate with an exclusive lock, such as most forms of `ALTER TABLE`.)

Restoring the Dump

Text files created by pg_dump are intended to be read in by the psql program. The general command form to restore a dump is

```
psql dbname < dumpfile
```

where `dumpfile` is the file output by the pg_dump command. The database `dbname` will not be created by this command, so you must create it yourself from `template0` before executing psql (e.g., with `createdb -T template0 dbname`). psql supports options similar to pg_dump for specifying the database server to connect to and the user name to use. See the psql reference page for more information. Non-text file dumps are restored using the pg_restore utility.

Before restoring an SQL dump, all the users who own objects or were granted permissions on objects in the dumped database must already exist. If they do not, the restore will fail to recreate the objects with the original ownership and/or permissions. (Sometimes this is what you want, but usually it is not.)

By default, the psql script will continue to execute after an SQL error is encountered. You might wish to run psql with the `ON_ERROR_STOP` variable set to alter that behavior and have psql exit with an exit status of 3 if an SQL error occurs:

```
psql --set ON_ERROR_STOP=on dbname < infile
```

Either way, you will only have a partially restored database. Alternatively, you can specify that the whole dump should be restored as a single transaction, so the restore is either fully completed or fully rolled back. This mode can be specified by passing the `-1` or `--single-transaction` command-line options to psql. When using this mode, be aware that even a minor error can rollback a restore that has already run for many hours. However, that might still be preferable to manually cleaning up a complex database after a partially restored dump.

The ability of pg_dump and psql to write to or read from pipes makes it possible to dump a database directly from one server to another, for example:

```
pg_dump -h host1 dbname | psql -h host2 dbname
```

Important: The dumps produced by pg_dump are relative to `template0`. This means that any languages, procedures, etc. added via `template1` will also be dumped by pg_dump. As a result, when restoring, if you are using a customized `template1`, you must create the empty database from `template0`, as in the example above.

After restoring a backup, it is wise to run `ANALYZE` on each database so the query optimizer has useful statistics.

Using pg_dumpall

pg_dump dumps only a single database at a time, and it does not dump information about roles or tablespaces (because those are cluster-wide rather than per-database). To support convenient dumping of the entire contents of a database cluster, the pg_dumpall program is provided. pg_dumpall backs up each database in a given cluster, and also preserves cluster-wide data such as role and tablespace definitions. The basic usage of this command is:

```
pg_dumpall > dumpfile
```

The resulting dump can be restored with psql:

```
psql -f dumpfile ivorysql
```

(Actually, you can specify any existing database name to start from, but if you are loading into an empty cluster then ivorysql should usually be used.) It is always necessary to have database superuser access when restoring a pg_dumpall dump, as that is required to restore the role and tablespace information. If you use tablespaces, make sure that the tablespace paths in the dump are appropriate for the new installation.

pg_dumpall works by emitting commands to re-create roles, tablespaces, and empty databases, then invoking pg_dump for each database. This means that while each database will be internally consistent, the snapshots of different databases are not synchronized.

Cluster-wide data can be dumped alone using the pg_dumpall `--globals-only` option. This is necessary to fully backup the cluster if running the pg_dump command on individual databases.

Handling Large Databases

Some operating systems have maximum file size limits that cause problems when creating large pg_dump output files. Fortunately, pg_dump can write to the standard output, so you can use standard Unix tools to work around this potential problem. There are several possible methods:

Use compressed dumps. You can use your favorite compression program, for example gzip:

```
pg_dump dbname | gzip > filename.gz
```

Reload with:

```
gunzip -c filename.gz | psql dbname
```

or:

```
cat filename.gz | gunzip | psql dbname
```

Use `split`. The `split` command allows you to split the output into smaller files that are acceptable in size to the underlying file system. For example, to make 2 gigabyte chunks:

```
pg_dump dbname | split -b 2G - filename
```

Reload with:

```
cat filename* | psql dbname
```

If using GNU split, it is possible to use it and gzip together:

```
pg_dump dbname | split -b 2G --filter='gzip > $FILE.gz'
```

It can be restored using zcat.

Use pg_dump’s custom dump format. If IvorySQL was built on a system with the zlib compression library installed, the custom dump format will compress data as it writes it to the output file. This will produce dump file sizes similar to using gzip, but it has the added advantage that tables can be restored selectively. The following command dumps a database using the custom dump format:

```
pg_dump -Fc dbname > filename
```

A custom-format dump is not a script for psql, but instead must be restored with pg_restore, for example:

```
pg_restore -d dbname filename
```

See the pg_dump and pg_restore reference pages for details.

For very large databases, you might need to combine split with one of the other two approaches.

Use pg_dump’s parallel dump feature. To speed up the dump of a large database, you can use pg_dump’s parallel mode. This will dump multiple tables at the same time. You can control the degree of parallelism with the -j parameter. Parallel dumps are only supported for the "directory" archive format.

```
pg_dump -j num -F d -f out.dir dbname
```

You can use pg_restore -j to restore a dump in parallel. This will work for any archive of either the "custom" or the "directory" archive mode, whether or not it has been created with pg_dump -j.

File System Level Backup

```
An alternative backup strategy is to directly copy the files that IvorySQL uses to
store the data in the database, You can use whatever method you prefer for doing file
system backups; for example:
```

```
tar -cf backup.tar /usr/local/pgsql/data
```

There are two restrictions, however, which make this method impractical, or at least inferior to the pg_dump method:

1. The database server must be shut down in order to get a usable backup. Half-way measures such as disallowing all connections will not work (in part because tar and similar tools do not take an atomic snapshot of the state of the file system, but also because of internal buffering within the server).Needless to say, you also need to shut down the server before restoring the data.
2. If you have dug into the details of the file system layout of the database, you might be tempted to try to

back up or restore only certain individual tables or databases from their respective files or directories. This will not work because the information contained in these files is not usable without the commit log files, `pg_xact/*`, which contain the commit status of all transactions. A table file is only usable with this information. Of course it is also impossible to restore only a table and the associated `pg_xact` data because that would render all other tables in the database cluster useless. So file system backups only work for complete backup and restoration of an entire database cluster.

An alternative file-system backup approach is to make a "consistent snapshot" of the data directory, if the file system supports that functionality (and you are willing to trust that it is implemented correctly). The typical procedure is to make a "frozen snapshot" of the volume containing the database, then copy the whole data directory (not just parts, see above) from the snapshot to a backup device, then release the frozen snapshot. This will work even while the database server is running. However, a backup created in this way saves the database files in a state as if the database server was not properly shut down; therefore, when you start the database server on the backed-up data, it will think the previous server instance crashed and will replay the WAL log. This is not a problem; just be aware of it (and be sure to include the WAL files in your backup). You can perform a `CHECKPOINT` before taking the snapshot to reduce recovery time.

If your database is spread across multiple file systems, there might not be any way to obtain exactly-simultaneous frozen snapshots of all the volumes. For example, if your data files and WAL log are on different disks, or if tablespaces are on different file systems, it might not be possible to use snapshot backup because the snapshots must be simultaneous. Read your file system documentation very carefully before trusting the consistent-snapshot technique in such situations.

If simultaneous snapshots are not possible, one option is to shut down the database server long enough to establish all the frozen snapshots. Another option is to perform a continuous archiving base backup,because such backups are immune to file system changes during the backup. This requires enabling continuous archiving just during the backup process; restore is done using continuous archive recovery

Another option is to use rsync to perform a file system backup. This is done by first running rsync while the database server is running, then shutting down the database server long enough to do an `rsync --checksum`. (`--checksum` is necessary because `rsync` only has file modification-time granularity of one second.) The second rsync will be quicker than the first, because it has relatively little data to transfer, and the end result will be consistent because the server was down. This method allows a file system backup to be performed with minimal downtime.

Note that a file system backup will typically be larger than an SQL dump. (pg_dump does not need to dump the contents of indexes for example, just the commands to recreate them.) However, taking a file system backup might be faster.

Continuous Archiving and Point-in-Time Recovery (PITR)

At all times,IvorySQLmaintains a write ahead log (WAL) in the `pg_wal/` subdirectory of the cluster's data directory. The log records every change made to the database's data files. This log exists primarily for crash-safety purposes: if the system crashes, the database can be restored to consistency by "replaying" the log entries made since the last checkpoint. However, the existence of the log makes it possible to use a third strategy for backing up databases: we can combine a file-system-level backup with backup of the WAL files. If recovery is needed, we restore the file system backup and then replay from the backed-up WAL files to bring the system to a current state. This approach is more complex to administer than either of the previous approaches, but it has some significant benefits:

- We do not need a perfectly consistent file system backup as the starting point. Any internal inconsistency in the backup will be corrected by log replay (this is not significantly different from what happens during crash recovery). So we do not need a file system snapshot capability, just tar or a similar archiving tool.
- Since we can combine an indefinitely long sequence of WAL files for replay, continuous backup can be achieved simply by continuing to archive the WAL files. This is particularly valuable for large databases, where it might not be convenient to take a full backup frequently.
- It is not necessary to replay the WAL entries all the way to the end. We could stop the replay at any point and have a consistent snapshot of the database as it was at that time. Thus, this technique supports point-in-time recovery: it is possible to restore the database to its state at any time since your base backup was taken.

- If we continuously feed the series of WAL files to another machine that has been loaded with the same base backup file, we have a warm standby system: at any point we can bring up the second machine and it will have a nearly-current copy of the database.

Note: pg_dump and pg_dumpall do not produce file-system-level backups and cannot be used as part of a continuous-archiving solution. Such dumps are logical and do not contain enough information to be used by WAL replay.

As with the plain file-system-backup technique, this method can only support restoration of an entire database cluster, not a subset. Also, it requires a lot of archival storage: the base backup might be bulky, and a busy system will generate many megabytes of WAL traffic that have to be archived. Still, it is the preferred backup technique in many situations where high reliability is needed.

To recover successfully using continuous archiving (also called "online backup" by many database vendors), you need a continuous sequence of archived WAL files that extends back at least as far as the start time of your backup. So to get started, you should set up and test your procedure for archiving WAL files before you take your first base backup. Accordingly, we first discuss the mechanics of archiving WAL files.For more information on how to create archives and backups and the key points during operation, please refer to doc。

Loading and unloading data

`COPY` moves data between IvorySQL tables and standard file-system files. `COPY TO` copies the contents of a table to a file, while `COPY FROM` copies data from a file to a table (appending the data to whatever is in the table already). `COPY TO` can also copy the results of a `SELECT` query.

If a column list is specified, `COPY TO` copies only the data in the specified columns to the file. For `COPY FROM`, each field in the file is inserted, in order, into the specified column. Table columns not specified in the `COPY FROM` column list will receive their default values.

`COPY` with a file name instructs the IvorySQL server to directly read from or write to a file. The file must be accessible by the IvorySQL user (the user ID the server runs as) and the name must be specified from the viewpoint of the server. When `PROGRAM` is specified, the server executes the given command and reads from the standard output of the program, or writes to the standard input of the program. The command must be specified from the viewpoint of the server, and be executable by the IvorySQL user. When `STDIN` or `STDOUT` is specified, data is transmitted via the connection between the client and the server.

Each backend running `COPY` will report its progress in the `pg_stat_progress_copy` view.

Synopsis

```
COPY table_name [ ( column_name [, ...] ) ]
    FROM { 'filename' | PROGRAM 'command' | STDIN }
    [ [ WITH ] ( option [, ...] ) ]
    [ WHERE condition ]

COPY { table_name [ ( column_name [, ...] ) ] | ( query ) }
    TO { 'filename' | PROGRAM 'command' | STDOUT }
    [ [ WITH ] ( option [, ...] ) ]

where option can be one of:

    FORMAT format_name
    FREEZE [ boolean ]
```

```
    DELIMITER 'delimiter_character'
    NULL 'null_string'
    HEADER [ boolean ]
    QUOTE 'quote_character'
    ESCAPE 'escape_character'
    FORCE_QUOTE { ( column_name [, ...] ) | * }
    FORCE_NOT_NULL ( column_name [, ...] )
    FORCE_NULL ( column_name [, ...] )
    ENCODING 'encoding_name'
```

For detailed parameter settings, please refer to doc.

Outputs

On successful completion, a COPY command returns a command tag of the form

```
COPY count
```

The count is the number of rows copied.

Note： psql will print this command tag only if the command was not COPY ⋯ TO STDOUT, or the equivalent psql meta-command \copy ⋯ to stdout. This is to prevent confusing the command tag with the data that was just printed.

Notes

COPY TO can be used only with plain tables, not views, and does not copy rows from child tables or child partitions. For example, COPY `table TO` copies the same rows as SELECT * FROM ONLY `table. The syntax `COPY (SELECT * FROM `table) TO ⋯` can be used to dump all of the rows in an inheritance hierarchy, partitioned table, or view.

COPY FROM can be used with plain, foreign, or partitioned tables or with views that have INSTEAD OF INSERT triggers.

You must have select privilege on the table whose values are read by COPY TO, and insert privilege on the table into which values are inserted by COPY FROM. It is sufficient to have column privileges on the column(s) listed in the command.

If row-level security is enabled for the table, the relevant SELECT policies will apply to COPY `table TO` statements. Currently, COPY FROM is not supported for tables with row-level security. Use equivalent INSERT statements instead.

Files named in a COPY command are read or written directly by the server, not by the client application. Therefore, they must reside on or be accessible to the database server machine, not the client. They must be accessible to and readable or writable by the IvorySQL user (the user ID the server runs as), not the client. Similarly, the command specified with PROGRAM is executed directly by the server, not by the client application, must be executable by the IvorySQL user. COPY naming a file or command is only allowed to database superusers or users who are granted one of the roles pg_read_server_files, pg_write_server_files, or pg_execute_server_program, since it allows reading or writing any file or running a program that the server has privileges to access.

Do not confuse COPY with the psql instruction \copy. \copy invokes COPY FROM STDIN or COPY TO STDOUT, and then fetches/stores the data in a file accessible to the psql client. Thus, file accessibility and access rights depend on the client rather than the server when \copy is used.

It is recommended that the file name used in `COPY` always be specified as an absolute path. This is enforced by the server in the case of `COPY TO`, but for `COPY FROM` you do have the option of reading from a file specified by a relative path. The path will be interpreted relative to the working directory of the server process (normally the cluster's data directory), not the client's working directory.

Executing a command with `PROGRAM` might be restricted by the operating system's access control mechanisms, such as SELinux.

`COPY FROM` will invoke any triggers and check constraints on the destination table. However, it will not invoke rules.

For identity columns, the `COPY FROM` command will always write the column values provided in the input data, like the `INSERT` option `OVERRIDING SYSTEM VALUE`.

`COPY` input and output is affected by `DateStyle`. To ensure portability to other IvorySQL installations that might use non-default `DateStyle` settings, `DateStyle` should be set to `ISO` before using `COPY TO`. It is also a good idea to avoid dumping data with `IntervalStyle` set to `sql_standard`, because negative interval values might be misinterpreted by a server that has a different setting for `IntervalStyle`.

Input data is interpreted according to `ENCODING` option or the current client encoding, and output data is encoded in `ENCODING` or the current client encoding, even if the data does not pass through the client but is read from or written to a file directly by the server.

`COPY` stops operation at the first error. This should not lead to problems in the event of a `COPY TO`, but the target table will already have received earlier rows in a `COPY FROM`. These rows will not be visible or accessible, but they still occupy disk space. This might amount to a considerable amount of wasted disk space if the failure happened well into a large copy operation. You might wish to invoke `VACUUM` to recover the wasted space.

`FORCE_NULL` and `FORCE_NOT_NULL` can be used simultaneously on the same column. This results in converting quoted null strings to null values and unquoted null strings to empty strings.

File Formats

Text Format

When the `text` format is used, the data read or written is a text file with one line per table row. Columns in a row are separated by the delimiter character. The column values themselves are strings generated by the output function, or acceptable to the input function, of each attribute's data type. The specified null string is used in place of columns that are null. `COPY FROM` will raise an error if any line of the input file contains more or fewer columns than are expected.

End of data can be represented by a single line containing just backslash-period (`\.`). An end-of-data marker is not necessary when reading from a file, since the end of file serves perfectly well; it is needed only when copying data to or from client applications using pre-3.0 client protocol.

Backslash characters (\) can be used in the `COPY` data to quote data characters that might otherwise be taken as row or column delimiters. In particular, the following characters must be preceded by a backslash if they appear as part of a column value: backslash itself, newline, carriage return, and the current delimiter character.

The specified null string is sent by `COPY TO` without adding any backslashes; conversely, `COPY FROM` matches the input against the null string before removing backslashes. Therefore, a null string such as `\N` cannot be confused with the actual data value `\N` (which would be represented as `\\N`).

The following special backslash sequences are recognized by `COPY FROM`:

Sequence	Represents
\b	Backspace (ASCII 8)
\f	Form feed (ASCII 12)

\n	Newline (ASCII 10)
\r	Carriage return (ASCII 13)
\t	Tab (ASCII 9)
\v	Vertical tab (ASCII 11)
\digits	Backslash followed by one to three octal digits specifies the byte with that numeric code
\xdigits	Backslash x followed by one or two hex digits specifies the byte with that numeric code

Presently, COPY TO will never emit an octal or hex-digits backslash sequence, but it does use the other sequences listed above for those control characters.

Any other backslashed character that is not mentioned in the above table will be taken to represent itself. However, beware of adding backslashes unnecessarily, since that might accidentally produce a string matching the end-of-data marker (\.) or the null string (\N by default). These strings will be recognized before any other backslash processing is done.

It is strongly recommended that applications generating COPY data convert data newlines and carriage returns to the \n and \r sequences respectively. At present it is possible to represent a data carriage return by a backslash and carriage return, and to represent a data newline by a backslash and newline. However, these representations might not be accepted in future releases. They are also highly vulnerable to corruption if the COPY file is transferred across different machines (for example, from Unix to Windows or vice versa).

All backslash sequences are interpreted after encoding conversion. The bytes specified with the octal and hex-digit backslash sequences must form valid characters in the database encoding.

COPY TO will terminate each row with a Unix-style newline ("\n"). Servers running on Microsoft Windows instead output carriage return/newline ("\r\n"), but only for COPY to a server file; for consistency across platforms, COPY TO STDOUT always sends "\n" regardless of server platform. COPY FROM can handle lines ending with newlines, carriage returns, or carriage return/newlines. To reduce the risk of error due to un-backslashed newlines or carriage returns that were meant as data, COPY FROM will complain if the line endings in the input are not all alike.

CSV Format

This format option is used for importing and exporting the Comma Separated Value (CSV) file format used by many other programs, such as spreadsheets. Instead of the escaping rules used by IvorySQL's standard text format, it produces and recognizes the common CSV escaping mechanism.

The values in each record are separated by the DELIMITER character. If the value contains the delimiter character, the QUOTE character, the NULL string, a carriage return, or line feed character, then the whole value is prefixed and suffixed by the QUOTE character, and any occurrence within the value of a QUOTE character or the ESCAPE character is preceded by the escape character. You can also use FORCE_QUOTE to force quotes when outputting non-NULL values in specific columns.

The CSV format has no standard way to distinguish a NULL value from an empty string. IvorySQL's COPY handles this by quoting. A NULL is output as the NULL parameter string and is not quoted, while a non-NULL value matching the NULL parameter string is quoted. For example, with the default settings, a NULL is written as an unquoted empty string, while an empty string data value is written with double quotes (""). Reading values follows similar rules. You can use FORCE_NOT_NULL to prevent NULL input comparisons for specific columns. You can also use FORCE_NULL to convert quoted null string data values to NULL.

Because backslash is not a special character in the CSV format, \., the end-of-data marker, could also appear as a data value. To avoid any misinterpretation, a \. data value appearing as a lone entry on a line is automatically quoted on output, and on input, if quoted, is not interpreted as the end-of-data marker. If you are loading a file created by another application that has a single unquoted column and might have a value of \., you might need to quote that value in the input file.

Note

CSV format, all characters are significant. A quoted value surrounded by white space, or any characters other than **DELIMITER**, will include those characters. This can cause errors if you import data from a system that pads **CSV** lines with white space out to some fixed width. If such a situation arises you might need to preprocess the **CSV** file to remove the trailing white space, before importing the data into IvorySQL.

Note

CSV format will both recognize and produce CSV files with quoted values containing embedded carriage returns and line feeds. Thus the files are not strictly one line per table row like text-format files.

Note

Many programs produce strange and occasionally perverse CSV files, so the file format is more a convention than a standard. Thus you might encounter some files that cannot be imported using this mechanism, and **COPY** might produce files that other programs cannot process.

Binary Format

The **binary** format option causes all data to be stored/read as binary format rather than as text. It is somewhat faster than the text and **CSV** formats, but a binary-format file is less portable across machine architectures and IvorySQL versions. Also, the binary format is very data type specific; for example it will not work to output binary data from a **smallint** column and read it into an **integer** column, even though that would work fine in text format.

The **binary** file format consists of a file header, zero or more tuples containing the row data, and a file trailer. Headers and data are in network byte order.

File Header

The file header consists of 15 bytes of fixed fields, followed by a variable-length header extension area.

The fixed fields are:

Signature

11-byte sequence PGCOPY\n\377\r\n\0 — note that the zero byte is a required part of the signature. (The signature is designed to allow easy identification of files that have been munged by a non-8-bit-clean transfer. This signature will be changed by end-of-line-translation filters, dropped zero bytes, dropped high bits, or parity changes.)

Flags field

32-bit integer bit mask to denote important aspects of the file format. Bits are numbered from 0 (LSB) to 31 (MSB). Note that this field is stored in network byte order (most significant byte first), as are all the integer fields used in the file format. Bits 16–31 are reserved to denote critical file format issues; a reader should abort if it finds an unexpected bit set in this range. Bits 0–15 are reserved to signal backwards-compatible format issues; a reader should simply ignore any unexpected bits set in this range. Currently only one flag bit is defined, and the rest must be zero:

Bit 16

If 1, OIDs are included in the data; if 0, not. Oid system columns are not supported in IvorySQL anymore, but the format still contains the indicator.

Header extension area length

32-bit integer, length in bytes of remainder of header, not including self. Currently, this is zero, and the first tuple follows immediately. Future changes to the format might allow additional data to be present in the header. A reader should silently skip over any header extension data it does not know what to do with.

The header extension area is envisioned to contain a sequence of self-identifying chunks. The flags field is not intended to tell readers what is in the extension area. Specific design of header extension contents is left for a later release.

This design allows for both backwards-compatible header additions (add header extension chunks, or set low-order flag bits) and non-backwards-compatible changes (set high-order flag bits to signal such changes, and add supporting data to the extension area if needed).

Tuples

Each tuple begins with a 16-bit integer count of the number of fields in the tuple. (Presently, all tuples in a table will have the same count, but that might not always be true.) Then, repeated for each field in the tuple, there is a 32-bit length word followed by that many bytes of field data. (The length word does not include itself, and can be zero.) As a special case, -1 indicates a NULL field value. No value bytes follow in the NULL case.

There is no alignment padding or any other extra data between fields.

Presently, all data values in a binary-format file are assumed to be in binary format (format code one). It is anticipated that a future extension might add a header field that allows per-column format codes to be specified.

To determine the appropriate binary format for the actual tuple data you should consult the PostgreSQL source, in particular the `*send` and `*recv` functions for each column's data type (typically these functions are found in the `src/backend/utils/adt/` directory of the source distribution).

If OIDs are included in the file, the OID field immediately follows the field-count word. It is a normal field except that it's not included in the field-count. Note that oid system columns are not supported in current versions of IvorySQL.

File Trailer

The file trailer consists of a 16-bit integer word containing -1. This is easily distinguished from a tuple's field-count word.

A reader should report an error if a field-count word is neither -1 nor the expected number of columns. This provides an extra check against somehow getting out of sync with the data.

Examples

The following example copies a table to the client using the vertical bar (|) as the field delimiter:

```
COPY country TO STDOUT (DELIMITER '|');
```

To copy data from a file into the `country` table:

```
COPY country TO STDOUT (DELIMITER '|');
```

```
To copy into a file just the countries whose names start with 'A':
```

```
COPY (SELECT * FROM country WHERE country_name LIKE 'A%') TO
'/usr1/proj/bray/sql/a_list_countries.copy';
```

To copy into a compressed file, you can pipe the output through an external compression program:

```
COPY country TO PROGRAM 'gzip > /usr1/proj/bray/sql/country_data.gz';
```

Here is a sample of data suitable for copying into a table from `STDIN`:

```
AF      AFGHANISTAN
AL      ALBANIA
DZ      ALGERIA
ZM      ZAMBIA
ZW      ZIMBABWE
```

> Note that the white space on each line is actually a tab character.

The following is the same data, output in binary format. The data is shown after filtering through the Unix utility `od -c`. The table has three columns; the first has type `char(2)`, the second has type `text`, and the third has type `integer`. All the rows have a null value in the third column.

```
0000000   P   G   C   O   P   Y  \n 377  \r  \n  \0  \0  \0  \0  \0  \0
0000020  \0  \0  \0  \0 003  \0  \0  \0 002   A   F  \0  \0  \0 013   A
0000040   F   G   H   A   N   I   S   T   A   N 377 377 377 377  \0 003
0000060  \0  \0  \0 002   A   L  \0  \0  \0 007   A   L   B   A   N   I
0000100   A 377 377 377 377  \0 003  \0  \0  \0 002   D   Z  \0  \0  \0
0000120 007   A   L   G   E   R   I   A 377 377 377 377  \0 003  \0  \0
0000140  \0 002   Z   M  \0  \0  \0 006   Z   A   M   B   I   A 377 377
0000160 377 377  \0 003  \0  \0  \0 002   Z   W  \0  \0  \0  \b   Z   I
0000200   M   B   A   B   W   E 377 377 377 377 377 377
```

The remaining details can see doc.

Performance Tips

Query performance can be affected by a variety of factors. Some of these factors can be controlled by the user, while others are fundamentals of the system’ s lower-level design.

Using EXPLAIN

IvorySQL devises a query plan for each query it receives. Choosing the right plan to match the query structure and the properties of the data is absolutely critical for good performance, so the system includes a complex planner that tries to choose good plans. You can use the `EXPLAIN` command to see what query plan the planner creates for any query. Plan-reading is an art that requires some experience to master, but this section attempts to cover the basics.

The examples use `EXPLAIN’ s default “text” output format, which is compact and convenient for

humans to read. If you want to feed `EXPLAIN’s output to a program for further analysis, you should use one of its machine-readable output formats (XML, JSON, or YAML) instead.

EXPLAIN Basics

The structure of a query plan is a tree of plan nodes. Nodes at the bottom level of the tree are scan nodes: they return raw rows from a table. There are different types of scan nodes for different table access methods: sequential scans, index scans, and bitmap index scans. There are also non-table row sources, such as `VALUES` clauses and set-returning functions in `FROM`, which have their own scan node types. If the query requires joining, aggregation, sorting, or other operations on the raw rows, then there will be additional nodes above the scan nodes to perform these operations. Again, there is usually more than one possible way to do these operations, so different node types can appear here too. The output of `EXPLAIN` has one line for each node in the plan tree, showing the basic node type plus the cost estimates that the planner made for the execution of that plan node. Additional lines might appear, indented from the node’s summary line, to show additional properties of the node. The very first line (the summary line for the topmost node) has the estimated total execution cost for the plan; it is this number that the planner seeks to minimize.

Here is a trivial example, just to show what the output looks like:

```
EXPLAIN SELECT * FROM tenk1;

                        QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

Since this query has no `WHERE` clause, it must scan all the rows of the table, so the planner has chosen to use a simple sequential scan plan. The numbers that are quoted in parentheses are (left to right):

- Estimated start-up cost. This is the time expended before the output phase can begin, e.g., time to do the sorting in a sort node.
- Estimated total cost. This is stated on the assumption that the plan node is run to completion, i.e., all available rows are retrieved. In practice a node’s parent node might stop short of reading all available rows (see the `LIMIT` example below).
- Estimated number of rows output by this plan node. Again, the node is assumed to be run to completion.
- Estimated average width of rows output by this plan node (in bytes).

The costs are measured in arbitrary units determined by the planner’s cost parameters .Traditional practice is to measure the costs in units of disk page fetches; that is, seq_page_cost is conventionally set to `1.0` and the other cost parameters are set relative to that. The examples in this section are run with the default cost parameters.

It’s important to understand that the cost of an upper-level node includes the cost of all its child nodes. It’s also important to realize that the cost only reflects things that the planner cares about. In particular, the cost does not consider the time spent transmitting result rows to the client, which could be an important factor in the real elapsed time; but the planner ignores it because it cannot change it by altering the plan. (Every correct plan will output the same row set, we trust.)

The `rows` value is a little tricky because it is not the number of rows processed or scanned by the plan node, but rather the number emitted by the node. This is often less than the number scanned, as a result of filtering by any `WHERE`-clause conditions that are being applied at the node. Ideally the top-level rows estimate will approximate the number of rows actually returned, updated, or deleted by the query.

Returning to our example:

```
EXPLAIN SELECT * FROM tenk1;
```

```
                        QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

These numbers are derived very straightforwardly. If you do:

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 'tenk1';
```

you will find that **tenk1** has 358 disk pages and 10000 rows. The estimated cost is computed as (disk pages read * seq_page_cost) + (rows scanned * cpu_tuple_cost). By default, **seq_page_cost** is 1.0 and **cpu_tuple_cost** is 0.01, so the estimated cost is (358 * 1.0) + (10000 * 0.01) = 458.

Now let's modify the query to add a **WHERE** condition:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 7000;

                        QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=7001 width=244)
   Filter: (unique1 < 7000)
```

Notice that the **EXPLAIN** output shows the **WHERE** clause being applied as a "filter" condition attached to the Seq Scan plan node. This means that the plan node checks the condition for each row it scans, and outputs only the ones that pass the condition. The estimate of output rows has been reduced because of the **WHERE** clause. However, the scan will still have to visit all 10000 rows, so the cost hasn't decreased; in fact it has gone up a bit (by 10000 * cpu_operator_cost, to be exact) to reflect the extra CPU time spent checking the **WHERE** condition.

The actual number of rows this query would select is 7000, but the **rows** estimate is only approximate. If you try to duplicate this experiment, you will probably get a slightly different estimate; moreover, it can change after each **ANALYZE** command, because the statistics produced by **ANALYZE** are taken from a randomized sample of the table.

Now, let's make the condition more restrictive:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100;

                                QUERY PLAN
---------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=5.07..229.20 rows=101 width=244)
   Recheck Cond: (unique1 < 100)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0)
         Index Cond: (unique1 < 100)
```

Here the planner has decided to use a two-step plan: the child plan node visits an index to find the locations of rows matching the index condition, and then the upper plan node actually fetches those rows from the table itself. Fetching rows separately is much more expensive than reading them sequentially, but because

not all the pages of the table have to be visited, this is still cheaper than a sequential scan. (The reason for using two plan levels is that the upper plan node sorts the row locations identified by the index into physical order before reading them, to minimize the cost of separate fetches. The "bitmap" mentioned in the node names is the mechanism that does the sorting.)

Now let's add another condition to the `WHERE` clause:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND stringu1 = 'xxx';

                                QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=5.04..229.43 rows=1 width=244)
   Recheck Cond: (unique1 < 100)
   Filter: (stringu1 = 'xxx'::name)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0)
         Index Cond: (unique1 < 100)
```

The added condition `stringu1 = 'xxx'` reduces the output row count estimate, but not the cost because we still have to visit the same set of rows. Notice that the `stringu1` clause cannot be applied as an index condition, since this index is only on the `unique1` column. Instead it is applied as a filter on the rows retrieved by the index. Thus the cost has actually gone up slightly to reflect this extra checking.

In some cases the planner will prefer a "simple" index scan plan:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 = 42;

                                QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using tenk1_unique1 on tenk1  (cost=0.29..8.30 rows=1 width=244)
   Index Cond: (unique1 = 42)
```

In this type of plan the table rows are fetched in index order, which makes them even more expensive to read, but there are so few that the extra cost of sorting the row locations is not worth it. You'll most often see this plan type for queries that fetch just a single row. It's also often used for queries that have an `ORDER BY` condition that matches the index order, because then no extra sorting step is needed to satisfy the `ORDER BY`. In this example, adding `ORDER BY unique1` would use the same plan because the index already implicitly provides the requested ordering.

The planner may implement an `ORDER BY` clause in several ways. The above example shows that such an ordering clause may be implemented implicitly. The planner may also add an explicit `sort` step:

```
EXPLAIN SELECT * FROM tenk1 ORDER BY unique1;
                           QUERY PLAN
-----------------------------------------------------------------
 Sort  (cost=1109.39..1134.39 rows=10000 width=244)
   Sort Key: unique1
   ->  Seq Scan on tenk1  (cost=0.00..445.00 rows=10000 width=244)
```

```
If a part of the plan guarantees an ordering on a prefix of the required sort keys,
then the planner may instead decide to use an `incremental sort` step:
```

```
EXPLAIN SELECT * FROM tenk1 ORDER BY four, ten LIMIT 100;
                                          QUERY PLAN
----------------------------------------------------------------------
----------------------------------
 Limit  (cost=521.06..538.05 rows=100 width=244)
   ->  Incremental Sort  (cost=521.06..2220.95 rows=10000 width=244)
         Sort Key: four, ten
         Presorted Key: four
         ->  Index Scan using index_tenk1_on_four on tenk1  (cost=0.29..1510.08
rows=10000 width=244)
```

Compared to regular sorts, sorting incrementally allows returning tuples before the entire result set has been sorted, which particularly enables optimizations with `LIMIT` queries. It may also reduce memory usage and the likelihood of spilling sorts to disk, but it comes at the cost of the increased overhead of splitting the result set into multiple sorting batches.

If there are separate indexes on several of the columns referenced in `WHERE`, the planner might choose to use an AND or OR combination of the indexes:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000;

                                     QUERY PLAN
-------------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=25.08..60.21 rows=10 width=244)
   Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))
   ->  BitmapAnd  (cost=25.08..25.08 rows=10 width=0)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0)
               Index Cond: (unique1 < 100)
         ->  Bitmap Index Scan on tenk1_unique2  (cost=0.00..19.78 rows=999 width=0)
               Index Cond: (unique2 > 9000)
```

But this requires visiting both indexes, so it's not necessarily a win compared to using just one index and treating the other condition as a filter. If you vary the ranges involved you'll see the plan change accordingly.

Here is an example showing the effects of `LIMIT`:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000 LIMIT 2;

                                     QUERY PLAN
-------------------------------------------------------------------------------------
 Limit  (cost=0.29..14.48 rows=2 width=244)
```

```
   ->  Index Scan using tenk1_unique2 on tenk1  (cost=0.29..71.27 rows=10 width=244)
         Index Cond: (unique2 > 9000)
         Filter: (unique1 < 100)
```

This is the same query as above, but we added a `LIMIT` so that not all the rows need be retrieved, and the planner changed its mind about what to do. Notice that the total cost and row count of the Index Scan node are shown as if it were run to completion. However, the Limit node is expected to stop after retrieving only a fifth of those rows, so its total cost is only a fifth as much, and that's the actual estimated cost of the query. This plan is preferred over adding a Limit node to the previous plan because the Limit could not avoid paying the startup cost of the bitmap scan, so the total cost would be something over 25 units with that approach.

Let's try joining two tables, using the columns we have been discussing:

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t1.unique2 = t2.unique2;

                                      QUERY PLAN
--------------------------------------------------------------------------------
 Nested Loop  (cost=4.65..118.62 rows=10 width=488)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.47 rows=10 width=244)
         Recheck Cond: (unique1 < 10)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36 rows=10 width=0)
               Index Cond: (unique1 < 10)
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.29..7.91 rows=1 width=244)
         Index Cond: (unique2 = t1.unique2)
```

In this plan, we have a nested-loop join node with two table scans as inputs, or children. The indentation of the node summary lines reflects the plan tree structure. The join's first, or "outer", child is a bitmap scan similar to those we saw before. Its cost and row count are the same as we'd get from `SELECT ⋯ WHERE unique1 < 10` because we are applying the `WHERE` clause `unique1 < 10` at that node. The `t1.unique2 = t2.unique2` clause is not relevant yet, so it doesn't affect the row count of the outer scan. The nested-loop join node will run its second, or "inner" child once for each row obtained from the outer child. Column values from the current outer row can be plugged into the inner scan; here, the `t1.unique2` value from the outer row is available, so we get a plan and costs similar to what we saw above for a simple `SELECT ⋯ WHERE t2.unique2 = `constant`` case. (The estimated cost is actually a bit lower than what was seen above, as a result of caching that's expected to occur during the repeated index scans on `t2`.) The costs of the loop node are then set on the basis of the cost of the outer scan, plus one repetition of the inner scan for each outer row (10 * 7.91, here), plus a little CPU time for join processing.

In this example the join's output row count is the same as the product of the two scans' row counts, but that's not true in all cases because there can be additional `WHERE` clauses that mention both tables and so can only be applied at the join point, not to either input scan. Here's an example:

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t2.unique2 < 10 AND t1.hundred < t2.hundred;
```

```
                                        QUERY PLAN
-------------------------------------------------------------------------------------
 Nested Loop  (cost=4.65..49.46 rows=33 width=488)
   Join Filter: (t1.hundred < t2.hundred)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.47 rows=10 width=244)
         Recheck Cond: (unique1 < 10)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36 rows=10 width=0)
               Index Cond: (unique1 < 10)
   ->  Materialize  (cost=0.29..8.51 rows=10 width=244)
         ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.29..8.46 rows=10
width=244)
               Index Cond: (unique2 < 10)
```

The condition `t1.hundred < t2.hundred` can't be tested in the `tenk2_unique2` index, so it's applied at the join node. This reduces the estimated output row count of the join node, but does not change either input scan.

Notice that here the planner has chosen to "materialize" the inner relation of the join, by putting a Materialize plan node atop it. This means that the `t2` index scan will be done just once, even though the nested-loop join node needs to read that data ten times, once for each row from the outer relation. The Materialize node saves the data in memory as it's read, and then returns the data from memory on each subsequent pass.

When dealing with outer joins, you might see join plan nodes with both "Join Filter" and plain "Filter" conditions attached. Join Filter conditions come from the outer join's `ON` clause, so a row that fails the Join Filter condition could still get emitted as a null-extended row. But a plain Filter condition is applied after the outer-join rules and so acts to remove rows unconditionally. In an inner join there is no semantic difference between these types of filters.

If we change the query's selectivity a bit, we might get a very different join plan:

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                                        QUERY PLAN
-------------------------------------------------------------------------------------
 Hash Join  (cost=230.47..713.98 rows=101 width=488)
   Hash Cond: (t2.unique2 = t1.unique2)
   ->  Seq Scan on tenk2 t2  (cost=0.00..445.00 rows=10000 width=244)
   ->  Hash  (cost=229.20..229.20 rows=101 width=244)
         ->  Bitmap Heap Scan on tenk1 t1  (cost=5.07..229.20 rows=101 width=244)
               Recheck Cond: (unique1 < 100)
               ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101
width=0)
                     Index Cond: (unique1 < 100)
```

Here, the planner has chosen to use a hash join, in which rows of one table are entered into an in-memory hash table, after which the other table is scanned and the hash table is probed for matches to each row.

Again note how the indentation reflects the plan structure: the bitmap scan on tenk1 is the input to the Hash node, which constructs the hash table. That's then returned to the Hash Join node, which reads rows from its outer child plan and searches the hash table for each one.

Another possible type of join is a merge join, illustrated here:

```
EXPLAIN SELECT *
FROM tenk1 t1, onek t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                                      QUERY PLAN
------------------------------------------------------------------------------
 Merge Join  (cost=198.11..268.19 rows=10 width=488)
   Merge Cond: (t1.unique2 = t2.unique2)
   ->  Index Scan using tenk1_unique2 on tenk1 t1  (cost=0.29..656.28 rows=101
width=244)
         Filter: (unique1 < 100)
   ->  Sort  (cost=197.83..200.33 rows=1000 width=244)
         Sort Key: t2.unique2
         ->  Seq Scan on onek t2  (cost=0.00..148.00 rows=1000 width=244)
```

Merge join requires its input data to be sorted on the join keys. In this plan the tenk1 data is sorted by using an index scan to visit the rows in the correct order, but a sequential scan and sort is preferred for onek, because there are many more rows to be visited in that table. (Sequential-scan-and-sort frequently beats an index scan for sorting many rows, because of the nonsequential disk access required by the index scan.)

One way to look at variant plans is to force the planner to disregard whatever strategy it thought was the cheapest, using the enable/disable flags .For example, if we're unconvinced that sequential-scan-and-sort is the best way to deal with table onek in the previous example, we could try

```
SET enable_sort = off;

EXPLAIN SELECT *
FROM tenk1 t1, onek t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                                      QUERY PLAN
------------------------------------------------------------------------
-----------------------
 Merge Join  (cost=0.56..292.65 rows=10 width=488)
   Merge Cond: (t1.unique2 = t2.unique2)
   ->  Index Scan using tenk1_unique2 on tenk1 t1  (cost=0.29..656.28 rows=101
width=244)
         Filter: (unique1 < 100)
   ->  Index Scan using onek_unique2 on onek t2  (cost=0.28..224.79 rows=1000
width=244)
```

which shows that the planner thinks that sorting **onek** by index-scanning is about 12% more expensive than sequential-scan-and-sort. Of course, the next question is whether it's right about that. We can investigate that using **EXPLAIN ANALYZE**, as discussed below.

EXPLAIN ANALYZE

It is possible to check the accuracy of the planner's estimates by using **EXPLAIN's `ANALYZE** option. With this option, **EXPLAIN** actually executes the query, and then displays the true row counts and true run time accumulated within each plan node, along with the same estimates that a plain **EXPLAIN** shows. For example, we might get a result like this:

```
EXPLAIN ANALYZE SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t1.unique2 = t2.unique2;

                                                                  QUERY PLAN
-----------------------------------------------------------------------------
-------------------------------------------------------------
 Nested Loop  (cost=4.65..118.62 rows=10 width=488) (actual time=0.128..0.377 rows=10
loops=1)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.47 rows=10 width=244) (actual
time=0.057..0.121 rows=10 loops=1)
         Recheck Cond: (unique1 < 10)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36 rows=10 width=0)
(actual time=0.024..0.024 rows=10 loops=1)
               Index Cond: (unique1 < 10)
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.29..7.91 rows=1 width=244)
(actual time=0.021..0.022 rows=1 loops=10)
         Index Cond: (unique2 = t1.unique2)
 Planning time: 0.181 ms
 Execution time: 0.501 ms
```

> Note that the "actual time" values are in milliseconds of real time, whereas the **cost** estimates are expressed in arbitrary units; so they are unlikely to match up. The thing that's usually most important to look for is whether the estimated row counts are reasonably close to reality. In this example the estimates were all dead-on, but that's quite unusual in practice.

In some query plans, it is possible for a subplan node to be executed more than once. For example, the inner index scan will be executed once per outer row in the above nested-loop plan. In such cases, the **loops** value reports the total number of executions of the node, and the actual time and rows values shown are averages per-execution. This is done to make the numbers comparable with the way that the cost estimates are shown. Multiply by the **loops** value to get the total time actually spent in the node. In the above example, we spent a total of 0.220 milliseconds executing the index scans on **tenk2**.

In some cases **EXPLAIN ANALYZE** shows additional execution statistics beyond the plan node execution times and row counts. For example, Sort and Hash nodes provide extra information:

```
EXPLAIN ANALYZE SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2 ORDER BY t1.fivethous;

                                                                  QUERY PLAN
-----------------------------------------------------------------------------
-------------------------------------------------------------------------
 Sort  (cost=717.34..718.09 rows=101 width=488) (actual time=7.761..7.774 rows=100
loops=1)
   Sort Key: t1.fivethous
   Sort Method: quicksort  Memory: 77kB
   ->  Hash Join  (cost=230.47..713.98 rows=101 width=488) (actual time=0.711..7.427
rows=100 loops=1)
         Hash Cond: (t2.unique2 = t1.unique2)
         ->  Seq Scan on tenk2 t2  (cost=0.00..445.00 rows=10000 width=244) (actual
time=0.007..2.583 rows=10000 loops=1)
         ->  Hash  (cost=229.20..229.20 rows=101 width=244) (actual time=0.659..0.659
rows=100 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 28kB
               ->  Bitmap Heap Scan on tenk1 t1  (cost=5.07..229.20 rows=101
width=244) (actual time=0.080..0.526 rows=100 loops=1)
                     Recheck Cond: (unique1 < 100)
                     ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101
width=0) (actual time=0.049..0.049 rows=100 loops=1)
                           Index Cond: (unique1 < 100)
 Planning time: 0.194 ms
 Execution time: 8.008 ms
```

The Sort node shows the sort method used (in particular, whether the sort was in-memory or on-disk) and the amount of memory or disk space needed. The Hash node shows the number of hash buckets and batches as well as the peak amount of memory used for the hash table. (If the number of batches exceeds one, there will also be disk space usage involved, but that is not shown.)

Another type of extra information is the number of rows removed by a filter condition:

```
EXPLAIN ANALYZE SELECT * FROM tenk1 WHERE ten < 7;

                                                QUERY PLAN
-----------------------------------------------------------------------------
--------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=7000 width=244) (actual time=0.016..5.107
rows=7000 loops=1)
   Filter: (ten < 7)
   Rows Removed by Filter: 3000
 Planning time: 0.083 ms
```

```
 Execution time: 5.905 ms
```

These counts can be particularly valuable for filter conditions applied at join nodes. The "Rows Removed" line only appears when at least one scanned row, or potential join pair in the case of a join node, is rejected by the filter condition.

A case similar to filter conditions occurs with "lossy" index scans. For example, consider this search for polygons containing a specific point:

```
EXPLAIN ANALYZE SELECT * FROM polygon_tbl WHERE f1 @> polygon '(0.5,2.0)';

                                                QUERY PLAN
-----------------------------------------------------------------------
-----------------------------------
 Seq Scan on polygon_tbl  (cost=0.00..1.05 rows=1 width=32) (actual time=0.044..0.044
rows=0 loops=1)
   Filter: (f1 @> '((0.5,2))'::polygon)
   Rows Removed by Filter: 4
 Planning time: 0.040 ms
 Execution time: 0.083 ms
```

```
The planner thinks (quite correctly) that this sample table is too small to bother
with an index scan, so we have a plain sequential scan in which all the rows got
rejected by the filter condition. But if we force an index scan to be used, we see:
```

```
SET enable_seqscan TO off;

EXPLAIN ANALYZE SELECT * FROM polygon_tbl WHERE f1 @> polygon '(0.5,2.0)';

                                                      QUERY PLAN
-----------------------------------------------------------------------
-------------------------------------------------------
 Index Scan using gpolygonind on polygon_tbl  (cost=0.13..8.15 rows=1 width=32)
(actual time=0.062..0.062 rows=0 loops=1)
   Index Cond: (f1 @> '((0.5,2))'::polygon)
   Rows Removed by Index Recheck: 1
 Planning time: 0.034 ms
 Execution time: 0.144 ms
```

Here we can see that the index returned one candidate row, which was then rejected by a recheck of the index condition. This happens because a GiST index is "lossy" for polygon containment tests: it actually returns the rows with polygons that overlap the target, and then we have to do the exact containment test on those rows.

`EXPLAIN` has a `BUFFERS` option that can be used with `ANALYZE` to get even more run time statistics:

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000;

                                                                  QUERY PLAN
-----------------------------------------------------------------------------
-------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=25.08..60.21 rows=10 width=244) (actual
time=0.323..0.342 rows=10 loops=1)
   Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))
   Buffers: shared hit=15
   ->  BitmapAnd  (cost=25.08..25.08 rows=10 width=0) (actual time=0.309..0.309 rows=0
loops=1)
         Buffers: shared hit=7
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0)
(actual time=0.043..0.043 rows=100 loops=1)
               Index Cond: (unique1 < 100)
               Buffers: shared hit=2
         ->  Bitmap Index Scan on tenk1_unique2  (cost=0.00..19.78 rows=999 width=0)
(actual time=0.227..0.227 rows=999 loops=1)
               Index Cond: (unique2 > 9000)
               Buffers: shared hit=5
 Planning time: 0.088 ms
 Execution time: 0.423 ms
```

The numbers provided by `BUFFERS` help to identify which parts of the query are the most I/O-intensive.

Keep in mind that because `EXPLAIN ANALYZE` actually runs the query, any side-effects will happen as usual, even though whatever results the query might output are discarded in favor of printing the `EXPLAIN` data. If you want to analyze a data-modifying query without changing your tables, you can roll the command back afterwards, for example:

```
BEGIN;

EXPLAIN ANALYZE UPDATE tenk1 SET hundred = hundred + 1 WHERE unique1 < 100;

                                                                  QUERY PLAN
-----------------------------------------------------------------------------
-------------------------------------------------------------
 Update on tenk1  (cost=5.07..229.46 rows=101 width=250) (actual time=14.628..14.628
rows=0 loops=1)
   ->  Bitmap Heap Scan on tenk1  (cost=5.07..229.46 rows=101 width=250) (actual
time=0.101..0.439 rows=100 loops=1)
         Recheck Cond: (unique1 < 100)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0)
(actual time=0.043..0.043 rows=100 loops=1)
               Index Cond: (unique1 < 100)
```

```
 Planning time: 0.079 ms
 Execution time: 14.727 ms

ROLLBACK;
```

As seen in this example, when the query is an `INSERT`, `UPDATE`, or `DELETE` command, the actual work of applying the table changes is done by a top-level Insert, Update, or Delete plan node. The plan nodes underneath this node perform the work of locating the old rows and/or computing the new data. So above, we see the same sort of bitmap table scan we've seen already, and its output is fed to an Update node that stores the updated rows. It's worth noting that although the data-modifying node can take a considerable amount of run time (here, it's consuming the lion's share of the time), the planner does not currently add anything to the cost estimates to account for that work. That's because the work to be done is the same for every correct query plan, so it doesn't affect planning decisions.

When an `UPDATE` or `DELETE` command affects an inheritance hierarchy, the output might look like this:

```
EXPLAIN UPDATE parent SET f2 = f2 + 1 WHERE f1 = 101;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Update on parent  (cost=0.00..24.53 rows=4 width=14)
   Update on parent
   Update on child1
   Update on child2
   Update on child3
   ->  Seq Scan on parent  (cost=0.00..0.00 rows=1 width=14)
         Filter: (f1 = 101)
   ->  Index Scan using child1_f1_key on child1  (cost=0.15..8.17 rows=1 width=14)
         Index Cond: (f1 = 101)
   ->  Index Scan using child2_f1_key on child2  (cost=0.15..8.17 rows=1 width=14)
         Index Cond: (f1 = 101)
   ->  Index Scan using child3_f1_key on child3  (cost=0.15..8.17 rows=1 width=14)
         Index Cond: (f1 = 101)
```

In this example the Update node needs to consider three child tables as well as the originally-mentioned parent table. So there are four input scanning subplans, one per table. For clarity, the Update node is annotated to show the specific target tables that will be updated, in the same order as the corresponding subplans.

The `Planning time` shown by `EXPLAIN ANALYZE` is the time it took to generate the query plan from the parsed query and optimize it. It does not include parsing or rewriting.

The `Execution time` shown by `EXPLAIN ANALYZE` includes executor start-up and shut-down time, as well as the time to run any triggers that are fired, but it does not include parsing, rewriting, or planning time. Time spent executing `BEFORE` triggers, if any, is included in the time for the related Insert, Update, or Delete node; but time spent executing `AFTER` triggers is not counted there because `AFTER` triggers are fired after completion of the whole plan. The total time spent in each trigger (either `BEFORE` or `AFTER`) is also shown separately. Note that deferred constraint triggers will not be executed until end of transaction and are thus not considered at all by `EXPLAIN ANALYZE`.

There are two significant ways in which run times measured by **EXPLAIN ANALYZE** can deviate from normal execution of the same query. First, since no output rows are delivered to the client, network transmission costs and I/O conversion costs are not included. Second, the measurement overhead added by **EXPLAIN ANALYZE** can be significant, especially on machines with slow **gettimeofday()** operating-system calls. You can use the pg_test_timing tool to measure the overhead of timing on your system.

EXPLAIN results should not be extrapolated to situations much different from the one you are actually testing; for example, results on a toy-sized table cannot be assumed to apply to large tables. The planner's cost estimates are not linear and so it might choose a different plan for a larger or smaller table. An extreme example is that on a table that only occupies one disk page, you'll nearly always get a sequential scan plan whether indexes are available or not. The planner realizes that it's going to take one disk page read to process the table in any case, so there's no value in expending additional page reads to look at an index. (We saw this happening in the **polygon_tbl** example above.)

There are cases in which the actual and estimated values won't match up well, but nothing is really wrong. One such case occurs when plan node execution is stopped short by a **LIMIT** or similar effect. For example, in the **LIMIT** query we used before,

```
EXPLAIN ANALYZE SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000 LIMIT 2;

                                                                  QUERY PLAN
----------------------------------------------------------------------------
-------------------------------------------------------------
 Limit  (cost=0.29..14.71 rows=2 width=244) (actual time=0.177..0.249 rows=2 loops=1)
   ->  Index Scan using tenk1_unique2 on tenk1  (cost=0.29..72.42 rows=10 width=244)
(actual time=0.174..0.244 rows=2 loops=1)
         Index Cond: (unique2 > 9000)
         Filter: (unique1 < 100)
         Rows Removed by Filter: 287
 Planning time: 0.096 ms
 Execution time: 0.336 ms
```

the estimated cost and row count for the Index Scan node are shown as though it were run to completion. But in reality the Limit node stopped requesting rows after it got two, so the actual row count is only 2 and the run time is less than the cost estimate would suggest. This is not an estimation error, only a discrepancy in the way the estimates and true values are displayed.

Merge joins also have measurement artifacts that can confuse the unwary. A merge join will stop reading one input if it's exhausted the other input and the next key value in the one input is greater than the last key value of the other input; in such a case there can be no more matches and so no need to scan the rest of the first input. This results in not reading all of one child, with results like those mentioned for **LIMIT**. Also, if the outer (first) child contains rows with duplicate key values, the inner (second) child is backed up and rescanned for the portion of its rows matching that key value. **EXPLAIN ANALYZE** counts these repeated emissions of the same inner rows as if they were real additional rows. When there are many outer duplicates, the reported actual row count for the inner child plan node can be significantly larger than the number of rows that are actually in the inner relation.

BitmapAnd and BitmapOr nodes always report their actual row counts as zero, due to implementation limitations.

Normally, **EXPLAIN** will display every plan node created by the planner. However, there are cases where the executor can determine that certain nodes need not be executed because they cannot produce any rows, based on parameter values that were not available at planning time. (Currently this can only happen for child nodes of an Append or MergeAppend node that is scanning a partitioned table.) When this happens,

those plan nodes are omitted from the EXPLAIN output and a Subplans Removed: `N` annotation appears instead.

Statistics Used by the Planner

Single-Column Statistics

As we saw in the previous section, the query planner needs to estimate the number of rows retrieved by a query in order to make good choices of query plans. This section provides a quick look at the statistics that the system uses for these estimates.

One component of the statistics is the total number of entries in each table and index, as well as the number of disk blocks occupied by each table and index. This information is kept in the table pg_class, in the columns reltuples and relpages. We can look at it with queries similar to this one:

```
SELECT relname, relkind, reltuples, relpages
FROM pg_class
WHERE relname LIKE 'tenk1%';

       relname        | relkind | reltuples | relpages
----------------------+---------+-----------+----------
 tenk1                | r       |     10000 |      358
 tenk1_hundred        | i       |     10000 |       30
 tenk1_thous_tenthous | i       |     10000 |       30
 tenk1_unique1        | i       |     10000 |       30
 tenk1_unique2        | i       |     10000 |       30
(5 rows)
```

Here we can see that tenk1 contains 10000 rows, as do its indexes, but the indexes are (unsurprisingly) much smaller than the table.

For efficiency reasons, reltuples and relpages are not updated on-the-fly, and so they usually contain somewhat out-of-date values. They are updated by VACUUM, ANALYZE, and a few DDL commands such as CREATE INDEX. A VACUUM or ANALYZE operation that does not scan the entire table (which is commonly the case) will incrementally update the reltuples count on the basis of the part of the table it did scan, resulting in an approximate value. In any case, the planner will scale the values it finds in pg_class to match the current physical table size, thus obtaining a closer approximation.

Most queries retrieve only a fraction of the rows in a table, due to WHERE clauses that restrict the rows to be examined. The planner thus needs to make an estimate of the selectivity of WHERE clauses, that is, the fraction of rows that match each condition in the WHERE clause. The information used for this task is stored in the pg_statistic system catalog. Entries in pg_statistic are updated by the ANALYZE and VACUUM ANALYZE commands, and are always approximate even when freshly updated.

Rather than look at pg_statistic directly, it's better to look at its view pg_stats when examining the statistics manually. pg_stats is designed to be more easily readable. Furthermore, pg_stats is readable by all, whereas pg_statistic is only readable by a superuser. (This prevents unprivileged users from learning something about the contents of other people's tables from the statistics. The pg_stats view is restricted to show only rows about tables that the current user can read.) For example, we might do:

```
SELECT attname, inherited, n_distinct,
       array_to_string(most_common_vals, E'\n') as most_common_vals
FROM pg_stats
```

```
WHERE tablename = 'road';

 attname | inherited | n_distinct |          most_common_vals
---------+-----------+------------+-----------------------------------
 name    | f         |  -0.363388 | I- 580                          Ramp+
         |           |            | I- 880                          Ramp+
         |           |            | Sp Railroad                        +
         |           |            | I- 580                             +
         |           |            | I- 680                          Ramp
 name    | t         |  -0.284859 | I- 880                          Ramp+
         |           |            | I- 580                          Ramp+
         |           |            | I- 680                          Ramp+
         |           |            | I- 580                             +
         |           |            | State Hwy 13                    Ramp
(2 rows)
```

Note that two rows are displayed for the same column, one corresponding to the complete inheritance hierarchy starting at the `road` table (`inherited=t`), and another one including only the `road` table itself (`inherited=f`).

The amount of information stored in `pg_statistic` by `ANALYZE`, in particular the maximum number of entries in the `most_common_vals` and `histogram_bounds` arrays for each column, can be set on a column-by-column basis using the `ALTER TABLE SET STATISTICS` command, or globally by setting the default_statistics_target configuration variable. The default limit is presently 100 entries. Raising the limit might allow more accurate planner estimates to be made, particularly for columns with irregular data distributions, at the price of consuming more space in `pg_statistic` and slightly more time to compute the estimates. Conversely, a lower limit might be sufficient for columns with simple data distributions.

Further details about the planner’s use of statistics can be found in doc.

Extended Statistics

It is common to see slow queries running bad execution plans because multiple columns used in the query clauses are correlated. The planner normally assumes that multiple conditions are independent of each other, an assumption that does not hold when column values are correlated. Regular statistics, because of their per-individual-column nature, cannot capture any knowledge about cross-column correlation. However, IvorySQL has the ability to compute multivariate statistics, which can capture such information.

Because the number of possible column combinations is very large, it’s impractical to compute multivariate statistics automatically. Instead, extended statistics objects, more often called just statistics objects, can be created to instruct the server to obtain statistics across interesting sets of columns.

Statistics objects are created using the `CREATE STATISTICS` command. Creation of such an object merely creates a catalog entry expressing interest in the statistics. Actual data collection is performed by `ANALYZE` (either a manual command, or background auto-analyze). The collected values can be examined in the `pg_statistic_ext_data` catalog.

`ANALYZE` computes extended statistics based on the same sample of table rows that it takes for computing regular single-column statistics. Since the sample size is increased by increasing the statistics target for the table or any of its columns (as described in the previous section), a larger statistics target will normally result in more accurate extended statistics, as well as more time spent calculating them.

The following subsections describe the kinds of extended statistics that are currently supported.

Functional Dependencies

The simplest kind of extended statistics tracks functional dependencies, a concept used in definitions of database normal forms. We say that column `b` is functionally dependent on column `a` if knowledge of the value of `a` is sufficient to determine the value of `b`, that is there are no two rows having the same value of `a` but different values of `b`. In a fully normalized database, functional dependencies should exist only on primary keys and superkeys. However, in practice many data sets are not fully normalized for various reasons; intentional denormalization for performance reasons is a common example. Even in a fully normalized database, there may be partial correlation between some columns, which can be expressed as partial functional dependency.

The existence of functional dependencies directly affects the accuracy of estimates in certain queries. If a query contains conditions on both the independent and the dependent column(s), the conditions on the dependent columns do not further reduce the result size; but without knowledge of the functional dependency, the query planner will assume that the conditions are independent, resulting in underestimating the result size.

To inform the planner about functional dependencies, `ANALYZE` can collect measurements of cross-column dependency. Assessing the degree of dependency between all sets of columns would be prohibitively expensive, so data collection is limited to those groups of columns appearing together in a statistics object defined with the `dependencies` option. It is advisable to create `dependencies` statistics only for column groups that are strongly correlated, to avoid unnecessary overhead in both `ANALYZE` and later query planning.

Here is an example of collecting functional-dependency statistics:

```
CREATE STATISTICS stts (dependencies) ON city, zip FROM zipcodes;

ANALYZE zipcodes;

SELECT stxname, stxkeys, stxddependencies
  FROM pg_statistic_ext join pg_statistic_ext_data on (oid = stxoid)
  WHERE stxname = 'stts';
 stxname | stxkeys |             stxddependencies
---------+---------+------------------------------------------
 stts    | 1 5     | {"1 => 5": 1.000000, "5 => 1": 0.423130}
(1 row)
```

Here it can be seen that column 1 (zip code) fully determines column 5 (city) so the coefficient is 1.0, while city only determines zip code about 42% of the time, meaning that there are many cities (58%) that are represented by more than a single ZIP code.

When computing the selectivity for a query involving functionally dependent columns, the planner adjusts the per-condition selectivity estimates using the dependency coefficients so as not to produce an underestimate.

Limitations of Functional Dependencies

Functional dependencies are currently only applied when considering simple equality conditions that compare columns to constant values, and `IN` clauses with constant values. They are not used to improve estimates for equality conditions comparing two columns or comparing a column to an expression, nor for range clauses, `LIKE` or any other type of condition.

When estimating with functional dependencies, the planner assumes that conditions on the involved columns are compatible and hence redundant. If they are incompatible, the correct estimate would be zero rows, but that possibility is not considered. For example, given a query like

```
SELECT * FROM zipcodes WHERE city = 'San Francisco' AND zip = '94105';
```

the planner will disregard the `city` clause as not changing the selectivity, which is correct. However, it will make the same assumption about

```
SELECT * FROM zipcodes WHERE city = 'San Francisco' AND zip = '90210';
```

even though there will really be zero rows satisfying this query. Functional dependency statistics do not provide enough information to conclude that, however.

In many practical situations, this assumption is usually satisfied; for example, there might be a GUI in the application that only allows selecting compatible city and ZIP code values to use in a query. But if that's not the case, functional dependencies may not be a viable option.

Multivariate N-Distinct Counts

Single-column statistics store the number of distinct values in each column. Estimates of the number of distinct values when combining more than one column (for example, for `GROUP BY a, b`) are frequently wrong when the planner only has single-column statistical data, causing it to select bad plans.

To improve such estimates, `ANALYZE` can collect n-distinct statistics for groups of columns. As before, it's impractical to do this for every possible column grouping, so data is collected only for those groups of columns appearing together in a statistics object defined with the `ndistinct` option. Data will be collected for each possible combination of two or more columns from the set of listed columns.

Continuing the previous example, the n-distinct counts in a table of ZIP codes might look like the following:

```
CREATE STATISTICS stts2 (ndistinct) ON city, state, zip FROM zipcodes;

ANALYZE zipcodes;

SELECT stxkeys AS k, stxdndistinct AS nd
  FROM pg_statistic_ext join pg_statistic_ext_data on (oid = stxoid)
  WHERE stxname = 'stts2';
-[ RECORD 1 ]-------------------------------------------------------
k  | 1 2 5
nd | {"1, 2": 33178, "1, 5": 33178, "2, 5": 27435, "1, 2, 5": 33178}
(1 row)
```

This indicates that there are three combinations of columns that have 33178 distinct values: ZIP code and state; ZIP code and city; and ZIP code, city and state (the fact that they are all equal is expected given that ZIP code alone is unique in this table). On the other hand, the combination of city and state has only 27435 distinct values.

It's advisable to create `ndistinct` statistics objects only on combinations of columns that are actually used for grouping, and for which misestimation of the number of groups is resulting in bad plans. Otherwise, the `ANALYZE` cycles are just wasted.

Multivariate MCV Lists

Another type of statistic stored for each column are most-common value lists. This allows very accurate estimates for individual columns, but may result in significant misestimates for queries with conditions on

multiple columns.

To improve such estimates, ANALYZE can collect MCV lists on combinations of columns. Similarly to functional dependencies and n-distinct coefficients, it's impractical to do this for every possible column grouping. Even more so in this case, as the MCV list (unlike functional dependencies and n-distinct coefficients) does store the common column values. So data is collected only for those groups of columns appearing together in a statistics object defined with the mcv option.

Continuing the previous example, the MCV list for a table of ZIP codes might look like the following (unlike for simpler types of statistics, a function is required for inspection of MCV contents):

```
CREATE STATISTICS stts3 (mcv) ON city, state FROM zipcodes;

ANALYZE zipcodes;

SELECT m.* FROM pg_statistic_ext join pg_statistic_ext_data on (oid = stxoid),
                pg_mcv_list_items(stxdmcv) m WHERE stxname = 'stts3';

 index |         values         | nulls | frequency | base_frequency
-------+------------------------+-------+-----------+----------------
     0 | {Washington, DC}       | {f,f} |  0.003467 |        2.7e-05
     1 | {Apo, AE}              | {f,f} |  0.003067 |        1.9e-05
     2 | {Houston, TX}          | {f,f} |  0.002167 |       0.000133
     3 | {El Paso, TX}          | {f,f} |     0.002 |       0.000113
     4 | {New York, NY}         | {f,f} |  0.001967 |       0.000114
     5 | {Atlanta, GA}          | {f,f} |  0.001633 |        3.3e-05
     6 | {Sacramento, CA}       | {f,f} |  0.001433 |        7.8e-05
     7 | {Miami, FL}            | {f,f} |    0.0014 |          6e-05
     8 | {Dallas, TX}           | {f,f} |  0.001367 |        8.8e-05
     9 | {Chicago, IL}          | {f,f} |  0.001333 |        5.1e-05
   ...
(99 rows)
```

This indicates that the most common combination of city and state is Washington in DC, with actual frequency (in the sample) about 0.35%. The base frequency of the combination (as computed from the simple per-column frequencies) is only 0.0027%, resulting in two orders of magnitude under-estimates.

It's advisable to create MCV statistics objects only on combinations of columns that are actually used in conditions together, and for which misestimation of the number of groups is resulting in bad plans. Otherwise, the ANALYZE and planning cycles are just wasted.

Controlling the Planner with Explicit JOIN Clauses

It is possible to control the query planner to some extent by using the explicit JOIN syntax. To see why this matters, we first need some background.

In a simple join query, such as:

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
```

the planner is free to join the given tables in any order. For example, it could generate a query plan that joins A to B, using the `WHERE` condition `a.id = b.id`, and then joins C to this joined table, using the other `WHERE` condition. Or it could join B to C and then join A to that result. Or it could join A to C and then join them with B — but that would be inefficient, since the full Cartesian product of A and C would have to be formed, there being no applicable condition in the `WHERE` clause to allow optimization of the join. (All joins in the IvorySQL executor happen between two input tables, so it' s necessary to build up the result in one or another of these fashions.) The important point is that these different join possibilities give semantically equivalent results but might have hugely different execution costs. Therefore, the planner will explore all of them to try to find the most efficient query plan.

When a query only involves two or three tables, there aren' t many join orders to worry about. But the number of possible join orders grows exponentially as the number of tables expands. Beyond ten or so input tables it' s no longer practical to do an exhaustive search of all the possibilities, and even for six or seven tables planning might take an annoyingly long time. When there are too many input tables, the IvorySQL planner will switch from exhaustive search to a genetic probabilistic search through a limited number of possibilities. (The switch-over threshold is set by the geqo_threshold run-time parameter.) The genetic search takes less time, but it won' t necessarily find the best possible plan.

When the query involves outer joins, the planner has less freedom than it does for plain (inner) joins. For example, consider:

```
SELECT * FROM a LEFT JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

Although this query' s restrictions are superficially similar to the previous example, the semantics are different because a row must be emitted for each row of A that has no matching row in the join of B and C. Therefore the planner has no choice of join order here: it must join B to C and then join A to that result. Accordingly, this query takes less time to plan than the previous query. In other cases, the planner might be able to determine that more than one join order is safe. For example, given:

```
SELECT * FROM a LEFT JOIN b ON (a.bid = b.id) LEFT JOIN c ON (a.cid = c.id);
```

it is valid to join A to either B or C first. Currently, only `FULL JOIN` completely constrains the join order. Most practical cases involving `LEFT JOIN` or `RIGHT JOIN` can be rearranged to some extent.

Explicit inner join syntax (`INNER JOIN`, `CROSS JOIN`, or unadorned `JOIN`) is semantically the same as listing the input relations in `FROM`, so it does not constrain the join order.

Even though most kinds of `JOIN` don' t completely constrain the join order, it is possible to instruct the IvorySQL query planner to treat all `JOIN` clauses as constraining the join order anyway. For example, these three queries are logically equivalent:

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a CROSS JOIN b CROSS JOIN c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

But if we tell the planner to honor the `JOIN` order, the second and third take less time to plan than the first. This effect is not worth worrying about for only three tables, but it can be a lifesaver with many tables.

To force the planner to follow the join order laid out by explicit `JOIN`s, set the join_collapse_limit run-time parameter to 1. (Other possible values are discussed below.)

You do not need to constrain the join order completely in order to cut search time, because it' s OK to use `JOIN` operators within items of a plain `FROM` list. For example, consider:

```
SELECT * FROM a CROSS JOIN b, c, d, e WHERE ...;
```

With **join_collapse_limit** = 1, this forces the planner to join A to B before joining them to other tables, but doesn't constrain its choices otherwise. In this example, the number of possible join orders is reduced by a factor of 5.

Constraining the planner's search in this way is a useful technique both for reducing planning time and for directing the planner to a good query plan. If the planner chooses a bad join order by default, you can force it to choose a better order via **JOIN** syntax — assuming that you know of a better order, that is. Experimentation is recommended.

A closely related issue that affects planning time is collapsing of subqueries into their parent query. For example, consider:

```
SELECT *
FROM x, y,
    (SELECT * FROM a, b, c WHERE something) AS ss
WHERE somethingelse;
```

This situation might arise from use of a view that contains a join; the view's **SELECT** rule will be inserted in place of the view reference, yielding a query much like the above. Normally, the planner will try to collapse the subquery into the parent, yielding:

```
SELECT * FROM x, y, a, b, c WHERE something AND somethingelse;
```

This usually results in a better plan than planning the subquery separately. (For example, the outer **WHERE** conditions might be such that joining X to A first eliminates many rows of A, thus avoiding the need to form the full logical output of the subquery.) But at the same time, we have increased the planning time; here, we have a five-way join problem replacing two separate three-way join problems. Because of the exponential growth of the number of possibilities, this makes a big difference. The planner tries to avoid getting stuck in huge join search problems by not collapsing a subquery if more than **from_collapse_limit FROM** items would result in the parent query. You can trade off planning time against quality of plan by adjusting this run-time parameter up or down.

from_collapse_limit and join_collapse_limit are similarly named because they do almost the same thing: one controls when the planner will "flatten out" subqueries, and the other controls when it will flatten out explicit joins. Typically you would either set **join_collapse_limit** equal to **from_collapse_limit** (so that explicit joins and subqueries act similarly) or set **join_collapse_limit** to 1 (if you want to control join order with explicit joins). But you might set them differently if you are trying to fine-tune the trade-off between planning time and run time.

Populating a Database

One might need to insert a large amount of data when first populating a database. This section contains some suggestions on how to make this process as efficient as possible.

Disable Autocommit

When using multiple **INSERT`s, turn off autocommit and just do one commit at the end. (In plain SQL, this means issuing `BEGIN** at the start and **COMMIT** at the end. Some client libraries might do this behind your back, in which case you need to make sure the library does it when you want it done.) If you allow each insertion to be committed separately, IvorySQL is doing a lot of work for each row that is added. An additional benefit of doing all insertions in one transaction is that if the insertion of one row were to fail then the insertion of all rows inserted up to that point would be rolled back, so you won't be stuck with partially loaded data.

Use COPY

Use **COPY** to load all the rows in one command, instead of using a series of **INSERT** commands. The **COPY** command is optimized for loading large numbers of rows; it is less flexible than **INSERT**, but incurs significantly less overhead for large data loads. Since **COPY** is a single command, there is no need to disable autocommit if you use this method to populate a table.

If you cannot use **COPY**, it might help to use **PREPARE** to create a prepared **INSERT** statement, and then use **EXECUTE** as many times as required. This avoids some of the overhead of repeatedly parsing and planning **INSERT**. Different interfaces provide this facility in different ways; look for "prepared statements" in the interface documentation.

Note that loading a large number of rows using **COPY** is almost always faster than using **INSERT**, even if **PREPARE** is used and multiple insertions are batched into a single transaction.

COPY is fastest when used within the same transaction as an earlier **CREATE TABLE** or **TRUNCATE** command. In such cases no WAL needs to be written, because in case of an error, the files containing the newly loaded data will be removed anyway. However, this consideration only applies when wal_level is **minimal** as all commands must write WAL otherwise.

Remove Indexes

If you are loading a freshly created table, the fastest method is to create the table, bulk load the table's data using **COPY**, then create any indexes needed for the table. Creating an index on pre-existing data is quicker than updating it incrementally as each row is loaded.

If you are adding large amounts of data to an existing table, it might be a win to drop the indexes, load the table, and then recreate the indexes. Of course, the database performance for other users might suffer during the time the indexes are missing. One should also think twice before dropping a unique index, since the error checking afforded by the unique constraint will be lost while the index is missing.

Remove Foreign Key Constraints

Just as with indexes, a foreign key constraint can be checked "in bulk" more efficiently than row-by-row. So it might be useful to drop foreign key constraints, load data, and re-create the constraints. Again, there is a trade-off between data load speed and loss of error checking while the constraint is missing.

What's more, when you load data into a table with existing foreign key constraints, each new row requires an entry in the server's list of pending trigger events (since it is the firing of a trigger that checks the row's foreign key constraint). Loading many millions of rows can cause the trigger event queue to overflow available memory, leading to intolerable swapping or even outright failure of the command. Therefore it may be necessary, not just desirable, to drop and re-apply foreign keys when loading large amounts of data. If temporarily removing the constraint isn't accept.

Increase maintenance_work_mem

Temporarily increasing the maintenance_work_mem configuration variable when loading large amounts of data can lead to improved performance. This will help to speed up **CREATE INDEX** commands and **ALTER TABLE ADD FOREIGN KEY** commands. It won't do much for **COPY** itself, so this advice is only useful when you are using one or both of the above techniques.

Increase max_wal_size

Temporarily increasing the max_wal_size configuration variable can also make large data loads faster. This is because loading a large amount of data into IvorySQL will cause checkpoints to occur more often than the normal checkpoint frequency (specified by the **checkpoint_timeout** configuration variable). Whenever a checkpoint occurs, all dirty pages must be flushed to disk. By increasing **max_wal_size** temporarily during bulk data loads, the number of checkpoints that are required can be reduced.

Disable WAL Archival and Streaming Replication

When loading large amounts of data into an installation that uses WAL archiving or streaming replication, it might be faster to take a new base backup after the load has completed than to process a large amount of

incremental WAL data. To prevent incremental WAL logging while loading, disable archiving and streaming replication, by setting wal_level to `minimal`, archive_mode to `off`, and max_wal_senders to zero. But note that changing these settings requires a server restart, and makes any base backups taken before unavailable for archive recovery and standby server, which may lead to data loss.

Aside from avoiding the time for the archiver or WAL sender to process the WAL data, doing this will actually make certain commands faster, because they do not to write WAL at all if `wal_level` is `minimal` and the current subtransaction (or top-level transaction) created or truncated the table or index they change. (They can guarantee crash safety more cheaply by doing an `fsync` at the end than by writing WAL.)

Run ANALYZE Afterwards

Whenever you have significantly altered the distribution of data within a table, running `ANALYZE` is strongly recommended. This includes bulk loading large amounts of data into the table. Running `ANALYZE` (or `VACUUM ANALYZE`) ensures that the planner has up-to-date statistics about the table. With no statistics or obsolete statistics, the planner might make poor decisions during query planning, leading to poor performance on any tables with inaccurate or nonexistent statistics. Note that if the autovacuum daemon is enabled, it might run `ANALYZE` automatically.

Some Notes about pg_dump

Dump scripts generated by pg_dump automatically apply several, but not all, of the above guidelines. To restore a pg_dump dump as quickly as possible, you need to do a few extra things manually. (Note that these points apply while restoring a dump, not while creating it. The same points apply whether loading a text dump with psql or using pg_restore to load from a pg_dump archive file.)

By default, pg_dump uses `COPY`, and when it is generating a complete schema-and-data dump, it is careful to load data before creating indexes and foreign keys. So in this case several guidelines are handled automatically. What is left for you to do is to:

- Set appropriate (i.e., larger than normal) values for `maintenance_work_mem` and `max_wal_size`.
- If using WAL archiving or streaming replication, consider disabling them during the restore. To do that, set `archive_mode` to `off`, `wal_level` to `minimal`, and `max_wal_senders` to zero before loading the dump. Afterwards, set them back to the right values and take a fresh base backup.
- Experiment with the parallel dump and restore modes of both pg_dump and pg_restore and find the optimal number of concurrent jobs to use. Dumping and restoring in parallel by means of the `-j` option should give you a significantly higher performance over the serial mode.
- Consider whether the whole dump should be restored as a single transaction. To do that, pass the `-1` or `--single-transaction` command-line option to psql or pg_restore. When using this mode, even the smallest of errors will rollback the entire restore, possibly discarding many hours of processing. Depending on how interrelated the data is, that might seem preferable to manual cleanup, or not. `COPY` commands will run fastest if you use a single transaction and have WAL archiving turned off.
- If multiple CPUs are available in the database server, consider using pg_restore's `--jobs` option. This allows concurrent data loading and index creation.
- Run `ANALYZE` afterwards.

Non-Durable Settings

Durability is a database feature that guarantees the recording of committed transactions even if the server crashes or loses power. However, durability adds significant database overhead, so if your site does not require such a guarantee, IvorySQL can be configured to run much faster. The following are configuration changes you can make to improve performance in such cases. Except as noted below, durability is still guaranteed in case of a crash of the database software; only an abrupt operating system crash creates a risk of data loss or corruption when these settings are used.

- Place the database cluster's data directory in a memory-backed file system (i.e., RAM disk). This eliminates all database disk I/O, but limits data storage to the amount of available memory (and perhaps swap).
- Turn off fsync; there is no need to flush data to disk.

- Turn off synchronous_commit; there might be no need to force WAL writes to disk on every commit. This setting does risk transaction loss (though not data corruption) in case of a crash of the database.
- Turn off full_page_writes; there is no need to guard against partial page writes.
- Increase max_wal_size and checkpoint_timeout; this reduces the frequency of checkpoints, but increases the storage requirements of `/pg_wal`.
- Create unlogged tables to avoid WAL writes, though it makes the tables non-crash-safe.

Data Migration

.1. Migration

Migration overview

Database migration refers to the process of transferring data from one database to another, and the databases at both ends may be PostgreSql, IvoySQL, MySQL, Oracle, SQL Server, etc. The migration process is a challenging, complex process that requires a thorough understanding of how databases work and their characteristics. If the application has been deployed to the production environment and is in a normal operating state, a smooth application migration is required after database migration to maintain uninterrupted business operation and no data loss.

After migration, databases and systems should meet the following requirements:

- The migrated database system should fully host the data of the original database system. Avoid data loss during migration that causes incomplete data to the new database system.
- The migrated database system should fully adapt to the functions of the original database. Avoid the inability to run or throw errors of the entire business system due to data types, syntax, and functions that are not supported after migration, and there is no alternative.
- The migrated database should be adapted to the upstream and downstream of the entire business system to ensure the stable and reliable operation of the entire business system.
- The comprehensive performance of the migrated database cannot be weaker than that of the original database, providing performance guarantee for the entire business system.

Migration tool——Ora2Pg

Ora2Pg is a free tool for migrating Oracle databases to an IvorySQL-compatible schema. It connects to your Oracle database, automatically scans and extracts its structure or data, and then generates SQL scripts that can be loaded into an IvorySQL database. Ora2Pg can migrate from a reverse-engineered Oracle database to a large enterprise database, or simply copy some Oracle data into an IvorySQL database. It is very easy to use and does not require any Oracle database knowledge without providing the parameters required to connect to Oracle Database.

Ora2Pg consists of a Perl script (ora2pg) and a Perl module(Ora2Pg.pm), the only thing that needs to be done is to modify its configuration file, ora2pg.conf, set the DSN to connect to the Oracle database, and an optional SCHEMA name. ONCE THAT’ S DONE, YOU ONLY NEED TO SET THE EXPORTED TYPE: TABLE (INCLUDING CONSTRAINTS AND INDEXES), VIEW, MVIEW, TABLESPACE, SEQUENCE, INDEXES, TRIGGER, GRANT, FUNCTION, PROCEDURE, PACKAGE, PARTITION, TYPE, INSERT OR COPY, FDW, QUERY, KETTLE, AND SYNONYM.

By default, Ora2Pg exports an SQL file, which can be executed through the IvorySQL client tool psql. When performing data migration, you can set the DSN of the target database in the configuration file to import data directly from Oracle into the IvorySQL database.

Object	Whether ora2pg is supported
view	yes
trigger	yes，In some cases, you need to modify the script manually
sequence	yes
function	yes
procedure	yes，In some cases, you need to modify the script manually
type	yes，In some cases, you need to modify the script manually

materialized view	yes，In some cases, you need to modify the script manually

Migrate Oracle Database to IvorySQL

Environment preparation

linux	Oracle Version	IvorySQL Version
Centos Stream 9	19.0.0.0	4.2

Environment-dependent installation

Install Perl

```
# dnf install -y perl perl-ExtUtils-CBuilder perl-ExtUtils-MakeMaker
# perl -v


This is perl 5, version 16, subversion 3 (v5.16.3) built for x86_64-linux-thread-multi
(with 44 registered patches, see perl -V for more detail)


Copyright 1987-2012, Larry Wall


Perl may be copied only under the terms of either the Artistic License or the
GNU General Public License, which may be found in the Perl 5 source kit.


Complete documentation for Perl, including FAQ lists, should be found on
this system using "man perl" or "perldoc perl".  If you have access to the
Internet, point your browser at http://www.perl.org/, the Perl Home Page.
```

Install the DBI module

DBI，Database Independent Interface，is the interface of the Perl language to connect to the database

Download address: https://cpan.metacpan.org/authors/id/T/TI/TIMB/DBI-1.643.tar.gz

```
# tar zxvf DBI-1.643.tar.gz
# cd DBI-1.643/
# perl Makefile.PL
# make
# make install
```

Install DBD-Oracle

Download address：https://sourceforge.net/projects/ora2pg/

Set environment variables; Load environment variables; Because ORACLE must be defined_ HOME environment variable; This example configures environment variables under the ivorysql user

```
export ORACLE_HOME=/usr/lib/oracle/18.3/client64
# tar -zxvf DBD-Oracle-1.76.tar.gz
# source /home/postgres/.bashrc
# cd DBD-Oracle-1.76
# perl Makefile.PL
# make
# make install
```

Install DBD-PG (optional)

Download address：https://metacpan.org/release/DBD-Pg/

Set environment variables：

```
export POSTGRES_HOME=/opt/ivorysql/3.2
# tar -zxvf DBD-Pg-3.80.tar.gz
# source /home/ivorysql/.bashrc
# cd DBD-Pg-3.8.0
# perl Makefile.PL
# make
# make install
```

Install Ora2pg

Download address：https://sourceforge.net/projects/ora2pg/

```
# tar -xjf  ora2pg-24.0.tar.bz2
# cd ora2pg-xx/
# perl Makefile.PL  PREFIX=<your_install_dir>
# make && make install
```

Installed in/usr/local/bin/directory by default Check the software environment:

```
# vi check.pl
#!/usr/bin/perl
use strict;
use ExtUtils::Installed;
my $inst= ExtUtils::Installed->new();
my @modules = $inst->modules();
```

```
foreach(@modules)
{
        my $ver = $inst->version($_) || "???";
        printf("%-12s --  %s\n", $_, $ver);

}
exit;
# perl check.pl
DBD::Oracle  --  1.76
DBD::Pg      --  3.8.0
DBI          --  1.643
Ora2Pg       --  24.0
Perl         --  5.16.3
```

Set environment variables

```
export PERL5LIB=<your_install_dir>
#export PERL5LIB=/usr/local/bin/
```

Source side preparation

Update oracle statistics to improve performance

```
BEGIN
DBMS_STATS.GATHER_SCHEMA_STATS('SH');
DBMS_STATS.GATHER_SCHEMA_STATS('SCOTT');
DBMS_STATS.GATHER_SCHEMA_STATS('HR');
DBMS_STATS.GATHER_DATABASE_STATS ;
DBMS_STATS.GATHER_DICTIONARY_STATS;
END;/
```

Query the source end object pair type

```
SYS@PROD1>set pagesize 200
SYS@PROD1>select distinct OBJECT_TYPE from  dba_objects where OWNER in
('SH','SCOTT','HR');
OBJECT_TYPE
-------------------
INDEX PARTITION
TABLE PARTITION
SEQUENCE
PROCEDURE
LOB                             X
```

```
TRIGGER
DIMENSION                      X
MATERIALIZED VIEW
TABLE
INDEX
VIEW
11 rows selected.
```

Ora2pg export table structure

Configure ora2pg.conf

By default, Ora2Pg will find the/etc/ora2pg/ora2pg.conf configuration file. If the file exists, you only need to execute:/usr/local/bin/ora2pg

```
cat /etc/ora2pg/ora2pg.conf.dist  | grep -v ^# |grep -v ^$ >ora2pg.conf
vi ora2pg.conf
# cat ora2pg.conf
ORACLE_HOME     /opt/oracle/product/19c/dbhome_1
ORACLE_DSN      dbi:Oracle:host=localhost;sid=ORCLCDB;port=1521
ORACLE_USER     system
ORACLE_PWD      oracle
SCHEMA          SH
EXPORT_SCHEMA  1
SKIP  fkeys ukeys checks
TYPE
TABLE,VIEW,GRANT,SEQUENCE,TABLESPACE,PROCEDURE,TRIGGER,FUNCTION,PACKAGE,PARTITION,TYPE
,MVIEW,QUERY,DBLINK,SYNONYM,DIRECTORY,TEST,TEST_VIEW
NLS_LANG    AMERICAN_AMERICA.UTF8
OUTPUT     sh.sql
```

1. Only one type of export can be executed at the same time, so the TYPE instruction must be unique. If you have more than one, only the last one will be found in the file. But I can export multiple types at the same time.
2. Please note that you can link multiple exports by providing a comma-separated list of export types to the TYPE directive, but in this case, you cannot use COPY or INSERT with other export types.
3. Some export types cannot or should not be directly loaded into the IvorySQL database, and still require little manual editing. This is the case for GRANT, TABLESPACE, TRIGGER, FUNCTION, PROCEDURE, TYPE, QUERY and PACKAGE export types, especially if you have PLSQL code or Oracle specific SQL.
4. For TABLESPACE, you must ensure that the file path exists on the system. For SYNONYM, you can ensure that the owner and schema of the object correspond to the new PostgreSQL database design.
5. It is recommended to export the table structure one type at a time to avoid

> other errors affecting each other.

Test connection

After setting the Oracle database DSN, you can execute ora2pg to check whether it is valid:

```
#  ora2pg -t SHOW_VERSION -c config/ora2pg.conf

Oracle Database 19c Enterprise Edition Release 19.0.0.0.0
```

Migration cost assessment

It is not easy to estimate the cost of the migration process from Oracle to PostgreSQL. In order to obtain a good evaluation of the migration cost, Ora2Pg will check all database objects, all functions and stored procedures to detect whether there are still some objects and PL/SQL code that cannot be automatically converted by Ora2Pg. Ora2Pg has a content analysis mode, which checks the Oracle database to generate a text report about the content contained in the Oracle database and the content that cannot be exported.

```
# ora2pg -t SHOW_REPORT --estimate_cost  -c ora2pg.conf
[========================>] 11/11 tables (100.0%) end of scanning.
[========================>] 11/11 objects types (100.0%) end of objects auditing.
-------------------------------------------------------------------------------
Ora2Pg v24.0 - Database Migration Report
-------------------------------------------------------------------------------
Version Oracle Database 19c Enterprise Edition Release 19.0.0.0.0
Schema  SH
Size    287.25 MB
-------------------------------------------------------------------------------
Object  Number  Invalid Estimated cost  Comments        Details
-------------------------------------------------------------------------------
DATABASE LINK   0       0       0       Database links will be exported as SQL/MED
IvorySQL's Foreign Data Wrapper (FDW) extensions using oracle_fdw.
DIMENSION       5       0       0
GLOBAL TEMPORARY TABLE  0       0       0       Global temporary table are not
supported by PostgreSQL and will not be exported. You will have to rewrite some
application code to match the PostgreSQL temporary table behavior.
INDEX   20      0       3.4     14 index(es) are concerned by the export, others are
automatically generated and will do so on PostgreSQL. Bitmap will be exported as
btree_gin index(es) and hash index(es) will be exported as b-tree index(es) if any.
Domain index are exported as b-tree but commented to be edited to mainly use FTS.
Cluster, bitmap join and IOT indexes will not be exported at all. Reverse indexes are
not exported too, you may use a trigram-based index (see pg_trgm) or a reverse()
function based index and search. Use 'varchar_pattern_ops', 'text_pattern_ops' or
'bpchar_pattern_ops' operators in your indexes to improve search with the LIKE
operator respectively into varchar, text or char columns.      11 bitmap index(es). 1
domain index(es). 2 b-tree index(es).

INDEX PARTITION 196     0       0       Only local indexes partition are exported,
```

```
they are build on the column used for the partitioning.

JOB     0       0       0       Job are not exported. You may set external cron job
with them.

MATERIALIZED VIEW       2       0       6       All materialized view will be exported
as snapshot materialized views, they are only updated when fully refreshed.

SYNONYM 0       0       0       SYNONYMs will be exported as views. SYNONYMs do not
exists with PostgreSQL but a common workaround is to use views or set the PostgreSQL
search_path in your session to access object outside the current schema.

TABLE   11      0       1.1     1 external table(s) will be exported as standard
table. See EXTERNAL_TO_FDW configuration directive to export as file_fdw foreign
tables or use COPY in your code if you just want to load data from external files.
Total number of rows: 1063384. Top 10 of tables sorted by number of rows:. sales has
918843 rows. costs has 82112 rows. customers has 55500 rows.
supplementary_demographics has 4500 rows. times has 1826 rows. promotions has 503
rows. products has 72 rows. countries has 23 rows. channels has 5 rows.
sales_transactions_ext has 0 rows. Top 10 of largest tables:.

TABLE PARTITION 56      0       5.6     Partitions are exported using table
inheritance and check constraint. Hash and Key partitions are not supported by
PostgreSQL and will not be exported.  56 RANGE partitions..

VIEW    1       0       1       Views are fully supported but can use specific
functions.


-------------------------------------------------------------------------------

Total   291     0       17.10   17.10 cost migration units means approximatively 1
man-day(s). The migration unit was set to 5 minute(s)
-------------------------------------------------------------------------------
Migration level : A-1
-------------------------------------------------------------------------------
Migration levels:

    A - Migration that might be run automatically

    B - Migration with code rewrite and a human-days cost up to 5 days

    C - Migration with code rewrite and a human-days cost above 5 days

Technical levels:

    1 = trivial: no stored functions and no triggers
```

```
    2 = easy: no stored functions but with triggers, no manual rewriting

    3 = simple: stored functions and/or triggers, no manual rewriting

    4 = manual: no stored functions but with triggers or views with code rewriting

    5 = difficult: stored functions and/or triggers with code rewriting

---------------------------------------------------------------------------
```

Export SH table structure

```
# ora2pg  -c ora2pg.conf
[=========================>] 11/11 tables (100.0%) end of scanning.

[=========================>] 12/12 tables (100.0%) end of table export.

[=========================>] 1/1 views (100.0%) end of output.

[=========================>] 0/0 sequences (100.0%) end of output.

[=========================>] 0/0 procedures (100.0%) end of procedures export.

[=========================>] 0/0 triggers (100.0%) end of output.

[=========================>] 0/0 functions (100.0%) end of functions export.

[=========================>] 0/0 packages (100.0%) end of output.

[=========================>] 56/56 partitions (100.0%) end of output.

[=========================>] 0/0 types (100.0%) end of output.

[=========================>] 2/2 materialized views (100.0%) end of output.
[=========================>] 0/0 dblink (100.0%) end of output.

[=========================>] 0/0 synonyms (100.0%) end of output.

[=========================>] 2/2 directory (100.0%) end of output.

Fixing function calls in output files....
```

Export SH user data

Configure the type of ora2pg.conf as COPY or INSERT

```
# cp ora2pg.conf sh_data.conf

# vi sh_data.conf

ORACLE_HOME     /usr/lib/oracle/18.3/client64

ORACLE_DSN      dbi:Oracle:host=10.85.10.6 ;sid=PROD1;port=1521

ORACLE_USER     system

ORACLE_PWD      oracle

SCHEMA          SH

EXPORT_SCHEMA  1

DISABLE_UNLOGGED  1

SKIP  fkeys ukeys checks

TYPE           COPY

NLS_LANG    AMERICAN_AMERICA.UTF8

OUTPUT     sh_data.sql
```

Export Data

```
# ora2pg  -c sh_data.conf

[=========================>] 11/11 tables (100.0%) end of scanning.

[=========================>] 5/5 rows (100.0%) Table CHANNELS (5 recs/sec)

[>                        ]       5/1063384 total rows (0.0%) - (0 sec., avg: 5
recs/sec).

[>                        ]     0/82112 rows (0.0%) Table COSTS_1995 (0 recs/sec)

[>                        ]       5/1063384 total rows (0.0%) - (0 sec., avg: 5
```

```
recs/sec).

[>                        ]     0/82112 rows (0.0%) Table COSTS_H1_1997 (0 recs/sec)

[>                        ]       5/1063384 total rows (0.0%) - (0 sec., avg: 5
recs/sec).

[>                        ]     0/82112 rows (0.0%) Table COSTS_1996 (0 recs/sec)

[>                        ]       5/1063384 total rows (0.0%) - (0 sec., avg: 5
recs/sec).

.............................................................

[=========================>] 4500/4500 rows (100.0%) Table SUPPLEMENTARY_DEMOGRAPHICS
(4500 recs/sec)

[========================> ] 1061558/1063384 total rows (99.8%) - (45 sec., avg: 23590
recs/sec).

[=========================>] 1826/1826 rows (100.0%) Table TIMES (1826 recs/sec)

[=========================>] 1063384/1063384 total rows (100.0%) - (45 sec., avg: 23630
recs/sec).

[=========================>] 1063384/1063384 rows (100.0%) on total estimated data (45
sec., avg: 23630 recs/sec)

Fixing function calls in output files...
```

To view the exported file:

```
# ls -lrt *.sql

-rw-r--r-- 1 root root 15716 Jul  2 21:21 TABLE_sh.sql

-rw-r--r-- 1 root root   858 Jul  2 21:21 VIEW_sh.sql

-rw-r--r-- 1 root root  2026 Jul  2 21:21 TABLESPACE_sh.sql

-rw-r--r-- 1 root root   345 Jul  2 21:21 SEQUENCE_sh.sql

-rw-r--r-- 1 root root  2382 Jul  2 21:21 GRANT_sh.sql
```

```
-rw-r--r-- 1 root root   344 Jul  2 21:21 TRIGGER_sh.sql

-rw-r--r-- 1 root root   346 Jul  2 21:21 PROCEDURE_sh.sql

-rw-r--r-- 1 root root   344 Jul  2 21:21 PACKAGE_sh.sql

-rw-r--r-- 1 root root   345 Jul  2 21:21 FUNCTION_sh.sql

-rw-r--r-- 1 root root  6771 Jul  2 21:21 PARTITION_sh.sql

-rw-r--r-- 1 root root   341 Jul  2 21:21 TYPE_sh.sql

-rw-r--r-- 1 root root   342 Jul  2 21:21 QUERY_sh.sql

-rw-r--r-- 1 root root   950 Jul  2 21:21 MVIEW_sh.sql

-rw-r--r-- 1 root root   344 Jul  2 21:21 SYNONYM_sh.sql

-rw-r--r-- 1 root root   926 Jul  2 21:21 DIRECTORY_sh.sql

-rw-r--r-- 1 root root   343 Jul  2 21:21 DBLINK_sh.sql

-rw-r--r-- 1 root root 55281235 Jul  2 17:11 sh_data.sql
```

Export HR and SCOTT user data in the same way.

Create orcl library in IvorySQL environment

Create ORCL database

```
# su - ivorysql

Last login: Tue Jul  2 20:04:30 CST 2019 on pts/3

$ createdb orcl

$ psql

psql (18.4)

Type "help" for help.
```

```
ivorysql=# \l

                                                 List of databases
   Name    |  Owner   | Encoding  | Collate | Ctype | ICU Locale | Locale Provider |
Access privileges
-----------+----------+-----------+---------+-------+------------+-----------------+
-----------------------
 ivorysql  | ivorysql | SQL_ASCII | C       | C     |            | libc            |
 orcl      | ivorysql | SQL_ASCII | C       | C     |            | libc            |
 postgres  | ivorysql | SQL_ASCII | C       | C     |            | libc            |
 template0 | ivorysql | SQL_ASCII | C       | C     |            | libc            |
=c/ivorysql          +
           |          |           |         |       |            |                 |
ivorysql=CTc/ivorysql
 template1 | ivorysql | SQL_ASCII | C       | C     |            | libc            |
=c/ivorysql          +
           |          |           |         |       |            |                 |
ivorysql=CTc/ivorysql

(5 rows)

ivorysql=#
```

Create SH, HR, SCOTT users:

```
$ psql orcl

psql (18.4)

Type "help" for help.

orcl=#

orcl=# create user sh with password 'sh';

CREATE ROLE
```

Migration Portal

Import table structure

Because of the materialized view, in TABLE_ The sh.sql contains the index of the materialized view, which will fail to create. You need to first create a table, then create a materialized view, and finally create an index. Cancel the materialized view index and create it separately later:

```
CREATE INDEX fw_psc_s_mv_chan_bix ON fweek_pscat_sales_mv (channel_id);

CREATE INDEX fw_psc_s_mv_promo_bix ON fweek_pscat_sales_mv (promo_id);

CREATE INDEX fw_psc_s_mv_subcat_bix ON fweek_pscat_sales_mv (prod_subcategory);

CREATE INDEX fw_psc_s_mv_wd_bix ON fweek_pscat_sales_mv (week_ending_day);

CREATE TEXT SEARCH CONFIGURATION en (COPY = pg_catalog.english);
ALTER TEXT SEARCH CONFIGURATION en ALTER MAPPING FOR hword, hword_part, word WITH
unaccent, english_stem;
```

```
psql orcl  -f  tab.sql.sql

ALTER TABLE PARTITION sh.sales OWNER TO sh;
COMMENT
COMMENT
COMMENT
COMMENT
COMMENT
COMMENT
COMMENT
ALTER TABLE
ALTER TABLE
ALTER TABLE
.................................
```

Authorize objects

```
cat psql orcl  -f  GRANT_sh.sql
CREATE USER SH WITH PASSWORD 'change_my_secret' LOGIN;
ALTER TABLE sh.fweek_pscat_sales_mv OWNER TO sh;
GRANT ALL ON  sh.fweek_pscat_sales_mv TO sh;
```

Import materialized view structure

Materialized views require relevant query permissions, so import permissions. Please keep up with users here

```
$  psql orcl sh -f  MVIEW_sh.sql
SELECT 0
SELECT 0
```

```
CREATE INDEX
CREATE INDEX
CREATE INDEX
CREATE INDEX
```

Import View

```
$  psql orcl  -f  VIEW_sh.sql
SET
SET
SET
CREATE VIEW
```

Import partition table

```
$  psql orcl  -f  PARTITION_sh.sql
SET
SET
SET
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
.........................
```

Import data

```
$  psql orcl   -f   sh_data.sql
SET
COPY 0
SET
COPY 0
SET
COPY 0
SET
COPY 0
SET
COPY 0
SET
COPY 0
```

```
SET
COPY 0
SET
COPY 4500
SET
COPY 1826
COMMIT
```

Data validation

Source database and target side extract part of objects for comparison：

```
SYS@PROD1>select count(*) from sh.products;
  COUNT(*)
----------
        72

orcl=#  select count(*) from sh.products;
 count
-------
    72
(1 row)
-------------------------------------------------------------------------------

SYS@PROD1>select count(*) from sh.channels;

  COUNT(*)

----------

         5

orcl=#   select count(*) from sh.channels;
 count
-------

     5

(1 row)

-------------------------------------------------------------------------

SYS@PROD1>select count(*) from sh.customers ;
```

```
  COUNT(*)
----------

     55500
orcl=# select count(*) from sh.customers ;
 count
-------
 55500
(1 row)
```

Generate migration template

When using, there are two options — project_ Base and — init_ Project indicates to ora2pg that he must create a project template, which contains the work tree, configuration files and scripts for exporting all objects from the Oracle database. Generate a generic configuration file. 1. Create script export_ Schema.sh to automatically perform all exports. 2. Create script import_ All.sh to automatically perform all imports. example：

```
mkdir -p  /ora2pg/migration

# ora2pg --project_base /ora2pg/migration/ --init_project test_project
Creating project test_project.
/ora2pg/migration//test_project/
        schema/
                dblinks/
                directories/
                functions/
                grants/
                mviews/
                packages/
                partitions/
                procedures/
                sequences/
                synonyms/
                tables/
                tablespaces/
                triggers/
                types/
                views/
        sources/
                functions/
                mviews/
                packages/
```

```
                partitions/
                procedures/
                triggers/
                types/
                views/
        data/
        config/
        reports/
Generating generic configuration file
Creating script export_schema.sh to automate all exports.
Creating script import_all.sh to automate all imports.
```

IvorySQL Developers

.1. Developer

Overview

IvorySQL provides unique additional functionality on top of the open source PostgreSQL database.

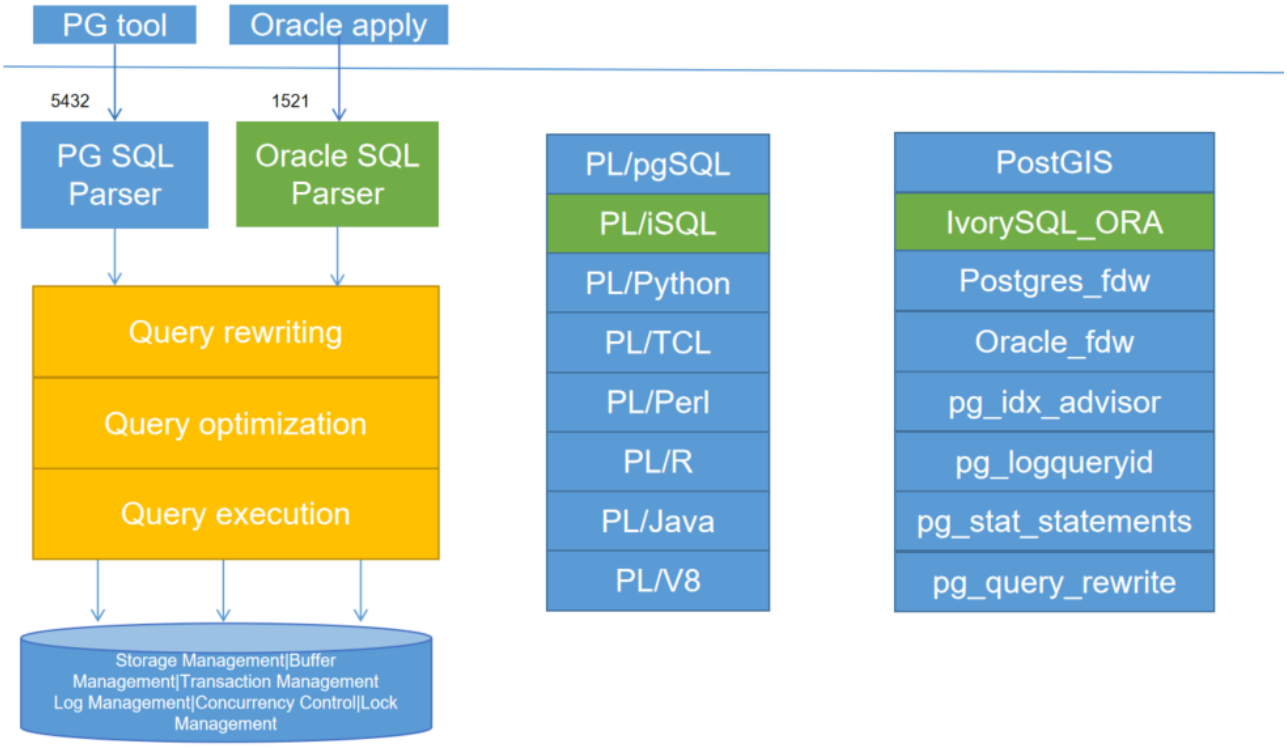
IvorySQL is committed to delivering value to its end-users through innovation and building on top of open source based database solutions.

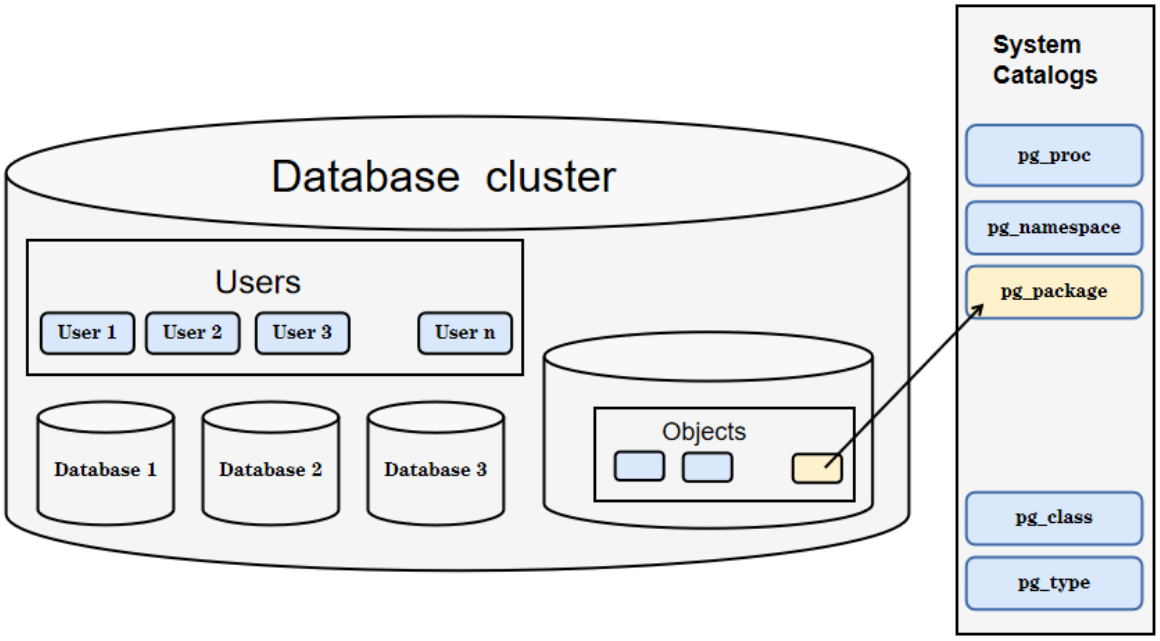
Our goal is to deliver a solution with high performance，scalability，reliability，and ease of use for small medium and large-scale enterprises.

The extended functionality provided by IvorySQL will enable users to build highly performant and scalable PostgreSQL database clusters with better database compatibility and administration.This simplifies the process of migration to PostgreSQLfrom other DBMS with enhanced database administration experiences.

Architecture Overview

The IvorySQL follows the same general architecture of PostgreSQL with some additions,but it does not deviate from its core philosophy.Thediagram below depicts essentially how IvorySQL operates.



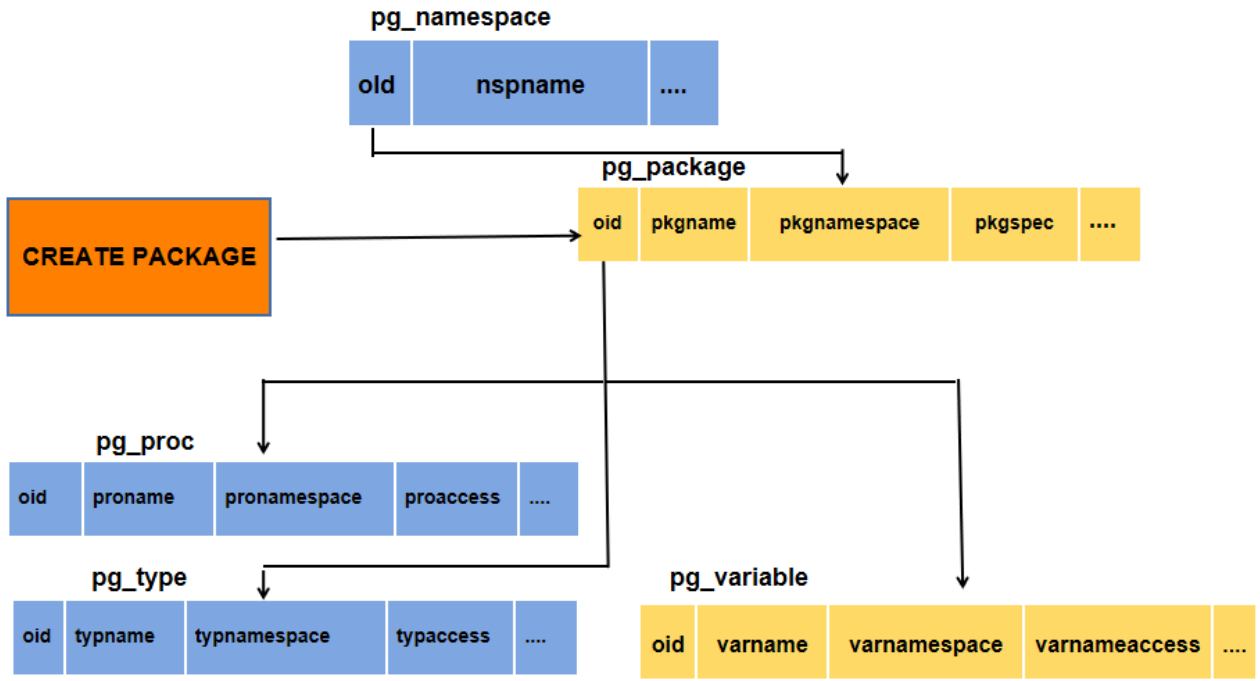


The yellow color in the diagram shows the new modules added by IvorySQL on top of existing PostgreSQL while IvorySQL has also made changes to existing modules and logical structures as well.

The most noteworthy of those modules that received updates for supporting oracle compatibility are backend parser and system catalogs.

Catalog changes

The following diagram depicts the changes made to PostgreSQL’ s existing directories and the additions that have been made.



Database Modeling

Creating a Database

The first test to see whether you can access the database server is to try to create a database. A running IvorySQL server can manage many databases. Typically, a separate database is used for each project or for each user.

Possibly, your site administrator has already created a database for your use. In that case you can omit this step and skip ahead to the next section.

To create a new database, in this example named **mydb**, you use the following command:

```
$ createdb mydb
```

If this produces no response then this step was successful and you can skip over the remainder of this section.

If you see a message similar to:

```
createdb: command not found
```

then IvorySQL was not installed properly. Either it was not installed at all or your shell' s search path was not set to include it. Try calling the command with an absolute path instead:

```
$ /usr/local/pgsql/bin/createdb mydb
```

The path at your site might be different. Contact your site administrator or check the installation instructions to correct the situation.

Another response could be this:

```
createdb: error: connection to server on socket "/tmp/.s.PGSQL.5432" failed: No such
file or directory
        Is the server running locally and accepting connections on that socket?
```

This means that the server was not started, or it is not listening where **createdb** expects to contact it. Again, check the installation instructions or consult the administrator.

Another response could be this:

```
createdb: error: connection to server on socket "/tmp/.s.PGSQL.5432" failed: FATAL:
role "joe" does not exist
```

where your own login name is mentioned. This will happen if the administrator has not created a IvorySQL user account for you. (IvorySQL user accounts are distinct from operating system user accounts.) If you are the administrator, You will need to become the operating system user under which IvorySQL was installed (usually **postgres**) to create the first user account. It could also be that you were assigned a IvorySQL user name that is different from your operating system user name; in that case you need to use the **-U** switch or set the **PGUSER** environment variable to specify your IvorySQL user name.

If you have a user account but it does not have the privileges required to create a database, you will see the

following:

```
createdb: error: database creation failed: ERROR:  permission denied to create
database
```

Not every user has authorization to create new databases. If IvorySQL refuses to create databases for you then the site administrator needs to grant you permission to create databases. Consult your site administrator if this occurs. If you installed IvorySQL yourself then you should log in for the purposes of this tutorial under the user account that you started the server as. [1]

You can also create databases with other names. IvorySQL allows you to create any number of databases at a given site. Database names must have an alphabetic first character and are limited to 63 bytes in length. A convenient choice is to create a database with the same name as your current user name. Many tools assume that database name as the default, so it can save you some typing. To create that database, simply type:

```
$ createdb
```

If you do not want to use your database anymore you can remove it. For example, if you are the owner (creator) of the database **mydb**, you can destroy it using the following command:

```
$ dropdb mydb
```

(For this command, the database name does not default to the user account name. You always need to specify it.) This action physically removes all files associated with the database and cannot be undone, so this should only be done with a great deal of forethought.

More about **createdb** and **dropdb** can be found in createdb and dropdb respectively.

Creating a New Table

You can create a new table by specifying the table name, along with all column names and their types:

```
CREATE TABLE weather (
    city            varchar(80),
    temp_lo         int,           -- low temperature
    temp_hi         int,           -- high temperature
    prcp            real,          -- precipitation
    date            date
);
```

You can enter this into **psql** with the line breaks. **psql** will recognize that the command is not terminated until the semicolon.

White space (i.e., spaces, tabs, and newlines) can be used freely in SQL commands. That means you can type the command aligned differently than above, or even all on one line. Two dashes (“--”) introduce comments. Whatever follows them is ignored up to the end of the line. SQL is case insensitive about key words and identifiers, except when identifiers are double-quoted to preserve the case (not done above).

varchar(80) specifies a data type that can store arbitrary character strings up to 80 characters in length. **int** is the normal integer type. **real** is a type for storing single precision floating-point numbers. **date** should be self-explanatory. (Yes, the column of type **date** is also named **date**. This might be convenient or confusing —

you choose.)

IvorySQL supports the standard SQL types `int`, `smallint`, `real`, `double precision`, `char(N)`, `varchar(N)`, `date`, `time`, `timestamp`, and `interval`, as well as other types of general utility and a rich set of geometric types. IvorySQL can be customized with an arbitrary number of user-defined data types. Consequently, type names are not key words in the syntax, except where required to support special cases in the SQL standard.

The second example will store cities and their associated geographical location:

```
CREATE TABLE cities (
    name            varchar(80),
    location        point
);
```

The `point` type is an example of a IvorySQL-specific data type.

Finally, it should be mentioned that if you don't need a table any longer or want to recreate it differently you can remove it using the following command:

```
DROP TABLE tablename;
```

Write to data

When a table is created, it contains no data. The first thing to do before a database can be of much use is to insert data. Data is inserted one row at a time. You can also insert more than one row in a single command, but it is not possible to insert something that is not a complete row. Even if you know only some column values, a complete row must be created.

To create a new row, use the INSERT command. The command requires the table name and column values.

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric
);

CREATE TABLE new_products (
    product_no int ,
    name varchar(255),
    price DECIMAL(10, 2),
    release_date DATE
);
```

An example command to insert a row would be:

```
INSERT INTO products VALUES (1, 'Cheese', 9.99);
```

The data values are listed in the order in which the columns appear in the table, separated by commas.

Usually, the data values will be literals (constants), but scalar expressions are also allowed.

The above syntax has the drawback that you need to know the order of the columns in the table. To avoid this you can also list the columns explicitly. For example, both of the following commands have the same effect as the one above:

```
INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese', 9.99);
INSERT INTO products (name, price, product_no) VALUES ('Cheese', 9.99, 1);
```

Many users consider it good practice to always list the column names.

If you don’t have values for all the columns, you can omit some of them. In that case, the columns will be filled with their default values. For example:

```
INSERT INTO products (product_no, name) VALUES (1, 'Cheese');
INSERT INTO products VALUES (1, 'Cheese');
```

The second form is a IvorySQL extension. It fills the columns from the left with as many values as are given, and the rest will be defaulted.

For clarity, you can also request default values explicitly, for individual columns or for the entire row:

```
INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese', DEFAULT);
INSERT INTO products DEFAULT VALUES;
```

You can insert multiple rows in a single command:

```
INSERT INTO products (product_no, name, price) VALUES
    (1, 'Cheese', 9.99),
    (2, 'Bread', 1.99),
    (3, 'Milk', 2.99);

INSERT INTO new_products (product_no, name, price, release_date) VALUES
    (1, 'A', 100.00, '2025-05-29'),
    (2, 'B', 150.50, '2024-11-20'),
    (3, 'C', 200.75, '2025-05-29');
```

It is also possible to insert the result of a query (which might be no rows, one row, or many rows):

```
INSERT INTO products (product_no, name, price)
  SELECT product_no, name, price FROM new_products
    WHERE release_date = '2025-05-29';
```

This provides the full power of the SQL query mechanism for computing the rows to be inserted.

Tip

When inserting a lot of data at the same time, consider using the COPY command. It is not as flexible as the INSERT command, but is more efficient.

Query Data

Combining Queries (`UNION`, `INTERSECT`, `EXCEPT`)

The results of two queries can be combined using the set operations union, intersection, and difference. The syntax is

```
query1 UNION [ALL] query2
query1 INTERSECT [ALL] query2
query1 EXCEPT [ALL] query2
```

where `query1` and `query2` are queries that can use any of the features discussed up to this point.

`UNION` effectively appends the result of `query2` to the result of `query1` (although there is no guarantee that this is the order in which the rows are actually returned). Furthermore, it eliminates duplicate rows from its result, in the same way as `DISTINCT`, unless `UNION ALL` is used.

`INTERSECT` returns all rows that are both in the result of `query1` and in the result of `query2`. Duplicate rows are eliminated unless `INTERSECT ALL` is used.

`EXCEPT` returns all rows that are in the result of `query1` but not in the result of `query2`. (This is sometimes called the difference between two queries.) Again, duplicates are eliminated unless `EXCEPT ALL` is used.

In order to calculate the union, intersection, or difference of two queries, the two queries must be “union compatible”, which means that they return the same number of columns and the corresponding columns have compatible data types.

Set operations can be combined, for example

```
query1 UNION query2 EXCEPT query3
```

which is equivalent to

```
(query1 UNION query2) EXCEPT query3
```

As shown here, you can use parentheses to control the order of evaluation. Without parentheses, `UNION` and `EXCEPT` associate left-to-right, but `INTERSECT` binds more tightly than those two operators. Thus

```
query1 UNION query2 INTERSECT query3
```

means

```
query1 UNION (query2 INTERSECT query3)
```

You can also surround an individual `query` with parentheses. This is important if the `query` needs to use any of the clauses discussed in following sections, such as `LIMIT`. Without parentheses, you'll get a syntax error, or else the clause will be understood as applying to the output of the set operation rather than one of its inputs. For example,

```
SELECT a FROM b UNION SELECT x FROM y LIMIT 10
```

is accepted, but it means

```
(SELECT a FROM b UNION SELECT x FROM y) LIMIT 10
```

not

```
SELECT a FROM b UNION (SELECT x FROM y LIMIT 10)
```

Parallel Query

How Parallel Query Works

When the optimizer determines that parallel query is the fastest execution strategy for a particular query, it will create a query plan that includes a Gather or Gather Merge node. Here is a simple example:

```
EXPLAIN SELECT * FROM pgbench_accounts WHERE filler LIKE '%x%';
                                     QUERY PLAN
------------------------------------------------------------------------------------
 Gather  (cost=1000.00..217018.43 rows=1 width=97)
   Workers Planned: 2
   ->  Parallel Seq Scan on pgbench_accounts  (cost=0.00..216018.33 rows=1 width=97)
         Filter: (filler ~~ '%x%'::text)
(4 rows)
```

In all cases, the `Gather` or `Gather Merge` node will have exactly one child plan, which is the portion of the plan that will be executed in parallel. If the `Gather` or `Gather Merge` node is at the very top of the plan tree, then the entire query will execute in parallel. If it is somewhere else in the plan tree, then only the portion of the plan below it will run in parallel. In the example above, the query accesses only one table, so there is only one plan node other than the `Gather` node itself; since that plan node is a child of the `Gather` node, it will run in parallel.

Using EXPLAIN, you can see the number of workers chosen by the planner. When the `Gather` node is reached during query execution, the process that is implementing the user's session will request a number of background worker processes equal to the number of workers chosen by the planner. The number of background workers that the planner will consider using is limited to at most max_parallel_workers_per_gather. The total number of background workers that can exist at any one time is limited by both max_worker_processes and max_parallel_workers. Therefore, it is possible for a parallel query to run with fewer workers than planned, or even with no workers at all. The optimal plan may depend on the number of workers that are available, so this can result in poor query performance. If this occurrence is frequent, consider increasing `max_worker_processes` and `max_parallel_workers` so that more workers can be run simultaneously or alternatively reducing `max_parallel_workers_per_gather` so that the planner requests fewer workers.

Every background worker process that is successfully started for a given parallel query will execute the parallel portion of the plan. The leader will also execute that portion of the plan, but it has an additional responsibility: it must also read all of the tuples generated by the workers. When the parallel portion of the plan generates only a small number of tuples, the leader will often behave very much like an additional worker, speeding up query execution. Conversely, when the parallel portion of the plan generates a large number of tuples, the leader may be almost entirely occupied with reading the tuples generated by the workers and performing any further processing steps that are required by plan nodes above the level of the `Gather` node or `Gather Merge` node. In such cases, the leader will do very little of the work of executing the parallel portion of the plan.

When the node at the top of the parallel portion of the plan is `Gather Merge` rather than `Gather`, it indicates that each process executing the parallel portion of the plan is producing tuples in sorted order, and that the leader is performing an order-preserving merge. In contrast, `Gather` reads tuples from the workers in whatever order is convenient, destroying any sort order that may have existed.

When Can Parallel Query Be Used?

There are several settings that can cause the query planner not to generate a parallel query plan under any circumstances. In order for any parallel query plans whatsoever to be generated, the following settings must be configured as indicated.

- max_parallel_workers_per_gather must be set to a value that is greater than zero. This is a special case of the more general principle that no more workers should be used than the number configured via `max_parallel_workers_per_gather`.

In addition, the system must not be running in single-user mode. Since the entire database system is running as a single process in this situation, no background workers will be available.

Even when it is in general possible for parallel query plans to be generated, the planner will not generate them for a given query if any of the following are true:

- The query writes any data or locks any database rows. If a query contains a data-modifying operation either at the top level or within a CTE, no parallel plans for that query will be generated. As an exception, the following commands, which create a new table and populate it, can use a parallel plan for the underlying `SELECT` part of the query:
- `CREATE TABLE ⋯ AS`
- `SELECT INTO`
- `CREATE MATERIALIZED VIEW`
- `REFRESH MATERIALIZED VIEW`
- The query might be suspended during execution. In any situation in which the system thinks that partial or incremental execution might occur, no parallel plan is generated. For example, a cursor created using DECLARE CURSOR will never use a parallel plan. Similarly, a PL/pgSQL loop of the form `FOR x IN query LOOP .. END LOOP` will never use a parallel plan, because the parallel query system is unable to verify that the code in the loop is safe to execute while parallel query is active.
- The query uses any function marked `PARALLEL UNSAFE`. Most system-defined functions are `PARALLEL SAFE`, but user-defined functions are marked `PARALLEL UNSAFE` by default.
- The query is running inside of another query that is already parallel. For example, if a function called by a parallel query issues an SQL query itself, that query will never use a parallel plan. This is a limitation of the current implementation, but it may not be desirable to remove this limitation, since it could result in a single query using a very large number of processes.

Even when parallel query plan is generated for a particular query, there are several circumstances under which it will be impossible to execute that plan in parallel at execution time. If this occurs, the leader will execute the portion of the plan below the `Gather` node entirely by itself, almost as if the `Gather` node were not present. This will happen if any of the following conditions are met:

- No background workers can be obtained because of the limitation that the total number of background workers cannot exceed max_worker_processes.

- No background workers can be obtained because of the limitation that the total number of background workers launched for purposes of parallel query cannot exceed max_parallel_workers.
- The client sends an Execute message with a non-zero fetch count. See the discussion of the extended query protocol. Since libpq currently provides no way to send such a message, this can only occur when using a client that does not rely on libpq. If this is a frequent occurrence, it may be a good idea to set max_parallel_workers_per_gather to zero in sessions where it is likely, so as to avoid generating query plans that may be suboptimal when run serially.

Parallel Plans

Because each worker executes the parallel portion of the plan to completion, it is not possible to simply take an ordinary query plan and run it using multiple workers. Each worker would produce a full copy of the output result set, so the query would not run any faster than normal but would produce incorrect results. Instead, the parallel portion of the plan must be what is known internally to the query optimizer as a partial plan; that is, it must be constructed so that each process that executes the plan will generate only a subset of the output rows in such a way that each required output row is guaranteed to be generated by exactly one of the cooperating processes. Generally, this means that the scan on the driving table of the query must be a parallel-aware scan.

Parallel Scans

The following types of parallel-aware table scans are currently supported.

- In a parallel sequential scan, the table’ s blocks will be divided into ranges and shared among the cooperating processes. Each worker process will complete the scanning of its given range of blocks before requesting an additional range of blocks.
- In a parallel bitmap heap scan, one process is chosen as the leader. That process performs a scan of one or more indexes and builds a bitmap indicating which table blocks need to be visited. These blocks are then divided among the cooperating processes as in a parallel sequential scan. In other words, the heap scan is performed in parallel, but the underlying index scan is not.
- In a parallel index scan or parallel index-only scan, the cooperating processes take turns reading data from the index. Currently, parallel index scans are supported only for btree indexes. Each process will claim a single index block and will scan and return all tuples referenced by that block; other processes can at the same time be returning tuples from a different index block. The results of a parallel btree scan are returned in sorted order within each worker process.

Other scan types, such as scans of non-btree indexes, may support parallel scans in the future.

Parallel Joins

Just as in a non-parallel plan, the driving table may be joined to one or more other tables using a nested loop, hash join, or merge join. The inner side of the join may be any kind of non-parallel plan that is otherwise supported by the planner provided that it is safe to run within a parallel worker. Depending on the join type, the inner side may also be a parallel plan.

- In a nested loop join, the inner side is always non-parallel. Although it is executed in full, this is efficient if the inner side is an index scan, because the outer tuples and thus the loops that look up values in the index are divided over the cooperating processes.
- In a merge join, the inner side is always a non-parallel plan and therefore executed in full. This may be inefficient, especially if a sort must be performed, because the work and resulting data are duplicated in every cooperating process.
- In a hash join (without the "parallel" prefix), the inner side is executed in full by every cooperating process to build identical copies of the hash table. This may be inefficient if the hash table is large or the plan is expensive. In a parallel hash join, the inner side is a parallel hash that divides the work of building a shared hash table over the cooperating processes.

Parallel Aggregation

IvorySQL supports parallel aggregation by aggregating in two stages. First, each process participating in the

parallel portion of the query performs an aggregation step, producing a partial result for each group of which that process is aware. This is reflected in the plan as a `Partial Aggregate` node. Second, the partial results are transferred to the leader via `Gather` or `Gather Merge`. Finally, the leader re-aggregates the results across all workers in order to produce the final result. This is reflected in the plan as a `Finalize Aggregate` node.

Because the `Finalize Aggregate` node runs on the leader process, queries that produce a relatively large number of groups in comparison to the number of input rows will appear less favorable to the query planner. For example, in the worst-case scenario the number of groups seen by the `Finalize Aggregate` node could be as many as the number of input rows that were seen by all worker processes in the `Partial Aggregate` stage. For such cases, there is clearly going to be no performance benefit to using parallel aggregation. The query planner takes this into account during the planning process and is unlikely to choose parallel aggregate in this scenario.

Parallel aggregation is not supported in all situations. Each aggregate must be safe for parallelism and must have a combine function. If the aggregate has a transition state of type `internal`, it must have serialization and deserialization functions. See CREATE AGGREGATE for more details. Parallel aggregation is not supported if any aggregate function call contains `DISTINCT` or `ORDER BY` clause and is also not supported for ordered set aggregates or when the query involves `GROUPING SETS`. It can only be used when all joins involved in the query are also part of the parallel portion of the plan.

Parallel Append

Whenever IvorySQL needs to combine rows from multiple sources into a single result set, it uses an `Append` or `MergeAppend` plan node. This commonly happens when implementing `UNION ALL` or when scanning a partitioned table. Such nodes can be used in parallel plans just as they can in any other plan. However, in a parallel plan, the planner may instead use a `Parallel Append` node.

When an `Append` node is used in a parallel plan, each process will execute the child plans in the order in which they appear, so that all participating processes cooperate to execute the first child plan until it is complete and then move to the second plan at around the same time. When a `Parallel Append` is used instead, the executor will instead spread out the participating processes as evenly as possible across its child plans, so that multiple child plans are executed simultaneously. This avoids contention, and also avoids paying the startup cost of a child plan in those processes that never execute it.

Also, unlike a regular `Append` node, which can only have partial children when used within a parallel plan, a `Parallel Append` node can have both partial and non-partial child plans. Non-partial children will be scanned by only a single process, since scanning them more than once would produce duplicate results. Plans that involve appending multiple results sets can therefore achieve coarse-grained parallelism even when efficient partial plans are not available. For example, consider a query against a partitioned table that can only be implemented efficiently by using an index that does not support parallel scans. The planner might choose a `Parallel Append` of regular `Index Scan` plans; each individual index scan would have to be executed to completion by a single process, but different scans could be performed at the same time by different processes.

enable_parallel_append can be used to disable this feature.

Parallel Plan Tips

If a query that is expected to do so does not produce a parallel plan, you can try reducing parallel_setup_cost or parallel_tuple_cost. Of course, this plan may turn out to be slower than the serial plan that the planner preferred, but this will not always be the case. If you don’ t get a parallel plan even with very small values of these settings (e.g., after setting them both to zero), there may be some reason why the query planner is unable to generate a parallel plan for your query.

When executing a parallel plan, you can use `EXPLAIN (ANALYZE, VERBOSE)` to display per-worker statistics for each plan node. This may be useful in determining whether the work is being evenly distributed between all plan nodes and more generally in understanding the performance characteristics of the plan.

Transaction

ABORT — abort the current transaction

Synopsis

```
ABORT [ WORK | TRANSACTION ] [ AND [ NO ] CHAIN ]
```

Description

ABORT rolls back the current transaction and causes all the updates made by the transaction to be discarded. This command is identical in behavior to the standard SQL command ROLLBACK, and is present only for historical reasons.

Parameters

- WORK TRANSACTION

Optional key words. They have no effect.

- AND CHAIN

If AND CHAIN is specified, a new transaction is immediately started with the same transaction characteristics (see SET TRANSACTION) as the just finished one. Otherwise, no new transaction is started.

Notes

Use COMMIT to successfully terminate a transaction.

Issuing ABORT outside of a transaction block emits a warning and otherwise has no effect.

Examples

To abort all changes:

```
ABORT;
```

Compatibility

This command is a IvorySQL extension present for historical reasons. ROLLBACK is the equivalent standard SQL command.

BEGIN — start a transaction block

Synopsis

```
BEGIN [ WORK | TRANSACTION ] [ transaction_mode [, ...] ]

where transaction_mode is one of:

    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ
UNCOMMITTED }
    READ WRITE | READ ONLY
```

```
    [ NOT ] DEFERRABLE
```

Description

BEGIN initiates a transaction block, that is, all statements after a **BEGIN** command will be executed in a single transaction until an explicit **COMMIT** or **ROLLBACK** is given. By default (without **BEGIN**), IvorySQL executes transactions in "autocommit" mode, that is, each statement is executed in its own transaction and a commit is implicitly performed at the end of the statement (if execution was successful, otherwise a rollback is done).

Statements are executed more quickly in a transaction block, because transaction start/commit requires significant CPU and disk activity. Execution of multiple statements inside a transaction is also useful to ensure consistency when making several related changes: other sessions will be unable to see the intermediate states wherein not all the related updates have been done.

If the isolation level, read/write mode, or deferrable mode is specified, the new transaction has those characteristics, as if **SET TRANSACTION** was executed.

Parameters

- **WORK TRANSACTION**

Optional key words. They have no effect.

Refer to SET TRANSACTION for information on the meaning of the other parameters to this statement.

Notes

START TRANSACTION has the same functionality as **BEGIN**.

Use **COMMIT** or **ROLLBACK** to terminate a transaction block.

Issuing **BEGIN** when already inside a transaction block will provoke a warning message. The state of the transaction is not affected. To nest transactions within a transaction block, use savepoints (see SAVEPOINT).

For reasons of backwards compatibility, the commas between successive **transaction_modes** can be omitted.

Examples

To begin a transaction block:

```
BEGIN;
```

Compatibility

BEGIN is a IvorySQL language extension. It is equivalent to the SQL-standard command **START TRANSACTION**, whose reference page contains additional compatibility information.

The **DEFERRABLE transaction_mode** is a IvorySQL language extension.

Incidentally, the **BEGIN** key word is used for a different purpose in embedded SQL. You are advised to be careful about the transaction semantics when porting database applications.

COMMIT — commit the current transaction

Synopsis

```
COMMIT [ WORK | TRANSACTION ] [ AND [ NO ] CHAIN ]
```

Description

`COMMIT` commits the current transaction. All changes made by the transaction become visible to others and are guaranteed to be durable if a crash occurs.

Parameters

- `WORK TRANSACTION`

Optional key words. They have no effect.

- `AND CHAIN`

If `AND CHAIN` is specified, a new transaction is immediately started with the same transaction characteristics (see SET TRANSACTION) as the just finished one. Otherwise, no new transaction is started.

Notes

Use ROLLBACK to abort a transaction.

Issuing `COMMIT` when not inside a transaction does no harm, but it will provoke a warning message. `COMMIT AND CHAIN` when not inside a transaction is an error.

Examples

To commit the current transaction and make all changes permanent:

```
COMMIT;
```

Compatibility

The command `COMMIT` conforms to the SQL standard. The form `COMMIT TRANSACTION` is a IvorySQL extension.

COMMIT PREPARED — commit a transaction that was earlier prepared for two-phase commit

Synopsis

```
COMMIT PREPARED transaction_id
```

Description

`COMMIT PREPARED` commits a transaction that is in prepared state.

Parameters

- `transaction_id`

The transaction identifier of the transaction that is to be committed.

Notes

To commit a prepared transaction, you must be either the same user that executed the transaction originally, or a superuser. But you do not have to be in the same session that executed the transaction.

This command cannot be executed inside a transaction block. The prepared transaction is committed immediately.

All currently available prepared transactions are listed in the `pg_prepared_xacts` system view.

Examples

Commit the transaction identified by the transaction identifier `foobar`:

```
COMMIT PREPARED 'foobar';
```

Compatibility

`COMMIT PREPARED` is a IvorySQL extension. It is intended for use by external transaction management systems, some of which are covered by standards (such as X/Open XA), but the SQL side of those systems is not standardized.

END — commit the current transaction

Synopsis

```
END [ WORK | TRANSACTION ] [ AND [ NO ] CHAIN ]
```

Description

`END` commits the current transaction. All changes made by the transaction become visible to others and are guaranteed to be durable if a crash occurs. This command is a IvorySQL extension that is equivalent to `COMMIT`.

Parameters

- `WORK TRANSACTION`

Optional key words. They have no effect.

- `AND CHAIN`

If `AND CHAIN` is specified, a new transaction is immediately started with the same transaction characteristics (see SET TRANSACTION) as the just finished one. Otherwise, no new transaction is started.

Notes

Use `ROLLBACK` to abort a transaction.

Issuing `END` when not inside a transaction does no harm, but it will provoke a warning message.

Examples

To commit the current transaction and make all changes permanent:

```
END;
```

Compatibility

END is a IvorySQL extension that provides functionality equivalent to COMMIT, which is specified in the SQL standard.

PREPARE TRANSACTION — prepare the current transaction for two-phase commit

Synopsis

```
PREPARE TRANSACTION transaction_id
```

Description

PREPARE TRANSACTION prepares the current transaction for two-phase commit. After this command, the transaction is no longer associated with the current session; instead, its state is fully stored on disk, and there is a very high probability that it can be committed successfully, even if a database crash occurs before the commit is requested.

Once prepared, a transaction can later be committed or rolled back with COMMIT PREPARED or ROLLBACK PREPARED, respectively. Those commands can be issued from any session, not only the one that executed the original transaction.

From the point of view of the issuing session, PREPARE TRANSACTION is not unlike a ROLLBACK command: after executing it, there is no active current transaction, and the effects of the prepared transaction are no longer visible. (The effects will become visible again if the transaction is committed.)

If the PREPARE TRANSACTION command fails for any reason, it becomes a ROLLBACK: the current transaction is canceled.

Parameters

- transaction_id

An arbitrary identifier that later identifies this transaction for COMMIT PREPARED or ROLLBACK PREPARED. The identifier must be written as a string literal, and must be less than 200 bytes long. It must not be the same as the identifier used for any currently prepared transaction.

Notes

PREPARE TRANSACTION is not intended for use in applications or interactive sessions. Its purpose is to allow an external transaction manager to perform atomic global transactions across multiple databases or other transactional resources. Unless you’re writing a transaction manager, you probably shouldn’t be using PREPARE TRANSACTION.

This command must be used inside a transaction block. Use BEGIN to start one.

It is not currently allowed to PREPARE a transaction that has executed any operations involving temporary tables or the session’s temporary namespace, created any cursors WITH HOLD, or executed LISTEN, UNLISTEN, or NOTIFY. Those features are too tightly tied to the current session to be useful in a transaction to be prepared.

If the transaction modified any run-time parameters with SET (without the LOCAL option), those effects persist after PREPARE TRANSACTION, and will not be affected by any later COMMIT PREPARED or ROLLBACK PREPARED.

Thus, in this one respect **PREPARE TRANSACTION** acts more like **COMMIT** than **ROLLBACK**.

All currently available prepared transactions are listed in the **pg_prepared_xacts** system view.

Caution

It is unwise to leave transactions in the prepared state for a long time. This will interfere with the ability of **VACUUM** to reclaim storage, and in extreme cases could cause the database to shut down to prevent transaction ID wraparound (see Section 25.1.5). Keep in mind also that the transaction continues to hold whatever locks it held. The intended usage of the feature is that a prepared transaction will normally be committed or rolled back as soon as an external transaction manager has verified that other databases are also prepared to commit.

If you have not set up an external transaction manager to track prepared transactions and ensure they get closed out promptly, it is best to keep the prepared-transaction feature disabled by setting max_prepared_transactions to zero. This will prevent accidental creation of prepared transactions that might then be forgotten and eventually cause problems.

Examples

Prepare the current transaction for two-phase commit, using **foobar** as the transaction identifier:

```
PREPARE TRANSACTION 'foobar';
```

Compatibility

PREPARE TRANSACTION is a IvorySQL extension. It is intended for use by external transaction management systems, some of which are covered by standards (such as X/Open XA), but the SQL side of those systems is not standardized.

ROLLBACK — abort the current transaction

Synopsis

```
ROLLBACK [ WORK | TRANSACTION ] [ AND [ NO ] CHAIN ]
```

Description

ROLLBACK rolls back the current transaction and causes all the updates made by the transaction to be discarded.

Parameters

- **WORK TRANSACTION**

Optional key words. They have no effect.

- **AND CHAIN**

If **AND CHAIN** is specified, a new transaction is immediately started with the same transaction characteristics (see SET TRANSACTION) as the just finished one. Otherwise, no new transaction is started.

Notes

Use **COMMIT** to successfully terminate a transaction.

Issuing `ROLLBACK` outside of a transaction block emits a warning and otherwise has no effect. `ROLLBACK AND CHAIN` outside of a transaction block is an error.

Examples

To abort all changes:

```
ROLLBACK;
```

Compatibility

The command `ROLLBACK` conforms to the SQL standard. The form `ROLLBACK TRANSACTION` is a IvorySQL extension.

ROLLBACK PREPARED — cancel a transaction that was earlier prepared for two-phase commit

Synopsis

```
ROLLBACK PREPARED transaction_id
```

Description

`ROLLBACK PREPARED` rolls back a transaction that is in prepared state.

Parameters

- `transaction_id`

The transaction identifier of the transaction that is to be rolled back.

Notes

To roll back a prepared transaction, you must be either the same user that executed the transaction originally, or a superuser. But you do not have to be in the same session that executed the transaction.

This command cannot be executed inside a transaction block. The prepared transaction is rolled back immediately.

All currently available prepared transactions are listed in the `pg_prepared_xacts` system view.

Examples

Roll back the transaction identified by the transaction identifier `foobar`:

```
ROLLBACK PREPARED 'foobar';
```

Compatibility

`ROLLBACK PREPARED` is a IvorySQL extension. It is intended for use by external transaction management systems, some of which are covered by standards (such as X/Open XA), but the SQL side of those systems is not standardized.

SAVEPOINT — define a new savepoint within the current transaction

Synopsis

```
SAVEPOINT savepoint_name
```

Description

`SAVEPOINT` establishes a new savepoint within the current transaction.

A savepoint is a special mark inside a transaction that allows all commands that are executed after it was established to be rolled back, restoring the transaction state to what it was at the time of the savepoint.

Parameters

- `savepoint_name`

The name to give to the new savepoint. If savepoints with the same name already exist, they will be inaccessible until newer identically-named savepoints are released.

Notes

Use `ROLLBACK TO` to rollback to a savepoint. Use `RELEASE SAVEPOINT` to destroy a savepoint, keeping the effects of commands executed after it was established.

Savepoints can only be established when inside a transaction block. There can be multiple savepoints defined within a transaction.

Examples

To establish a savepoint and later undo the effects of all commands executed after it was established:

```
BEGIN;
    INSERT INTO table1 VALUES (1);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (2);
    ROLLBACK TO SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (3);
COMMIT;
```

The above transaction will insert the values 1 and 3, but not 2.

To establish and later destroy a savepoint:

```
BEGIN;
    INSERT INTO table1 VALUES (3);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (4);
    RELEASE SAVEPOINT my_savepoint;
```

```
COMMIT;
```

The above transaction will insert both 3 and 4.

To use a single savepoint name:

```
BEGIN;
    INSERT INTO table1 VALUES (1);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (2);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (3);

    -- rollback to the second savepoint
    ROLLBACK TO SAVEPOINT my_savepoint;
    SELECT * FROM table1;               -- shows rows 1 and 2

    -- release the second savepoint
    RELEASE SAVEPOINT my_savepoint;

    -- rollback to the first savepoint
    ROLLBACK TO SAVEPOINT my_savepoint;
    SELECT * FROM table1;               -- shows only row 1
COMMIT;
```

The above transaction shows row 3 being rolled back first, then row 2.

Compatibility

SQL requires a savepoint to be destroyed automatically when another savepoint with the same name is established. In IvorySQL, the old savepoint is kept, though only the more recent one will be used when rolling back or releasing. (Releasing the newer savepoint with `RELEASE SAVEPOINT` will cause the older one to again become accessible to `ROLLBACK TO SAVEPOINT` and `RELEASE SAVEPOINT`.) Otherwise, `SAVEPOINT` is fully SQL conforming.

SET CONSTRAINTS — set constraint check timing for the current transaction

Synopsis

```
SET CONSTRAINTS { ALL | name [, ...] } { DEFERRED | IMMEDIATE }
```

Description

`SET CONSTRAINTS` sets the behavior of constraint checking within the current transaction. `IMMEDIATE` constraints are checked at the end of each statement. `DEFERRED` constraints are not checked until transaction commit. Each constraint has its own `IMMEDIATE` or `DEFERRED` mode.

Upon creation, a constraint is given one of three characteristics: `DEFERRABLE INITIALLY DEFERRED`, `DEFERRABLE INITIALLY IMMEDIATE`, or `NOT DEFERRABLE`. The third class is always `IMMEDIATE` and is not affected by the `SET CONSTRAINTS` command. The first two classes start every transaction in the indicated mode, but their behavior can be changed within a transaction by `SET CONSTRAINTS`.

`SET CONSTRAINTS` with a list of constraint names changes the mode of just those constraints (which must all be deferrable). Each constraint name can be schema-qualified. The current schema search path is used to find the first matching name if no schema name is specified. `SET CONSTRAINTS ALL` changes the mode of all deferrable constraints.

When `SET CONSTRAINTS` changes the mode of a constraint from `DEFERRED` to `IMMEDIATE`, the new mode takes effect retroactively: any outstanding data modifications that would have been checked at the end of the transaction are instead checked during the execution of the `SET CONSTRAINTS` command. If any such constraint is violated, the `SET CONSTRAINTS` fails (and does not change the constraint mode). Thus, `SET CONSTRAINTS` can be used to force checking of constraints to occur at a specific point in a transaction.

Currently, only `UNIQUE`, `PRIMARY KEY`, `REFERENCES` (foreign key), and `EXCLUDE` constraints are affected by this setting. `NOT NULL` and `CHECK` constraints are always checked immediately when a row is inserted or modified (not at the end of the statement). Uniqueness and exclusion constraints that have not been declared `DEFERRABLE` are also checked immediately.

The firing of triggers that are declared as “constraint triggers” is also controlled by this setting — they fire at the same time that the associated constraint should be checked.

Notes

Because IvorySQL does not require constraint names to be unique within a schema (but only per-table), it is possible that there is more than one match for a specified constraint name. In this case `SET CONSTRAINTS` will act on all matches. For a non-schema-qualified name, once a match or matches have been found in some schema in the search path, schemas appearing later in the path are not searched.

This command only alters the behavior of constraints within the current transaction. Issuing this outside of a transaction block emits a warning and otherwise has no effect.

Compatibility

This command complies with the behavior defined in the SQL standard, except for the limitation that, in IvorySQL, it does not apply to `NOT NULL` and `CHECK` constraints. Also, IvorySQL checks non-deferrable uniqueness constraints immediately, not at end of statement as the standard would suggest.

SET TRANSACTION — set the characteristics of the current transaction

Synopsis

```
SET TRANSACTION transaction_mode [, ...]
SET TRANSACTION SNAPSHOT snapshot_id
SET SESSION CHARACTERISTICS AS TRANSACTION transaction_mode [, ...]

where transaction_mode is one of:

    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ
UNCOMMITTED }
    READ WRITE | READ ONLY
    [ NOT ] DEFERRABLE
```

Description

The `SET TRANSACTION` command sets the characteristics of the current transaction. It has no effect on any subsequent transactions. `SET SESSION CHARACTERISTICS` sets the default transaction characteristics for subsequent transactions of a session. These defaults can be overridden by `SET TRANSACTION` for an individual transaction.

The available transaction characteristics are the transaction isolation level, the transaction access mode (read/write or read-only), and the deferrable mode. In addition, a snapshot can be selected, though only for the current transaction, not as a session default.

The isolation level of a transaction determines what data the transaction can see when other transactions are running concurrently:

- `READ COMMITTED`

A statement can only see rows committed before it began. This is the default.

- `REPEATABLE READ`

All statements of the current transaction can only see rows committed before the first query or data-modification statement was executed in this transaction.

- `SERIALIZABLE`

All statements of the current transaction can only see rows committed before the first query or data-modification statement was executed in this transaction. If a pattern of reads and writes among concurrent serializable transactions would create a situation which could not have occurred for any serial (one-at-a-time) execution of those transactions, one of them will be rolled back with a `serialization_failure` error.

The SQL standard defines one additional level, `READ UNCOMMITTED`. In IvorySQL `READ UNCOMMITTED` is treated as `READ COMMITTED`.

The transaction isolation level cannot be changed after the first query or data-modification statement (`SELECT`, `INSERT`, `DELETE`, `UPDATE`, `FETCH`, or `COPY`) of a transaction has been executed. See Chapter 13 for more information about transaction isolation and concurrency control.

The transaction access mode determines whether the transaction is read/write or read-only. Read/write is the default. When a transaction is read-only, the following SQL commands are disallowed: `INSERT`, `UPDATE`, `DELETE`, and `COPY FROM` if the table they would write to is not a temporary table; all `CREATE`, `ALTER`, and `DROP` commands; `COMMENT`, `GRANT`, `REVOKE`, `TRUNCATE`; and `EXPLAIN ANALYZE` and `EXECUTE` if the command they would execute is among those listed. This is a high-level notion of read-only that does not prevent all writes to disk.

The `DEFERRABLE` transaction property has no effect unless the transaction is also `SERIALIZABLE` and `READ ONLY`. When all three of these properties are selected for a transaction, the transaction may block when first acquiring its snapshot, after which it is able to run without the normal overhead of a `SERIALIZABLE` transaction and without any risk of contributing to or being canceled by a serialization failure. This mode is well suited for long-running reports or backups.

The `SET TRANSACTION SNAPSHOT` command allows a new transaction to run with the same snapshot as an existing transaction. The pre-existing transaction must have exported its snapshot with the `pg_export_snapshot` function. That function returns a snapshot identifier, which must be given to `SET TRANSACTION SNAPSHOT` to specify which snapshot is to be imported. The identifier must be written as a string literal in this command, for example `'00000003-0000001B-1'`. `SET TRANSACTION SNAPSHOT` can only be executed at the start of a transaction, before the first query or data-modification statement (`SELECT`, `INSERT`, `DELETE`, `UPDATE`, `FETCH`, or `COPY`) of the transaction. Furthermore, the transaction must already be set to `SERIALIZABLE` or `REPEATABLE READ` isolation level (otherwise, the snapshot would be discarded immediately, since `READ COMMITTED` mode takes a new snapshot for each command). If the importing transaction uses `SERIALIZABLE` isolation level, then the transaction that exported the snapshot must also use that isolation level. Also, a non-read-only serializable transaction cannot import a snapshot from a read-only transaction.

If SET TRANSACTION is executed without a prior START TRANSACTION or BEGIN, it emits a warning and otherwise has no effect.

It is possible to dispense with SET TRANSACTION by instead specifying the desired transaction_modes in BEGIN or START TRANSACTION. But that option is not available for SET TRANSACTION SNAPSHOT.

The session default transaction modes can also be set or examined via the configuration parameters default_transaction_isolation, default_transaction_read_only, and default_transaction_deferrable. (In fact SET SESSION CHARACTERISTICS is just a verbose equivalent for setting these variables with SET.) This means the defaults can be set in the configuration file, via ALTER DATABASE, etc. Consult Chapter 20 for more information.

The current transaction's modes can similarly be set or examined via the configuration parameters transaction_isolation, transaction_read_only, and transaction_deferrable. Setting one of these parameters acts the same as the corresponding SET TRANSACTION option, with the same restrictions on when it can be done. However, these parameters cannot be set in the configuration file, or from any source other than live SQL.

Examples

To begin a new transaction with the same snapshot as an already existing transaction, first export the snapshot from the existing transaction. That will return the snapshot identifier, for example:

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT pg_export_snapshot();
 pg_export_snapshot
---------------------
 00000003-0000001B-1
(1 row)
```

Then give the snapshot identifier in a SET TRANSACTION SNAPSHOT command at the beginning of the newly opened transaction:

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET TRANSACTION SNAPSHOT '00000003-0000001B-1';
```

Compatibility

These commands are defined in the SQL standard, except for the DEFERRABLE transaction mode and the SET TRANSACTION SNAPSHOT form, which are IvorySQL extensions.

SERIALIZABLE is the default transaction isolation level in the standard. In IvorySQL the default is ordinarily READ COMMITTED, but you can change it as mentioned above.

In the SQL standard, there is one other transaction characteristic that can be set with these commands: the size of the diagnostics area. This concept is specific to embedded SQL, and therefore is not implemented in the IvorySQL server.

The SQL standard requires commas between successive transaction_modes, but for historical reasons IvorySQL allows the commas to be omitted.

START TRANSACTION — start a transaction block

Synopsis

```
START TRANSACTION [ transaction_mode [, ...] ]

where transaction_mode is one of:

    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ
UNCOMMITTED }
    READ WRITE | READ ONLY
    [ NOT ] DEFERRABLE
```

Description

This command begins a new transaction block. If the isolation level, read/write mode, or deferrable mode is specified, the new transaction has those characteristics, as if `SET TRANSACTION` was executed. This is the same as the `BEGIN` command.

Parameters

Refer to SET TRANSACTION for information on the meaning of the parameters to this statement.

Compatibility

In the standard, it is not necessary to issue `START TRANSACTION` to start a transaction block: any SQL command implicitly begins a block. IvorySQL’s behavior can be seen as implicitly issuing a `COMMIT` after each command that does not follow `START TRANSACTION` (or `BEGIN`), and it is therefore often called “autocommit”. Other relational database systems might offer an autocommit feature as a convenience.

The `DEFERRABLE transaction_mode` is a IvorySQL language extension.

The SQL standard requires commas between successive `transaction_modes`, but for historical reasons IvorySQL allows the commas to be omitted.

See also the compatibility section of SET TRANSACTION.

Sql Reference

Lexical Structure

SQL input consists of a sequence of commands. A command is composed of a sequence of tokens, terminated by a semicolon (“;”). The end of the input stream also terminates a command. Which tokens are valid depends on the syntax of the particular command.

A token can be a key word, an identifier, a quoted identifier, a literal (or constant), or a special character symbol. Tokens are normally separated by whitespace (space, tab, newline), but need not be if there is no ambiguity (which is generally only the case if a special character is adjacent to some other token type).

For example, the following is (syntactically) valid SQL input:

```
SELECT * FROM MY_TABLE;
UPDATE MY_TABLE SET A = 5;
```

```
INSERT INTO MY_TABLE VALUES (3, 'hi there');
```

This is a sequence of three commands, one per line (although this is not required; more than one command can be on a line, and commands can usefully be split across lines).

Additionally, comments can occur in SQL input. They are not tokens, they are effectively equivalent to whitespace.

The SQL syntax is not very consistent regarding what tokens identify commands and which are operands or parameters. The first few tokens are generally the command name, so in the above example we would usually speak of a "SELECT", an "UPDATE", and an "INSERT" command. But for instance the `UPDATE` command always requires a `SET` token to appear in a certain position, and this particular variation of `INSERT` also requires a `VALUES` in order to be complete.

Identifiers and Key Words

Tokens such as `SELECT`, `UPDATE`, or `VALUES` in the example above are examples of key words, that is, words that have a fixed meaning in the SQL language. The tokens `MY_TABLE` and `A` are examples of identifiers. They identify names of tables, columns, or other database objects, depending on the command they are used in. Therefore they are sometimes simply called "names". Key words and identifiers have the same lexical structure, meaning that one cannot know whether a token is an identifier or a key word without knowing the language. A complete list of key words can be found in Appendix C.

SQL identifiers and key words must begin with a letter (`a`-`z`, but also letters with diacritical marks and non-Latin letters) or an underscore (`_`). Subsequent characters in an identifier or key word can be letters, underscores, digits (`0`-`9`), or dollar signs (`$`). Note that dollar signs are not allowed in identifiers according to the letter of the SQL standard, so their use might render applications less portable. The SQL standard will not define a key word that contains digits or starts or ends with an underscore, so identifiers of this form are safe against possible conflict with future extensions of the standard.

The system uses no more than `NAMEDATALEN`-1 bytes of an identifier; longer names can be written in commands, but they will be truncated. By default, `NAMEDATALEN` is 64 so the maximum identifier length is 63 bytes. If this limit is problematic, it can be raised by changing the `NAMEDATALEN` constant in `src/include/pg_config_manual.h`.

Key words and unquoted identifiers are case insensitive. Therefore:

```
UPDATE MY_TABLE SET A = 5;
```

can equivalently be written as:

```
uPDaTE my_TabLE SeT a = 5;
```

A convention often used is to write key words in upper case and names in lower case, e.g.:

```
UPDATE my_table SET a = 5;
```

There is a second kind of identifier: the delimited identifier or quoted identifier. It is formed by enclosing an arbitrary sequence of characters in double-quotes (`"`). A delimited identifier is always an identifier, never a key word. So `"select"` could be used to refer to a column or table named "select", whereas an unquoted `select` would be taken as a key word and would therefore provoke a parse error when used where a table or column name is expected. The example can be written with quoted identifiers like this:

```
UPDATE "my_table" SET "a" = 5;
```

Quoted identifiers can contain any character, except the character with code zero. (To include a double quote, write two double quotes.) This allows constructing table or column names that would otherwise not be possible, such as ones containing spaces or ampersands. The length limitation still applies.

Quoting an identifier also makes it case-sensitive, whereas unquoted names are always folded to lower case. For example, the identifiers `FOO`, `foo`, and `"foo"` are considered the same by IvorySQL, but `"Foo"` and `"FOO"` are different from these three and each other. (The folding of unquoted names to lower case in IvorySQL is incompatible with the SQL standard, which says that unquoted names should be folded to upper case. Thus, `foo` should be equivalent to `"FOO"` not `"foo"` according to the standard. If you want to write portable applications you are advised to always quote a particular name or never quote it.)

A variant of quoted identifiers allows including escaped Unicode characters identified by their code points. This variant starts with `U&` (upper or lower case U followed by ampersand) immediately before the opening double quote, without any spaces in between, for example `U&"foo"`. (Note that this creates an ambiguity with the operator `&`. Use spaces around the operator to avoid this problem.) Inside the quotes, Unicode characters can be specified in escaped form by writing a backslash followed by the four-digit hexadecimal code point number or alternatively a backslash followed by a plus sign followed by a six-digit hexadecimal code point number. For example, the identifier `"data"` could be written as

```
U&"d\0061t\+000061"
```

The following less trivial example writes the Russian word “slon” (elephant) in Cyrillic letters:

```
U&"\0441\043B\043E\043D"
```

If a different escape character than backslash is desired, it can be specified using the `UESCAPE` clause after the string, for example:

```
U&"d!0061t!+000061" UESCAPE '!'
```

The escape character can be any single character other than a hexadecimal digit, the plus sign, a single quote, a double quote, or a whitespace character. Note that the escape character is written in single quotes, not double quotes, after `UESCAPE`.

To include the escape character in the identifier literally, write it twice.

Either the 4-digit or the 6-digit escape form can be used to specify UTF-16 surrogate pairs to compose characters with code points larger than U+FFFF, although the availability of the 6-digit form technically makes this unnecessary. (Surrogate pairs are not stored directly, but are combined into a single code point.)

If the server encoding is not UTF-8, the Unicode code point identified by one of these escape sequences is converted to the actual server encoding; an error is reported if that’s not possible.

Constants

There are three kinds of implicitly-typed constants in IvorySQL: strings, bit strings, and numbers. Constants can also be specified with explicit types, which can enable more accurate representation and more efficient handling by the system. These alternatives are discussed in the following subsections.

String Constants

A string constant in SQL is an arbitrary sequence of characters bounded by single quotes (`'`), for example

'This is a string'. To include a single-quote character within a string constant, write two adjacent single quotes, e.g., 'Dianne''s horse'. Note that this is not the same as a double-quote character (").

Two string constants that are only separated by whitespace with at least one newline are concatenated and effectively treated as if the string had been written as one constant. For example:

```
SELECT 'foo'
'bar';
```

is equivalent to:

```
SELECT 'foobar';
```

but:

```
SELECT 'foo'     'bar';
```

is not valid syntax. (This slightly bizarre behavior is specified by SQL; IvorySQL is following the standard.)

String Constants With C-Style Escapes

IvorySQL also accepts "escape" string constants, which are an extension to the SQL standard. An escape string constant is specified by writing the letter E (upper or lower case) just before the opening single quote, e.g., E' foo'. (When continuing an escape string constant across lines, write E only before the first opening quote.) Within an escape string, a backslash character (\) begins a C-like backslash escape sequence, in which the combination of backslash and following character(s) represent a special byte value.

Table 5.1. Backslash Escape Sequences

Backslash Escape Sequence	Interpretation
\b	backspace
\f	form feed
\n	newline
\r	carriage return
\`o*, \`\`oo*, \`\`ooo*` (o = 0–7)	octal byte value
\x*`h, `\x`hh` (h* = 0–9, A–F)	hexadecimal byte value
\u*`xxxx, `\U`xxxxxxxx` (x* = 0–9, A–F)	16 or 32-bit hexadecimal Unicode character value

Any other character following a backslash is taken literally. Thus, to include a backslash character, write two backslashes (\\). Also, a single quote can be included in an escape string by writing \', in addition to the normal way of ''.

It is your responsibility that the byte sequences you create, especially when using the octal or hexadecimal escapes, compose valid characters in the server character set encoding. A useful alternative is to use Unicode escapes or the alternative Unicode escape syntax, ; then the server will check that the character conversion is possible.

Caution

If the configuration parameter standard_conforming_strings is off, then IvorySQL recognizes

backslash escapes in both regular and escape string constants. However, as of IvorySQL, the default is `on`, meaning that backslash escapes are recognized only in escape string constants. This behavior is more standards-compliant, but might break applications which rely on the historical behavior, where backslash escapes were always recognized. As a workaround, you can set this parameter to `off`, but it is better to migrate away from using backslash escapes. If you need to use a backslash escape to represent a special character, write the string constant with an `E`.In addition to `standard_conforming_strings`, the configuration parameters escape_string_warning and backslash_quote govern treatment of backslashes in string constants.The character with the code zero cannot be in a string constant.

String Constants With Unicode Escapes

IvorySQL also supports another type of escape syntax for strings that allows specifying arbitrary Unicode characters by code point. A Unicode escape string constant starts with `U&` (upper or lower case letter U followed by ampersand) immediately before the opening quote, without any spaces in between, for example `U&'foo'`. (Note that this creates an ambiguity with the operator `&`. Use spaces around the operator to avoid this problem.) Inside the quotes, Unicode characters can be specified in escaped form by writing a backslash followed by the four-digit hexadecimal code point number or alternatively a backslash followed by a plus sign followed by a six-digit hexadecimal code point number. For example, the string `'data'` could be written as

```
U&'d\0061t\+000061'
```

The following less trivial example writes the Russian word “slon” (elephant) in Cyrillic letters:

```
U&'\0441\043B\043E\043D'
```

If a different escape character than backslash is desired, it can be specified using the `UESCAPE` clause after the string, for example:

```
U&'d!0061t!+000061' UESCAPE '!'
```

The escape character can be any single character other than a hexadecimal digit, the plus sign, a single quote, a double quote, or a whitespace character.

To include the escape character in the string literally, write it twice.

Either the 4-digit or the 6-digit escape form can be used to specify UTF-16 surrogate pairs to compose characters with code points larger than U+FFFF, although the availability of the 6-digit form technically makes this unnecessary. (Surrogate pairs are not stored directly, but are combined into a single code point.)

If the server encoding is not UTF-8, the Unicode code point identified by one of these escape sequences is converted to the actual server encoding; an error is reported if that’ s not possible.

Also, the Unicode escape syntax for string constants only works when the configuration parameter standard_conforming_strings is turned on. This is because otherwise this syntax could confuse clients that parse the SQL statements to the point that it could lead to SQL injections and similar security issues. If the parameter is set to off, this syntax will be rejected with an error message.

Dollar-Quoted String Constants

While the standard syntax for specifying string constants is usually convenient, it can be difficult to understand when the desired string contains many single quotes or backslashes, since each of those must be doubled. To allow more readable queries in such situations, IvorySQL provides another way, called “dollar quoting”, to write string constants. A dollar-quoted string constant consists of a dollar sign (`$`), an optional “tag” of zero or more characters, another dollar sign, an arbitrary sequence of characters that

makes up the string content, a dollar sign, the same tag that began this dollar quote, and a dollar sign. For example, here are two different ways to specify the string “Dianne’s horse” using dollar quoting:

```
$$Dianne's horse$$
$SomeTag$Dianne's horse$SomeTag$
```

Notice that inside the dollar-quoted string, single quotes can be used without needing to be escaped. Indeed, no characters inside a dollar-quoted string are ever escaped: the string content is always written literally. Backslashes are not special, and neither are dollar signs, unless they are part of a sequence matching the opening tag.

It is possible to nest dollar-quoted string constants by choosing different tags at each nesting level. This is most commonly used in writing function definitions. For example:

```
$function$
BEGIN
    RETURN ($1 ~ $q$[\t\r\n\v\\]$q$);
END;
$function$
```

Here, the sequence `$q$[\t\r\n\v\\]$q$` represents a dollar-quoted literal string `[\t\r\n\v\\]`, which will be recognized when the function body is executed by IvorySQL. But since the sequence does not match the outer dollar quoting delimiter `$function$`, it is just some more characters within the constant so far as the outer string is concerned.

The tag, if any, of a dollar-quoted string follows the same rules as an unquoted identifier, except that it cannot contain a dollar sign. Tags are case sensitive, so `$tag$String content$tag$` is correct, but `$TAG$String content$tag$` is not.

A dollar-quoted string that follows a keyword or identifier must be separated from it by whitespace; otherwise the dollar quoting delimiter would be taken as part of the preceding identifier.

Dollar quoting is not part of the SQL standard, but it is often a more convenient way to write complicated string literals than the standard-compliant single quote syntax. It is particularly useful when representing string constants inside other constants, as is often needed in procedural function definitions. With single-quote syntax, each backslash in the above example would have to be written as four backslashes, which would be reduced to two backslashes in parsing the original string constant, and then to one when the inner string constant is re-parsed during function execution.

Bit-String Constants

Bit-string constants look like regular string constants with a `B` (upper or lower case) immediately before the opening quote (no intervening whitespace), e.g., `B'1001'`. The only characters allowed within bit-string constants are `0` and `1`.

Alternatively, bit-string constants can be specified in hexadecimal notation, using a leading `X` (upper or lower case), e.g., `X'1FF'`. This notation is equivalent to a bit-string constant with four binary digits for each hexadecimal digit.

Both forms of bit-string constant can be continued across lines in the same way as regular string constants. Dollar quoting cannot be used in a bit-string constant.

Numeric Constants

Numeric constants are accepted in these general forms:

```
digits
digits.[digits][e[+-]digits]
[digits].digits[e[+-]digits]
digitse[+-]digits
```

where **digits** is one or more decimal digits (0 through 9). At least one digit must be before or after the decimal point, if one is used. At least one digit must follow the exponent marker (**e**), if one is present. There cannot be any spaces or other characters embedded in the constant. Note that any leading plus or minus sign is not actually considered part of the constant; it is an operator applied to the constant.

These are some examples of valid numeric constants:

```
42 3.5 4. .001 5e2 1.925e-3
```

A numeric constant that contains neither a decimal point nor an exponent is initially presumed to be type **integer** if its value fits in type **integer** (32 bits); otherwise it is presumed to be type **bigint** if its value fits in type **bigint** (64 bits); otherwise it is taken to be type **numeric**. Constants that contain decimal points and/or exponents are always initially presumed to be type **numeric**.

The initially assigned data type of a numeric constant is just a starting point for the type resolution algorithms. In most cases the constant will be automatically coerced to the most appropriate type depending on context. When necessary, you can force a numeric value to be interpreted as a specific data type by casting it. For example, you can force a numeric value to be treated as type **real** (**float4**) by writing:

```
REAL '1.23'  -- string style
1.23::REAL   -- IvorySQL (historical) style
```

These are actually just special cases of the general casting notations discussed next.

Constants Of Other Types

A constant of an arbitrary type can be entered using any one of the following notations:

```
type 'string'
'string'::type
CAST ( 'string' AS type )
```

The string constant’s text is passed to the input conversion routine for the type called **type**. The result is a constant of the indicated type. The explicit type cast can be omitted if there is no ambiguity as to the type the constant must be (for example, when it is assigned directly to a table column), in which case it is automatically coerced.

The string constant can be written using either regular SQL notation or dollar-quoting.

It is also possible to specify a type coercion using a function-like syntax:

```
typename ( 'string' )
```

but not all type names can be used in this way.

The `::`, `CAST()`, and function-call syntaxes can also be used to specify run-time type conversions of arbitrary expressions. To avoid syntactic ambiguity, the `` `type 'string'` `` syntax can only be used to specify the type of a simple literal constant. Another restriction on the `` `type 'string'` `` syntax is that it does not work for array types; use `::` or `CAST()` to specify the type of an array constant.

The `CAST()` syntax conforms to SQL. The `` `type 'string'` `` syntax is a generalization of the standard: SQL specifies this syntax only for a few data types, but IvorySQL allows it for all types. The syntax with `::` is historical IvorySQL usage, as is the function-call syntax.

Operators

An operator name is a sequence of up to `NAMEDATALEN`-1 (63 by default) characters from the following list:

\+ - * / < > = ~ ! @ # % ^ & | ` ?

There are a few restrictions on operator names, however:

- `--` and `/*` cannot appear anywhere in an operator name, since they will be taken as the start of a comment.
- A multiple-character operator name cannot end in `+` or `-`, unless the name also contains at least one of these characters:

```
~ ! @ # % ^ & | ` ?
```

For example, `@-` is an allowed operator name, but `*-` is not. This restriction allows IvorySQL to parse SQL-compliant queries without requiring spaces between tokens.

When working with non-SQL-standard operator names, you will usually need to separate adjacent operators with spaces to avoid ambiguity. For example, if you have defined a prefix operator named `@`, you cannot write `X*@Y`; you must write `X* @Y` to ensure that IvorySQL reads it as two operator names not one.

Special Characters

Some characters that are not alphanumeric have a special meaning that is different from being an operator. Details on the usage can be found at the location where the respective syntax element is described. This section only exists to advise the existence and summarize the purposes of these characters.

- A dollar sign (`$`) followed by digits is used to represent a positional parameter in the body of a function definition or a prepared statement. In other contexts the dollar sign can be part of an identifier or a dollar-quoted string constant.
- Parentheses (`()`) have their usual meaning to group expressions and enforce precedence. In some cases parentheses are required as part of the fixed syntax of a particular SQL command.
- Brackets (`[]`) are used to select the elements of an array.
- Commas (`,`) are used in some syntactical constructs to separate the elements of a list.
- The semicolon (`;`) terminates an SQL command. It cannot appear anywhere within a command, except within a string constant or quoted identifier.
- The colon (`:`) is used to select “slices” from arrays. In certain SQL dialects (such as Embedded SQL), the colon is used to prefix variable names.
- The asterisk (`*`) is used in some contexts to denote all the fields of a table row or composite value. It also has a special meaning when used as the argument of an aggregate function, namely that the aggregate does not require any explicit parameter.
- The period (`.`) is used in numeric constants, and to separate schema, table, and column names.

Comments

A comment is a sequence of characters beginning with double dashes and extending to the end of the line, e.g.:

```
-- This is a standard SQL comment
```

Alternatively, C-style block comments can be used:

```
/* multiline comment
 * with nesting: /* nested block comment */
 */
```

where the comment begins with `/ and extends to the matching occurrence of/`. These block comments nest, as specified in the SQL standard but unlike C, so that one can comment out larger blocks of code that might contain existing block comments.

A comment is removed from the input stream before further syntax analysis and is effectively replaced by whitespace.

Operator Precedence

Table 5.2 shows the precedence and associativity of the operators in IvorySQL. Most operators have the same precedence and are left-associative. The precedence and associativity of the operators is hard-wired into the parser. Add parentheses if you want an expression with multiple operators to be parsed in some other way than what the precedence rules imply.

Table 5.2. Operator Precedence (highest to lowest)

Operator/Element	Associativity	Description
`.`	left	table/column name separator
`::`	left	IvorySQL-style typecast
`[ ]`	left	array element selection
`+ -`	right	unary plus, unary minus
`^`	left	exponentiation
`* / %`	left	multiplication, division, modulo
`+ -`	left	addition, subtraction
(any other operator)	left	all other native and user-defined operators
`BETWEEN IN LIKE ILIKE SIMILAR`		range containment, set membership, string matching
`< > = <= >= <>`		comparison operators
`IS ISNULL NOTNULL`		`IS TRUE`, `IS FALSE`, `IS NULL`, `IS DISTINCT FROM`, etc.
`NOT`	right	logical negation
`AND`	left	logical conjunction
`OR`	left	logical disjunction

Note that the operator precedence rules also apply to user-defined operators that have the same names as the built-in operators mentioned above. For example, if you define a “” operator for some custom data

type it will have the same precedence as the built-in "" operator, no matter what yours does.

When a schema-qualified operator name is used in the OPERATOR syntax, as for example in:

```
SELECT 3 OPERATOR(pg_catalog.+) 4;
```

the OPERATOR construct is taken to have the default precedence shown in Table 5.2 for "any other operator". This is true no matter which specific operator appears inside OPERATOR().

> **Note**
>
> In particular, <= >= and <> used to be treated as generic operators; IS tests used to have higher priority; and NOT BETWEEN and related constructs acted inconsistently, being taken in some cases as having the precedence of NOT rather than BETWEEN. These rules were changed for better compliance with the SQL standard and to reduce confusion from inconsistent treatment of logically equivalent constructs. In most cases, these changes will result in no behavioral change, or perhaps in "no such operator" failures which can be resolved by adding parentheses. However there are corner cases in which a query might change behavior without any parsing error being reported.

Value Expressions

Value expressions are used in a variety of contexts, such as in the target list of the SELECT command, as new column values in INSERT or UPDATE, or in search conditions in a number of commands. The result of a value expression is sometimes called a scalar, to distinguish it from the result of a table expression (which is a table). Value expressions are therefore also called scalar expressions (or even simply expressions). The expression syntax allows the calculation of values from primitive parts using arithmetic, logical, set, and other operations.

A value expression is one of the following:

- A constant or literal value
- A column reference
- A positional parameter reference, in the body of a function definition or prepared statement
- A subscripted expression
- A field selection expression
- An operator invocation
- A function call
- An aggregate expression
- A window function call
- A type cast
- A collation expression
- A scalar subquery
- An array constructor
- A row constructor
- Another value expression in parentheses (used to group subexpressions and override precedence)

In addition to this list, there are a number of constructs that can be classified as an expression but do not follow any general syntax rules. These generally have the semantics of a function or operator . An example is the IS NULL clause.

A column can be referenced in the form:

```
correlation.columnname
```

`correlation` is the name of a table (possibly qualified with a schema name), or an alias for a table defined by means of a `FROM` clause. The correlation name and separating dot can be omitted if the column name is unique across all the tables being used in the current query.

Positional Parameters

A positional parameter reference is used to indicate a value that is supplied externally to an SQL statement. Parameters are used in SQL function definitions and in prepared queries. Some client libraries also support specifying data values separately from the SQL command string, in which case parameters are used to refer to the out-of-line data values. The form of a parameter reference is:

```
$number
```

For example, consider the definition of a function, `dept`, as:

```
CREATE FUNCTION dept(text) RETURNS dept
    AS $$ SELECT * FROM dept WHERE name = $1 $$
    LANGUAGE SQL;
```

Here the `$1` references the value of the first function argument whenever the function is invoked.

Subscripts

If an expression yields a value of an array type, then a specific element of the array value can be extracted by writing

```
expression[subscript]
```

or multiple adjacent elements (an “array slice”) can be extracted by writing

```
expression[lower_subscript:upper_subscript]
```

(Here, the brackets `[ ]` are meant to appear literally.) Each `subscript` is itself an expression, which will be rounded to the nearest integer value.

In general the array `expression` must be parenthesized, but the parentheses can be omitted when the expression to be subscripted is just a column reference or positional parameter. Also, multiple subscripts can be concatenated when the original array is multidimensional. For example:

```
mytable.arraycolumn[4]
mytable.two_d_column[17][34]
$1[10:42]
```

```
(arrayfunction(a,b))[42]
```

The parentheses in the last example are required.

Field Selection

If an expression yields a value of a composite type (row type), then a specific field of the row can be extracted by writing

```
expression.fieldname
```

In general the row `expression` must be parenthesized, but the parentheses can be omitted when the expression to be selected from is just a table reference or positional parameter. For example:

```
mytable.mycolumn
$1.somecolumn
(rowfunction(a,b)).col3
```

(Thus, a qualified column reference is actually just a special case of the field selection syntax.) An important special case is extracting a field from a table column that is of a composite type:

```
(compositecol).somefield
(mytable.compositecol).somefield
```

The parentheses are required here to show that `compositecol` is a column name not a table name, or that `mytable` is a table name not a schema name in the second case.

You can ask for all fields of a composite value by writing `.*`:

```
(compositecol).*
```

This notation behaves differently depending on context.

Operator Invocations

There are two possible syntaxes for an operator invocation:

`expression operator expression` (binary infix operator)
`operator expression` (unary prefix operator)

where the `operator` token follows the syntax rules , or is one of the key words `AND`, `OR`, and `NOT`, or is a qualified operator name in the form:

```
OPERATOR(schema.operatorname)
```

Which particular operators exist and whether they are unary or binary depends on what operators have been defined by the system or the user.

Function Calls

The syntax for a function call is the name of a function (possibly qualified with a schema name), followed by its argument list enclosed in parentheses:

```
function_name ([expression [, expression ... ]] )
```

For example, the following computes the square root of 2:

```
sqrt(2)
```

```
Other functions can be added by the user.
```

When issuing queries in a database where some users mistrust other users,

The arguments can optionally have names attached.

> **Note**
>
> A function that takes a single argument of composite type can optionally be called using field-selection syntax, and conversely field selection can be written in functional style. That is, the notations `col(table)` and `table.col` are interchangeable. This behavior is not SQL-standard but is provided in IvorySQL because it allows use of functions to emulate “computed fields”.

Aggregate Expressions

An aggregate expression represents the application of an aggregate function across the rows selected by a query. An aggregate function reduces multiple inputs to a single output value, such as the sum or average of the inputs. The syntax of an aggregate expression is one of the following:

```
aggregate_name (expression [ , ... ] [ order_by_clause ] ) [ FILTER ( WHERE
filter_clause ) ]
aggregate_name (ALL expression [ , ... ] [ order_by_clause ] ) [ FILTER ( WHERE
filter_clause ) ]
aggregate_name (DISTINCT expression [ , ... ] [ order_by_clause ] ) [ FILTER ( WHERE
filter_clause ) ]
aggregate_name ( * ) [ FILTER ( WHERE filter_clause ) ]
aggregate_name ( [ expression [ , ... ] ] ) WITHIN GROUP ( order_by_clause ) [ FILTER
( WHERE filter_clause ) ]
```

where `aggregate_name` is a previously defined aggregate (possibly qualified with a schema name) and `expression` is any value expression that does not itself contain an aggregate expression or a window function call. The optional `order_by_clause` and `filter_clause` are described below.

The first form of aggregate expression invokes the aggregate once for each input row. The second form is the same as the first, since `ALL` is the default. The third form invokes the aggregate once for each distinct value of the expression (or distinct set of values, for multiple expressions) found in the input rows. The fourth form invokes the aggregate once for each input row; since no particular input value is specified, it is generally only useful for the `count() aggregate function. The last form is used with *ordered-set` aggregate functions, which are described below.

Most aggregate functions ignore null inputs, so that rows in which one or more of the expression(s) yield null are discarded. This can be assumed to be true, unless otherwise specified, for all built-in aggregates.

For example, `count(*)` yields the total number of input rows; `count(f1)` yields the number of input rows in which `f1` is non-null, since `count` ignores nulls; and `count(distinct f1)` yields the number of distinct non-null values of `f1`.

Ordinarily, the input rows are fed to the aggregate function in an unspecified order. In many cases this does not matter; for example, `min` produces the same result no matter what order it receives the inputs in. However, some aggregate functions (such as `array_agg` and `string_agg`) produce results that depend on the ordering of the input rows. When using such an aggregate, the optional `order_by_clause` can be used to specify the desired ordering. The `order_by_clause` has the same syntax as for a query-level `ORDER BY` clause, except that its expressions are always just expressions and cannot be output-column names or numbers. For example:

```
SELECT array_agg(a ORDER BY b DESC) FROM table;
```

When dealing with multiple-argument aggregate functions, note that the `ORDER BY` clause goes after all the aggregate arguments. For example, write this:

```
SELECT string_agg(a, ',' ORDER BY a) FROM table;
```

not this:

```
SELECT string_agg(a ORDER BY a, ',') FROM table;  -- incorrect
```

The latter is syntactically valid, but it represents a call of a single-argument aggregate function with two `ORDER BY` keys (the second one being rather useless since it's a constant).

If `DISTINCT` is specified in addition to an `order_by_clause`, then all the `ORDER BY` expressions must match regular arguments of the aggregate; that is, you cannot sort on an expression that is not included in the `DISTINCT` list.

> **Note**
>
> The ability to specify both `DISTINCT` and `ORDER BY` in an aggregate function is a IvorySQL extension.

Placing `ORDER BY` within the aggregate's regular argument list, as described so far, is used when ordering the input rows for general-purpose and statistical aggregates, for which ordering is optional. There is a subclass of aggregate functions called ordered-set aggregates for which an `order_by_clause` is required, usually because the aggregate's computation is only sensible in terms of a specific ordering of its input rows. Typical examples of ordered-set aggregates include rank and percentile calculations. For an ordered-set aggregate, the `order_by_clause` is written inside `WITHIN GROUP (…)`, as shown in the final syntax alternative above. The expressions in the `order_by_clause` are evaluated once per input row just like regular aggregate arguments, sorted as per the `order_by_clause`'s requirements, and fed to the aggregate function as input arguments. (This is unlike the case for a non-`WITHIN GROUP` `order_by_clause`, which is not treated as argument(s) to the aggregate function.) The argument expressions preceding `WITHIN GROUP`, if any, are called direct arguments to distinguish them from the aggregated arguments listed in the `order_by_clause`. Unlike regular aggregate arguments, direct arguments are evaluated only once per aggregate call, not once per input row. This means that they can contain variables only if those variables are grouped by `GROUP BY`; this restriction is the same as if the direct arguments were not inside an aggregate expression at all. Direct arguments are typically used for things like percentile fractions, which only make sense as a single value per aggregation calculation. The direct argument list can be empty; in this case, write just `()` not `(*)`. (IvorySQL will actually accept either spelling, but only the first way conforms to the SQL standard.)

An example of an ordered-set aggregate call is:

```
SELECT percentile_cont(0.5) WITHIN GROUP (ORDER BY income) FROM households;
 percentile_cont
-----------------
           50489
```

which obtains the 50th percentile, or median, value of the `income` column from table `households`. Here, `0.5` is a direct argument; it would make no sense for the percentile fraction to be a value varying across rows.

If `FILTER` is specified, then only the input rows for which the `filter_clause` evaluates to true are fed to the aggregate function; other rows are discarded. For example:

```
SELECT
    count(*) AS unfiltered,
    count(*) FILTER (WHERE i < 5) AS filtered
FROM generate_series(1,10) AS s(i);
 unfiltered | filtered
------------+----------
         10 |        4
(1 row)
```

Other aggregate functions can be added by the user.

An aggregate expression can only appear in the result list or `HAVING` clause of a `SELECT` command. It is forbidden in other clauses, such as `WHERE`, because those clauses are logically evaluated before the results of aggregates are formed.

When an aggregate expression appears in a subquery,the aggregate is normally evaluated over the rows of the subquery. But an exception occurs if the aggregate's arguments (and `filter_clause` if any) contain only outer-level variables: the aggregate then belongs to the nearest such outer level, and is evaluated over the rows of that query. The aggregate expression as a whole is then an outer reference for the subquery it appears in, and acts as a constant over any one evaluation of that subquery. The restriction about appearing only in the result list or `HAVING` clause applies with respect to the query level that the aggregate belongs to.

Window Function Calls

A window function call represents the application of an aggregate-like function over some portion of the rows selected by a query. Unlike non-window aggregate calls, this is not tied to grouping of the selected rows into a single output row — each row remains separate in the query output. However the window function has access to all the rows that would be part of the current row's group according to the grouping specification (`PARTITION BY` list) of the window function call. The syntax of a window function call is one of the following:

```
function_name ([expression [, expression ... ]]) [ FILTER ( WHERE filter_clause ) ]
OVER window_name
function_name ([expression [, expression ... ]]) [ FILTER ( WHERE filter_clause ) ]
OVER ( window_definition )
function_name ( * ) [ FILTER ( WHERE filter_clause ) ] OVER window_name
function_name ( * ) [ FILTER ( WHERE filter_clause ) ] OVER ( window_definition )
```

where window_definition has the syntax

```
[ existing_window_name ]
[ PARTITION BY expression [, ...] ]
[ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [,
...] ]
[ frame_clause ]
```

The optional frame_clause can be one of

```
{ RANGE | ROWS | GROUPS } frame_start [ frame_exclusion ]
{ RANGE | ROWS | GROUPS } BETWEEN frame_start AND frame_end [ frame_exclusion ]
```

where frame_start and frame_end can be one of

```
UNBOUNDED PRECEDING
offset PRECEDING
CURRENT ROW
offset FOLLOWING
UNBOUNDED FOLLOWING
```

and frame_exclusion can be one of

```
EXCLUDE CURRENT ROW
EXCLUDE GROUP
EXCLUDE TIES
EXCLUDE NO OTHERS
```

Here, expression represents any value expression that does not itself contain window function calls.

window_name is a reference to a named window specification defined in the query's WINDOW clause. Alternatively, a full window_definition can be given within parentheses, using the same syntax as for defining a named window in the WINDOW clause; see the SELECT reference page for details. It's worth pointing out that OVER wname is not exactly equivalent to OVER (wname ···); the latter implies copying and modifying the window definition, and will be rejected if the referenced window specification includes a frame clause.

The PARTITION BY clause groups the rows of the query into partitions, which are processed separately by the window function. PARTITION BY works similarly to a query-level GROUP BY clause, except that its expressions are always just expressions and cannot be output-column names or numbers. Without PARTITION BY, all rows produced by the query are treated as a single partition. The ORDER BY clause determines the order in which the rows of a partition are processed by the window function. It works similarly to a query-level ORDER BY clause, but likewise cannot use output-column names or numbers. Without ORDER BY, rows are processed in an unspecified order.

The frame_clause specifies the set of rows constituting the window frame, which is a subset of the current partition, for those window functions that act on the frame instead of the whole partition. The set of rows in the frame can vary depending on which row is the current row. The frame can be specified in RANGE, ROWS or GROUPS mode; in each case, it runs from the frame_start to the frame_end. If frame_end is omitted, the end defaults to CURRENT ROW.

A `frame_start` of `UNBOUNDED PRECEDING` means that the frame starts with the first row of the partition, and similarly a `frame_end` of `UNBOUNDED FOLLOWING` means that the frame ends with the last row of the partition.

In `RANGE` or `GROUPS` mode, a `frame_start` of `CURRENT ROW` means the frame starts with the current row's first peer row (a row that the window's `ORDER BY` clause sorts as equivalent to the current row), while a `frame_end` of `CURRENT ROW` means the frame ends with the current row's last peer row. In `ROWS` mode, `CURRENT ROW` simply means the current row.

In the `offset PRECEDING` and `offset FOLLOWING` frame options, the `offset` must be an expression not containing any variables, aggregate functions, or window functions. The meaning of the `offset` depends on the frame mode:

- In `ROWS` mode, the `offset` must yield a non-null, non-negative integer, and the option means that the frame starts or ends the specified number of rows before or after the current row.
- In `GROUPS` mode, the `offset` again must yield a non-null, non-negative integer, and the option means that the frame starts or ends the specified number of peer groups before or after the current row's peer group, where a peer group is a set of rows that are equivalent in the `ORDER BY` ordering. (There must be an `ORDER BY` clause in the window definition to use `GROUPS` mode.)
- In `RANGE` mode, these options require that the `ORDER BY` clause specify exactly one column. The `offset` specifies the maximum difference between the value of that column in the current row and its value in preceding or following rows of the frame. The data type of the `offset` expression varies depending on the data type of the ordering column. For numeric ordering columns it is typically of the same type as the ordering column, but for datetime ordering columns it is an `interval`. For example, if the ordering column is of type `date` or `timestamp`, one could write `RANGE BETWEEN '1 day' PRECEDING AND '10 days' FOLLOWING`. The `offset` is still required to be non-null and non-negative, though the meaning of "non-negative" depends on its data type.

In any case, the distance to the end of the frame is limited by the distance to the end of the partition, so that for rows near the partition ends the frame might contain fewer rows than elsewhere.

Notice that in both `ROWS` and `GROUPS` mode, `0 PRECEDING` and `0 FOLLOWING` are equivalent to `CURRENT ROW`. This normally holds in `RANGE` mode as well, for an appropriate data-type-specific meaning of "zero".

The `frame_exclusion` option allows rows around the current row to be excluded from the frame, even if they would be included according to the frame start and frame end options. `EXCLUDE CURRENT ROW` excludes the current row from the frame. `EXCLUDE GROUP` excludes the current row and its ordering peers from the frame. `EXCLUDE TIES` excludes any peers of the current row from the frame, but not the current row itself. `EXCLUDE NO OTHERS` simply specifies explicitly the default behavior of not excluding the current row or its peers.

The default framing option is `RANGE UNBOUNDED PRECEDING`, which is the same as `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`. With `ORDER BY`, this sets the frame to be all rows from the partition start up through the current row's last `ORDER BY` peer. Without `ORDER BY`, this means all rows of the partition are included in the window frame, since all rows become peers of the current row.

Restrictions are that `frame_start` cannot be `UNBOUNDED FOLLOWING`, `frame_end` cannot be `UNBOUNDED PRECEDING`, and the `frame_end` choice cannot appear earlier in the above list of `frame_start` and `frame_end` options than the `frame_start` choice does — for example `RANGE BETWEEN CURRENT ROW AND `offset PRECEDING` is not allowed. But, for example, `ROWS BETWEEN 7 PRECEDING AND 8 PRECEDING` is allowed, even though it would never select any rows.

If `FILTER` is specified, then only the input rows for which the `filter_clause` evaluates to true are fed to the window function; other rows are discarded. Only window functions that are aggregates accept a `FILTER` clause.

Other window functions can be added by the user. Also, any built-in or user-defined general-purpose or statistical aggregate can be used as a window function. (Ordered-set and hypothetical-set aggregates cannot presently be used as window functions.)

The syntaxes using `are used for calling parameter-less aggregate functions as window functions, for example count() OVER (PARTITION BY x ORDER BY y)`. The asterisk (`*`) is customarily not used for window-specific functions. Window-specific functions do not allow `DISTINCT` or `ORDER BY` to be used within the function argument list.

Window function calls are permitted only in the `SELECT` list and the `ORDER BY` clause of the query.

Type Casts

A type cast specifies a conversion from one data type to another. IvorySQL accepts two equivalent syntaxes for type casts:

```
CAST ( expression AS type )
expression::type
```

The `CAST` syntax conforms to SQL; the syntax with `::` is historical IvorySQL usage.

When a cast is applied to a value expression of a known type, it represents a run-time type conversion. The cast will succeed only if a suitable type conversion operation has been defined. Notice that this is subtly different from the use of casts with constants. A cast applied to an unadorned string literal represents the initial assignment of a type to a literal constant value, and so it will succeed for any type (if the contents of the string literal are acceptable input syntax for the data type).

An explicit type cast can usually be omitted if there is no ambiguity as to the type that a value expression must produce (for example, when it is assigned to a table column); the system will automatically apply a type cast in such cases. However, automatic casting is only done for casts that are marked "OK to apply implicitly" in the system catalogs. Other casts must be invoked with explicit casting syntax. This restriction is intended to prevent surprising conversions from being applied silently.

It is also possible to specify a type cast using a function-like syntax:

```
typename ( expression )
```

However, this only works for types whose names are also valid as function names. For example, `double precision` cannot be used this way, but the equivalent `float8` can. Also, the names `interval`, `time`, and `timestamp` can only be used in this fashion if they are double-quoted, because of syntactic conflicts. Therefore, the use of the function-like cast syntax leads to inconsistencies and should probably be avoided.

> **Note**
>
> The function-like syntax is in fact just a function call. When one of the two standard cast syntaxes is used to do a run-time conversion, it will internally invoke a registered function to perform the conversion. By convention, these conversion functions have the same name as their output type, and thus the "function-like syntax" is nothing more than a direct invocation of the underlying conversion function. Obviously, this is not something that a portable application should rely on. For further details see CREATE CAST.

Collation Expressions

The `COLLATE` clause overrides the collation of an expression. It is appended to the expression it applies to:

```
expr COLLATE collation
```

where `collation` is a possibly schema-qualified identifier. The `COLLATE` clause binds tighter than operators; parentheses can be used when necessary.

If no collation is explicitly specified, the database system either derives a collation from the columns involved in the expression, or it defaults to the default collation of the database if no column is involved in the expression.

The two common uses of the COLLATE clause are overriding the sort order in an ORDER BY clause, for example:

```
SELECT a, b, c FROM tbl WHERE ... ORDER BY a COLLATE "C";
```

and overriding the collation of a function or operator call that has locale-sensitive results, for example:

```
SELECT * FROM tbl WHERE a > 'foo' COLLATE "C";
```

Note that in the latter case the COLLATE clause is attached to an input argument of the operator we wish to affect. It doesn't matter which argument of the operator or function call the COLLATE clause is attached to, because the collation that is applied by the operator or function is derived by considering all arguments, and an explicit COLLATE clause will override the collations of all other arguments. (Attaching non-matching COLLATE clauses to more than one argument, however, is an error.) Thus, this gives the same result as the previous example:

```
SELECT * FROM tbl WHERE a COLLATE "C" > 'foo';
```

But this is an error:

```
SELECT * FROM tbl WHERE (a > 'foo') COLLATE "C";
```

because it attempts to apply a collation to the result of the > operator, which is of the non-collatable data type boolean.

Scalar Subqueries

A scalar subquery is an ordinary SELECT query in parentheses that returns exactly one row with one column. The SELECT query is executed and the single returned value is used in the surrounding value expression. It is an error to use a query that returns more than one row or more than one column as a scalar subquery. (But if, during a particular execution, the subquery returns no rows, there is no error; the scalar result is taken to be null.) The subquery can refer to variables from the surrounding query, which will act as constants during any one evaluation of the subquery.

For example, the following finds the largest city population in each state:

```
SELECT name, (SELECT max(pop) FROM cities WHERE cities.state = states.name)
    FROM states;
```

Array Constructors

An array constructor is an expression that builds an array value using values for its member elements. A simple array constructor consists of the key word ARRAY, a left square bracket [, a list of expressions (separated by commas) for the array element values, and finally a right square bracket]. For example:

```
SELECT ARRAY[1,2,3+4];
  array
---------
 {1,2,7}
```

```
(1 row)
```

By default, the array element type is the common type of the member expressions, determined using the same rules as for UNION or CASE constructs. You can override this by explicitly casting the array constructor to the desired type, for example:

```
SELECT ARRAY[1,2,22.7]::integer[];
  array
----------
 {1,2,23}
(1 row)
```

This has the same effect as casting each expression to the array element type individually.

Multidimensional array values can be built by nesting array constructors. In the inner constructors, the key word ARRAY can be omitted. For example, these produce the same result:

```
SELECT ARRAY[ARRAY[1,2], ARRAY[3,4]];
     array
---------------
 {{1,2},{3,4}}
(1 row)

SELECT ARRAY[[1,2],[3,4]];
     array
---------------
 {{1,2},{3,4}}
(1 row)
```

Since multidimensional arrays must be rectangular, inner constructors at the same level must produce sub-arrays of identical dimensions. Any cast applied to the outer ARRAY constructor propagates automatically to all the inner constructors.

Multidimensional array constructor elements can be anything yielding an array of the proper kind, not only a sub-ARRAY construct. For example:

```
CREATE TABLE arr(f1 int[], f2 int[]);

INSERT INTO arr VALUES (ARRAY[[1,2],[3,4]], ARRAY[[5,6],[7,8]]);

SELECT ARRAY[f1, f2, '{{9,10},{11,12}}'::int[]] FROM arr;
                     array
------------------------------------------------
 {{{1,2},{3,4}},{{5,6},{7,8}},{{9,10},{11,12}}}
(1 row)
```

You can construct an empty array, but since it's impossible to have an array with no type, you must explicitly cast your empty array to the desired type. For example:

```
SELECT ARRAY[]::integer[];
 array
-------
 {}
(1 row)
```

It is also possible to construct an array from the results of a subquery. In this form, the array constructor is written with the key word `ARRAY` followed by a parenthesized (not bracketed) subquery. For example:

```
SELECT ARRAY(SELECT oid FROM pg_proc WHERE proname LIKE 'bytea%');
                              array
---------------------------------------------------------------
 {2011,1954,1948,1952,1951,1244,1950,2005,1949,1953,2006,31,2412}
(1 row)

SELECT ARRAY(SELECT ARRAY[i, i*2] FROM generate_series(1,5) AS a(i));
              array
----------------------------------
 {{1,2},{2,4},{3,6},{4,8},{5,10}}
(1 row)
```

The subquery must return a single column. If the subquery's output column is of a non-array type, the resulting one-dimensional array will have an element for each row in the subquery result, with an element type matching that of the subquery's output column. If the subquery's output column is of an array type, the result will be an array of the same type but one higher dimension; in this case all the subquery rows must yield arrays of identical dimensionality, else the result would not be rectangular.

The subscripts of an array value built with `ARRAY` always begin with one.

Row Constructors

A row constructor is an expression that builds a row value (also called a composite value) using values for its member fields. A row constructor consists of the key word `ROW`, a left parenthesis, zero or more expressions (separated by commas) for the row field values, and finally a right parenthesis. For example:

```
SELECT ROW(1,2.5,'this is a test');
```

The key word `ROW` is optional when there is more than one expression in the list.

A row constructor can include the syntax `rowvalue.*`, which will be expanded to a list of the elements of the row value, just as occurs when the `.*` syntax is used at the top level of a `SELECT` list .For example, if table `t` has columns `f1` and `f2`, these are the same:

```
SELECT ROW(t.*, 42) FROM t;
SELECT ROW(t.f1, t.f2, 42) FROM t;
```

Note

Before PostgreSQL 8.2, the `.*` syntax was not expanded in row constructors, so that writing `ROW(t.*, 42)` created a two-field row whose first field was another row value. The new behavior is usually more useful. If you need the old behavior of nested row values, write the inner row value without `.*`, for instance `ROW(t, 42)`.

By default, the value created by a `ROW` expression is of an anonymous record type. If necessary, it can be cast to a named composite type — either the row type of a table, or a composite type created with `CREATE TYPE AS`. An explicit cast might be needed to avoid ambiguity. For example:

```
CREATE TABLE mytable(f1 int, f2 float, f3 text);

CREATE FUNCTION getf1(mytable) RETURNS int AS 'SELECT $1.f1' LANGUAGE SQL;

-- No cast needed since only one getf1() exists
SELECT getf1(ROW(1,2.5,'this is a test'));
 getf1
-------
     1
(1 row)

CREATE TYPE myrowtype AS (f1 int, f2 text, f3 numeric);

CREATE FUNCTION getf1(myrowtype) RETURNS int AS 'SELECT $1.f1' LANGUAGE SQL;

-- Now we need a cast to indicate which function to call:
SELECT getf1(ROW(1,2.5,'this is a test'));
ERROR:  function getf1(record) is not unique

SELECT getf1(ROW(1,2.5,'this is a test')::mytable);
 getf1
-------
     1
(1 row)

SELECT getf1(CAST(ROW(11,'this is a test',2.5) AS myrowtype));
 getf1
-------
    11
(1 row)
```

Row constructors can be used to build composite values to be stored in a composite-type table column, or to be passed to a function that accepts a composite parameter. Also, it is possible to compare two row values or test a row with `IS NULL` or `IS NOT NULL`, for example:

```
SELECT ROW(1,2.5,'this is a test') = ROW(1, 3, 'not the same');

SELECT ROW(table.*) IS NULL FROM table;  -- detect all-null rows
```

Expression Evaluation Rules

The order of evaluation of subexpressions is not defined. In particular, the inputs of an operator or function are not necessarily evaluated left-to-right or in any other fixed order.

Furthermore, if the result of an expression can be determined by evaluating only some parts of it, then other subexpressions might not be evaluated at all. For instance, if one wrote:

```
SELECT true OR somefunc();
```

then `somefunc()` would (probably) not be called at all. The same would be the case if one wrote:

```
SELECT somefunc() OR true;
```

Note that this is not the same as the left-to-right "short-circuiting" of Boolean operators that is found in some programming languages.

As a consequence, it is unwise to use functions with side effects as part of complex expressions. It is particularly dangerous to rely on side effects or evaluation order in `WHERE` and `HAVING` clauses, since those clauses are extensively reprocessed as part of developing an execution plan. Boolean expressions (`AND`/`OR`/`NOT` combinations) in those clauses can be reorganized in any manner allowed by the laws of Boolean algebra.

When it is essential to force evaluation order, a `CASE` construct can be used. For example, this is an untrustworthy way of trying to avoid division by zero in a `WHERE` clause:

```
SELECT ... WHERE x > 0 AND y/x > 1.5;
```

But this is safe:

```
SELECT ... WHERE CASE WHEN x > 0 THEN y/x > 1.5 ELSE false END;
```

A `CASE` construct used in this fashion will defeat optimization attempts, so it should only be done when necessary. (In this particular example, it would be better to sidestep the problem by writing `y > 1.5*x` instead.)

`CASE` is not a cure-all for such issues, however. One limitation of the technique illustrated above is that it does not prevent early evaluation of constant subexpressions. As described in Section 38.7, functions and operators marked `IMMUTABLE` can be evaluated when the query is planned rather than when it is executed. Thus for example

```
SELECT CASE WHEN x > 0 THEN x ELSE 1/0 END FROM tab;
```

is likely to result in a division-by-zero failure due to the planner trying to simplify the constant subexpression, even if every row in the table has `x > 0` so that the `ELSE` arm would never be entered at run time.

While that particular example might seem silly, related cases that don’ t obviously involve constants can occur in queries executed within functions, since the values of function arguments and local variables can be inserted into queries as constants for planning purposes. Within PL/pgSQL functions, for example, using an `IF-THEN-ELSE` statement to protect a risky computation is much safer than just nesting it in a `CASE` expression.

Another limitation of the same kind is that a `CASE` cannot prevent evaluation of an aggregate expression contained within it, because aggregate expressions are computed before other expressions in a `SELECT` list or `HAVING` clause are considered. For example, the following query can cause a division-by-zero error despite seemingly having protected against it:

```
SELECT CASE WHEN min(employees) > 0
            THEN avg(expenses / employees)
       END
    FROM departments;
```

The `min()` and `avg()` aggregates are computed concurrently over all the input rows, so if any row has `employees` equal to zero, the division-by-zero error will occur before there is any opportunity to test the result of `min()`. Instead, use a `WHERE` or `FILTER` clause to prevent problematic input rows from reaching an aggregate function in the first place.

Calling Functions

IvorySQL allows functions that have named parameters to be called using either positional or named notation. Named notation is especially useful for functions that have a large number of parameters, since it makes the associations between parameters and actual arguments more explicit and reliable. In positional notation, a function call is written with its argument values in the same order as they are defined in the function declaration. In named notation, the arguments are matched to the function parameters by name and can be written in any order.

In either notation, parameters that have default values given in the function declaration need not be written in the call at all. But this is particularly useful in named notation, since any combination of parameters can be omitted; while in positional notation parameters can only be omitted from right to left.

IvorySQL also supports mixed notation, which combines positional and named notation. In this case, positional parameters are written first and named parameters appear after them.

The following examples will illustrate the usage of all three notations, using the following function definition:

```
CREATE FUNCTION concat_lower_or_upper(a text, b text, uppercase boolean DEFAULT false)
RETURNS text
AS
$$
 SELECT CASE
        WHEN $3 THEN UPPER($1 || ' ' || $2)
        ELSE LOWER($1 || ' ' || $2)
        END;
$$
LANGUAGE SQL IMMUTABLE STRICT;
```

Function `concat_lower_or_upper` has two mandatory parameters, `a` and `b`. Additionally there is one optional parameter `uppercase` which defaults to `false`. The `a` and `b` inputs will be concatenated, and forced to either upper or lower case depending on the `uppercase` parameter. The remaining details of this function definition are not important here .

Using Positional Notation

Positional notation is the traditional mechanism for passing arguments to functions in IvorySQL. An example is:

```
SELECT concat_lower_or_upper('Hello', 'World', true);
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

All arguments are specified in order. The result is upper case since `uppercase` is specified as `true`. Another example is:

```
SELECT concat_lower_or_upper('Hello', 'World');
 concat_lower_or_upper
-----------------------
 hello world
(1 row)
```

Here, the `uppercase` parameter is omitted, so it receives its default value of `false`, resulting in lower case output. In positional notation, arguments can be omitted from right to left so long as they have defaults.

Using Named Notation

In named notation, each argument's name is specified using => to separate it from the argument expression. For example:

```
SELECT concat_lower_or_upper(a => 'Hello', b => 'World');
 concat_lower_or_upper
-----------------------
 hello world
(1 row)
```

Again, the argument `uppercase` was omitted so it is set to `false` implicitly. One advantage of using named notation is that the arguments may be specified in any order, for example:

```
SELECT concat_lower_or_upper(a => 'Hello', b => 'World', uppercase => true);
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)

SELECT concat_lower_or_upper(a => 'Hello', uppercase => true, b => 'World');
 concat_lower_or_upper
-----------------------
```

```
 HELLO WORLD
(1 row)
```

An older syntax based on ":=" is supported for backward compatibility:

```
SELECT concat_lower_or_upper(a := 'Hello', uppercase := true, b := 'World');
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

Using Mixed Notation

The mixed notation combines positional and named notation. However, as already mentioned, named arguments cannot precede positional arguments. For example:

```
SELECT concat_lower_or_upper('Hello', 'World', uppercase => true);
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

In the above query, the arguments `a` and `b` are specified positionally, while `uppercase` is specified by name. In this example, that adds little except documentation. With a more complex function having numerous parameters that have default values, named or mixed notation can save a great deal of writing and reduce chances for error.

> **Note**
>
> Named and mixed call notations currently cannot be used when calling an aggregate function (but they do work when an aggregate function is used as a window function).

Oracle Compatible Features

Changing tables

syntax

```
ALTER TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
action;

action:
    ADD ( add_coldef [, ... ] )
    | MODIFY ( modify_coldef [, ... ] )
    | DROP [ COLUMN ] ( column_name [, ... ] )
```

```
add_coldef:
    column_name data_type

modify_coldef:
    column_name data_type alter_using

alter_using:
    USING expression
```

parameters

`name` Table name. `column_name` Column name. `data_type` Column type. `expression` The value expression. `ADD keyword` Adds a column to the table, either one or more columns. `MODIFY keyword` Modify a column of the table, you can modify one or more columns. `DROP keyword` Deletes a column of a table, you can delete one or more columns. `USING keyword` Modifies the value of a column.

Example

```
ADD:
create table tb_test1(id int, flg char(10));

alter table tb_test1 add (name varchar);

ALTER TABLE tb_test1
    ADD adress varchar,
    ADD num int,
    ADD flg1 char;

\d tb_test1
                  Table "public.tb_test1"
 Column |      Type       | Collation | Nullable | Default
--------+-----------------+-----------+----------+---------
 id     | pg_catalog.int4 |           |          |
 flg    | char(10)        |           |          |
 name   | varchar2(4000)  |           |          |
 adress | varchar2(4000)  |           |          |
 num    | pg_catalog.int4 |           |          |
 flg1   | char(1)         |           |          |

MODIFY:
create table tb_test2(id int, flg char(10), num varchar);

insert into tb_test2 values('1', 2, '3');
```

```
ALTER TABLE tb_test2 ALTER COLUMN id TYPE char;

\d tb_test2
                Table "public.tb_test2"
 Column |      Type      | Collation | Nullable | Default
--------+----------------+-----------+----------+---------
 id     | char(1)        |           |          |
 flg    | char(10)       |           |          |
 num    | varchar2(4000) |           |          |

DROP:
create table tb_test3(id int, flg1 char(10), flg2 char(11), flg3 char(12), flg4
char(13),
                       flg5 char(14), flg6 char(15));

ALTER TABLE tb_test3 DROP id;

\d tb_test3
              Table "public.tb_test3"
 Column |   Type    | Collation | Nullable | Default
--------+-----------+-----------+----------+---------
 flg1   | char(10) |           |          |
 flg2   | char(11) |           |          |
 flg3   | char(12) |           |          |
 flg4   | char(13) |           |          |
 flg5   | char(14) |           |          |
 flg6   | char(15) |           |          |
```

Delete table

Syntax

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
DELETE [ FROM ] [ ONLY ] table_name [ * ] [ [ AS ] alias ]
    [ USING using_list ]
    [ WHERE condition | WHERE CURRENT OF cursor_name ]
    [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]
```

Parameters

`table_name` The name of the table. `alias` The table alias. `using_list` A list of table expressions that allow columns from other tables to appear in the WHERE condition. `condition` An expression that returns a boolean type value. `cursor_name` The name of the cursor to be used in the WHERE CURRENT OF case. `output_expression` An expression that is calculated by DELETE and returned after each row is deleted. `output_name` The name of the returned column.

uses

```
create table tb_test4(id int, flg char(10));

insert into tb_test4 values(1, '2'), (5, '6');

delete from tb_test4 where id = 1;

table tb_test4;
 id |    flg
----+------------
 5  | 6
(1 row)
```

Update table

Syntax

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
UPDATE [ ONLY ] table_name [ * ] [ [ AS ] alias ]
    SET { [ table_name | alias ] column_name = { expression | DEFAULT }
| ( [ table_name | alias ] column_name [, ...] ) = [ ROW ]    ( { expression | DEFAULT
} [, ...] )
| ( [ table_name | alias ] column_name [, ...] ) = ( sub-SELECT )
        } [, ...]
    [ FROM from_list ]
    [ WHERE condition | WHERE CURRENT OF cursor_name ]
    [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]
```

parameters

`table_name` Table name. `alias` Table alias. `column_name` Column name. `expression` Value expression. `sub-SELECT` select clause. `from_list` Table expression. `condition` An expression that returns a value of type boolean. `cursor_name` The name of the cursor to be used in the WHERE CURRENT OF case. `output_expression` An expression that is computed by DELETE and returned after each row is deleted. `output_name` The name of the column being returned.

Example

```
create table tb_test5(id int, flg char(10));

insert into tb_test5 values(1, '2'), (3, '4'), (5, '6');

update tb_test5 a set a.id = 33 where a.id = 3;
```

```
table tb_test5;
Id  |    flg
----+------------
  1 | 2
  5 | 6
 33 | 4
(3 rows)
```

GROUP BY

Example

```
set ivorysql.compatible_mode to oracle;

create table students(student_id varchar(20) primary key ,
student_name varchar(40),
student_pid int);

select student_id,student_name from students group by student_id;
ERROR:  column "students.student_name" must appear in the GROUP BY clause or be used
in an aggregate function
```

UNION

Example

```
SELECT 100 AS value FROM DUAL UNION SELECT 200 AS value FROM DUAL UNION SELECT 100 AS
value FROM DUAL;
 value
-------
   100
   200
(2 rows)
```

Minus Operator

Syntax

```
select_statement MINUS [ ALL | DISTINCT ] select_statement;
```

Parameters

`select_statement` Any SELECT statement without the ORDER BY, LIMIT, FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE, and FOR KEY SHARE clauses. `ALL keyword` contains duplicate row results. `DISTINCT keyword` shows the elimination of duplicate rows.

Example

```
select * from generate_series(1, 3) g(i) MINUS select * from generate_series(1, 3)
g(i) where i = 1;
 i
---
 2
 3
(2 rows)
```

Escape characters

Overview

Use q\' to escape special characters. q\' escaped characters are usually used after ! [] {} () \<> and other escaping characters, you can also use \, letters, numbers, \=, +, -, *, \&, \$, \%, #, etc., no spaces are allowed.

Example of

```
select q''' is goog '';
  ?column?
------------
 ' is goog
(1 row)
```

Sequence

Syntax

```
SELECT [ database {schema} | schema ] sequence {nextval | currval};
```

Parameters

`sequence`Sequence Name.

Example

```
create sequence sq;

select sq.nextval;
 nextval
-----------
       1
(1 row)
```

```
select sq.currval;
 nextval
-----------
      1
(1 row)
```

Compatible with time and date functions

from_tz

Purpose

```
Convert the given timestamp without time zone to the specified timestamp with time
zone, or return NULL if the specified time zone or timestamp is NULL.
```

Parameters

Parameters	Description
day	Timestamp without time zone
tz	Specified time zone

Example

```
select from_tz('2021-11-08 09:12:39','Asia/Shanghai') from dual;
              from_tz
-----------------------------------
 2021-11-08 09:12:39 Asia/Shanghai
(1 row)

select from_tz('2021-11-08 09:12:39','SAST') from dual;
         from_tz
--------------------------
 2021-11-08 09:12:39 SAST

select from_tz(NULL,'SAST') from dual;
 from_tz
---------

(1 row)

select from_tz('2021-11-08 09:12:31',NULL) from dual;
 from_tz
---------
```

```
(1 row)
```

systimestamp

Purpose

```
Get the timestamp of the current database system.
```

Example

```
select oracle.systimestamp();
         systimestamp
-------------------------------
 2021-12-02 14:38:59.879642+08
(1 row)

select systimestamp;
      statement_timestamp
-------------------------------
 2021-12-02 14:39:33.262828+08
```

sys_extract_utc

Purpose

```
Converts the given timestamp with time zone to UTC time without time zone.
```

Parameters Description

Parameters	Description
day	Need to convert time stamp with time zone

Example

```
select sys_extract_utc('2018-03-28 11:30:00.00 +09:00'::timestamptz) from dual;
   sys_extract_utc
---------------------
 2018-03-28 02:30:00
(1 row)

select oracle.sys_extract_utc(NULL) from dual;
 sys_extract_utc
-----------------
```

```
(1 row)
```

sessiontimezone

Purpose

```
Gets the time zone of the current session.
```

Example

```
select sessiontimezone() from dual;
 sessiontimezone
-----------------
 PRC
(1 row)

set timezone to UTC;

 select oracle.sessiontimezone();
 sessiontimezone
-----------------
 UTC
(1 row)
```

next_day

Purpose

```
next_day returns the date of the first weekday with the same format name, which is
later than the current date. The return type is always DATE, regardless of the date's
data type. The return value has the same hour, minute, and second parts as the
Parameters date.
```

ParametersDescription

Parameters	Description
value	Start Timestamp
weekday	The day of the week, can be "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday" or 0,1,2,3,4,5,6,0 for Sunday

Example

```
select next_day(to_timestamp('2020-02-29 14:40:50', 'YYYY-MM-DD HH24:MI:SS'),
'Tuesday') from dual;
```

```
      next_day
---------------------
 2020-03-03 14:40:50
(1 row)

select next_day('2020-07-01 19:43:51 +8'::timestamptz, 1) from dual;
      next_day
---------------------
 2020-07-05 19:43:51
(1 row)
```

last_day

Purpose

```
last_day returns the last day of the month in which the slot date falls.
```

ParametersDescription

Parameters	Description
value	Specified timestamp

Example

```
select last_day(timestamp '2020-05-17 13:27:19') from dual;
      last_day
---------------------
 2020-05-31 13:27:19
(1 row)

select last_day('2020-11-29 19:20:40 +08'::timestamptz) from dual;
      last_day
---------------------
 2020-11-30 19:20:40
(1 row)
```

add_months

Purpose

```
add_months returns the date plus an integer month. date Parameters can be date-time
values or any value that can be implicitly converted to DATE. integer Parameters can
be an integer or any value that can be implicitly converted to an integer.
```

ParametersDescription

Parameters	Description
day	oracle.date type, the timestamp that needs to be changed
value	A shaping data, the number of months to be added

Example

```
select add_months(date '2020-02-15',7) from dual;
     add_months
---------------------
 2020-09-15 00:00:00
(1 row)

select add_months(timestamp '2018-12-15 19:12:09',12) from dual;
     add_months
---------------------
 2019-12-15 19:12:09
(1 row)
```

sysdate

Purpose

```
sysdate returns the operating system time of the database server.
```

Example

```
select sysdate;
  statement_sysdate
---------------------
 2021-12-09 16:20:34
(1 row)

select sysdate();
       sysdate
---------------------
 2021-12-09 16:21:39
(1 row)
```

new_time

Purpose

```
Convert the time of the first time zone to the time of the second time zone. The time
zones include "ast", "adt", "bst", "bdt", "cst", "cdt", "est", "edt", "gmt", "hst",
"hdt", "mst", "mdt", "nst", "pst", "pdt", "yst", "ydt".
```

ParametersDescription

Parameters	Description
day	Timestamp to be converted
tz1	Timestamped time zones
tz2	Target time zone

Example

```
select new_time(timestamp '2020-12-12 17:45:18', 'AST', 'ADT') from dual;
      new_time
---------------------
 2020-12-12 18:45:18
(1 row)

select new_time(timestamp '2020-12-12 17:45:18', 'BST', 'BDT') from dual;
      new_time
---------------------
 2020-12-12 18:45:18
(1 row)

select new_time(timestamp '2020-12-12 17:45:18', 'CST', 'CDT') from dual;
      new_time
---------------------
 2020-12-12 18:45:18
(1 row)
```

trunc

Purpose

```
The trunc function returns a date, truncated in the specified format. fmt includes
"Y", "YY", "YYYY", "YYYY", "YEAR", "SYYYY", "SYEAR", "I", "IY", "IYY", "IYYY", "Q",
"WW", "Iw", "W", "DAY", "DY", "D", " MONTH", "MONn", "MM", "RM", "CC", "SCC", "DDD",
"DD", "J", "HH", "HH12", "HH24", "MI".
```

ParametersDescription

Parameters	Description

value	The specified date (oracle.date, timestamp, timestamptz)
fmt	The specified format, if omitted, defaults to "DDD"

Example

```
select trunc(timestamp '2020-07-28 19:16:12', 'Q');
        trunc
---------------------
 2020-07-01 00:00:00
(1 row)

select trunc(timestamptz '2020-09-27 18:30:21 + 08', 'MONTH');
         trunc
-------------------------
 2020-09-01 00:00:00+08
(1 row)
```

round

Purpose

```
The round function returns a date, rounded to the specified format. fmt includes "Y",
"YY", "YYYY", "YYYY", "YEAR", "SYYYY", "SYEAR", "I", "IY", "IYY", "IYYY", "Q", "WW",
"Iw", "W", "DAY", "DY", "D ", "MONTH", "MONn", "MM", "RM", "CC", "SCC", "DDD", "DD",
"J", "HH", "HH12", "HH24", "MI".
```

ParametersDescription

Parameters	Description
value	The date being converted (oracle.date, timestamp, timestamptz)
fmt	The specified format, if omitted, defaults to "DDD"

Example

```
select round(timestamp '2050-06-12 16:40:55', 'IYYY');
        round
---------------------
 2050-01-03 00:00:00
(1 row)
```

Compatible conversion and comparison and NULL-related functions

TO_CHAR

Purpose

TO_CHAR (str,[fmt]) Converts the input Parameters to a TEXT data type value according to the given format. If fmt is omitted, the data will be converted to a TEXT value in the system default format. If str is null, the function returns null.

Parameters

`str` Input Parameters (any type). `fmt` Input format Parameters, see format fmt for details.

Example

```
select to_char('3 2:20:05' );
     to_char
 -----------------
  3 days 02:20:05
 (1 row)

select to_char('4.00'::numeric);
  to_char
 ---------
  4
 (1 row)

select to_char(NULL);
  to_char
 ---------

 (1 row)

select to_char(123,'xx');
  to_char
 ---------
  7b
 (1 row)
```

TO_NUMBER

Purpose

TO_NUMBER(str,[fmt1]) Converts the input Parameters str to a value of the NUMREIC data type according to the given format. If fmt1 is omitted, the data will be converted to a NUMERIC value in the system default format. If str is NUMERIC, the function returns str. If str calculates to null, the function returns null. If it cannot be converted to the NUMERIC data type, the function returns an error.

Parameters

`str` input Parameters include the following data types (double precision, numeric, text, integer, etc., but must be implicitly converted to numeric). `fmt1` Input format Parameters, see format fmt1 for details.

Example

```
select to_number(1210.73::numeric, 9999.99::numeric);
 to_number
-----------
   1210.73
(1 row)

select to_number(NULL);
 to_number
-----------

(1 row)

select to_number('123'::text);
 to_number
-----------
   123
(1 row)
```

TO_DATE

Purpose

TO_DATE(str,[fmt]) Converts the input Parameters str to a date data type value according to the given format. If fmt is omitted, the data will be converted to a date value in the system default format. If str is null, the function returns null. If fmt is J, for Julian, then char must be an integer. The function returns an error if it cannot be converted to DATE.

Parameters

`str` input Parameters (integer, text, can be implicitly converted to the above type, string that matches the date format). `fmt` input format Parameters, see format fmt for details.

Example

```
select to_date('50-11-28 ','RR-MM-dd ');
       to_date
---------------------
 1950-11-28 00:00:00
(1 row)

select to_date(2454336, 'J');
```

```
       to_date
---------------------
 2007-08-23 00:00:00
(1 row)

select to_date('2019/11/22', 'yyyy-mm-dd');
       to_date
---------------------
 2019-11-22 00:00:00
(1 row)

select to_date('20-11-28 10:14:22','YY-MM-dd hh24:mi:ss');
       to_date
---------------------
 2020-11-28 10:14:22
(1 row)

select to_date('2019/11/22');
       to_date
---------------------
 2019-11-22 00:00:00
(1 row)

select to_date('2019/11/27 10:14:22');
       to_date
---------------------
 2019-11-27 10:14:22
(1 row)

select to_date('2020','RR');
       to_date
---------------------
 2020-01-01 00:00:00
(1 row)

select to_date(NULL);
 to_date
---------

(1 row)

select to_date('-4712-07-23 14:31:23', 'syyyy-mm-dd hh24:mi:ss');
```

```
       to_date
----------------------
 -4712-07-23 14:31:23
(1 row)
```

TO_TIMESTAMP

Purpose

TO_TIMESTAMP(str,[fmt]) Converts the input Parameters str to a timestamp without a time zone according to the given format. If fmt is omitted, the data is converted to a timestamp with no time zone value in the system default format. If str is null, the function returns null. If it cannot be converted to a timestamp without a time zone, the function returns an error.

Parameters

`str` input Parameters (double precision,text, which can be implicitly converted to the above type). `fmt` Input format Parameters, see format fmt for details.

Example

```
select to_timestamp(1212121212.55::numeric);
       to_timestamp
---------------------------
 2008-05-30 12:20:12.55
(1 row)

select to_timestamp('2020/03/03 10:13:18 +5:00', 'YYYY/MM/DD HH:MI:SS TZH:TZM');
      to_timestamp
------------------------
 2020-03-03 13:13:18
(1 row)

select to_timestamp(NULL,NULL);
 to_timestamp
--------------

(1 row)
```

TO_YMINTERVAL

Purpose

TO_YMINTERVAL(str) Converts the input Parameters str time interval to a time interval in the year-to-month range. Only the year and month are processed, other parts are omitted. If the input Parameters is NULL, the function returns NULL, and if the input Parameters is in the wrong format, the function returns an error.

Parameters

str Input Parameters (text, can be implicitly converted to text type, must be in time interval format. (SQL interval format compatible with SQL standard, ISO duration format compatible with ISO 8601:2004 standard).

Example

```
select to_yminterval('P1Y-2M2D');
 to_yminterval
---------------
 +000000000-10
(1 row)

select to_yminterval('P1Y2M2D');
 to_yminterval
---------------
 +000000001-02
(1 row)

select to_yminterval('-01-02');
 to_yminterval
---------------
 -000000001-02
(1 row)
```

TO_DSINTERVAL

Purpose

TO_DSINTERVAL(str) converts the time interval of the input Parameters str to a time interval in the range of days to seconds. Input Parameters include: day, hour, minute, second and microsecond. If the input Parameters is NULL, the function returns NULL, and if the input Parameters contains the year and month or is in the wrong format, the function returns an error.

Parameters

str Input Parameters (text, can be implicitly converted to text type, must be in time interval format. (SQL interval format compatible with SQL standard, ISO duration format compatible with ISO 8601:2004 standard).

Example

```
select to_dsinterval('100 00 :02 :00');
         to_dsinterval
------------------------------
 +000000100 00:02:00.000000000
(1 row)
```

```
select to_dsinterval('-100 00:02:00');
         to_dsinterval
-------------------------------
 -000000100 00:02:00.000000000
(1 row)

select to_dsinterval(NULL);
 to_dsinterval
---------------

(1 row)
```

TO_TIMESTAMP_TZ

Purpose

TO_TIMESTAMP_TZ(str,[fmt]) Converts the input Parameters str to a timestamp with a time zone according to the given format. If fmt is omitted, the data will be converted to a timestamp with a time zone value in the system default format. If str is null, the function returns null. If it cannot be converted to a timestamp with a time zone, the function returns an error.

Parameters

`str` input Parameters (text, which can be implicitly converted to a text type). `fmt` Enter format Parameters, see format fmt for details.

Example

```
select to_timestamp_tz('2019','yyyy');
    to_timestamp_tz
------------------------
 2019-01-01 00:00:00+08
(1 row)

select to_timestamp_tz('2019-11','yyyy-mm');
    to_timestamp_tz
------------------------
 2019-11-01 00:00:00+08
(1 row)

select to_timestamp_tz('2003/12/13 10:13:18 +7:00');
    to_timestamp_tz
------------------------
 2003-12-13 11:13:18+08
(1 row)

select to_timestamp_tz('2019/12/13 10:13:18 +5:00', 'YYYY/MM/DD HH:MI:SS TZH:TZM');
```

```
    to_timestamp_tz
------------------------
 2019-12-13 13:13:18+08
(1 row)

select to_timestamp_tz(NULL);
 to_timestamp_tz
-----------------

(1 row)
```

GREATEST

Purpose

GREATEST(expr1,expr2,⋯) Gets the maximum value in the input list of one or more expressions. If the result of any expr calculation is NULL, the function returns NULL.

Parameters

```
expr1` Enter Parameters (of any type).
`expr2` Enter Parameters (of any type).
`...
```

Example

```
select greatest('a','b','A','B');
 greatest
----------
 b
(1 row)

select greatest(',','.','/',';','!','@','?');
 greatest
----------
 @
(1 row)

select greatest('瀚','高','数','据','库');
 greatest
----------
 高
(1 row)
```

```
SELECT greatest('HARRY', 'HARRIOT', 'HARRA');
 greatest
----------
 HARRY
(1 row)

SELECT greatest('HARRY', 'HARRIOT', NULL);
 greatest
----------

(1 row)

SELECT greatest(1.1, 2.22, 3.33);
 greatest
----------
     3.33
(1 row)

SELECT greatest('A', 6, 7, 5000, 'E', 'F','G') A;
 a
---
 G
(1 row)
```

LEAST

Purpose

LEAST(expr1,expr2,⋯) Gets the smallest value in the input list of one or more expressions. If the result of any expr calculation is NULL, the function returns NULL.

Parameters

```
expr1` Enter Parameters (of any type).
`expr2` Enter Parameters (of any type).
`...
```

Example

```
SELECT least(1,' 2', '3' );
 least
-------
     1
(1 row)
```

```
SELECT least(NULL, NULL, NULL);
 least
-------

(1 row)

SELECT least('A', 6, 7, 5000, 'E', 'F','G') A;
  a
------
 5000
(1 row)

select least(1,3,5,10);
 least
-------
     1
(1 row)

select least('a','A','b','B');
 least
-------
 A
(1 row)

select least(',','.','/',';','!','@');
 least
-------
 !
(1 row)

select least('瀚','高','据','库');
 least
-------
 库
(1 row)

SELECT least('HARRY', 'HARRIOT', NULL);
 least
-------

(1 row)
```

Overview

NLS_LENGTH_SEMANTICS enables you to create CHAR and VARCHAR2 columns using byte or character length semantics. Existing columns are not affected. In this case, the default semantics is BYTE.

Syntax

```
SET NLS_LENGTH_SEMANTICS TO [NONE | BYTE | CHAR];
```

Note on the range of values

```
BYTE: The data is stored in byte length.
CHAR:Data is stored in character length.
NONE:Data is stored using native IvorySQL storage.
```

Example

--Test “CHAR”

```
create table test(a varchar2(5));
CREATE TABLE

SET NLS_LENGTH_SEMANTICS TO CHAR;
SET

SHOW NLS_LENGTH_SEMANTICS;
 nls_length_semantics
----------------------
 char
(1 row)

insert into test values ('Hello,Mr.li');
INSERT 0 1
```

--Test “BYTE”

```
SET NLS_LENGTH_SEMANTICS TO BYTE;
SET

SHOW NLS_LENGTH_SEMANTICS;
 nls_length_semantics
----------------------
```

```
 byte
(1 row)

insert into test values ('Hello,Mr.li');
2021-12-14 15:28:11.906 HKT [6774] ERROR:  value too long for type varchar2(5 byte)
2021-12-14 15:28:11.906 HKT [6774] STATEMENT:  insert into test values
('Hello,Mr.li');
ERROR:  value too long for type varchar2(5 byte)
```

VARCHAR2(size)

Overview

Variable length strings with maximum length bytes or characters. You must specify the size for VARCHAR2. The minimum size is 1 byte or 1 character.

Syntax

```
VARCHAR2(size)
```

Example

```
create table test(a varchar2(5));
CREATE TABLE

SET NLS_LENGTH_SEMANTICS TO CHAR;
SET

SHOW NLS_LENGTH_SEMANTICS;
 nls_length_semantics
----------------------
 char
(1 row)

insert into test values ('Hello,Mr.li');
INSERT 0 1
```

PL/iSQL

PL/iSQL is IvorySQL’s procedural language for writing custom functions, procedures and packages for IvorySQL. PL/iSQL is derived from IvorySQL’s PL/pgsql with some added features, but syntactically PL/iSQL is closer to Oracle’s PL/SQL. This document Describes the basic structure and construction of PL/iSQL programs.

Structure of PL/iSQL Programs

iSQL is a procedural block structure language that supports four different program types, PACKAGES,

PROCEDURES, FUNCTIONS, and TRIGGERS. iSQL supports four different program types, PACKAGES, PROCEDURES, FUNCTIONS, and TRIGGERS. iSQL uses the same block structure for each type of supported program. A block consists of up to three parts: a declaration part, an executable, and an exception part. The declaration and exception sections are optional.

```
[DECLARE
      declarations]
    BEGIN
      statements
  [ EXCEPTION
      WHEN <exception_condition> THEN
        statements]
    END;
```

A block can consist of at least one executable section Contains one or more iSQL statements in the BEGIN and END keywords.

```
CREATE OR REPLACE FUNCTION null_func() RETURN VOID AS
BEGIN
    NULL;
END;
/
```

All keywords are case-insensitive. Identifiers are implicitly converted to lowercase unless double-quoted, just as they are in normal SQL commands. The declaration section can be used to declare variables and cursors, and depending on the context in which the block is used, the declaration section can begin with the keyword DECLARE.

```
CREATE OR REPLACE FUNCTION null_func() RETURN VOID AS
DECLARE
    quantity integer := 30;
    c_row pg_class%ROWTYPE;
    r_cursor refcursor;
    CURSOR c1 RETURN pg_proc%ROWTYPE;
BEGIN
    NULL;
end;
/
```

An optional exception section can also be included in a BEGIN - END block. The exception section begins with the keyword EXCEPTION and continues until the end of the block in which it appears. If a statement within the block throws an exception, program control goes to the exception section, which may or may not handle the thrown exception, depending on the contents of the exception and exception sections.

```
CREATE OR REPLACE FUNCTION reraise_test() RETURN void AS
BEGIN
```

```
    BEGIN
        RAISE syntax_error;
        EXCEPTION
            WHEN syntax_error THEN

            BEGIN
                raise notice 'exception % thrown in inner block, reraising', sqlerrm;
                RAISE;
                EXCEPTION
                WHEN OTHERS THEN
                    raise notice 'RIGHT - exception % caught in inner block', sqlerrm;
                END;
            END;
    EXCEPTION
        WHEN OTHERS THEN
        raise notice 'WRONG - exception % caught in outer block', sqlerrm;
END;
/
```

> **Note**
>
> Like PL/pgSQL, PL/iSQL uses BEGIN/END to group statements, and do not confuse them with the SQL commands of the same name used for transaction control. PL/iSQL’s BEGIN/END are used only for grouping; they do not start or end transactions

psql support for PL/iSQL programs

To create a PL/iSQL program from a psql client, you can use a syntax similar to PL/pgSQL’s $$

```
CREATE FUNCTION func() RETURNS void as
$$
..
end$$ language plisql;
```

Alternatively, you can use the Oracle-compliant syntax of references and language specifications without $$ and end the program definition with / (forward slash). The */ (forward slash) must be on the newline character

```
CREATE FUNCTION func() RETURN void AS
...
END;
/
```

PROCEDURES

```
CREATE [OR REPLACE] PROCEDURE procedure_name [(parameter_list)]
is
[DECLARE]
    -- variable declaration
BEGIN
    -- stored procedure body
END;
/
```

FUNCTIONS

```
CREATE [OR REPLACE] FUNCTION function_name ([parameter_list])
RETURN return_type AS
[DECLARE]
    -- variable declaration
BEGIN
    -- function body
   return statement
END;
/
```

PACKAGES

PACKAGE HEADER

```
CREATE [ OR REPLACE ] PACKAGE [schema.] *package_name* [invoker_rights_clause] [IS |
AS]
   item_list[, item_list ...]
END [*package_name*];


invoker_rights_clause:
     AUTHID [CURRENT_USER | DEFINER]

item_list:
[
   function_declaration    |
   procedure_declaration   |
   type_definition         |
   cursor_declaration      |
```

```
   item_declaration
]


function_declaration:
     FUNCTION function_name [(parameter_declaration[, ...])] RETURN datatype;

procedure_declaration:
     PROCEDURE procedure_name [(parameter_declaration[, ...])]

type_definition:
     record_type_definition      |
     ref_cursor_type_definition

cursor_declaration:
   CURSOR name [(cur_param_decl[, ...])] RETURN rowtype;

item_declaration:
     cursor_declaration             |
     cursor_variable_declaration    |
     record_variable_declaration    |
     variable_declaration           |

record_type_definition:
   TYPE record_type IS RECORD  ( variable_declaration [, variable_declaration]... ) ;

ref_cursor_type_definition:
   TYPE type IS REF CURSOR [ RETURN type%ROWTYPE ];

cursor_variable_declaration:
   curvar curtype;

record_variable_declaration:
   recvar { record_type | rowtype_attribute | record_type%TYPE };

variable_declaration:
   varname datatype [ [ NOT NULL ] := expr ]

parameter_declaration:
   parameter_name [IN] datatype [[:= | DEFAULT] expr]
```

```
CREATE [ OR REPLACE ] PACKAGE BODY [schema.] package_name [IS | AS]
   [item_list[, item_list ...]] |
   item_list_2 [, item_list_2 ...]
   [initialize_section]
END [package_name];


initialize_section:
   BEGIN statement[, ...]

item_list:
[
   function_declaration    |
   procedure_declaration   |
   type_definition         |
   cursor_declaration      |
   item_declaration
]

item_list_2:
[
   function_declaration
   function_definition
   procedure_declaration
   procedure_definition
   cursor_definition
]

function_definition:
   FUNCTION function_name [(parameter_declaration[, ...])] RETURN datatype  [IS | AS]
   [declare_section] body;

procedure_definition:
   PROCEDURE procedure_name [(parameter_declaration[, ...])] [IS | AS]
   [declare_section] body;

cursor_definition:
   CURSOR name [(cur_param_decl[, ...])] RETURN rowtype IS select_statement;

body:
   BEGIN statement[, ...] END [name];
```

```
statement:
   [<<LABEL>>] pl_statments[, ...];
```

Hierarchy Search

Syntax

```
 {
 CONNECT BY [ NOCYCLE ] [PRIOR] condition [AND [PRIOR] condition]... [ START WITH
condition ]
 | START WITH condition CONNECT BY [ NOCYCLE ] [PRIOR] condition [AND [PRIOR]
condition]...
 }
```

- The CONNECT BY query syntax begins with the CONNECT BY keywords, which define hierarchical interdependencies between parent and child rows. The result must be further qualified by specifying the PRIOR keyword in the conditional part of the CONNECT BY clause.

The PRIOR PRIOR keyword is a unary operator that relates the previous row to the current row. This keyword can be used to the left or right of the equality condition.

START WITH This clause specifies the line from which the hierarchy begins.

NOCYCLE No operation statement. Currently only supported by the syntax. This clause indicates that data is returned even if a loop exists.

ADDITIONAL COLUMN

LEVEL Returns the level of the current row in the hierarchy, starting at 1 at the root node and incrementing by 1 at each level thereafter.

CONNECT_BY_ROOT expr Returns the parent column of the current row in the hierarchy.

SYS_CONNECT_BY_PATH(col, chr) It is a function that returns the value of the column from the root to the current node, separated by the character "chr".

Limitations

This function currently has the following limitations.

- Additional columns can be used for most expressions, such as function calls, CASE statements and general expressions, but there are some unsupported columns, such as ROW, TYPECAST, COLLATE, GROUPING clauses, etc.
- In case two or more columns are the same, you may need to output the column name, Example such as

> SELECT CONNECT_BY_ROOT col AS "col1", CONNECT_BY_ROOT col AS "col2" ….

- Indirect operators or "*" are not supported
- Loop detection is not supported

Global Unique Index

Create global unique index

Syntax

```
CREATE UNIQUE INDEX [IF NOT EXISTS] name ON table_name [USING method] (columns) GLOBAL
```

Example

```
CREATE UNIQUE INDEX myglobalindex on mytable(bid) GLOBAL;
```

Global uniqueness assurance

During the creation of a globally unique index, the system performs an index scan on all existing partitions and raises an error if it finds duplicate entries from other partitions than the current one. Example.

Command

```
create table gidxpart (a int, b int, c text) partition by range (a);
create table gidxpart1 partition of gidxpart for values from (0) to (100000);
create table gidxpart2 partition of gidxpart for values from (100000) to (199999);
insert into gidxpart (a, b, c) values (42, 572814, 'inserted first on gidxpart1');
insert into gidxpart (a, b, c) values (150000, 572814, 'inserted second on
gidxpart2');
create unique index on gidxpart (b) global;
```

Output

```
ERROR:  could not create unique index "gidxpart1_b_idx"
DETAIL:  Key (b)=(572814) is duplicated.
```

Insertions and updates

Global uniqueness guarantee for insertions and updates

During global unique index creation, the system performs an index scan on all existing partitions and raises an error if duplicate items are found in other partitions than the current one.

Example

Command

```
create table gidx_part (a int, b int, c text) partition by range (a);
create table gidxpart (a int, b int, c text) partition by range (a);
create table gidxpart1 partition of gidxpart for values from (0) to (10);
create table gidxpart2 partition of gidxpart for values from (10) to (100);
```

```
create unique index gidx_u on gidxpart using btree(b) global;


insert into gidxpart values (1, 1, 'first');
insert into gidxpart values (11, 11, 'eleventh');
insert into gidxpart values (2, 11, 'duplicated (b)=(11) on other partition');
```

Output

```
ERROR:  duplicate key value violates unique constraint "gidxpart2_b_idx"
DETAIL:  Key (b)=(11) already exists.
```

Append and detach

Global uniqueness guarantee for append statements

When appending a new table to a partitioned table with a globally unique index, the system performs a duplicate check on all existing partitions. If a duplicate item is found in an existing partition that matches a tuple in the appended table, an error is raised and the append fails.

Appending requires a SHARE LOCK on all existing partitions. If one of the partitions is doing a concurrent INSERT, the append will wait for it to complete first. This can be improved in a future release

Example

Command

```
create table gidxpart (a int, b int, c text) partition by range (a);
create table gidxpart1 partition of gidxpart for values from (0) to (100000);
insert into gidxpart (a, b, c) values (42, 572814, 'inserted first on gidxpart1');
create unique index on gidxpart (b) global;
create table gidxpart2 (a int, b int, c text);
insert into gidxpart2 (a, b, c) values (150000, 572814, 'dup inserted on gidxpart2');

alter table gidxpart attach partition gidxpart2 for values from (100000) to (199999);
```

Output

```
ERROR:  could not create unique index "gidxpart1_b_idx"
DETAIL:  Key (b)=(572814) is duplicated.
```

.2. Community contribution

Summary

Illustration

IvorySQL is maintained by a core development team, which has commit access to the main repository of IvorySQL on GitHub. We are eager to get contributions from members of the wider IvorySQL community. If

you want to see your changes to code or documents added to IvorySQL and appear in future versions, you need to understand the content of this section.

IvorySQL community welcomes and appreciates all types of contributions and we are looking forward to your participation!

Principles of Conduct

Every member, contributor and leader should read our principles of conduct. We promise that everyone can participate in community and pay equal attention to everyone, no matter who.

We are committed to acting and interacting in a way that contributes to the establishment of an open, enthusiastic, diverse, inclusive and healthy community.

Description of Community Governance

Our team is a continuously open team, focusing on parts of IvorySQL. In our team, there are reviewers, submitters and maintainers, and we have one or more repositories. The decision for the team is made by the maintainer. The typical promotion path for IvorySQL developers is from user to reviewer, then submitter and maintainer. But getting more roles doesn’ t mean you have any privileges to other community members. Everyone in the IvorySQL community is equal and has the responsibility to cooperate constructively with other contributors to build a friendly community. These roles are natural rewards for your significant contributions to the development of IvorySQL, and provide you with more rights in the development workflow to improve your efficiency. At the same time， this requires you to undertake some additional duty:

Team honor: now you are already one of the team reviewers/submitters/maintainers, it means that you represent the project and your team members. So, please be Mr.Nice Guy to defend the reputation of the team.

Responsibility: submitters/maintainers have the right to merge pull requests, therefore, they take additional responsibility to deal with the consequences of accepting changes to the code base or documents. When a bug occurs, they should fix it. If they can not solve it, they should roll back the project. Also, they need to help the release manager solve any problems found in the test cycle.

Contributor’ s Guide

Before contributing, we need to know the current version of IvorySQL and the version of the document.At present, we maintain versions after version 5.4. Our version follows the update pace of PG. Please update to the latest version before contributing. After that, we need to read the format requirements carefully and be familiar with code format, code comment format, issue format, pull PR title format, document contribution format, and article contribution format. These can help you become a contributor of IvorySQL as soon as possible.

Preparation before Contribution

Getting started

IvorySQL is developed on GitHub. Anyone who wishes to contribute to it must have a Github account and be familiar with Git tools and workflow. It is also recommended that you follow the developer’ s mailing list since some of the contributions may generate more detailed discussions there.

Once you have your GitHub account, fork this repository so that you can have your private copy to start hacking on and to use as a source of pull requests.

Licensing of IvorySQL contributions

If the contribution you’ re submitting is original work, you can assume that IvorySQL will release it as part of an overall IvorySQL release available to the downstream consumers under the Apache License, Version 2.0.

If the contribution you’ re submitting is NOT original work you have to indicate the name of the license and also make sure that it is similar in terms to the Apache License 2.0. Apache Software Foundation maintains a

list of these licenses under Category A. In addition to that, you may be required to make proper attributions.

Finally, keep in mind that it is NEVER a good idea to remove licensing headers from the work that is not your original one. Even if you are using parts of the file that originally had a licensing header at the top you should err on the side of preserving it. As always, if you are not quite sure about the licensing implications of your contributions, feel free to reach out to us on the developer mailing list.

What Contribution Can You Make

Code Contribution

You can upload your modified bugs, new functions and other codes to your personal warehouse, and finally submit PR requests to merge them on the official website: https://github.com/IvorySQL/IvorySQL.

Document Contribution(https://www.ivorysql.org/zh-CN/docs/intro)

The IvorySQL community provides Chinese and English documents. English documents are saved in … document repository, Chinese documents are saved in i18n document repository. You can contribute to one of them or both.

When you add or update a plugin compatibility document, you must also update the plugin summary table in IvorySQL Ecosystem Plugin Compatibility List. Keep the plugin name, version, function description, use cases, and link to the detail page in sync so that the overview page and the detail page stay consistent.

Test IvorySQL and Report Bugs

GitHub: https://github.com/IvorySQL/IvorySQL

Gitee:https://gitee.com/IvorySQL/

Participate in the Construction of IvorySQL Website

IvorySQL official website:https://github.com/IvorySQL/Ivory-www

Answer Questions on the Mailing List

Mailing List website:https://lists.ivorysql.org/

Contribute Article

You can submit your article to the blog in the IvorySQL-WWW code warehouse, or send it to the mailbox renjiao@highgo.com.

How to Contribute

Coding Guidelines

Your chances of getting feedback and seeing your code merged into the project greatly depend on how granular your changes are. If you happen to have a bigger change in mind, we highly recommend engaging on the developer’ s mailing list first and sharing your proposal with us before you spend a lot of time writing code. Even when your proposal gets validated by the community, we still recommend doing the actual work as a series of small, self-contained commits. This makes the reviewer’ s job much easier and increases the timeliness of feedback.

When it comes to C and C++ parts of IvorySQL, we follow PostgreSQL Coding Conventions. In addition to that:

For C and Perl code, please run pgindent if necessary. We recommend using git diff --color when reviewing your changes so that you don’ t have any spurious whitespace issues in the code that you submit.

All new functionality that is contributed to IvorySQL should be covered by regression tests that are

contributed alongside it. If you are uncertain about how to test or document your work, please raise the question on the ivorysql-hackers mailing list and the developer community will do its best to help you.

At the very minimum, you should always be running

```
make installcheck world
```

to make sure that you’re not breaking anything.

Changes applicable to upstream PostgreSQL

If the change you’re working on touches functionality that is common between PostgreSQL and IvorySQL, you may be asked to forward-port it to PostgreSQL. This is not only so that we keep reducing the delta between the two projects, but also so that any change that is relevant to PostgreSQL can benefit from a much broader review of the upstream PostgreSQL community. In general, it is a good idea to keep both codebases handy so you can be sure whether your changes may need to be forward-ported.

Patch submission

Once you are ready to share your work with the IvorySQL core team and the rest of the IvorySQL community, you should first create or claim the corresponding GitHub Issue, then push all the commits to a branch in your own repository forked from the official IvorySQL and send us a pull request. Every pull request must correspond to at least one Issue so that the change request, discussion, review, and merge result remain traceable.

Patch review

A submitted pull request with passing validation checks is assumed to be available for peer review. Peer review is the process that ensures that contributions to IvorySQL are of high quality and align well with the road map and community expectations. Every member of the IvorySQL community is encouraged to review pull requests and provide feedback. Since you don’t have to be a core team member to be able to do that, we recommend following a stream of pull reviews to anybody who’s interested in becoming a long-term contributor to IvorySQL.

One outcome of the peer review could be a consensus that you need to modify your pull request in certain ways. GitHub allows you to push additional commits into a branch from which a pull request was sent. Those additional commits will be then visible to all of the reviewers.

A peer review converges when it receives at least one +1 and no -1s votes from the participants. At that point, you should expect one of the core team members to pull your changes into the project.

At any time during the patch review, you may experience delays based on the availability of reviewers and core team members. Please be patient. That being said, don’t get discouraged either. If you’re not getting expected feedback for a few days add a comment asking for updates on the pull request itself or send an email to the mailing list.

Direct commits to the repository

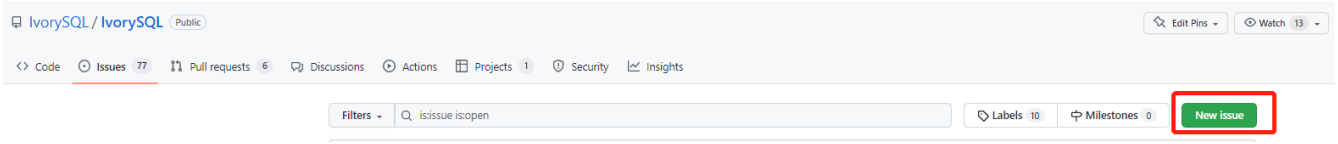
On occasion, you will see core team members committing directly to the repository without going through the pull request workflow. This is reserved for small changes only and the rule of thumb we use is this: if the change touches any functionality that may result in a test failure, then it has to go through a pull request workflow. If, on the other hand, the change is in the non-functional part of the codebase (such as fixing a typo inside of a comment block) core team members can decide to just commit to the repository directly.

Submit Issue

First: Get into New issue page:

1 Enter IvorySQL official website:https://github.com/IvorySQL/IvorySQL

2 Click New issue



Second:Select the issue type

1 bug report

```
Title:
```

```
## Bug Report
Describe the bug


\### IvorySQL Version
The version of IvorySQL you are using

\### OS Version (uname -a)
Operating system version(uname -a)

\### Configuration options  ( config.status --config )


\### Current Behavior


\### Expected behavior/code


\### Step to reproduce


\### Additional context that can be helpful for identifying the problem
```

2 Enhancement

```
Title:
```

```
## Enhancement
Describe the functions that you expect to be strengthened
```

3 Feature Request

```
Title:
```

```
## Feature Request
Describe the feature that you expect to be real
```

Third: Submit

Click submit new issue button. WELL DONE!

Contribute Code

First: Fork ivorysql.org warehouse

1 Open the ivorysql warehouse: https://github.com/IvorySQL/IvorySQL

2 Click the fork button in the upper right corner, Wait for the fork to finish

Before you start coding or editing documents, make sure there is a corresponding GitHub Issue for the change. You can create a new Issue or claim an existing one, but do not submit a PR without an associated Issue.

Second: Clone the warehouse to local

```
cd $working_dir #  $working_dir can be replaced by the directory where you want to
place repo. For example, `cd ~/Documents/GitHub`

git clone git@github.com:$user/IvorySQL.git # `$user` can be replaced by your GitHub
ID.
```

Third: Create a new Branch

```
cd $working_dir/IvorySQL

git checkout -b new-branch-name
```

Fourth: Edit Document or Modify Code

You can modify the code in new-branch-name.

Fifth: Generate commit

```
Git add <file>

Git commit -m  “commit-message”
```

Sixth: Push the modification to the remote end

```
Git push -u origin new-branch-name
```

Seventh: Create a Pull Request

1 Open your warehouse: https://github.com/$user/IvorySQL ($user is your GitHub ID) .

2 Click Compare & pull request button and create a PR.

3 In the PR title or description, explicitly reference the corresponding Issue number, for example `Fixes #123` or `Refs #123`.

Submit PR

A PR submission should contain only one function or one bug. Prohibit submitting multiple functions at one time.

Every GitHub PR must have a corresponding Issue. A PR without an associated Issue does not meet the community contribution process requirements.

Fill in PR information

```
Fix test
Describe the function
```

```
leave a comment
Give a detailed description of the submission function
```

Reference the related Issue in the PR description and keep the Issue number visible to reviewers, for example `Fixes #123`.

Submit PR

Click Create pull request button. WELL DONE!

Edit Documents

Preparation

(1) Download Markdown or Typora document editor.

(2) Check whether the source warehouse has updates. If there are updates, please update and synchronize to your own warehouse first. Refer to the following steps to update to the latest version:

```
git remote

git fetch upstream

git merge upstream/main
```

```
git push
```

(3) Familiar with format [_8:::_pecification].

Where to Contribute

The IvorySQL community provides Chinese and English documents. English documents are saved in IvorySQL document repository, Chinese documents are saved in i18n document repository. You can contribute to one of them or both.

You can start from any of following to help improve the IvorySQL documents on the IvorySQL website:

(1) Prepare complete documents.

(2) Fix incorrect spelling and formatting (Punctuation, space, indentation, code block, etc) .

(3) Improper or outdated instructions corrected or updated.

(4) Add missing content (sentences, paragraphs, or new documents) .

(5) Translate document from English to Chinese, or from Chinese to English.

(6) Submit, reply and resolve document issues or document-i18n issues.

(7) (Advanced) View pull requests created by others.

Specification

The IvorySQL document is written in 'markdown'. To ensure the quality and consistency of the format, certain Markdown rules should be followed when modifying and updating the document.

Markdown Specification

1 Titles are used incrementally from the first level, and skipping is prohibited. For example: The third level title cannot be used directly under the first level title; The fourth level title cannot be used directly under the second level title.

2 The title must use the ATX style uniformly. Indicate the title level by adding # before the title.

3 The leading symbol # of the title must be followed by a blank space.

4 The leading symbol "#" of the title can only be followed by one blank space and then the title content. There can be no more than one space.

5 The title must appear at the beginning of a line, there must be no space before the # sign of the title.

6 Only Chinese and English question marks, back quotes, Chinese and English single and double quotes and other symbols can appear at the end of the title. Other symbols such as colon, comma, period and exclamation point cannot be used at the end of the title.

7 One line must be empty above the title.

8 The same title cannot appear continuously in the document. If the first level title is # TiDB architecture, the next level title cannot be # # TiDB architecture. If it is not a continuous title, the title content can be repeated.

9 Only one first level title in document.

10 In general, except for TOC.md files, which can be indented by two spaces, other .md files must be indented by four spaces by default foe each level of indentation.

11 Tab is not allowed in documents(including code blocks) . If indentation is required, spaces must be uniformly used instead.

12 Continuous blank lines are prohibited.

13 Multiple spaces are not allowed after the block reference symbol > . Only one space can be used, followed by the reference content.

14 When using a ordered list, it must start from 1 and increase in order.

15 When using a list, the identifier (+, -, * or number) of each list item can only be left blank, followed by the list content.

16 The list (includeing ordered and unordered lists) must be empty before and after each line.

17 There must be one blank line before and after the code block.

18 Exposed URLs are prohibited in documents. If you want users to click and open the URL directly, wrap the URL with a pair of angle brackets (<URL>) . If the exposed URL must be used due to special circumstances, and the user does not need to open it by clicking, a pair of back quatation marks (`URL`) will be used to wrap the URL.

19 When using bold, italic and other emphasis effects, redundant spaces are prohibited in the emphasis identifier, such as `text` .

20 No extra space is allowed in the code block wrapped by a single backquote, such as ` text ` .

21 No extra spaces are allowed on both sides of the link text, such as [Link](https://www.example.com/)

22 The link must have a link path. [Empty link]() and [empty link](#) are not allowed.

Example

1 Titles are used incrementally from the first level, and skipping is prohibited.

```
# Heading 1
### Heading 3

We skipped out a 2nd level heading in this document
```

2 The title must use the ATX style uniformly. Indicate the title level by adding # before the title.

```
# Heading 1
## Heading 2
### Heading 3
#### Heading 4
## Another Heading 2
### Another Heading 3
```

3 The leading symbol # of the title must be followed by a blank space. Multiple spaces after # are prohibited, and spaces before # are prohibited.

Incorrect Example:

```
# Heading 1
## Heading 2
```

Correct Example:

```
# Heading 1
## Heading 2
```

4 Only Chinese and English question marks, back quotes, Chinese and English single and double quotes and other symbols can appear at the end of the title.

Incorrect Example:

```
# This is a heading.
```

Correct Example:

```
# This is a heading
```

5 One line must be empty above the title.

Incorrect Example:

```
# Heading 1
Some text
Some more text## Heading 2
```

Correct Example:

```
# Heading 1
Some text
Some more text

## Heading 2
```

6 he same title cannot appear continuously in the document. If the first level title is # TiDB architecture, the next level title cannot be ## TiDB architecture. If it is not a continuous title, the title content can be repeated.

Incorrect Example:

```
# Some text
```

```
## Some text
```

Correct Example:

```
# Some text

## Some more text
```

7 Only one first level title in document.

Incorrect Example:

```
# Top level heading

# Another top-level heading
```

Correct Example:

```
# Title

## Heading

## Another heading
```

8 In general, except for TOC.md files, which can be indented by two spaces, other .md files must be indented by four spaces by default foe each level of indentation.

Incorrect Example:

```
* List item
  * Nested list item indented by 3 spaces
```

Correct Example:

```
* List item
    * Nested list item indented by 4 spaces
```

9 Tab is not allowed in documents(including code blocks) . If indentation is required, spaces must be uniformly used instead.

Incorrect Example:

```
Some text
    * hard tab character used to indent the list item
```

Correct Example:

```
Some text
  * Spaces used to indent the list item instead
```

10 Continuous blank lines are prohibited.

Incorrect Example:

```
Some text here



Some more text here
```

Correct Example:

```
Some text here

Some more text here
```

11 Multiple spaces are not allowed after the block reference symbol > . Only one space can be used, followed by the reference content.

Incorrect Example:

```
>  This is a blockquote with bad indentation>  there should only be one.
```

Correct Example:

```
> This is a blockquote with correct> indentation.
```

12 When using a ordered list, it must start from 1 and increase in order.

Incorrect Example:

```
1. Do this.
1. Do that.
1. Done.
```

```
0. Do this.
1. Do that.
2. Done.
```

```
Correct Example:
```

```
1. Do this.
2. Do that.
3. Done.
```

13 When using a list, the identifier (+, -, * or number) of each list item can only be left blank, followed by the list content.

Correct Example:

```
* Foo
* Bar
* Baz

1. Foo
  * Bar
1. Baz
```

14 The list (includeing ordered and unordered lists) must be empty before and after each line.

Incorrect Example:

```
Some text* Some* List

1. Some2. List

Some text
```

Correct Example:

```
Some text

* Some
* List

1. Some
2. List

Some text
```

15 There must be one blank line before and after the code block.

Incorrect Example:

````
Some text
```

Code block
```
```

Another code block
```
Some more text
````

Correct Example:

````
Some text

```

Code block
```


```

Another code block
```

Some more text
````

16 Exposed URLs are prohibited in documents. If you want users to click and open the URL directly, wrap the URL with a pair of angle brackets (<URL>) . If the exposed URL must be used due to special circumstances, and the user does not need to open it by clicking, a pair of back quatation marks (`URL`) will be used to wrap the URL.

Incorrect Example:

```
For more information, see https://www.example.com/.
```

Correct Example:

```
For more information, see <https://www.example.com/>.
```

17 When using bold, italic and other emphasis effects, redundant spaces are prohibited in the emphasis identifier, such as `text` .

Incorrect Example:

```
Here is some ** bold ** text.

Here is some * italic * text.
```

```
Here is some more __ bold __ text.

Here is some more _ italic _ text.
```

Correct Example:

```
Here is some **bold** text.

Here is some *italic* text.

Here is some more __bold__ text.

Here is some more _italic_ text.
```

18 No extra space is allowed in the code block wrapped by a single backquote, such as ` text `.

Incorrect Example:

```
some text
 some text
```

Correct Example:

```
some text
```

19 No extra spaces are allowed on both sides of the link text, such as [Link](https://www.example.com/) .
Incorrect Example:

```
[a link](https://www.example.com/)
```

Correct Example:

```
[a link](https://www.example.com/)
```

20 The link must have a link path. [Empty link]() and [empty link](#) are not allowed.

Incorrect Example:

```
[an empty link]()

[an empty fragment](#)
```

Correct Example:

```
[a valid link](https://example.com/)

[a valid fragment](#fragment)
```

21 Code blocks in the document are wrapped with three backquote, and the use of indented four-space code blocks is prohibited.

Incorrect Example:

```
Some text.

  # Indented code

More text.
```

Correct Example:

````
```ruby
# Fenced code
```

More text.
````

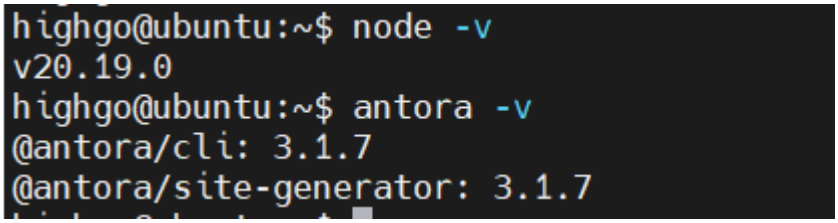
Environmental preparation

In order to test your modifications, you need to prepare the following environment.

- `Node.js` install
- `Antora` install

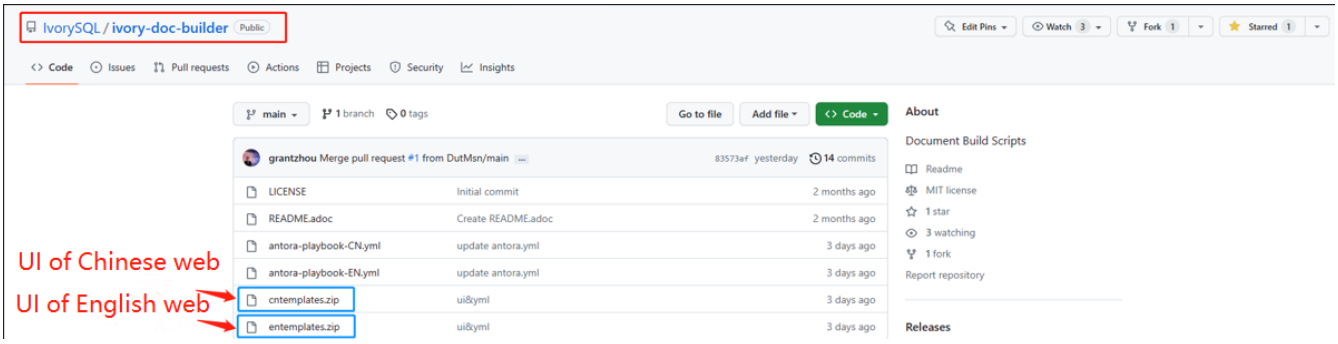
Please refer to Antora docs。

After installation, the following display on the terminal indicates successful installation.



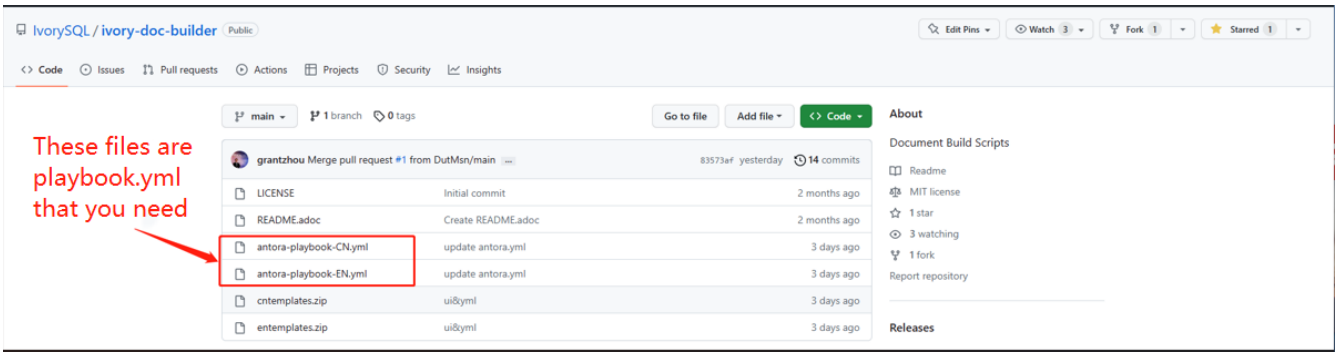
Generate web pages

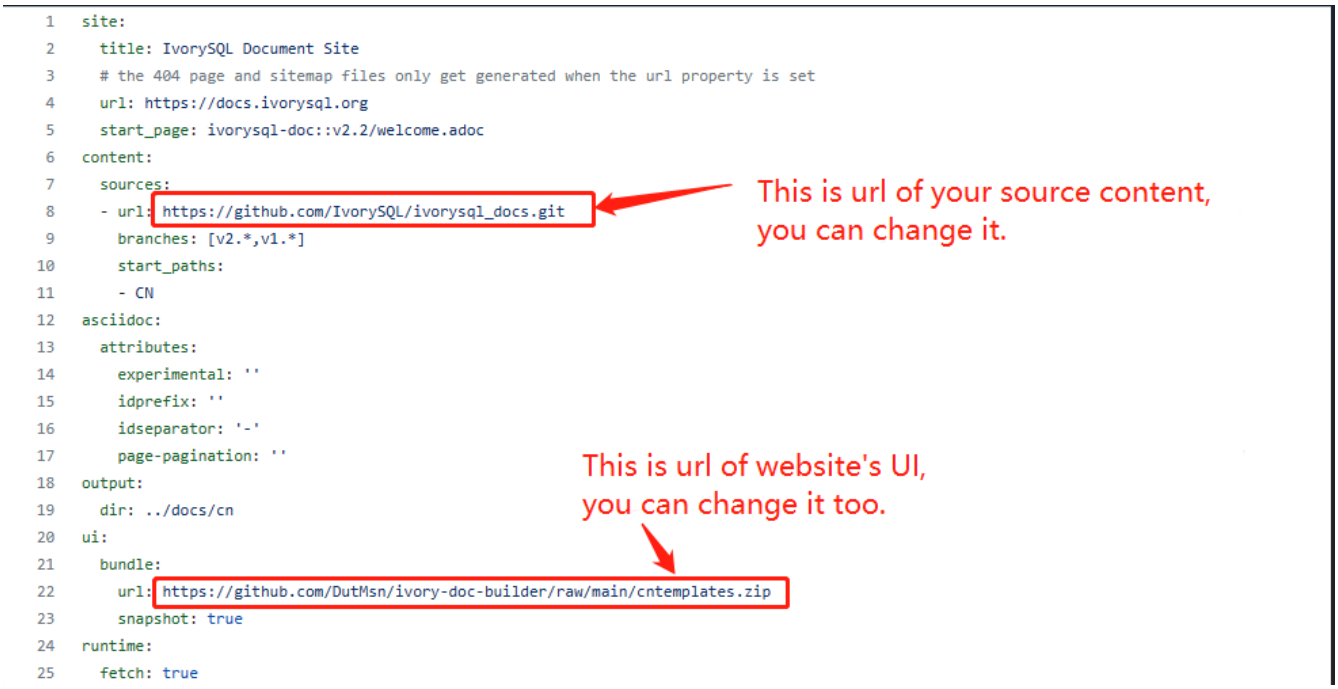
- Firstly, you need to know the location of the corresponding UI for the webpage, as shown in the following figure:



The UI templates for both Chinese and English web pages are basically the same, so when making modifications, it is best to ensure that both templates are modified at the same time. After uploading the modified UI to your personal Github, you can consider generating your modified web page locally.

IvorySQL Document Site is built by `Antora`. Before running `Antora`, remember to modify the corresponding `playbook.yml` file.






```
site:
  title: IvorySQL Document Site
  # the 404 page and sitemap files only get generated when the url property is set
  url: https://docs.ivorysql.org
  start_page: ivorysql-doc::v2.2/welcome.adoc
content:
  sources:
  - url: https://github.com/IvorySQL/ivorysql_docs.git
    branches: [v2.*,v1.*]
    start_paths:
    - CN
asciidoc:
  attributes:
    experimental: ''
    idprefix: ''
    idseparator: '-'
    page-pagination: ''
output:
  dir: ../docs/cn
ui:
  bundle:
    url: https://github.com/DutMsn/ivory-doc-builder/raw/main/cntemplates.zip
    snapshot: true
runtime:
  fetch: true
```

After completing the above process, please run the command `antora antora-playbook.yml --stacktrace` on the terminal, and then patiently wait. After the successful operation is completed, you can view the webpage you have generated.

You can start uploading to our ivorysql_web, the process of submitting PR is the same as before. Thank you for your contribution to the community－. We will consider whether to update the website after the review.

Submit Blog

Preparation

1 Download Markdown or Typora .

2 Check whether the source warehouse (https://github.com/IvorySQL/Ivory-www) has updates. If there are updates, please update and synchronize to your own warehouse first. Refer to the following steps to update to the latest version:

```
# Download source code
git clone https://github.com/IvorySQL/Ivory-www.git
# Synchronize updates warehouse
git pull
```

3 Familiar with format ([_8:::_specification_2])

Where to Congtribute

The IvorySQL community provides Chinese and English documents. English documents are saved in IvorySQL document repository, Chinese documents are saved in i18n document repository. You can contribute to one of them or both.

How to Contribute

Let’ s take a quick look at the information about the maintenance of the IvorySQL blog before contributing. It is helpful for you to submit blog and to be a contributor.

(1) Clone code to local warehouse

```
git clone https://github.com/IvorySQL/Ivory-www.git
```

(2) Create a branch

```
git checkout -b <branch-name>
```

(3) Create a directory of your own articles in the blog directory, and please name your own directory according to the ([Specification](#7.4 Specification)) .

```
# Make English blog directory and files
cd Ivory-www/blog
mkdir <YEAR-MONTH-DAY-title>
cd <YEAR-MONTH-DAY-title>
touch index.md
# Make Chinese blog directory and files
cd Ivory-www/i18n/zh-CN/docusaurus-plugin-content-blog
mkdir <YEAR-MONTH-DAY-title>
cd <YEAR-MONTH-DAY-title>
touch index.md
```

(4) Write the blog to publish in index.md, put the required pictures in the blog in the same directory as index.md.

(5) Submit Blog

```
git add <file-path>
git commit -m "<message>"
git push origin <branch-name>:<branch-name>
```

Specification

Submit specifications

(1) Format of folder naming: year-month-day-foldername

```
Example: 2022-1-28-ivorysql-arrived
```

(2) File property is index.md

(3) Picture property is .png, and put the pictures to be uploaded into the folder to be submitted in advance.

Notice:The name of every picture is unique and cannot be repeated.

Example: po-one.png

Write blog

Blogs are written in markdown or Typora, you can understand the design of blog by reading Blog | Docusaurus.

(1) The header of blog includes the following information:

```
---
slug: IvorySQL
title: Welcome to IvorySQL community
authors: [official]
authorTwitter: IvorySql
tags: [IvorySQL, Welcome, Database, Join Us]
---
```

Prompt:You can copy the above template to your file and edit it.

Notice:1) Add one space after slug, title, authors, tags.

2) The name of every slug is unique, the Chinese and English versions of the same blog can be the same.

(2) Text format

The text paragraph title is h2/"Second level title";

The body uses the default font size.

(3) Naming format of inserted pictures

[Hello](Hello-banner.png)

(4) Naming format of inserted hyperlink

[name](link)

Github page Download source code and published packages.

Website Contribution Guide

IvorySQL Document Site uses **Antora** to build. Also, IvorySQL Document Site is open source. It consists of three parts, such as **ivorysql_docs**, **ivorysql_web** and **ivory-doc-builder**.

Welcome everyone who is willing to participate in open source work to join us, and remember to follow our code of conduct -.

How to Contribute

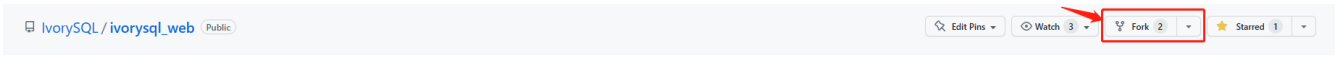
Due to the fact that IvorySQL Document Site is all hosted on Github, this allows any users to **fork** our document repository into their personal repository, make modifications to it, and then submit a PR. After being reviewed by our open source team, the modifications can be updated to our Document Site.

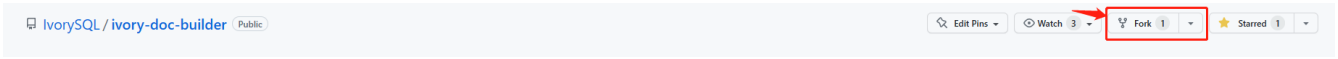
In order to achieve the goat of correcting document errors more conveniently, you first need to establish a personal warehouse according to the size you want to update. As follows:

- If you want to modify the existing content or add a new page, you only need to **fork** ivorysql_docs to your personal repository.



- If you want to participate more deeply in the construction of IvorySQL Document Site, in addition to the **ivorysql_docs**, you also need to **fork** ivorysql_web and ivory-doc-builder to your personal repository.





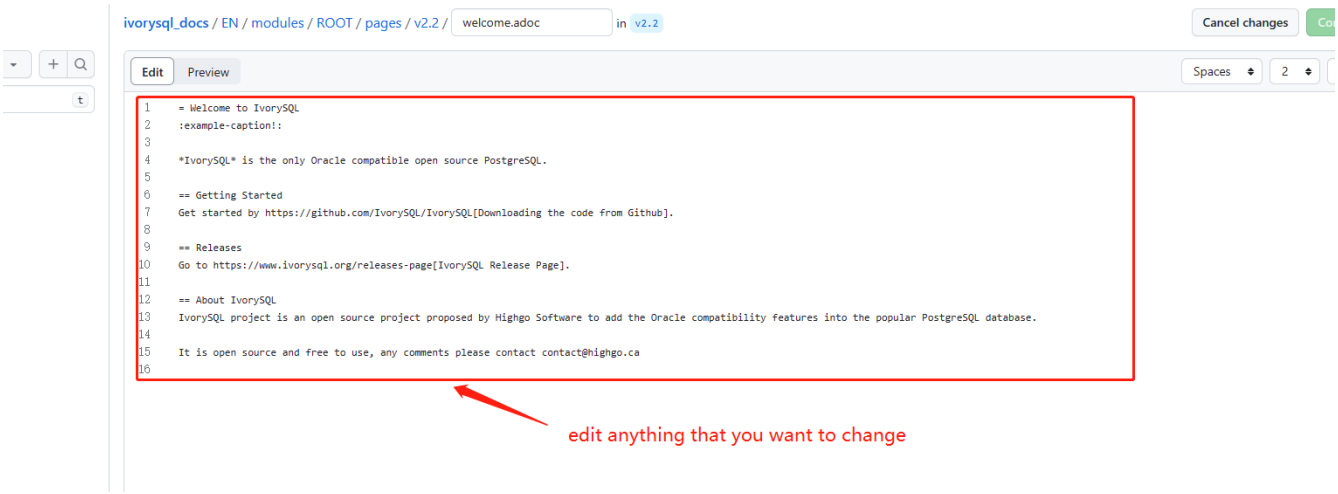
Modify Content

This section will introduce the process of modifying webpage content after discovering that it is inappropriate.

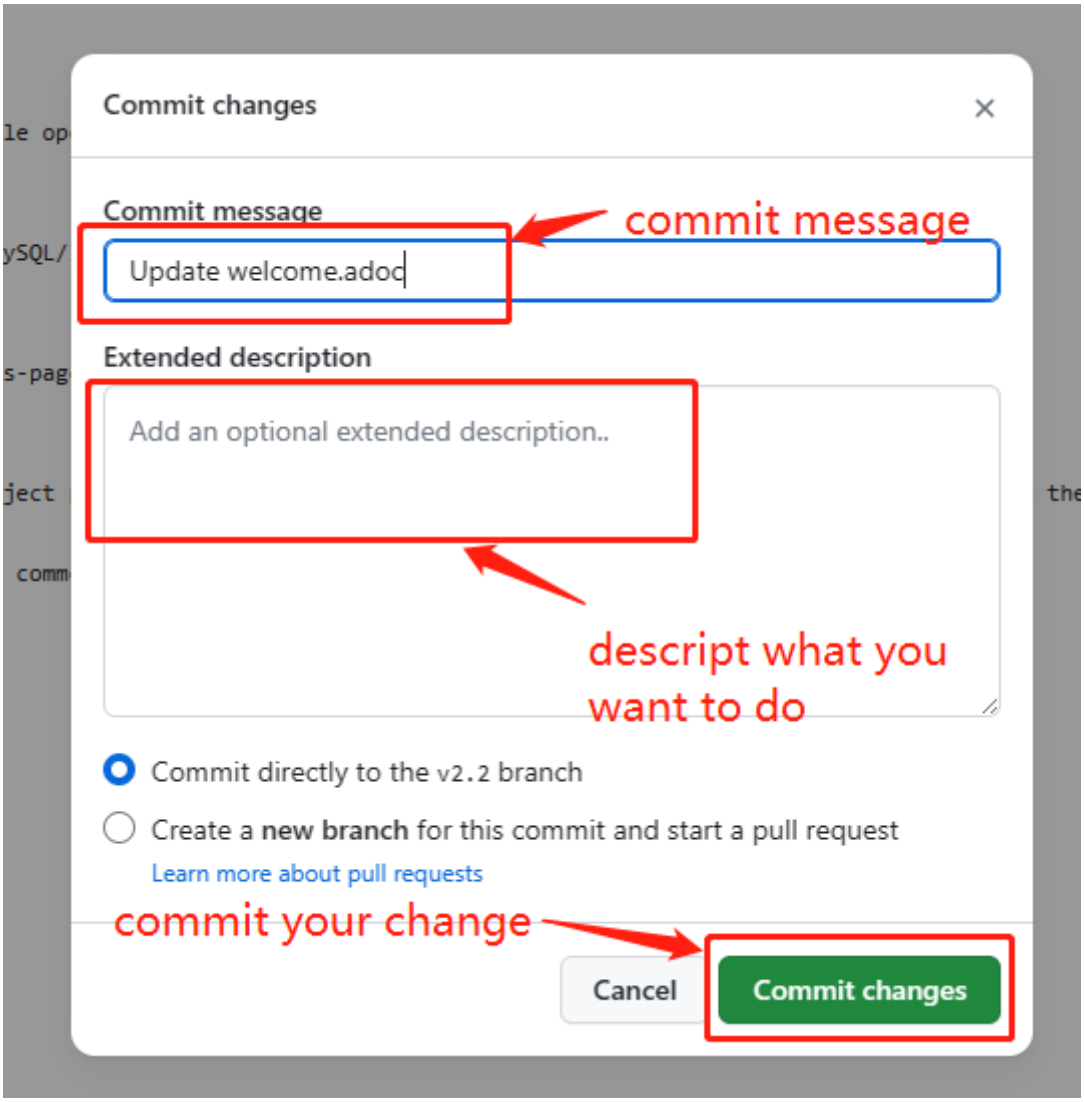
- In the upper right corner of a webpage with incorrect content, there is a button called **edit this page**, click on the button. As shown in the figure:



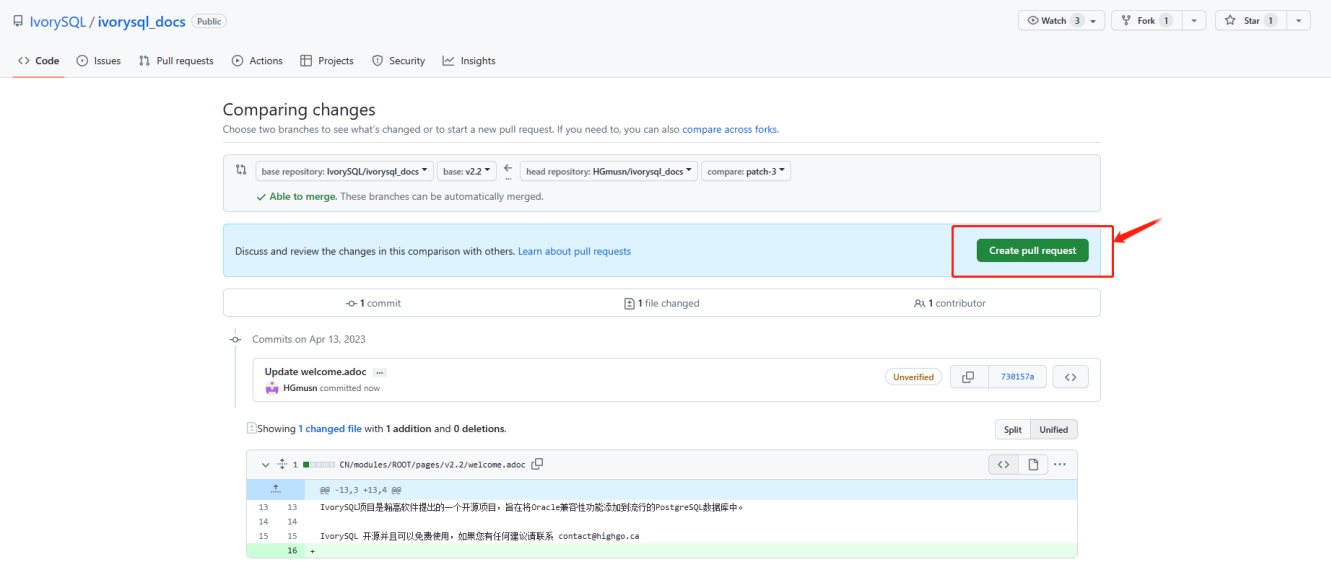
- After clicking, it will redirect to the editing page where we store the current page source **.adoc** file. Please modify the content in the **Asciidoc** format. As shown in the figure:



- After editing is completed. As shown in the figure:



- After confirming the update. As shown in the figure:



- Next, the relevant person of the open source team will be responsible for reviewing the content you submitted. After the review is completed, the updates you submitted will appear on the corresponding page.

Add Page

This section will introduce how to add new page components to a website, and the modifications involved in adding new pages mainly include the following:

- Add the `.adoc` file in the `CN/modules/ROOT/pages/vX.X`.
- Modify the `CN/modules/ROOT/nav.adoc`. If the modification involves modifying or adding images, please modify the images in `images`.
- Add the `.adoc` file in the `EN/modules/ROOT/pages/vX.X`.
- Modify the `EN/modules/ROOT/nav.adoc`. If the modification involves modifying or adding images, please modify the images in `images`.

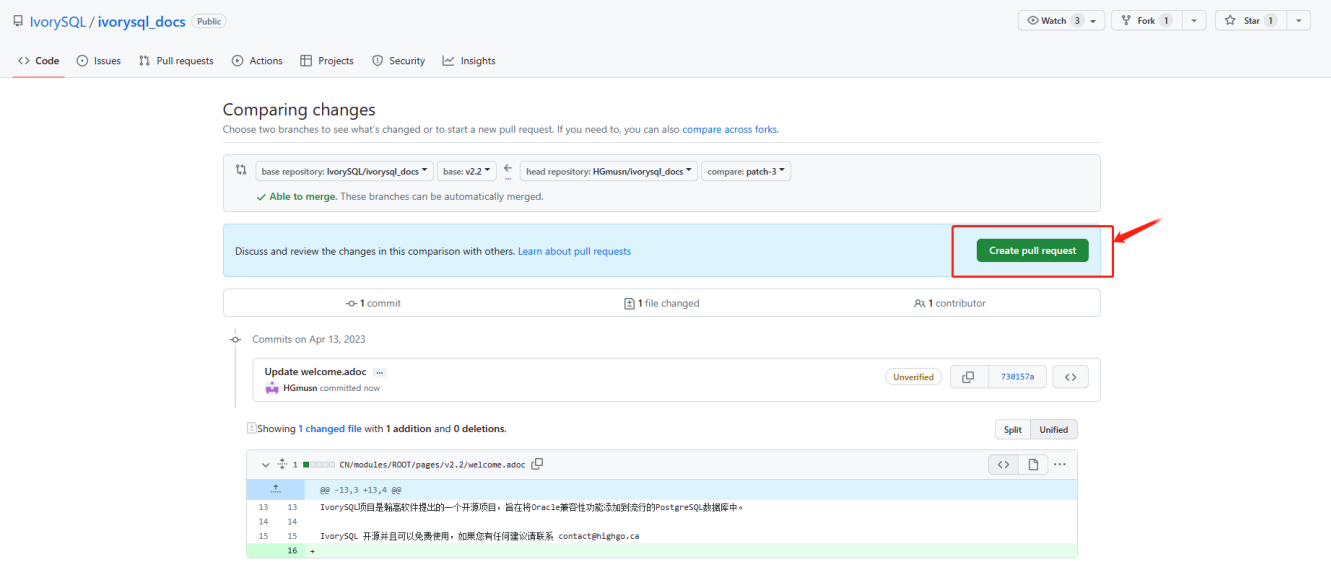
1. Firstly, you need to download the warehouse that you `fork` from `IvorySQL` to your personal computer.

```
git clone https://github.com/$username$/ivorysql_docs.git
```

2. Then, place the `.adoc` file to be added in the correct directory, remembering that both Chinese and English files should be prepared (Chinese and English files should have the same name), and each file should be placed in the correct directory. At the same time, modify the corresponding `nav.adoc` file (the modification method can refer to the existing content of the file).
3. After the above modifications are completed, submit them to the personal warehouse first.

```
git add .
git commit -m "$describe your change$"
git push
```

4. Afterwards, submit the PR as follows



Test

If you are not satisfied with simply submitting on the webpage or only modifying the webpage content, or if you want to modify the webpage UI, this section will help you.

Before reading this section, you need to confirm whether your Github personal repository has **forked** ivorysql_docs, ivorysql_web and ivory-doc-builder.

Deploy web pages

The deployment of web pages is currently the responsibility of the open-source team. We value every submission and issue, so please do not worry about your contribution to the community being buried.

Tip

If you don’ t have much time, you can send an email to ivorysql-docs@ivorysql.org. We will have dedicated staff to handle every your letter, and we looking forward to hearing from you.

IvorySQL Architecture Design

Query Processing

..1. Dual Parser

The dual parser framework is divided into two main parts: the SQL side and the server-side programming language.

Lexical and Syntax Separation on the SQL Side

The basic approach is to introduce a new set of Oracle-compatible syntax and lexical rules. When Oracle compatibility is enabled, the parser follows Oracle-style syntax analysis to generate the corresponding syntax tree.

Implementation Details: Under src/backend/, create a new directory named oracle_parser. Copy scan.l and gram.y from src/backend/parser/ to the oracle_parser directory, renaming them to ora_gram.y and ora_scan.l. Add Oracle-style syntax and lexical analysis code to these files. Copy keywords.c to the oracle_parser directory to store Oracle-specific keywords. Compile the oracle_parser directory into a dynamic library named libparser_oracle.so. When Oracle compatibility is enabled, the ivorysql.conf file is appended to the end of the postgresql.conf file. Add liboracle_parser to the shared_preload_libraries parameter in ivorysql.conf to ensure the dynamic library is loaded automatically when the database starts.

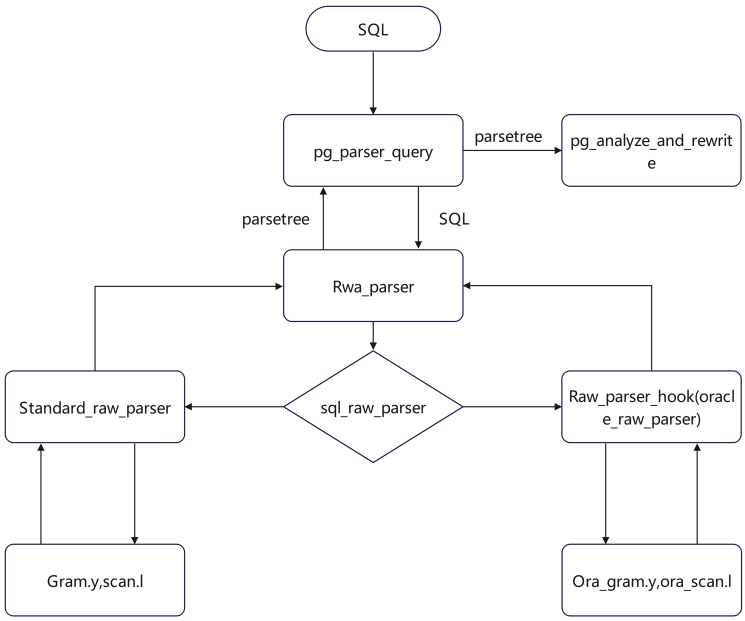
A new function pointer ora_raw_parser is introduced. When the libparser_oracle.so library is loaded, the _PG_init() function assigns the address of oracle_raw_parser() to ora_raw_parser. The _PG_fini() function resets ora_raw_parser to NULL when the compatibility mode is switched.

Each backend process calls the BackendInitialize() function, which sets port→connmode based on the port number the backend process is connected to. If the port is an Oracle-compatible port, connmode is set to 'o'; otherwise, it is set to 'p'.

After this, PostgresMain() calls InitIvorysql(). If port→connmode is 'o', the function SetConfigOption("ivorysql.compatible_mode", "oracle", PGC_USERSET, PGC_S_OVERRIDE) is called. Since this parameter has an assign_hook, executing assign_hook() within SetConfigOption() is equivalent to calling assign_compatible_mode(), which sets sql_raw_parser = ora_raw_parser.

During SQL statement analysis, the function pg_parse_query()→raw_parser() calls either standard_parser() or ora_raw_parser() through the function pointer sql_raw_parser.

The diagram below illustrates what happens during SQL statement analysis.



Lexical and Syntax Separation for Server-Side Programming Language

A new language, pliSQL, is added to the system table pg_language.

Implementation Details: Copy the plpgsql directory from the PostgreSQL source code and rename it to plisql. Rename all files in the plisql directory to start with plisql. Since plpgsql is a language, the registration functions for plpgsql (e.g., plpgsql_validator, plpgsql_call_handler, plpgsql_inline_handler) are renamed to start with plisql. All other functions in the directory are also renamed to start with plisql.

The plisql directory is built as a plugin. If the database mode is Oracle during initdb, this plugin is created. The plugin registers the pliSQL language in the database system tables.

The pliSQL side does not have its own lexical rules; it relies on the lexical rules of the SQL side. Therefore, for pliSQL, Oracle-compatible lexical rules are enforced. The main modification is in the plisql_scanner_init function, which calls ora_scanner_init(). The internal_yylex() function in the plisql directory calls ora_core_yylex.

The syntax rules for pliSQL are defined in plisql/src/pl_gram.y, where the syntax rules for Oracle-compatible PL/SQL blocks are implemented.

When creating a function on the SQL side, if no language is specified:

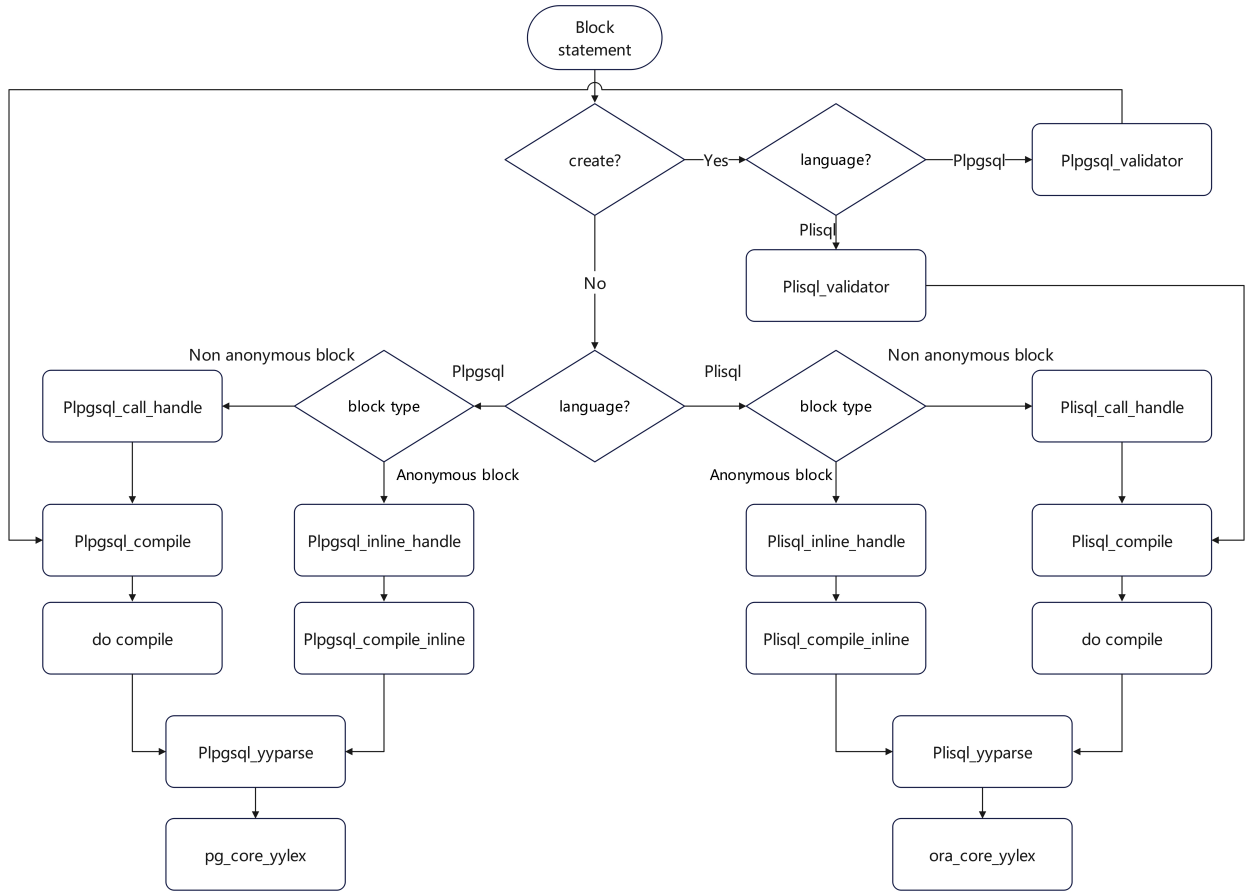
- In Oracle compatibility mode, the default language is plisql.
- In PostgreSQL compatibility mode, the default language is plpgsql.

In oracle_parser, ora_gram.y defaults to plisql, while in the PostgreSQL parser, gram.y defaults to plpgsql.

For anonymous blocks, if no language is specified:

- In Oracle compatibility mode, the default language is plisql.
- In PostgreSQL compatibility mode, the default language is plpgsql.

The ExecuteDoStmt function also determines the default language based on the compatibility mode.

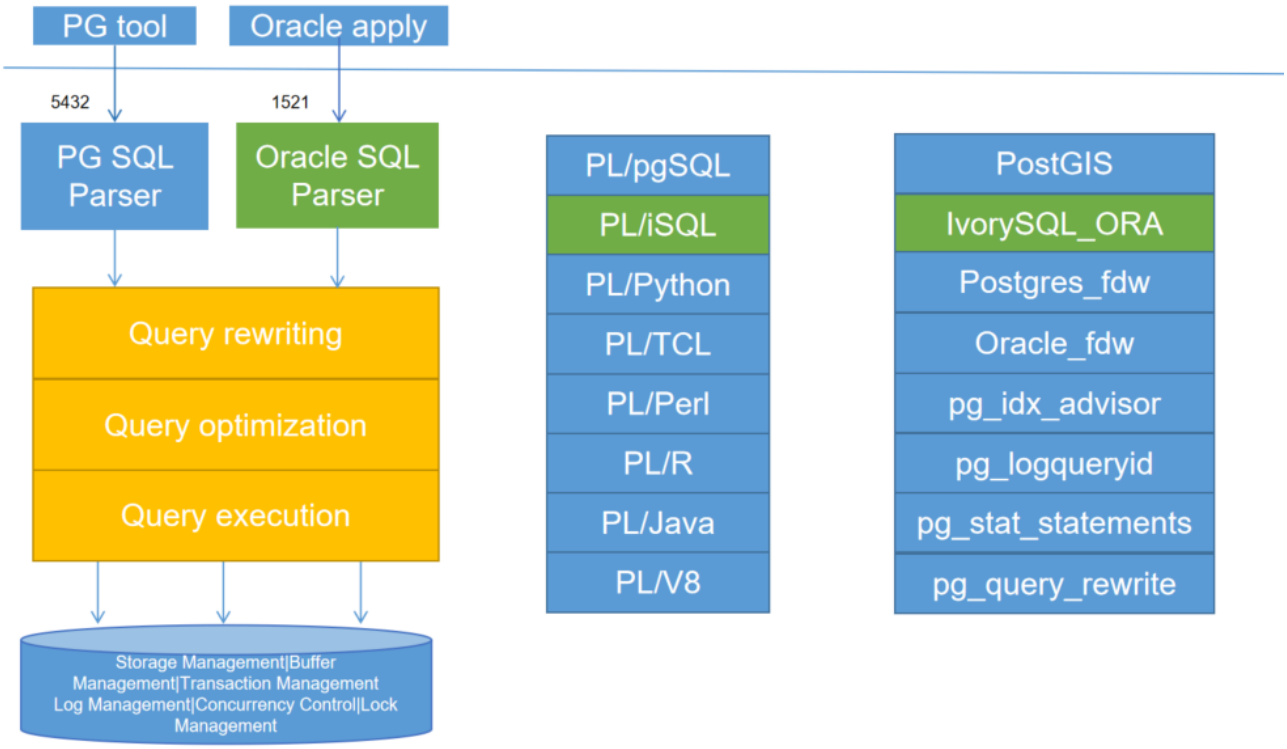


Compatibility Framework

..1. Ivorysql frame design

Chapter 1. Objective

- In order to minimize changes to the original PostgreSQL, it is compatible with Oracle. We need to implement a framework for dual-parser, dual-port, modal PL/PGSQL to implement PL/iSQL. The implementation flowchart is as follows:



Function

Dual-port design

- The Ivorysql port 5432 is kept compatible with the original PostgreSQL situation, so Ivorysql uses another separate port to log in, which defaults to 1521. Log in from this port, and the Oracle compatibility mode is used by default. If you need to log in from port 5432 and also enter compatibility mode, you must configure it using `SET ivorysql.compatible_mode=oracle`.

Parser module design

- In order to minimize the interference between Oracle syntax and PostgreSQL syntax, a parser module has been added to handle oracle-related syntax.

Added PL/iSQL procedural language

- Added PL/iSQL procedural language compatible with Oracle’ s PL/SQL procedural language.

1.1. GUC Framework

New GUC variables

In order to be compatible with Oracle, it is necessary to add some variables for controlling the database execution results on the basis of the original GUC variables, so as to achieve the same behavior as Oracle.

In order to better add compatible GUC parameters and to minimize changes to the PG kernel source code, we need to design a framework to add GUC to a unified location.

Achieve

When adding guc parameters, we need to add them uniformly in the ivy_guc.c. Where `Ivy_ConfigureNamesBool`, `Ivy_ConfigureNamesInt`, `Ivy_ConfigureNamesString`, `Ivy_ConfigureNamesReal` and `Ivy_ConfigureNamesEnum` represent 5 different types of guc parameters. When adding guc parameters, simply add the value of guc to the corresponding array.

New variables(currently)

Variable	Description
ivorysql.compatible_mode	Indicates which database (pg/oracle) is currently compatible with, which can be viewed through the show command. The set command changes this variable, and the reset command resets it to the database mode at the time of connection. Resetting all will affect this variable
ivorysql.database_mode	Indicates the current database schema (pg/oracle), which can be viewed through the show command. The set/reset/reset all command does not affect this variable
ivorysql.datetime_ignore_nls_mask	Indicates whether the date format will be affected by the NLS parameter. The default value is 0, which can be set using the set command. The reset command resets the date format, and the reset all command resets the variable
ivorysql.enable_emptystring_to_NULL	The value is (on/off), and when this variable is on, it will convert the inserted empty string into a NULL value for storage
ivorysql.identifier_case_switch	Set character case conversion mode
ivorysql.listen_address	Indicates the address for compatibility mode listening. When initializing the database, read the configuration from the ivorysql.conf file, modify the value in the configuration file, and restart the database to take effect. This can be viewed through the show command
ivorysql.port	Indicates the port number for connecting in compatibility mode. When initializing the database, read the configuration from the ivorysql.conf file and modify the value in the configuration file. To take effect, restart the database and view it through the show command
nls_date_format	Represents the default date format, which can be viewed through the show command and defaults to 'YYYY-MM-DD'. It can be set through the set command and reset back to the default value through the reset command. The reset all command will reset this variable
nls_length_semantics	Compatible with oracle parameters of the same name, controlling the size of memory occupied by a character
nls_timestamp_format	Compatible with oracle parameters of the same name, controlling date format with time
nls_timestamp_tz_format	Compatible with oracle parameters of the same name, controlling the date format with time zone
shared_preload_libraries	When initializing the database, read from the ivorysql.conf file and view it through the show command. Modify the value in the configuration file and restart the database to take effect.

Example

ivorysql.datetime_ignore_nls_mask

The optional values of this GUC variable are from 0 to 15.

optional values	Types not formatted through NLS
0	Does not block any types, all time formats are formatted by NLS.

1	date
2	timestamp
3	date、timestamp
4	timestamptz
5	date、timestamptz
6	timestamp、timestamptz
7	date、timestamp、timestamptz
8	timestampltz
9	date、timestampltz
10	timestamp、timestampltz
11	date、timestamp、timestampltz
12	timestamptz、timestampltz
13	date、timestamptz、timestampltz
14	timestamp、timestamptz、timestampltz
15	date、timestamp、timestamptz、timestampltz

- Usage Example(date)

check value of nls_date_format and datetime_ignore_nls_mask

```
ivorysql=# set ivorysql.compatible_mode to oracle;
SET
ivorysql=# show nls_date_format;
 nls_date_format
-----------------
 YYYY-MM-DD
(1 row)
ivorysql=# show ivorysql.datetime_ignore_nls_mask;
 ivorysql.datetime_ignore_nls_mask
-----------------------------------
 0
(1 row)
```

create a table for testing

```
ivorysql=# create table test_nls_date(a int, created_at date);
CREATE TABLE
```

insert data

```
ivorysql=# insert into test_nls_date values(1, '2024/04/05');
INSERT 0 1
ivorysql=# select * from test_nls_date;
```

```
 a | created_at
---+------------
 1 | 2024-04-05
(1 row)
```

modify nls_date_format

```
ivorysql=# set nls_date_format to 'yy-mm-dd';
SET
```

Insert NLS formatted data and view, insert successfully.

```
ivorysql=# insert into test_nls_date values(2, '24/04/15');
INSERT 0 1
ivorysql=# select * from test_nls_date;
 a | created_at
---+------------
 1 | 24-04-05
 2 | 24-04-15
(2 rows)
```

Changing the date type to not undergo NLS processing, and inserting the same data, such as changing it to 1 (3, 5, 7, etc.), will result in an error upon data insertion. NLS formatting will not affect the query results for dates.

```
ivorysql=# set ivorysql.datetime_ignore_nls_mask to 1;
SET
ivorysql=# insert into test_nls_date values(3, '24/05/15');
ERROR:  date/time field value out of range: "24/05/15"
LINE 1: insert into test_nls_date values(3, '24/05/15');
                                            ^
HINT:  Perhaps you need a different "datestyle" setting.
ivorysql=#  select * from test_nls_date;
 a | created_at
---+------------
 1 | 24-04-05
 2 | 24-04-15
(2 rows)
```

1.2. Dual-mode design

Objective

- In order to support both the PG mode and Oracle-compatible mode, IvorySQL allows specifying the -m parameter during initdb to obtain either a PG mode database or an Oracle-compatible mode database.

- If the -m parameter is not specified, it defaults to Oracle-compatible mode.
- If the -m parameter is specified as pg, the database will no longer be compatible with Oracle syntax.

Function

- initdb -m initialization requires judgment of different modes, among which oracle mode requires the execution of SQL statements postgres_oracle.bki. The default is Oracle compatibility mode, and the process is as follows:
- Startup: When starting, it determines whether it is an Oracle compatibility mode based on the initialization mode;

```
Description:
database_mode: Used to indicate initialization mode;
database_mode=DB_PG, PG mode, and cannot be switched;
database_mode=DB_ORACLE,  oracle compatibility mode;
```

Test cases

```
Initialize the PG mode:
./initdb -D ../data -m pg

Initialize the Oracle compatibility mode:
./initdb -D ../data -m oracle

or
./initdb -D ../data
```

1.3. initdb Process

IvorySQL supports two database modes during the initialization process:

- PG Mode: Maintains compatibility with native PostgreSQL.
- Oracle Mode (default): Provides Oracle syntax compatibility and enhanced features.

Users can specify the initialization mode using the initdb command parameters to meet compatibility requirements for different scenarios.

Parameter Parsing and Processing

During the initial phase, initdb parses the input command-line parameters.

Parameter	Description	Options	Default Value
-m	Specifies the database mode	oracle/pg	oracle
-C	Sets the case conversion mode for Oracle compatibility	interchange/normal/lowercase	interchange

Parameter Parsing Workflow:

1.Inherits PostgreSQL’s existing parameter processing mechanism.

2.Adds parsing logic for the mode selection parameter -m.

3.Introduces a processing module for the case conversion parameter -C.

File Path Initialization

The setup_data_file_paths() function is executed to configure critical file paths:

```
if (DB_PG == database_mode)
    set_input(&bki_file, "postgres.bki");
else
    set_input(&bki_file, "postgres_oracle.bki");
```

Path Verification Mechanism:

1.Verifies the existence of the BKI files (postgres_oracle.bki / postgres.bki).

2.Confirms the availability of the configuration file templates.

3.Establishes the system directory structure corresponding to the database mode.

Data Directory Initialization Process

The core initialization operations are performed by the initialize_data_directory() function:

Directory Structure Creation

Calls create_data_directory() to create the main data directory (PGDATA).

Uses create_xlog_or_symlink() to establish the WAL log directory.

Iteratively creates standard subdirectories such as base and global.

Configuration File Initialization

Calls set_null_conf() to create an empty postgresql.conf file.

In Oracle mode, additionally creates the ivorysql.conf configuration file.

Calls setup_config() to write configuration information to postgresql.conf. In Oracle mode, additional configuration information is written to ivorysql.conf.

Template Database Bootstrapping

Executes bootstrap_template1() to load the corresponding BKI file and initialize the template1 template database.

IvorySQL additionally sets the database mode (oracle/pg) and case conversion mode for the template1 template database.

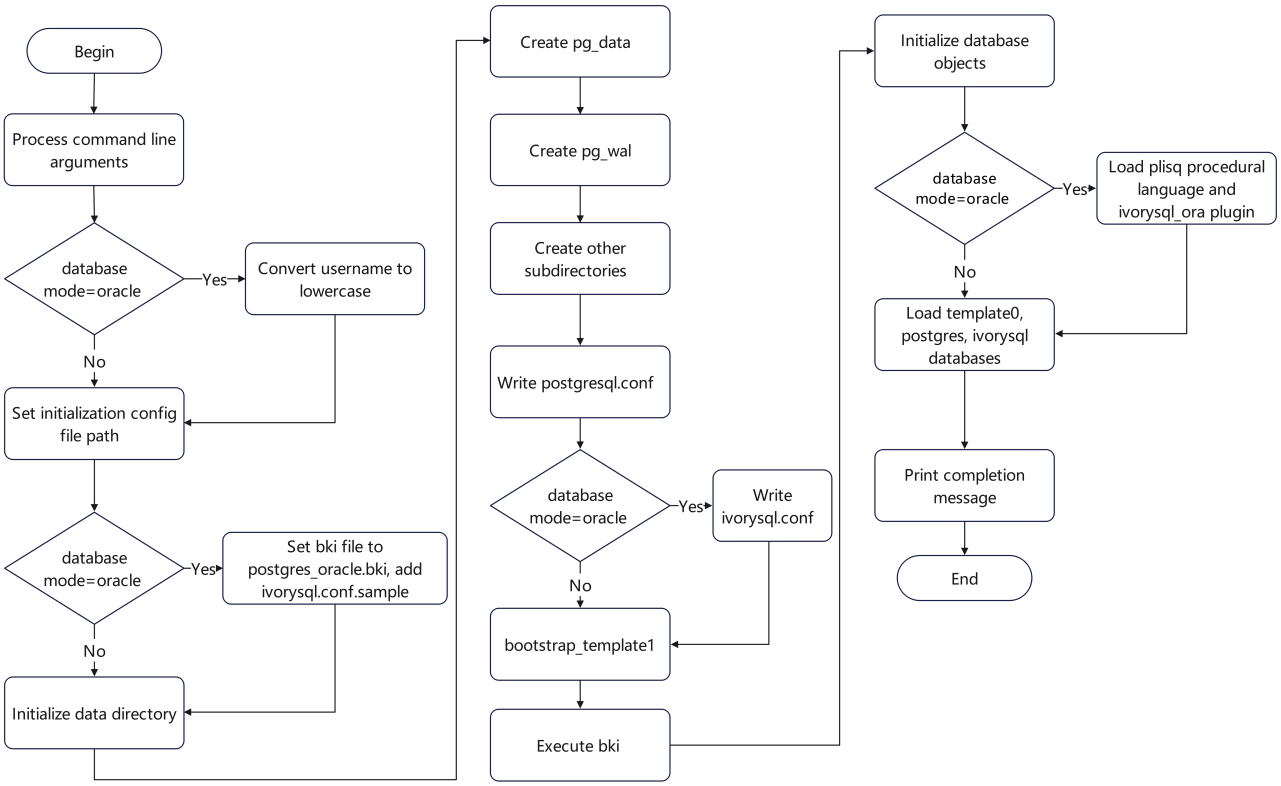
load_plisql(): Installs the PL/iSQL procedural language for Oracle PL/SQL compatibility.

load_ivorysql_ora(): Loads the core Oracle compatibility layer extension.

make_ivorysql(): Creates the default ivorysql database.

When the database mode is Oracle, usernames are forcibly converted to lowercase.

> **Note**
>
> In PostgreSQL, "bki" stands for "Bootstrap Kit." The Bootstrap Kit is a collection of tools and files used internally by PostgreSQL to bootstrap the database system.
>
> The Bootstrap Kit is primarily used to initialize the foundational structure of the PostgreSQL database system, including the system catalog, system tables, and other necessary system objects. These system catalogs and tables store metadata information about the PostgreSQL database system, such as database objects, table structures, and index information.
>
> The genbki.pl script processes system table files like pg_class.h, pg_namespace.h, and pg_proc.h, and generates postgres.bki for initdb to load. This primarily involves loading initialization data, such as creating system tables, system functions, and predefined types.
>
> During the compilation phase before initdb, genbki.pl generates both postgres.bki and postgres_ora.bki. While postgres.bki remains unchanged from PostgreSQL, postgres_ora.bki includes additional system objects for Oracle compatibility.



Compatibility Features

1..1. like

The LIKE syntax in Oracle and IvorySQL is identical. Their difference lies in the expression types: Oracle supports using the LIKE keyword with wildcards for fuzzy queries on columns of numeric, date, and string types. Native PostgreSQL only supports string types, not date or numeric types. IvorySQL achieves Oracle-

compatible LIKE operator functionality by extending data type support and operator overloading.

Implementation Principle

In PostgreSQL, the fundamental string type is text, so LIKE is text-based. Other PostgreSQL types implicitly convert to text, enabling automatic conversion without creating operators. In IvorySQL, the Oracle-compatible string type is varchar2. Therefore, a LIKE operator for varchar2 is created, and other Oracle types also utilize the LIKE operator by implicitly converting to varchar2 without requiring additional operator creation.

In the previous implementation of Oracle-compatible data types, IvorySQL established implicit conversions from certain data types (such as integer, float8, float4) to varchar2, but not directly to text.

To achieve LIKE operator compatibility for these types, there are two approaches:

1、Add a separate LIKE operator for each individual type.

2、Implement a base LIKE operator for varchar2.

In the second approach, since IvorySQL already supports implicit conversions from float8, integer, number, etc., to varchar2, these data types can share the same operator. Thus, only a single LIKE operator for varchar2 needs to be created.

Type Conversion

You can use the following SQL statement to check existing implicit type conversions.

```
-- Check existing implicit conversion paths (where 9503 is the OID of varchar2)
SELECT t1.typname AS source_type, t2.typname AS target_type
FROM pg_cast C
JOIN pg_type t1 ON C.castsource = t1.OID
JOIN pg_type t2 ON C.casttarget = t2.OID
WHERE C.casttarget = 9503;
```

After reviewing, it can be observed that all the types requiring compatibility have implicit conversions defined in this table. Therefore, these types can be directly converted to the text type for fuzzy query operations.

Core Function Implementation

```
CREATE OR REPLACE FUNCTION sys.varchar2like(varchar2, varchar2)
RETURNS bool AS $$
SELECT $1::text LIKE $2::text;
$$ LANGUAGE SQL IMMUTABLE STRICT;

CREATE OPERATOR ~~ (
  PROCEDURE = sys.varchar2like,
  LEFTARG   = varchar2,
  RIGHTARG  = varchar2
);
```

The first code segment defines a function named sys.varchar2like, which accepts two parameters of type

varchar2. By converting them to PostgreSQL’s native text type, it performs standard LIKE pattern matching and ultimately returns a boolean value indicating whether the match is successful.

The second code segment creates an operator named ~~, which uses the previously defined varchar2like function as its implementation. It specifies that both the left and right operands of this operator must be of type varchar2.

This establishes an Oracle-compatible LIKE operator in IvorySQL. When users employ the ~~ operator, it essentially invokes PostgreSQL’s native LIKE functionality after type conversion, thereby achieving semantic compatibility with Oracle’s LIKE behavior.

1..2. RowID

Objective

IvorySQL provides Oracle-compatible RowID functionality. RowID is a pseudo-column automatically generated by the database when a table is created, returning the address of each row in the database.

RowID should have the following characteristics:

1. Logically identifies each row with a unique value
2. Allows quick querying and modification of other columns in the table via ROWID, but cannot be inserted or modified itself
3. Users can control whether this feature is enabled

Implementation Principles

In IvorySQL, the system column ctid represents the physical location of a data row in a table, also known as the tuple identifier, which consists of a pair of values (block number and row index). The ctid allows for quick lookup of data rows in a table, behaving similarly to Oracle’s RowID. However, the ctid value may change (e.g., during UPDATE or VACUUM FULL), making it unsuitable as a long-term row identifier.

We chose a composite type consisting of the table’s OID and a sequence value as the RowID value, where the sequence is a system column. If the RowID functionality is enabled, a sequence named table-id_rowid_seq is created simultaneously with the table. Additionally, in the heap_form_tuple constructor, the length of HeapTupleHeaderData is increased by 8 bytes, and the td→t_infomask = HEAP_HASROWID bit is set to indicate the presence of the RowID.

When the RowID functionality is enabled via the GUC parameter, or by the WITH ROWID option in table creation, or by executing ALTER TABLE … SET WITH ROWID on an existing table, a sequence is created by adding a sequence creation command.

```
/*
         * Build a CREATE SEQUENCE command to create the sequence object,
         * and add it to the list of things to be done before this CREATE/ALTER TABLE
         */
        seqstmt = makeNode(CreateSeqStmt);
        seqstmt->with_rowid = true;
        seqstmt->sequence = makeRangeVar(snamespace, sname, -1);
        seqstmt->options = lcons(makeDefElem("as",
                                             (Node *) makeTypeNameFromOid(INT8OID, -1),
                                             -1),
                                 seqstmt->options);
        seqstmt->options = lcons(makeDefElem("nocache",
```

```
                                NULL,
                                -1),
                            seqstmt->options);
```

To enable fast querying of a row using the RowID pseudo-column, a `UNIQUE` index is automatically created on the RowID column of the table by default to provide efficient query performance.

The implementation of the RowID column as a system attribute column is achieved by adding a new system column in heap.c.

```
/*
 * Compatible Oracle ROWID pseudo column.
 */
static const FormData_pg_attribute a7 = {
    .attname = {"rowid"},
    .atttypid = ROWIDOID,
    .attlen = -1,
    .attnum = RowIdAttributeNumber,
    .attcacheoff = -1,
    .atttypmod = -1,
    .attbyval = false,
    .attalign = TYPALIGN_SHORT,
    .attstorage = TYPSTORAGE_PLAIN,
    .attnotnull = true,
    .attislocal = true,
};
```

A boolean field `relhasrowid` has been added to the `pg_class` system table to indicate whether the `WITH ROWID` option was specified during table creation. If the `WITH ROWID` option is included when creating a table, `relhasrowid` is set to `t`; otherwise, it is set to `f`. This value is also updated when a user executes the `ALTER TABLE ⋯ SET WITH ROWID` or `ALTER TABLE ⋯ SET WITHOUT ROWID` command.

```
       /* T if we generate ROWIDs for rows of rel */
    bool        relhasrowid BKI_DEFAULT(f);
```

Regarding RowID storage, if the RowID pseudo-column functionality is enabled, the `heap_form_tuple` function will add 8 bytes to the `HeapTupleHeaderData` to store the sequence value, depending on whether `tdhasrowid` in the `TupleDesc` parameter is `true`. In the `heap_prepare_insert` function, the `nextval` of the sequence is obtained and stored in the corresponding position in the `HeapTupleHeader`.

```
       if (relation->rd_rel->relhasrowid)
    {
        //  Get the sequence next value
        seqnum = nextval_internal(relation->rd_rowdSeqid, true);
        //  Set the HeapTupleHeader
        HeapTupleSetRowId(tup, seqnum);
```

```
}
```

1..3. OUT Parameter

IvorySQL provides Oracle-compatible OUT parameter functionality, including functions and procedures with OUT parameters, support for OUT parameters in anonymous blocks, and libpq support for OUT parameters.

Implementation Principles

Functions with OUT Parameters

For PL/pgSQL functions, when creating a function, the system table pg_proc stores the total number of parameters (including OUT parameters) and their corresponding data types.

In the interpret_function_parameter_list() function, which processes function parameters, it checks the parameter mode. If the mode is IN OUT, the parameter cannot have a default value.

In the make_return_stmt function, the error handling for detecting OUT parameters has been removed.

By modifying the FuncnameGetCandidates function, function lookup now matches all parameters, including OUT parameters.

During function compilation, a row variable is constructed to hold OUT parameter variables and the return value variable. The compiled function’ s return type (function→fn_rettype) is modified to RECORDOID.

During function execution, the ExecInitFunc function calls the new ExecInitFuncOutParams function to construct OUT parameter computation nodes. The plisql_out_param function separates the function return value and OUT parameter values from the tuple and assigns values to the OUT parameters externally.

Support for OUT Parameters in Anonymous Blocks

To support binding variables in the form of colon placeholders, the ora_scan.l file was modified to add syntax that returns ORAPARAM when a colon placeholder is encountered. In the ora_gram.y file, handling for ORAPARAM was added to the c_expr and plassign_target assignment syntax, constructing an OraParamRef node.

New DO + USING syntax was added:

```
DO [ LANGUAGE lang_name ] code [USING IN | OUT | IN OUT, ...]
```

The ora_gram.y file was modified to support the DO+USING syntax. The DoStmt structure now includes a paramsmode field to store a list of binding variable modes and other information for anonymous blocks.

Modifications to the PBE (Parse, Bind, Execute) process include: In the exec_parse_message function, the parameter mode is identified based on the parameter type OID passed from the application interface. In the exec_bind_message function, for anonymous blocks with DO+USING, the parameter modes following USING are identified, and parameter information is passed to the executor.

Execution of anonymous blocks with OUT parameters: 1. In the PortalStart function, for anonymous block statements, the CreateTupleDescFromParams function is called to construct parameter description information.

1. A new value, PROKIND_ANONYMOUS_BLOCK, was added to the PLiSQL_function member fn_prokind to indicate an anonymous block.
2. In the plisql_exec_function function, anonymous blocks with OUT parameters are evaluated. The plisql_anonymous_return_out_parameter function is called to construct a PLiSQL_row type variable for OUT parameters, and the evaluated row type variable is used as the return value of the anonymous block function.

The libpq interface was modified to support binding by position and by parameter name, involving changes to the SQL layer, PL/pgSQL layer, and libpq interface layer.

1. SQL Layer: A system function get_parameter_description was implemented on the server side. This function returns the relationship between variable names and their positions based on the SQL statement. It is used in libpq interface functions.

The first row of the result shows the SQL type in the name column, followed by rows displaying placeholder names and position information.

```
ivorysql=# select * from get_parameter_description('insert into t values(:x, :y);');
 name  | position
-------+----------
 false |        0
 :x    |        1
 :y    |        2
(3 rows)
```

Support for anonymous block statements is not yet implemented.

1. PL/pgSQL Layer: The PL/pgSQL block adjusts the internal identification of parameters based on their position or name.

The execution function retrieves parameter values and type information from the binding handle.

For OUT parameters, a special handling of return column names is applied. If a parameter is an OUT parameter, its column name is formatted as _column_xxx, where xxx is the position of the OUT parameter. This allows assigning values to OUT parameters from the result set based on the binding position and return position.

At the PLiSQL execution layer, parameter names are converted to internal identifiers (e.g., $number) based on their position. When returning to the client, description information is sent to libpq, ensuring that the returned column names are constructed from parameter names. The libpq side assigns values to OUT parameters based on these column names.

1. libpq Interface Layer: Provides functions for preparing, binding, and executing statements, similar to the corresponding OCI functions.

The general calling process is as follows: Use IvyHandleAlloc to allocate statement and error handles. Call IvyStmtPrepare to prepare the statement. Call IvyBindByPos or IvyBindByName to bind parameters. Call IvyStmtExecute to execute, which can be repeated. Call IvyFreeHandle to release the statement and error handles.

Additionally, a series of interface functions have been implemented, including Ivyconnectdb, Ivystatus, Ivyexec, IvyresultStatus, IvyCreatePreparedStatement, IvybindOutParameterByPos, IvyexecPreparedStatement, IvyexecPreparedStatement2, Ivynfields, Ivyntuples, and Ivyclear.

1..4. %Type & %Rowtype

Purpose

IvorySQL provides Oracle-compatible PL/SQL data type functionality, including %TYPE and %ROWTYPE.

Implementation description

If the reference changes, variables declared with %TYPE or %ROWTYPE will change accordingly

This is a passive process. The current implementation records the dependency between functions (stored procedures) and tablename.columnname. When the referenced type changes, the function (stored procedure) cache is invalidated based on this dependency. As a result, when the function is called, it undergoes forced compilation, ensuring that the function retrieves the latest variable type.

A field named `prostatus` is added to the system table `pg_proc` to indicate the status of a function (stored procedure), with three possible states: validate (v), invalidate (i), and N/A (n). After a function is successfully compiled, this status is set to valid.

When parsing function content, the functions `plisql_parse_cwordtype`, `plisql_parse_wordrowtype`, and `plisql_parse_cwordrowtype` identify objects referenced by %TYPE and %ROWTYPE, record them in the `plisql_referenced_object` linked list, and finally add them to the `pg_depend` system table.

Add a new dependency type, DEPENDENCY_TYPE = 't', to represent %TYPE or %ROWTYPE dependencies. When adding object reference relationships to the `pg_depend` system table, set the dependency type to 't'.

When performing operations on a table (such as modifying the table type or deleting the table), check the `pg_depend` system table. If there exists a dependency type `deptype='t'` and the dependent object is a function, call the `plisql_free_function` function to clear the function cache and update the function status `prostatus` in the `pg_proc` system table to N/A (n).

Variables declared with %TYPE inherit the constraints of the referenced variable.

Add bool notnull member in structure PLiSQL_type;

```
/*
 * Postgres data type
 */
typedef struct PLiSQL_type
{
    bool        notnull;        /* the type is built by variable%type,
                         * isnull or notnull of the variable */
```

In the `plisql_parse_wordtype` or `plisql_parse_cwordtype` functions responsible for parsing %TYPE type functions, determine if the referenced variable type specifies a NOT NULL constraint, and set the `bool notnull` attribute of the returned `datatype` member to `true`. In the `decl_statement` grammar of the `pl_gram.y` file, assign the `notnull` attribute of the `PLiSQL_variable *var` variable based on the `bool notnull` member of `PLiSQL_type`. This way, the constraints of the referenced variable are inherited.

Use table_name%ROWTYPE or view_name%ROWTYPE as the parameter type of a function / stored procedure or the return type of a function

Add support of %ROWTYPE for func_type in ora_gram.y

```
| type_function_name attrs '%' ROWTYPE
                {
                    $$ = makeTypeNameFromNameList(lcons(makeString($1), $2));
                    $$->row_type = true;
                    $$->location = @1;
                }
```

Add a member `bool row_type` to the `TypeName` struct to indicate whether %ROWTYPE is specified.

```
typedef struct TypeName
{
    bool        pct_type;   /* %TYPE specified? */
    bool        row_type;   /* %ROWTYPE specified? */
```

In the `LookupTypeName` function, if the `row_type` member of `TypeName` is `TRUE`, obtain the schema name and table name from the `names` member of `TypeName`, and then retrieve the table's `typeoid`.

Enhancement to INSERT statement

In `ora_gram.y`, add new syntax to support `VALUES` without requiring parentheses '(' afterward.

```
values_clause_no_parens:
            VALUES columnref
                {
                    SelectStmt *n = makeNode(SelectStmt);
                    n->valuesLists = list_make1(list_make1($2));
                    n->valuesIsrow = true;
                    $$ = (Node *) n;
                }
```

Add a field `bool valuesIsrow` to the `SelectStmt` struct to indicate that `values` is a row.

When the `transformInsertStmt` function processes the `INSERT ⋯ VALUES` statement, if `valuesIsrow` is `true`, calls the new function `transformRowExpression` to convert `row_variable` into the equivalent `row_variable.field1, ⋯, row_variable.fieldN`.

Enhancement to UPDATE statement

When transforming an UPDATE statement, i.e., when calling the `transformUpdateStmt` function, if in Oracle compatibility mode, invoke the newly added `transformIvyUpdateTargetList` function. In this new function, for cases where the `origTlist` (i.e., `targetList`) does not contain a name of `row`, execute the `transformUpdateTargetList` function following the original UPDATE transform process.

For cases where the `origTlist` parameter contains a name `row`, since `row` can be used as a column name in PostgreSQL but is a reserved keyword in Oracle and cannot be used as a column name, it is necessary to determine whether `row` is a column in the table being updated. If `row` is not a column in the table to be updated, call the new function `transformUpdateRowTargetList` to convert the sql statement

```
UPDATE table_name SET ROW = row_variable [WHERE ⋯];
```

into equivalent

```
UPDATE table_name SET table_name.column1 = row_variable.column1, table_name.column2 =
row_variable.filed2,⋯ table_name.columnN = row_variable.columnN [WHERE ⋯];
```

If the variable `row_variable` in the statement `UPDATE table_name SET ROW = row_variable` is not a

composite type, execute the `transformUpdateTargetList` function following the original UPDATE transform process. If the variable `row_variable` in the statement is a composite type, the column named `row` in the table is also a composite type, and their type OIDs match, execute the `transformUpdateTargetList` function following the original UPDATE transform process. In all other cases, call the new function `transformUpdateRowTargetList` for processing.

1..5. NLS Parameters

IvorySQL provides Oracle-compatible NLS (National Language Support) parameter functionality, including the following parameters:

Parameter Name	Description
ivorysql.datetime_ignore_nls_mask	Indicates whether the date format ignores the influence of NLS parameters. Default is 0.
nls_length_semantics	Oracle-compatible parameter that specifies whether the size unit for CHAR, VARCHAR, and VARCHAR2 type modifiers is bytes or characters.
nls_date_format	Specifies the default date format, which can be viewed using the SHOW command. Default is 'YYYY-MM-DD'.
nls_timestamp_format	Oracle-compatible parameter that controls the format of timestamps.
nls_timestamp_tz_format	Oracle-compatible parameter that controls the format of timestamps with time zones.
nls_territory	Oracle-compatible parameter that specifies the default region for the database.
nls_iso_currency	Oracle-compatible parameter that assigns a unique currency symbol to a specified country or region.
nls_currency	Oracle-compatible parameter that specifies the symbol for the local currency, corresponding to the placeholder L in numeric string formats.

Implementation Principles

Parameter `nls_length_semantics`

In IvorySQL, data types have an attribute modifier called typmod, which provides additional information about the type. For example, in `VARCHAR(n)`, n is the type modifier. When creating or modifying table columns, you can specify the length type, such as:

```
ivorysql=# create table t1(name varchar2(2 byte));
```

For columns of type `CHAR`, `VARCHAR`, or `VARCHAR2`, if the length type is not explicitly specified, IvorySQL uses the value of the `nls_length_semantics` parameter to determine the length type. The possible values are `BYTE` and `CHAR`, with `BYTE` being the default.

Note that the `nls_length_semantics` parameter only affects newly created columns and has no impact on existing columns.

In the syntax parsing file ora_gram.y, the following code converts the original `CHAR`, `VARCHAR`, or `VARCHAR2` types to oracharchar or oracharbyte based on `nls_length_semantics`.

```
CharacterWithLength:  character '(' Iconst ')'
                {
                    if (ORA_PARSER == compatible_db)
                    {
```

```
                    if (strcmp($1, "bpchar"))
                    {
                        if (nls_length_semantics == NLS_LENGTH_CHAR)
                            $1 = "oravarcharchar";
                        else
                            $1 = "oravarcharbyte";
                    }
                    else
                    {
                        if (nls_length_semantics == NLS_LENGTH_CHAR)
                            $1 = "oracharchar";
                        else
                            $1 = "oracharbyte";
                    }

                    $$ = OracleSystemTypeName($1);
                    $$->typmods = list_make1(makeIntConst($3, @3));
                    $$->location = @1;
                }
                else
                {
                    ...
                }
            }
        ;
```

The input/output functions for the oracharchar and oracharbyte data types in IvorySQL include:

```
oravarcharchartypmodout()
oravarcharbytetypmodout()
oracharbytetypmodout()
oracharchartypmodout()
```

These functions call the C-language function `anychar_typmodout()`, which adjusts the output to include `BYTE` or `CHAR` based on the value of `nls_length_semantics`.

Another role of `nls_length_semantics` is to limit column lengths in tables. If the original `VARCHAR` type is converted to `oracharchar`, the function `oravarcharchar()` is called, and `pg_mbcharcliplen()` calculates the character length instead of the byte length.

```
Datum
oravarcharchar(PG_FUNCTION_ARGS)
{
    VarChar    *source = PG_GETARG_VARCHAR_PP(0);
```

```
    int32       typmod = PG_GETARG_INT32(1);
    bool        isExplicit = PG_GETARG_BOOL(2);
    int32       len,
                maxlen;
    size_t      maxmblen;
    char       *s_data;

    len = VARSIZE_ANY_EXHDR(source);
    s_data = VARDATA_ANY(source);
    maxlen = typmod - VARHDRSZ;

    /* No work if typmod is invalid or supplied data fits it already */
    if (maxlen < 0 || len <= maxlen)
        PG_RETURN_VARCHAR_P(source);

    maxmblen = pg_mbcharcliplen(s_data, len, maxlen);

    ...
}
```

GUC Parameter `datetime_ignore_nls_mask`

This parameter is defined as an int value, where the lower four bits indicate whether to ignore NLS parameter influence on the corresponding date-time formats. The mask definitions are:

```
#define ORADATE_MASK            0x01
#define ORATIMESTAMP_MASK       0x02
#define ORATIMESTAMPTZ_MASK     0x04
#define ORATIMESTAMPLTZ_MASK    0x08
```

In the source code, this GUC parameter is used in the following functions:

```
oradate_in()
oratimestamp_in()
oratimestampltz_in()
oratimestamptz_in()
```

If the corresponding mask is set, the native PostgreSQL processing function is called; otherwise, the compatibility code is invoked, and NLS formats are ignored.

GUC Parameters `nls_date_format`/`nls_timestamp_format`/`nls_timestamp_tz_format`

These three GUC parameters serve as format strings in the function `ora_do_to_timestamp()` for checking and parsing input strings. Their default values are:

```
char       *nls_date_format = "YYYY-MM-DD";
char       *nls_timestamp_format = "YYYY-MM-DD HH24:MI:SS.FF6";
char       *nls_timestamp_tz_format = "YYYY-MM-DD HH24:MI:SS.FF6 TZH:TZM";
```

Setting these values to "pg" disables NLS-specific behavior, reverting to PostgreSQL’s default behavior.

GUC Parameters `nls_currency`/`nls_iso_currency`/`nls_territory`

Currently, `nls_territory` and `nls_iso_currency` support the values CHINA and AMERICA.

The default values are:

```
char       *nls_territory = "AMERICA";
char       *nls_currency = "$";
char       *nls_iso_currency = "AMERICA";
```

These parameters will be used in the Oracle-compatible function `to_number()`.



The `to_number()` function has not yet been implemented.

1..6. Function and stored procedure

Purpose

PostgreSQL supports functions and stored procedures, but there are syntax differences between PostgreSQL and Oracle. To enable Oracle’s PL/SQL statements to run on IvorySQL—i.e., to achieve "syntax compatibility"—IvorySQL adopts the following solution: If an Oracle clause has a counterpart clause with the same function in IvorySQL, it is directly mapped to the corresponding IvorySQL clause; otherwise, only the syntax of the Oracle clause is implemented, while its function is not.

Implementation description

PL/iSQL is the name of the procedural language in IvorySQL, specifically designed to be compatible with Oracle’s PL/SQL statements. To achieve compatibility with Oracle-style syntax for functions and stored procedures, corresponding adjustments need to be made to the psql client tool, the SQL layer, and the PL/iSQL layer.

Client tool psql

Oracle’s sqlplus tool uses a slash (/) to terminate functions and stored procedures. IvorySQL’s client tool, psql, needs to be compatible with the same syntax. This means that the conventional mechanism—where statements are sent to the server upon encountering a semicolon—becomes ineffective when dealing with Oracle-style function and stored procedure commands; instead, a slash (/) is used to send the commands.

To this end the following fields are added to the PsqlScanStateData structure:

```
    bool        cancel_semicolon_terminator; /* not send command when semicolon found
*/

    /*
     * State to track boundaries of Oracle ANONYMOUS BLOCK.
     * Case 1: Statements starting with << ident >> is Oracle anonymous block.
```

```
     */
    int     token_count;            /* # of tokens, not blank or newline since start
of statement */
    bool        anonymous_label_start;      /* T if the first token is "<<" */
    bool        anonymous_label_ident;  /* T if the second token is an identifier */
    bool        anonymous_label_end;        /* T if the third token is ">>" */

    /*
     * Case 2: DECLARE BEGIN ... END is Oracle anonymous block syntax.
     * DECLARE can also be a PostgreSQL cursor declaration statement, we need to tell
this.
     */
    bool        maybe_anonymous_declare_start;  /* T if the first token is DECLARE */
    int     token_cursor_idx;       /* the position of keyword CURSOR in SQL statement
*/

    /*
     * Case 3: DECLARE BEGIN ... END is Oracle anonymous block syntax.
     * BEGIN can also be a PostgreSQL transaction statement.
     */
    bool        maybe_anonymous_begin_start;    /* T if the first token is BEGIN */
```

Meanwhile, modify ora_psqlscan.l and add or update the corresponding lexical rules. Below is a code snippet example:

```
{
    if (is_oracle_slash(cur_state, cur_state->scanline))
    {
        /* Terminate lexing temporarily */
        cur_state->cancel_semicolon_terminator = false;
        cur_state->maybe_anonymous_declare_start = false;
        cur_state->maybe_anonymous_begin_start = false;
        cur_state->anonymous_label_start = false;
        cur_state->anonymous_label_ident = false;
        cur_state->anonymous_label_end = false;
        cur_state->start_state = YY_START;
        cur_state->token_count = 0;
        cur_state->token_cursor_idx = 0;
        cur_state->identifier_count = 0;
        cur_state->begin_depth = 0;
        cur_state->ora_plsql_expect_end_symbol = END_SYMBOL_INVALID;
        return LEXRES_SEMI;
    }
```

```
    ECHO;
```

The psql tool needs to detect the meaning of the slash (/), to avoid identifying slashes in comments and other parts as terminators. To this end, a separate interface is_oracle_slash is added in the oracle_fe_utils/ora_psqlscan.l file for detection.

```
bool
is_oracle_slash(PsqlScanState state, const char *line)
{
    bool result = false;

    switch (state->start_state)
    {
        case INITIAL:
        case xqs:   /* treat these like INITIAL */
            {
                int len, i;
                bool has_slash = false;

                len = strlen(line);
                for (i = 0; i < len; i++)
                {
                    /* allow special char */
                    if (line[i] == '\t' ||
                        line[i] == '\n' ||
                        line[i] == '\r' ||
                        line[i] == ' ')
                        continue;

                    if (line[i] == '/')
                    {
                        if (has_slash)
                            break;
                        has_slash = true;
                        continue;
                    }
                    /* others */
                    break;
                }

                if (i == len && has_slash)
                    result = true;
            }
```

```
            break;
        default:
            break;
    }

    return result;
}
```

SQL layer

The SQL layer needs to be able to recognize the creation syntax for functions and stored procedures, and this is achieved by modifying ora_base_yylex. This function prefetches and caches tokens: if the token follows Oracle syntax, it organizes an SCONST and sends it to the PL/SQL layer; otherwise, it retrieves the previously preread tokens from the stack and processes them according to the native PostgreSQL logic.

The following fields are added to the ora_base_yy_extra_type structure:

```
    /*
     * The native PG only cache one-token info include yylloc, yylval and token
     * number in yyextra, IvorySQL cache multiple tokens info using two arrays.
     */
    int max_pushbacks;      /* the max size of cache array */
    int loc_pushback;       /* # of used tokens */
    int num_pushbacks;      /* # of cached tokens */
    int *pushback_token;            /* token number array */
    TokenAuxData *pushback_auxdata; /* auxdata array */

    OraBodyStyle body_style;
    int          body_start;
    int          body_level;
```

Add operation interfaces for the token stack:

push_back_token
forward_token
ora_internal_yylex
internal_yylex

⊠⊠ In the ora_base_yylex function, when creating functions, procedures, or anonymous blocks, some tokens are preread. These tokens are cached into the stack using the aforementioned structure, and this is done to construct an SCONST that conforms to Oracle PL/SQL syntax and send it to the PL/iSQL layer for processing. For details, please refer to the source code.

PL/iSQL layer

This part mainly modifies the pl_gram.y file to achieve compatibility with the syntax of PL/SQL functions and stored procedures. It enables compatibility with Oracle PL/SQL syntax forms without affecting PostgreSQL’ s native PL/pgSQL. Below is a code example for the compatibility of the DECLARE section; for more details, please refer to the IvorySQL source code.

```
/*
 * The declaration section of the outermost block in Oracle does not have the DECLARE
keyword.
 */
ora_outermost_pl_block: ora_decl_sect K_BEGIN proc_sect exception_sect K_END opt_label
                    {
                        PLiSQL_stmt_block *new;

                        new = palloc0(sizeof(PLiSQL_stmt_block));

                        new->cmd_type   = PLISQL_STMT_BLOCK;
                        new->lineno     = plisql_location_to_lineno(@2);
                        new->stmtid     = ++plisql_curr_compile->nstatements;
                        new->label      = $1.label;
                        new->n_initvars = $1.n_initvars;
                        new->initvarnos = $1.initvarnos;
                        new->body       = $3;
                        new->exceptions = $4;

                        check_labels($1.label, $6, @6);
                        plisql_ns_pop();

                        $$ = (PLiSQL_stmt *)new;
                    }
                ;

ora_decl_sect: opt_block_label opt_ora_decl_start opt_ora_decl_stmts
                {
                        if ($2)
                        {
                                if ($1 == NULL)
                                {
                                        plisql_ns_push(NULL, PLISQL_LABEL_BLOCK);
                                }
                        }
                }
                opt_ora_decl_stmts
                {
                        if ($4)
                        {
                                plisql_IdentifierLookup = IDENTIFIER_LOOKUP_NORMAL;
                                $$.label          = ($1 == NULL ?
```

```
plisql_curr_compile->namelabel : $1);
                                if ($2 && $1 == NULL)
                                        $$.popname = true;
                                else
                                        $$.popname = false;
                                /* Remember variables declared in decl_stmts */
                                $$.n_initvars =
plisql_add_initdatums(&($$.initvarnos));
                        }
                        else
                        {
                                plisql_IdentifierLookup = IDENTIFIER_LOOKUP_NORMAL;
                                $$.label          = ($1 == NULL ?
plisql_curr_compile->namelabel : $1);
                                $$.n_initvars = 0;
                                if ($2 && $1 == NULL)
                                        $$.popname = true;
                                else
                                        $$.popname = false;
                                $$.initvarnos = NULL;
                        }
                }
                ;

opt_ora_decl_start: K_DECLARE
                    {
                        /* Forget any variables created before block */
                        plisql_add_initdatums(NULL);
                        /*
                         * Disable scanner lookup of identifiers while
                         * we process the decl_stmts
                         */
                        plisql_IdentifierLookup = IDENTIFIER_LOOKUP_DECLARE;
                        $$ = true;
                    }
                | /*EMPTY*/
                    {
                        /* Forget any variables created before block */
                        plisql_add_initdatums(NULL);
                        /*
                         * Disable scanner lookup of identifiers while
                         * we process the decl_stmts
                         */
```

```
                        plisql_IdentifierLookup = IDENTIFIER_LOOKUP_DECLARE;
                        $$ = false;
                    }
                ;

opt_ora_decl_stmts:
                ora_decl_stmts
                    {
                        $$ = true;
                    }
                | /*EMPTY*/
                    {
                        $$ = false;
                    }

ora_decl_stmts: ora_decl_stmts ora_decl_stmt
                | ora_decl_stmt
                ;

ora_decl_stmt: decl_statement
                ;
```

1..7. Nested Subfunctions

Objective

- Nested subfunctions refer to functions or procedures defined inside another function, stored procedure, or anonymous block; they are also called subprocs or inner functions.
- Parent functions are the outer functions, stored procedures, or anonymous blocks that host nested subfunctions and are responsible for invoking them during execution.

Implementation Notes

Syntax Recognition for Nested Subfunctions

Detecting Nested Definitions

When a `DECLARE` block contains a `function ⋯ is/as begin ⋯ end` construct, `pl_gram.y` calls `plisql_build_subproc_function()` (similar to creating a regular function and updating the entry in `pg_proc`):

1. Create a `PLiSQL_subproc_function` entry in the parent `PLiSQL_function`’ s `subprocfuncs[]` array to store the name, arguments, return type, and other attributes, and record the index `fno` as the identifier of this subfunction.
2. Call `plisql_check_subprocfunc_properties()` to validate the combination of declaration and definition attributes.

Storing Datum Entries

Nested subfunctions share the parent’ s datum table. During compilation, `PLiSQL_function→datums` describes variables and record fields inside the subfunction, while `PLiSQL_execstate→datums` keeps the

runtime values.

Preserving Polymorphic Templates

If the subfunction uses polymorphic parameters, the parser stores its source code in `subprocfunc→src` and sets `has_poly_argument` to `true` so that the executor can recompile it for each distinct argument type.

Recompiling the Parent Program

1. The parent `PLiSQL_function` gains a `subprocfuncs` array, each element being the `PLiSQL_subproc_function` created earlier.
2. Each `PLiSQL_subproc_function` has a `HTAB *poly_tab` pointer that is initialized on the first compilation when `has_poly_argument` is `true`. The hash key is `PLiSQL_func_hashkey`, which records the subfunction's `fno` and input argument types; the value is the compiled `PLiSQL_function *` execution context.

Name Resolution During Invocation

1. PostgreSQL builds a `ParseState` structure during compilation. `plisql_subprocfunc_ref()` locates the parent `PLiSQL_function` through `ParseState→p_subprocfunc_hook()` and calls `plisql_ns_lookup()` to gather all `fno` values for subfunctions sharing the same name, then selects the best match based on argument count and types.
2. When `FuncExpr` nodes are created, the subfunction call is tagged for later execution: `function_from = FUNC_FROM_SUBPROCFUNC`, `parent_func` points to the parent `PLiSQL_function`, and `funcid = fno`.
3. In `plisql_call_handler()`, when `function_from == FUNC_FROM_SUBPROCFUNC`, the runtime fetches the appropriate `PLiSQL_subproc_function` via the pair `(parent_func, fno)`:
   a. For non-polymorphic subfunctions, reuse the precompiled action tree stored in `subprocfunc→function`.
   b. For polymorphic subfunctions, probe `poly_tab`; if there is no cached plan, call `plisql_dynamic_compile_subproc()` to compile one and store it in the cache.
4. Before execution, `plisql_init_subprocfunc_globalvar()` forks relevant entries from the parent's datum table so the subfunction can access the latest parent variables without polluting the parent scope. After execution, `plisql_assign_out_subprocfunc_globalvar()` writes back the necessary variables.

Module Design

PL/iSQL Grammar Extensions

- `pl_gram.y` adds productions for subprocedure declarations and nested definitions, and records metadata such as `lastoutvardno` and subprocedure descriptors.
- Nested subfunctions can reference variables from the parent scope, other subprocedures, and user-defined types.

Whenever a `function … is/as begin … end` construct is seen inside a `DECLARE` block, `pl_gram.y` invokes `plisql_build_subproc_function()`:

1. Insert a `PLiSQL_subproc_function` entry into the parent `PLiSQL_function→subprocfuncs[]`, storing the name, arguments, return type, and other attributes, and assign an index `fno`.
2. Call `plisql_check_subprocfunc_properties()` to verify that declarations and definitions are consistent and to prevent duplicate or missing declarations from introducing semantic errors.

Datum Storage

The parent program's datum tables hold the variables accessible to nested subfunctions during compilation and execution:

1. `PLiSQL_function→datums` preserves the variable and record metadata visible during compilation.

2. `PLiSQL_execstate→datums` carries the live values at runtime.

Polymorphic Templates

When a subfunction contains polymorphic arguments, the parser will:

1. Copy the subfunction source text into `subprocfunc→src`.
2. Set `has_poly_argument = true` to prepare for dynamic recompilation based on actual argument types.

Parent Recompilation

- The parent `PLiSQL_function` includes a `subprocfuncs` array, with each element corresponding to a `PLiSQL_subproc_function`.
- Each `PLiSQL_subproc_function` maintains an optional `HTAB *poly_tab`; when `has_poly_argument` is `true`, the cache is initialized on the first compile. Keys are `PLiSQL_func_hashkey` (subfunction `fno` plus argument types), and values are the compiled `PLiSQL_function` plans.

Parser Hooks

During compilation, PostgreSQL creates a `ParseState`. `plisql_subprocfunc_ref()` plugs into `ParseState→p_subprocfunc_hook`, reusing the namespace lookup logic to gather candidates. `plisql_get_subprocfunc_detail()` then chooses the best match based on argument count, types, and named parameters, enabling overloaded dispatch.

FuncExpr Annotation

When constructing `FuncExpr` nodes, the compiler attaches metadata so the executor can recognize nested calls:

- `function_from = FUNC_FROM_SUBPROCFUNC`.
- `parent_func` references the owning `PLiSQL_function`.
- `funcid = fno`, enabling direct lookup of the subfunction definition.

Nested Subfunction Lookup

- `plisql_subprocfunc_ref()` implements `ParseState→p_subprocfunc_hook` and reuses the namespace search to find nested subfunctions.
- `plisql_get_subprocfunc_detail()` applies matching rules for argument count, type, and naming to pick the optimal overload.

Execution Path

1. `plisql_call_handler()` checks `function_from`; if it is a nested subfunction, the handler locates `PLiSQL_subproc_function` via `(parent_func, fno)`.
2. For regular subfunctions, reuse the cached plan stored in `subprocfunc→function`.
3. For polymorphic subfunctions, consult `poly_tab`; on a miss, call `plisql_dynamic_compile_subproc()` to build and cache a specialized plan.

Variable Synchronization

- `plisql_init_subprocfunc_globalvar()` copies the relevant entries from the parent datum table before the subfunction runs to expose the latest state.
- `plisql_assign_out_subprocfunc_globalvar()` writes back OUT/INOUT variables after execution to keep parent and child scopes consistent without mutual pollution.

Statement Dispatch in psql

- `psqlscan.l` adjusts the push/pop logic of `proc_func_define_level` and `begin_depth` so the nested

subfunction body is transmitted to the SQL engine as a whole.

- Statements are sent only when the nesting depth returns to zero and a semicolon is reached, avoiding partial dispatch of subfunction blocks.

Retrieving Return Information on the SQL Side

- Regular functions obtain metadata via `funcid` from `pg_proc`; nested subfunctions rely on `FuncExpr.parent_func`, which holds the parent `PLiSQL_function`.
- A set of callback pointers (registered through `plisql_register_internal_func()`) allows the SQL layer to fetch nested subfunction names, return types, and OUT parameter information on demand.

1..8. Force View

Purpose

- Force View offers Oracle-compatible `CREATE [OR REPLACE] FORCE VIEW` and `ALTER VIEW ⋯ COMPILE` behavior, allowing developers to persist a placeholder view even when dependencies are missing and then switch it back to a normal view once dependencies are available.
- The system records the original SQL text and identifier case mode in a dedicated catalog entry so that automatic or manual recompilation can restore the view with Oracle-style warnings and error reporting.

Implementation Description

In PostgreSQL, a base table referenced by a view cannot be dropped unless `CASCADE` is used. Oracle, however, allows the base table to be dropped while retaining the Force View and marking it as invalid. To align with Oracle semantics, IvorySQL introduces new grammar rules and a catalog that stores Force View metadata so that the Force state can be shared across sessions.

Grammar and Parsing Entry Points

Force View Syntax Support

- Register the Force View keywords in `src/backend/oracle_parser/ora_gram.y`.
- Set `ViewStmt→force` during the grammar phase, and generate the `AT_ForceViewCompile` option when parsing `AlterTableStmt`.
- Preserve `ViewStmt→stmt_literal` for later reuse when the placeholder is materialized.

```
/* Insert or update the pg_force_view catalog as needed */
if (need_store)
{
    ......

    StoreForceViewQuery(address.objectId, stmt->replace, ident_case, stmt-
>stmt_literal ? stmt->stmt_literal : queryString);
}
```

AST Field Extensions

- Add `bool force` and `char *stmt_literal` to `ViewStmt` in `src/include/nodes/parsenodes.h`.
- Define `AT_ForceViewCompile` consistently between `parsenodes.h` and `tablecmds.c` so that `ALTER VIEW ⋯ COMPILE` flows into the regular alter-table machinery.
- `parse_analyze` still follows the native path; Force mode intervenes only after a parsing error occurs.

Parsing Entry

- `DefineView()` in `src/backend/commands/view.c` now handles both normal and Force views.
- The function first attempts normal parsing inside a `PG_TRY/PG_CATCH`; if the semantic analysis fails, it checks `stmt→force` to decide whether to enter the Force View branch.
- On success it continues to `StoreViewQuery()`; on failure it falls back to the Force View logic so that the view object still exists.

Force View Metadata

`pg_force_view` Catalog

- `src/include/catalog/pg_force_view.h` defines the catalog table, whose fields include the view OID (`fvoid`), identifier case (`ident_case`), and the original SQL text (`source`).
- The unique index `pg_force_view_fvoid_index` plus the syscache `FORCEVIEWOID` (declared in `src/include/catalog/syscache_info.h`) enable lookups by view OID.
- The catalog enables TOAST so lengthy SQL definitions are preserved in full, covering complex migration scripts.

```
CATALOG(pg_force_view,9120,ForceViewRelationId)
{
    /* oid of force view */
    Oid         fvoid;

    /* see IDENT_CASE__xxx constants below */
    char        ident_case;

#ifdef CATALOG_VARLEN           /* variable-length fields start here */

    /* sql definition */
    text        source;
#endif
} FormData_pg_force_view;
```

View State

- `bool rel_is_force_view(Oid relid)` in `src/backend/commands/view.c` determines whether a view is in Force state by checking whether the `_RETURN` rule exists.
- Force views still register as `relkind = RELKIND_VIEW`, avoiding extra compatibility branches.
- `pg_class.relhasrules` is set to `false` while in placeholder mode, which becomes part of the detection logic.

Creation and Replacement Flow

Normal View

- After successful parsing, `DefineView()` calls `DefineVirtualRelation()` to populate `pg_class`, `pg_attribute`, and related catalogs.
- `StoreViewQuery()` generates the `_RETURN` rule and records dependencies.
- This path never touches `pg_force_view`; the view is immediately ready for use.

Force View

- `CreateForceVirtualPlaceholder()` in `src/backend/commands/view.c` creates or reuses a placeholder view:
- If the view does not exist, it calls `DefineVirtualRelation()` to create the base object without a `_RETURN` rule.
- If a Force View already exists, it reuses the current record and updates column definitions or cleans up legacy metadata.
- If a normal view exists and `OR REPLACE` is specified, it invokes `make_view_invalid()` to invalidate the old definition before installing the placeholder.
- `StoreForceViewQuery()` persists `stmt_literal` and the current `ivorysql.identifier_case_switch` to `pg_force_view` so identifier case semantics can be restored later.
- After the placeholder is created, the client receives `WARNING: View created with compilation errors`, indicating the view cannot yet be used.

Dependency Invalidations and Rollback

Active Invalidation Logic

- `make_view_invalid()` is triggered when dependencies are dropped, altered, or otherwise compromised.
- The routine removes the `_RETURN` rule, clears `pg_depend` entries, resets `pg_class.relhasrules`, and truncates `pg_attribute` column metadata.
- It also captures `CREATE FORCE VIEW ⋯ AS <pg_get_viewdef>` and saves it in `pg_force_view`, while setting `ident_case` to `IDENT_CASE_UNDEFINE` to indicate the SQL is system-generated.

Observable Behavior After Invalidation

- The view remains visible in metadata catalogs but runtime access detects the Force flag.
- Because `_RETURN` is missing, the executor calls `compile_force_view()` when opening the view; if compilation fails, it raises `view "<schema>.<name>" has errors`.
- Users can recover by issuing `ALTER VIEW ⋯ COMPILE` or `CREATE OR REPLACE FORCE VIEW` once dependencies are ready.

Automatic and Manual Compilation

Automatic Compilation Triggers

- `addRangeTableEntry()` in `parse_relation.c` and the target-relation open logic in `parse_clause.c` call `compile_force_view()` after detecting a Force view.
- The function reruns `raw_parser` and `parse_analyze`; on success it reinstalls the `_RETURN` rule, and on failure it aborts the statement with the encountered error.

Manual Compilation

- `AT_ForceViewCompile` executes during phase 2 in `tablecmds.c`, acquiring `AccessExclusiveLock` before invoking `compile_force_view()`.
- Successful compilation behaves like a normal `ALTER VIEW`; failures emit `WARNING: View altered with compilation errors`, and the view stays in placeholder mode.

Column Checks and Metadata Updates

- `compile_force_view()` reads `pg_force_view.source`, rebuilds a `ViewStmt`, and calls `compile_force_view_internal()`.
- The routine uses `checkViewColumns()` to compare legacy columns, allowing additions but rejecting incompatible type changes; new columns are applied through `AT_AddColumnToView()`.
- `_RETURN` is regenerated via `StoreViewQuery()`, and `DeleteForceView()` removes the catalog record so the

view becomes a standard one again.

1..9. Case Conversion

Purpose

To meet Oracle’s quoted identifier case compatibility requirements, IvorySQL has designed three case conversion modes for quoted identifiers.

Implementation Details

If the parameter `-C` is appended during database initialization, with values of `normal/interchange/lowercase`, the `Initdb.c→main()` function in the code will process this parameter and set the global variable `caseswitchmode` according to the parameter value. Then the `initdb` command will start a `postgres` process in `-boot` mode to set up the `template1` template database, while passing the parameter `-C ivorysql.identifier_case_switch=caseswitchmode` to the new process.

This newly started backend process will write the `identifier_case_switch` information to the `pg_control` file through the following code:

```
BootstrapModeMain() -> BootStrapXLOG();
    /* save database compatible level value */
    ControlFile->dbmode = bootstrap_database_mode;
    ControlFile->casemode = identifier_case_switch;

    /* some additional ControlFile fields are set in WriteControlFile() */
    WriteControlFile();
```

When a user starts the database using the `pg_ctl` command, the `postmaster` process will read the contents of the `pg_control` file. The code call path is:

```
PostmasterMain()-->SetCaseGucOption()-->GetCaseSwitchModeFromControl()
```

After reading the parameter value, the `SetConfigOption()` function is called to assign the value.

When each new backend process starts, since it is forked from the `postmaster` process, it automatically has the same `ivorysql.identifier_case_switch` parameter value. First, it processes the `startup` packet, and if it contains `database` or `user` parameters, the parameter values are processed accordingly based on `identifier_case_switch`.

The source code is:

```
BackendMain()->BackendInitialize()-->ProcessStartupPacket()
```

Additionally, when processing user SQL statements, if they contain identifiers, the same processing is performed. The code is distributed in: SplitIdentifierString(), quoteOneName() and SplitGUCList() functions.

1..10. sys_guid Function

Purpose

IvorySQL’s sys_guid() is a powerful random number generation function that generates and returns a 16-byte database-level unique identifier (raw value).

Implementation Description

The sys_guid() function of IvorySQL is implemented by modifying the code of the uuid-ossp plugin. To make full use of various underlying libraries of uuid, the following logic is adopted:

1. If the uuid-ossp exists in the system, uuid_make() will be used;
2. If the uuid-e2fs exists in the system, uuid_generate_random() will be used;
3. Otherwise use arc4random();

Meanwhile modify the code so that IvorySQL can load uuid-ossp extension automatically.

1..11. Empty String to NULL

Purpose

In Oracle databases, empty strings ('') are treated as NULL values, which is an important Oracle feature. To maintain compatibility with this Oracle behavior, IvorySQL provides the empty string to NULL conversion functionality. When this feature is enabled, empty strings in SQL statements are automatically converted to NULL values, ensuring behavioral consistency for Oracle applications running on IvorySQL.

Implementation

The parameter `ivorysql.enable_emptystring_to_NULL` corresponds to the GUC variable `enable_emptystring_to_NULL`.

In the file `ora_scan.l`, you can see how this variable is used:

```
case xe:
    yylval->str = litbufdup(yyscanner);
    if (strcmp(yylval->str, "") == 0 &&
        ORA_PARSER == compatible_db &&
        enable_emptystring_to_NULL)
    {
        return NULL_P;
    }
    return SCONST;
```

Where `xe` represents strings enclosed in quotes:

```
<xe> extended quoted strings (support backslash escape sequences)
```

The logic of the above code is that during lexical analysis, if an empty string is encountered and the empty-to-NULL conversion feature is enabled, it returns `NULL_P`; otherwise, it returns `SCONST`.

In the grammar analysis file `ora_gram.y`, the statement `insert into abc values('');` is parsed as follows:

```
values_clause:
    VALUES '(' expr_list ')'
    {
        SelectStmt *n = makeNode(SelectStmt);
```

```
        n->valuesLists = list_make1($3);
        $$ = (Node *) n;
    }

expr_list:      a_expr
        {
            $$ = list_make1($1);
        }

a_expr:       c_expr                    { $$ = $1; }

c_expr:       AexprConst                { $$ = $1; }

AexprConst: Iconst
       | Sconst
           {
               $$ = makeStringConst($1, @1);
           }
       | NULL_P
           {
               $$ = makeNullAConst(@1);
           }
```

The above code constructs corresponding nodes to handle `NULL_P` or `SCONST` returned from the lexical analysis.

1..12. CALL INTO

Purpose

Currently, PostgreSQL’s CALL statement has the following limitations:

- Does not support the INTO clause;
- Cannot call functions that return values;
- Cannot assign results to client-side variables (i.e., Oracle’s binding variables / host variables).

To enhance compatibility with Oracle, IvorySQL has implemented support for the CALL func(…) INTO :var; syntax, allowing users to receive function return values through binding variables (e.g., :x). This behavior (including precision checks and error handling) aligns with Oracle’s standards.

Overall Design Approach

Since PostgreSQL/IvorySQL inherently does not support direct assignment to client-side variables at the SQL level, this solution adopts a "client-side rewriting + server-side collaboration" approach:

- When the CALL statement includes binding variables (e.g., :x):

```
The client rewrites it into a special anonymous PL block (DO $$ ... $$);
Sends it using the extended query protocol to transmit parameter type and precision
information;
The server executes this anonymous block and returns the result to the client;
The client then writes the result to the corresponding binding variable.
```

- When the CALL statement does not contain binding variables:

```
The behavior remains fully consistent with native PostgreSQL, using the simple
query protocol without any rewriting.
```

Implementation Details

psql

To ensure compatibility with the CALL [INTO] statement in the interface, it must be converted into an anonymous PL/iSQL block, leveraging the support for OUT parameters in anonymous blocks to achieve functional equivalence.

This requires the get_parameter_description function to correctly identify CALL statements and, when encountering CALL INTO, return the rewritten PL statement.

Correspondingly, the get_hostvariables routine needs to store this information (such as whether it is a CALL statement, whether it includes INTO, the rewritten statement, etc.) in the HostVariable structure. The definition of HostVariable is as follows:

```
typedef struct HostVariable
{
    HostVariableEntry *hostvars;
    int     length;
    bool    isdostmt;
    bool    iscallstmt; // Whether it is from a CALL statement
    char    *convertcall;   // Rewritten statement
} HostVariable;
```

Server-side

On the server side, modifications are required in the syntax parser section. Add CALL INTO grammar rules in ora_gram.y, and generate rewritten PL statements in the action section, such as "x := add(1,2);".

```
CallStmt:   CALL func_application
                {
                    CallStmt *n = makeNode(CallStmt);
                    n->funccall = castNode(FuncCall, $2);
                    $$ = (Node *)n;
                }
            | CALL func_application INTO ORAPARAM
```

```
                {
                    CallStmt *n = makeNode(CallStmt);
                    OraParamRef *hostvar = makeNode(OraParamRef);
                    char    *callstr = NULL;
                    n->funccall = castNode(FuncCall, $2);
                    hostvar->number = 0;
                    hostvar->location = @4;
                    hostvar->name = $4;
                    n->hostvariable = hostvar;
                    callstr = pnstrdup(pg_yyget_extra(yyscanner)-
>core_yy_extra.scanbuf + @2, @3 - @2);
                    n->callinto = psprintf("%s := %s;", $4, callstr);
                    pfree(callstr);
                    $$ = (Node *)n;
                }
        ;
```

The CallStmt structure needs to store the INTO clause and the converted PL statement.

```
typedef struct CallStmt
{
  NodeTag   type;
  FuncCall  *funccall;    /* from the parser */
  FuncExpr  *funcexpr;    /* transformed call, with only input args */
  List    *outargs;    /* transformed output-argument expressions */
  OraParamRef *hostvariable; /* only used for get_parameter_description() */
  char    *callinto;    /* rewrite CALL INTO to a PL assign stmt */
} CallStmt;
```

To distinguish between regular DO statements and anonymous blocks converted from CALL statements, a new keyword GENERATED FROM CALL is added to the syntax.

```
opt_do_from_where:
            GENERATED FROM CALL         { $$ = true; }
            | /*EMPTY*/                 { $$ = false; }
        ;
```

The generated DoStmt node will set do_from_call = true for executor identification.

```
typedef struct DoStmt
{
  NodeTag   type;
  List    *args;      /* List of DefElem nodes */
```

```
  List    *paramsmode;  /* List of parameters mode */
  List    *paramslen;   /* List of length for parameter datatypes */
  bool    do_from_call;  /* True if DoStmt is come from CallStmt */
} DoStmt;
```

In the IVY interface, placeholder information is obtained through a Set-Returning Function (SRF) called get_parameter_description. This function needs to identify the type of input statement and return the rewritten PL/iSQL assignment statement when encountering CALL INTO statements.

To achieve this, IvorySQL has extended the return structure (TupleDesc) of this function: a new hint field has been added specifically to return the rewritten PL code for CALL INTO statements; for other types of statements, this field remains NULL.

Additionally, the first field of the first tuple in the original function result set previously used only true/false to distinguish whether the statement was an anonymous block. To more accurately identify statement types (especially CALL statements), it has now been modified to return the corresponding parse tree’s CommandTag.

All these metadata details are ultimately encapsulated into a user context structure to enable efficient passing and reuse across multiple invocations of the SRF function.

```
{
    OraParamExtralData *extral;
    const char     *cmdtag;
    char        *callintoexpr;
} outparam_fctx;
```

Interface layer

The IVY-prefixed interfaces involved in CALL include:

IvyStmtExecute

IvyStmtExecute2

IvyexecPreparedStatement

IvyexecPreparedStatement2

In the aforementioned interfaces, user-provided CALL [INTO] statements are rewritten into a "special" anonymous block statement. To clearly identify such anonymous blocks converted from CALL statements, a dedicated type has been added to the statement type definitions of the interface.

The purpose of this type is to correctly recognize such statements in IvyHandleDostmt and generate execution statements in the form of:DO ... USING ⋯ — GENERATED FROM CALL

```
typedef enum IvyStmtType
{
    IVY_STMT_UNKNOW,
    IVY_STMT_DO,
    IVY_STMT_DOFROMCALL, /* new statement type */
    IVY_STMT_DOHANDLED,
```

```
    IVY_STMT_OTHERS
} IvyStmtType;
```

When rewriting CALL statements, if encountering a CALL INTO statement that invokes a function, the interface needs to internally adjust the order of bound variables. This adjustment is completely transparent to users: when binding parameters, users only need to follow the order in which the variables appear in the CALL statement—that is, the variable in the INTO clause is positioned last in the original statement.

However, in the rewritten special anonymous block, this INTO variable will appear as the left-hand side (i.e., the first parameter) of an assignment expression. Therefore, the interface must internally adjust the binding order correctly to ensure the execution logic matches the user's expectations.

All interface routines involved in this logic must implement this handling. The relevant routines are as follows:

Ivyreplacenamebindtoposition

Ivyreplacenamebindtoposition2

Ivyreplacenamebindtoposition3

In interfaces such as IvyexecPreparedStatement and IvyexecPreparedStatement2, users must explicitly provide paramvalues, paramlengths, paramformats, and parammode for each parameter. For CALL statements, the order of elements in these parameter arrays must be adjusted according to the rewritten anonymous block structure to ensure binding consistency with the execution logic.

Among them, IvyexecPreparedStatement2 is more specialized: it requires users to additionally provide an output binding list of type IvyBindOutInfo*. This list is not only used to bind OUT parameters but is also utilized by IvyAssignPLISQLOutParameter to identify the data type of each OUT parameter when retrieving PL/iSQL procedure results. Therefore, when processing CALL statements, the interface first reorders the user-provided IvyBindOutInfo* list (moving the INTO-bound output variable to the first position) and then writes it into the IvyPreparedStatement statement handle for subsequent assignment.

Regarding precision handling for output parameters: When there is a mismatch between the precision of an output binding variable in a CALL statement and the actual returned value, the system may either raise an error or automatically truncate—the specific behavior depends on whether the data type of the binding variable exactly matches the parameter type declared in the procedure/function.

In the PL/iSQL inline handler, the precise data type of each OUT parameter can be obtained through ParamListInfo during the binding phase. If the currently executed anonymous block is a special DoStmt converted from a CALL statement, the system performs the following checks during assignment:

If the type recorded in ParamListInfo exactly matches the formal parameter type of the function/stored procedure, a forced type conversion is applied for assignment. Otherwise, an implicit type conversion is used for assignment. This mechanism is designed to be compatible with Oracle's behavior, ensuring safe and reasonable assignment even when types do not fully match.

```
-- Original CALL statement:
CALL my_func(:in1, :in2) INTO :out;
-- Rewritten as:
do $$BEGIN
  :out := my_func(:in1, :in2);
END$$ using
  out INOUT, in1 INOUT, in2 INOUT
  paramslength -1,-1,-1
GENERATED FROM CALL;
```

Built-in Functions

1..1. sys_context

IvorySQL provides compatibility with the Oracle built-in function `SYS_CONTEXT('namespace', 'parameter' [, length ])`, which returns the value of the parameter associated with the given context at the current moment. It can be used in both SQL and PL/SQL languages.

IvorySQL provides the following built-in namespaces:

- USERENV - Describes the current session.
- SYS_SESSION_ROLES - Indicates whether a specified role is currently enabled for the session.

Implementation Principle

The implementation principle of `SYS_CONTEXT` involves dynamically querying system tables and PostgreSQL's built-in functions to ensure real-time results. It uses `SECURITY INVOKER` to ensure the function executes with the caller's permissions, avoiding privilege escalation issues. The implementation logic is as follows:

```
CREATE OR REPLACE FUNCTION sys.sys_context(a varchar2, b varchar2)
RETURNS varchar2 AS $$
DECLARE
    res varchar2;
BEGIN
    IF upper(a) = 'USERENV' THEN
      CASE upper(b)
          WHEN 'CURRENT_SCHEMA' THEN
            SELECT current_schema() INTO res;
          WHEN 'LANG' THEN
            SELECT sys.get_lang() INTO res;
          ...
          ELSE
            RAISE EXCEPTION 'invalid USERENV parameter: %', b;
      END CASE;
    ELSIF upper(a) = 'SYS_SESSION_ROLES' THEN
        CASE upper(b)
            WHEN 'LOGIN' THEN
                SELECT CASE WHEN rolcanlogin = 't' THEN 'TRUE' ELSE 'FALSE' END INTO
res FROM pg_roles WHERE oid = current_user::regrole::oid;
...
            ELSE
                RAISE EXCEPTION 'invalid SYS_SESSION_ROLES parameter: %', b;
        END CASE;
    ELSE
      SELECT current_setting(a||'.'||b, true) INTO res;
    END IF;
```

```
    RETURN res;
END;
$$ LANGUAGE plisql SECURITY INVOKER;
```

Parameters supported by namespace **USERENV**

Parameter Name	Return Value
CURRENT_SCHEMA	The name of the currently active default schema. This value may change during the duration of a session through use of an ALTER SESSION SET CURRENT_SCHEMA statement. This may also change during the duration of a session to reflect the owner of any active definer’ s rights object. When used directly in the body of a view definition, this returns the default schema used when executing the cursor that is using the view; it does not respect views used in the cursor as being definer’ s rights.
CURRENT_SCHEMAID	Identifier of the currently active default schema.
SESSION_USER	The name of the session user (the user who logged on). This may change during the duration of a database session as Real Application Security sessions are attached or detached. For enterprise users, returns the schema. For other users, returns the database user name. If a Real Application Security session is currently attached to the database session, returns user XS$NULL.
SESSION_USERID	The identifier of the session user (the user who logged on).
PROXY_USER	Name of the database user who opened the current session on behalf of SESSION_USER.
PROXY_USERID	Identifier of the database user who opened the current session on behalf of SESSION_USER.
CURRENT_USER	The name of the database user whose privileges are currently active. This may change during the duration of a database session as Real Application Security sessions are attached or detached, or to reflect the owner of any active definer’ s rights object. When no definer’ s rights object is active, CURRENT_USER returns the same value as SESSION_USER. When used directly in the body of a view definition, this returns the user that is executing the cursor that is using the view; it does not respect views used in the cursor as being definer’ s rights. For enterprise users, returns schema. If a Real Application Security user is currently active, returns user XS$NULL.
CURRENT_USERID	The identifier of the database user whose privileges are currently active.
CURRENT_EDITION_NAME	The name of the current edition.
CLIENT_PROGRAM_NAME	The name of the program used for the database session.

Parameter Name	Return Value
IP_ADDRESS	The IP address of the client.
HOST	Name of the host machine from which the client has connected.
ISDBA	Returns TRUE if the user has been authenticated as having DBA privileges either through the operating system or through a password file.
LANG	The abbreviated name for the language, a shorter form than the existing 'LANGUAGE' parameter.
LANGUAGE	The language and territory currently used by your session, along with the database character set, in this form:language_territory.characterset
NLS_DATE_FORMAT	The date format for the session.
PLATFORM_SLASH	The slash character that is used as the file path delimiter for your platform.

DB_NAME	Name of the database as specified in the DB_NAME initialization parameter.
SID	The session ID.
SESSIONID	The auditing session identifier. You cannot use this attribute in distributed SQL statements.
CLIENT_INFO	Returns up to 64 bytes of user session information that can be stored by an application using the DBMS_APPLICATION_INFO package.
ENTRYID	The current audit entry number. The audit entryid sequence is shared between fine-grained audit records and regular audit records. You cannot use this attribute in distributed SQL statements. The correct auditing entry identifier can be seen only through an audit handler for standard or fine-grained audit.
TERMINAL	The operating system identifier for the client of the current session. In distributed SQL statements, this attribute returns the identifier for your local session. In a distributed environment, this is supported only for remote SELECT statements, not for remote INSERT, UPDATE, or DELETE operations. (The return length of this parameter may vary by operating system.)

Parameters supported by namespace `SYS_SESSION_ROLES`

Parameter Name	Return Value
DBA	Returns TRUE if the current user is a database administrator.
LOGIN	Returns TRUE if the current user is a login role.
CREATEROLE	Returns TRUE if the current session’ s user has the privilege to create roles.
CREATEDB	Returns TRUE if the current session’ s user has the privilege to create databases.

1..2. userenv

IvorySQL provides compatibility with Oracle’ s built-in function `USERENV('parameter')`, which is used to return information about the current session. This is a legacy function, and IvorySQL recommends taht you can use the `SYS_CONTEXT` function with its built-in `USERENV` namespace for current functionality.

Implementation Principle

`USERENV` parses function calls through Bison rules, checks parameter validity, and maps them to corresponding SQL functions. The parsing rules are implemented in `ora_gram.y`, with the following logic:

```
USERENV '(' Sconst ')'
    {
        char *normalized_param = downcase_identifier($3, strlen($3), true, true);

        #define CHECK_AND_CALL(param, func_name) \
            if (strcmp(normalized_param, param) == 0) \
                $$ = (Node *) makeFuncCall(OracleSystemFuncName(func_name), NIL,
COERCE_EXPLICIT_CALL, @1);

        CHECK_AND_CALL("client_info", "get_client_info")
        else CHECK_AND_CALL("entryid", "get_entryid")
```

```
        else CHECK_AND_CALL("terminal", "get_terminal")
        else CHECK_AND_CALL("isdba", "get_isdba")
        else CHECK_AND_CALL("lang", "get_lang")
        else CHECK_AND_CALL("language", "get_language")
        else CHECK_AND_CALL("sessionid", "get_sessionid")
        else CHECK_AND_CALL("sid", "get_sid")
        else
            ereport(ERROR,
                    (errcode(ERRCODE_FEATURE_NOT_SUPPORTED),
                     errmsg("invalid USERENV parameter: \"%s\".", $3)));
        #undef CHECK_AND_CALL
    }
```

The specific functionality is implemented in `builtin_functions—1.0.sql`. For example:

```
CREATE OR REPLACE FUNCTION sys.get_language()
RETURNS varchar2
AS $$
    SELECT (regexp_split_to_array(current_setting('lc_monetary'), '\\.'))[1]||'.'
||pg_client_encoding();
$$ LANGUAGE sql STRICT;
```

Supported parameters

Parameter Name	Return Value
sid	The user ID of the current session.
sessionid	Returns the audit session identifier.
language	Returns the language, territory, and character set of the current database session.
lang	Returns the ISO abbreviation of the language name.
isdba	Returns "TRUE" if the user has been authenticated as an administrator or has administrator privileges through the operating system or password file; otherwise, returns "FALSE".

1.4. GB18030 Character Set

Objective

PostgreSQL provides comprehensive server-side support for the GB18030 character set. GB18030 is a Chinese national standard designed to include all Chinese characters and various minority scripts, aiming for alignment with Unicode. Proper configuration and use of the GB18030 character set within PostgreSQL are essential for processing and storing Chinese data that must comply with this standard.

Server-side GB18030 support should provide the following features:

1. Support for GB18030 as a server encoding: initdb -E GB18030 is available, and SHOW server_encoding displays GB18030.

2. Provide bidirectional conversion between GB18030 and UTF8.
3. Support for multibyte character boundary determination.

Implementation Details

Specifying GB18030 or GB18030_2022 with the -E Option during initdb

PostgreSQL has historically supported the GB18030-2000 standard as a client-side encoding. Support for conversion between the GB18030_2022 character set and UTF-8 is provided via an extension.

To enable GB18030 as a server encoding, modifications are made to pg_enc, and new low-level functions are added to the PostgreSQL encoding framework for invocation by the core system.

A global variable, is_load_gb18030_2022, is introduced with a default value of true. When the -E option is used during initdb, the get_encoding_id function checks the specified encoding name. If the name is gb18030_2022, it is internally mapped to the gb18030 encoding ID, and the is_load_gb18030_2022 flag is set to true. If the -E option is GB18030, the flag is set to false.

At the appropriate stage in the startup process, the system checks this flag to determine if the extension should be loaded. If required, the load_gb18030_2022 function is executed, and the gb18030_2022 extension is added to the shared_preload_libraries parameter in ivorysql.conf.

```
if (encoding_name && *encoding_name)
{
    encoding_name_modify = pg_strdup(encoding_name);
    if(pg_strcasecmp(encoding_name,"gb18030_2022") == 0)
    {
        encoding_name_modify = pg_strdup("gb18030");
        is_load_gb18030_2022 =  true;
    }
    else if(pg_strcasecmp(encoding_name,"gb18030") == 0)
    is_load_gb18030_2022 = false;

    if ((enc = pg_valid_server_encoding((const char *)encoding_name_modify)) >= 0)
        return enc;
}
```

Multibyte Character Handling

Function pointers for GB18030 are added in wchar.c:

pg_gb180302wchar_with_len(const unsigned char *from, pg_wchar *to, int len) gb18030 → wchar

pg_wchar2gb18030_with_len(const pg_wchar *from, unsigned char *to, int len) wchar → gb18030

pg_gb18030_mblen(const unsigned char *s)： Returns 1/2/4.

pg_gb18030_dsplen(const unsigned char *s)： Calculates the display width of a character. ASCII characters have a width of 1, while others are also treated as having a width of 1 .

pg_gb18030_verifier(const unsigned char *s, int len)： Verifies that a byte sequence is a valid GB18030 character, rejecting illegal sequences.

Client-Server Interaction

Receiving Data

When a client using UTF-8 encoding connects to the server, the server, upon receiving data, invokes its internal utf8_to_gb18030 function. This converts the data to the GB18030 format, which is then validated and stored.

Sending Data

When the same client executes a SELECT query, the server reads the native GB18030 data from disk or memory. It then calls the gb18030_to_utf8 function to convert the data to UTF-8 format before sending it to the client via the network protocol.

A new data file, GB18030-2022.xml, is introduced. This file is parsed by a Perl script to generate mapping files that provide the logic for the gb18030_to_utf8() and utf8_to_gb18030() conversion functions. The implementation prioritizes a table-driven approach, falling back to algorithmic mapping for ranges not covered by the tables.

```
static inline uint32
unicode_to_utf8word(uint32 c)
{
    uint32      word;

    if (c <= 0x7F)
    {
        word = c;
    }
    else if (c <= 0x7FF)
    {
        word = (0xC0 | ((c >> 6) & 0x1F)) << 8;
        word |= 0x80 | (c & 0x3F);
    }
    else if (c <= 0xFFFF)
    {
        word = (0xE0 | ((c >> 12) & 0x0F)) << 16;
        word |= (0x80 | ((c >> 6) & 0x3F)) << 8;
        word |= 0x80 | (c & 0x3F);
    }
    else
    {
        word = (0xF0 | ((c >> 18) & 0x07)) << 24;
        word |= (0x80 | ((c >> 12) & 0x3F)) << 16;
        word |= (0x80 | ((c >> 6) & 0x3F)) << 8;
        word |= 0x80 | (c & 0x3F);
    }

    return word;
```

```
}

static uint32
conv_18030_2022_to_utf8(uint32 code)
{
#define conv18030(minunicode, mincode, maxcode) \
    if (code >= mincode && code <= maxcode) \
        return unicode_to_utf8word(gb_linear(code) - gb_linear(mincode) + minunicode)

    conv18030(0x0452, 0x8130D330, 0x8136A531);
    conv18030(0x2643, 0x8137A839, 0x8138FD38);
    conv18030(0x361B, 0x8230A633, 0x8230F237);
    conv18030(0x3CE1, 0x8231D438, 0x8232AF32);
    conv18030(0x4160, 0x8232C937, 0x8232F837);
    conv18030(0x44D7, 0x8233A339, 0x8233C931);
    conv18030(0x478E, 0x8233E838, 0x82349638);
    conv18030(0x49B8, 0x8234A131, 0x8234E733);
    conv18030(0x9FA6, 0x82358F33, 0x8336C738);
    conv18030(0xE865, 0x8336D030, 0x84308534);
    conv18030(0xFA2A, 0x84309C38, 0x84318537);
    conv18030(0xFFE6, 0x8431A234, 0x8431A439);
    conv18030(0x10000, 0x90308130, 0xE3329A35);
    /* No mapping exists */
    return 0;
}
```

Reference

.1. Tool Reference

List of tools

This part contains reference information for IvorySQL client applications and utilities. Not all of these commands are of general utility; some might require special privileges. The common feature of Client Applications is that they can be run on any host, independent of where the database server resides.

When specified on the command line, user and database names have their case preserved — the presence of spaces or special characters might require quoting. Table names and other identifiers do not have their case preserved, except where documented, and might require quoting.

category	Tool name	Description
Client Applications	clusterdb	Reclusters the tables that were previously marked as clustered, acting as a wrapper around the SQL CLUSTER command.
	createdb	Creates a new IvorySQL database, optionally specifying owner, encoding, locale or tablespace, just like running `CREATE DATABASE`.
	createuser	Creates a new IvorySQL role when run by a superuser or someone with `CREATEROLE`, mirroring `CREATE ROLE`.
	dropdb	Removes an existing database when run by its owner or a superuser, wrapping `DROP DATABASE`.
	dropuser	Deletes an existing role and corresponds to `DROP ROLE`; superuser or `CREATEROLE` privileges are required.
	ecpg	Preprocesses C source files that embed SQL, turning statements into libpq calls so the code can then be compiled with a normal toolchain.
	pg_amcheck	Runs amcheck routines to validate indexes and heap data, allowing filtered and parallel corruption checks.
	pg_basebackup	Takes an online base backup over the replication protocol for PITR or standby seeding, copying the entire cluster while managing backup mode automatically.
	pgbench	Executes customizable benchmark scripts (TPC-B style by default) to measure throughput, concurrency and latency of an IvorySQL instance.

	pg_config	Prints build-time configuration details (include paths, library directories, version, etc.) for the currently installed IvorySQL binaries.
	pg_dump	Creates a consistent backup of a single database in plain SQL or archive formats that can later be restored with pg_restore.
	pg_dumpall	Sequentially dumps every database plus global objects into one SQL script by invoking pg_dump per database.
	pg_isready	Returns the connection status of a target server so scripts can detect whether IvorySQL is accepting connections.
	pg_receivewal	Streams WAL segments from a running cluster via the replication protocol and writes them to disk for archiving or downstream recovery.
	pg_recvlogical	Manages logical-replication slots and streams decoded changes over a replication connection for custom consumers.
	pg_restore	Replays archives produced by pg_dump (non-plain formats) directly into a database or onto stdout, supporting selective and reordered restores.
	pg_verifybackup	Validates a plain-format backup created by `pg_basebackup` against its `backup_manifest`, detecting missing or corrupted files before restore.
	psql	Terminal client that runs SQL interactively or from scripts, providing meta-commands, variables and scripting conveniences.
	reindexdb	Rebuilds indexes for a database, schema or table set, invoking `REINDEX` under the hood.
	vacuumdb	Automates `VACUUM` (optionally `ANALYZE`) for selected databases or tables to reclaim space and refresh optimizer statistics.
Server Applications	initdb(1)	Initializes a new database cluster by creating the data directory, catalog tables and default databases (`postgres`, `template1`, `template0`).

Server Applications	initdb(2)	Must be run as the future cluster owner; controls file permissions plus the default locale, encoding and ICU/libc settings for the cluster.
	pg_archivecleanup	Cleans archived WAL files, typically via `archive_cleanup_command`, so old segments are removed once they are no longer needed.
	pg_checksums	Enables, disables or verifies cluster-wide data checksums while the server is stopped, rewriting blocks as needed.
	pg_controldata	Displays cluster control information (catalog version, checkpoint and WAL status) by reading `pg_control`.
	pg_ctl	Initializes, starts, stops, restarts or queries the status of the IvorySQL server, handling log redirection and daemonization.
	pg_resetwal	Resets the WAL and selected control fields after corruption so the server can start, to be followed by a dump and rebuild.
	pg_rewind(1)	Syncs a diverged cluster with another timeline by copying only the changed blocks since the split, enabling quick re-attachment as a standby.
	pg_rewind(2)	Requires retained WAL or archive access plus `wal_log_hints` or checksums; after rewinding, recovery replays new WAL before the node can serve.
	pg_test_fsync	Benchmarks different `wal_sync_method` options to identify the fastest durable fsync strategy on the host OS and storage.
	pg_test_timing	Measures timer resolution and detects backward time jumps, helping interpret `EXPLAIN ANALYZE` accuracy.
	pg_upgrade	Reuses existing data files to move between major releases quickly, checking binary compatibility and external module support.
	pg_waldump	Reads and prints WAL records for debugging or education; requires direct read access to the data directory.
	postgres	The actual IvorySQL server executable that accepts client connections and forks per-session backend processes.

Client Applications

clusterdb

Synopsis

clusterdb [**connection-option**⋯] [**--verbose** | **-v**] [**--table** | **-t table**] ⋯ [**dbname**]

```
clusterdb` [*`connection-option`*...] [ `--verbose` | `-v` ] `--all` | `-a
```

Options

clusterdb accepts the following command-line arguments:

- **-a --all**

Cluster all databases.

- **[-d] dbname dbname**

Specifies the name of the database to be clustered, when **-a** / **--all** is not used. If this is not specified, the database name is read from the environment variable **PGDATABASE**. If that is not set, the user name specified for the connection is used. The **dbname** can be a connection string. If so, connection string parameters will override any conflicting command line options.

- **-e --echo**

Echo the commands that clusterdb generates and sends to the server.

- **-q --quiet**

Do not display progress messages.

- **-t table --table=table**

Cluster **table** only. Multiple tables can be clustered by writing multiple **-t** switches.

- **-v --verbose**

Print detailed information during processing.

- **-V --version**

Print the clusterdb version and exit.

- **-? --help**

Show help about clusterdb command line arguments, and exit.clusterdb also accepts the following command-line arguments for connection parameters:

- **-h host --host=host**

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket.

- **-p port --port=port**

Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections.

- `-U username --username=username`

User name to connect as.

- `-w --no-password`

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

- `-W --password`

Force clusterdb to prompt for a password before connecting to a database.This option is never essential, since clusterdb will automatically prompt for a password if the server demands password authentication. However, clusterdb will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing `-W` to avoid the extra connection attempt.

- `--maintenance-db=dbname`

Specifies the name of the database to connect to to discover which databases should be clustered, when `-a` / `--all` is used. If not specified, the `postgres` database will be used, or if that does not exist, `template1` will be used. This can be a connection string. If so, connection string parameters will override any conflicting command line options. Also, connection string parameters other than the database name itself will be re-used when connecting to other databases.

Environment

- `PGDATABASE PGHOST PGPORT PGUSER`

Default connection parameters

- `PG_COLOR`

Specifies whether to use color in diagnostic messages. Possible values are `always`, `auto` and `never`.

This utility, like most other IvorySQL utilities, also uses the environment variables supported by libpq

Diagnostics

In case of difficulty, see CLUSTER and psql for discussions of potential problems and error messages. The database server must be running at the targeted host. Also, any default connection settings and environment variables used by the libpq front-end library will apply.

Examples

To cluster the database `test`:

```
$ clusterdb test
```

To cluster a single table `foo` in a database named `xyzzy`:

```
$ clusterdb --table=foo xyzzy
```

createdb

createdb — create a new IvorySQL database

Synopsis

`createdb` [`connection-option`⋯] [`option`⋯] [`dbname` [`description`]]

Options

createdb accepts the following command-line arguments:

- `dbname`

Specifies the name of the database to be created. The name must be unique among all IvorySQL databases in this cluster. The default is to create a database with the same name as the current system user.

- `description`

Specifies a comment to be associated with the newly created database.

- `-D tablespace --tablespace=tablespace`

Specifies the default tablespace for the database. (This name is processed as a double-quoted identifier.)

- `-e --echo`

Echo the commands that createdb generates and sends to the server.

- `-E encoding --encoding=encoding`

Specifies the character encoding scheme to be used in this database.

- `-l locale --locale=locale`

Specifies the locale to be used in this database. This is equivalent to specifying both `--lc-collate` and `--lc-ctype`.

- `--lc-collate=`locale``

Specifies the LC_COLLATE setting to be used in this database.

- `--lc-ctype=locale`

Specifies the LC_CTYPE setting to be used in this database.

- `--icu-locale=`locale``

Specifies the ICU locale ID to be used in this database, if the ICU locale provider is selected.

- `--locale-provider={libc|icu}`

Specifies the locale provider for the database’ s default collation.

- `-O `owner` --owner=`owner``

Specifies the database user who will own the new database. (This name is processed as a double-quoted identifier.)

- `-T template --template=template`

Specifies the template database from which to build this database. (This name is processed as a double-quoted identifier.)

- `-V --version`

Print the createdb version and exit.

- `-? --help`

Show help about createdb command line arguments, and exit.

The options `-D`, `-l`, `-E`, `-O`, and `-T` correspond to options of the underlying SQL command `CREATE DATABASE`; see there for more information about them.

createdb also accepts the following command-line arguments for connection parameters:

- `-h host --host=host`

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket.

- `-p port --port=port`

Specifies the TCP port or the local Unix domain socket file extension on which the server is listening for connections.

- `-U username --username=username`

User name to connect as.

- `-w --no-password`

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

- `-W --password`

Force createdb to prompt for a password before connecting to a database.This option is never essential, since createdb will automatically prompt for a password if the server demands password authentication. However, createdb will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing `-W` to avoid the extra connection attempt.

- `--maintenance-db=`dbname`

Specifies the name of the database to connect to when creating the new database. If not specified, the `postgres` database will be used; if that does not exist (or if it is the name of the new database being created), `template1` will be used. This can be a connection string. If so, connection string parameters will override any conflicting command line options.

Environment

- `PGDATABASE`

If set, the name of the database to create, unless overridden on the command line.

- `PGHOST PGPORT PGUSER`

Default connection parameters. `PGUSER` also determines the name of the database to create, if it is not specified on the command line or by `PGDATABASE`.

- `PG_COLOR`

Specifies whether to use color in diagnostic messages. Possible values are `always`, `auto` and `never`.

This utility, like most other IvorySQL utilities, also uses the environment variables supported by libpq

Diagnostics

In case of difficulty, see CREATE DATABASE and psql for discussions of potential problems and error messages. The database server must be running at the targeted host. Also, any default connection settings and environment variables used by the libpq front-end library will apply.

Examples

To create the database **demo** using the default database server:

```
$ createdb demo
```

To create the database **demo** using the server on host **eden**, port 5000, using the **template0** template database, here is the command-line command and the underlying SQL command:

```
$ createdb -p 5000 -h eden -T template0 -e demo
CREATE DATABASE demo TEMPLATE template0;
```

createuser

createuser — define a new IvorySQL user account

Synopsis

createuser [**connection-option**⋯] [**option**⋯] [**username**]

Options

createuser accepts the following command-line arguments:

- **username**

Specifies the name of the IvorySQL user to be created.

- **-c number --connection-limit=number**

Set a maximum number of connections for the new user. The default is to set no limit.

- **-d --createdb**

The new user will be allowed to create databases.

- **-D --no-createdb**

The new user will not be allowed to create databases. This is the default.

- **-e --echo**

Echo the commands that createuser generates and sends to the server.

- **-E --encrypted**

This option is obsolete but still accepted for backward compatibility.

- **-g role --role=role**

Indicates role to which this role will be added immediately as a new member. Multiple roles to which this

role will be added as a member can be specified by writing multiple `-g` switches.

- `-i --inherit`

The new role will automatically inherit privileges of roles it is a member of. This is the default.

- `-I --no-inherit`

The new role will not automatically inherit privileges of roles it is a member of.

- `--interactive`

Prompt for the user name if none is specified on the command line, and also prompt for whichever of the options `-d` / `-D`, `-r` / `-R`, `-s` / `-S` is not specified on the command line.

- `-l --login`

The new user will be allowed to log in (that is, the user name can be used as the initial session user identifier). This is the default.

- `-L --no-login`

The new user will not be allowed to log in. (A role without login privilege is still useful as a means of managing database permissions.)

- `-P --pwprompt`

If given, createuser will issue a prompt for the password of the new user. This is not necessary if you do not plan on using password authentication.

- `-r --createrole`

The new user will be allowed to create new roles. That is, this user will have `CREATEROLE` privilege.

- `-R --no-createrole`

The new user will not be allowed to create new roles. This is the default.

- `-s --superuser`

The new user will be a superuser.

- `-S --no-superuser`

The new user will not be a superuser. This is the default.

- `-V --version`

Print the createuser version and exit.

- `--replication`

The new user will have the `REPLICATION` privilege, which is described more fully in the documentation for CREATE ROLE.

- `--no-replication`

The new user will not have the `REPLICATION` privilege, which is described more fully in the documentation for CREATE ROLE.

- `-? --help`

Show help about createuser command line arguments, and exit.

createuser also accepts the following command-line arguments for connection parameters:

- `-h host --host=host`

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket.

- `-p port --port=port`

Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections.

- `-U username --username=username`

User name to connect as (not the user name to create).

- `-w --no-password`

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

- `-W --password`

Force createuser to prompt for a password (for connecting to the server, not for the password of the new user).This option is never essential, since createuser will automatically prompt for a password if the server demands password authentication. However, createuser will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing `-W` to avoid the extra connection attempt.

Environment

- `PGHOST PGPORT PGUSER`

Default connection parameters

- `PG_COLOR`

Specifies whether to use color in diagnostic messages. Possible values are `always`, `auto` and `never`.

This utility, like most other IvorySQL utilities, also uses the environment variables supported by libpq

Diagnostics

In case of difficulty, see CREATE ROLE and psql for discussions of potential problems and error messages. The database server must be running at the targeted host. Also, any default connection settings and environment variables used by the libpq front-end library will apply.

Examples

To create a user `joe` on the default database server:

```
$ createuser joe
```

To create a user `joe` on the default database server with prompting for some additional attributes:

```
$ createuser --interactive joe
Shall the new role be a superuser? (y/n) n
```

```
Shall the new role be allowed to create databases? (y/n) n
Shall the new role be allowed to create more new roles? (y/n) n
```

To create the same user **joe** using the server on host **eden**, port 5000, with attributes explicitly specified, taking a look at the underlying command:

```
$ createuser -h eden -p 5000 -S -D -R -e joe
CREATE ROLE joe NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT LOGIN;
```

To create the user **joe** as a superuser, and assign a password immediately:

```
$ createuser -P -s -e joe
Enter password for new role: xyzzy
Enter it again: xyzzy
CREATE ROLE joe PASSWORD 'md5b5f5ba1a423792b526f799ae4eb3d59e' SUPERUSER CREATEDB
CREATEROLE INHERIT LOGIN;
```

In the above example, the new password isn’ t actually echoed when typed, but we show what was typed for clarity. As you see, the password is encrypted before it is sent to the client.

dropdb

dropdb — remove a IvorySQL database

Synopsis

dropdb [**connection-option**⋯] [**option**⋯] **dbname**

Options

dropdb accepts the following command-line arguments:

- **dbname**

Specifies the name of the database to be removed.

- **-e --echo**

Echo the commands that dropdb generates and sends to the server.

- **-f --force**

Attempt to terminate all existing connections to the target database before dropping it. See DROP DATABASE for more information on this option.

- **-i --interactive**

Issues a verification prompt before doing anything destructive.

- **-V --version**

Print the dropdb version and exit.

- **--if-exists**

Do not throw an error if the database does not exist. A notice is issued in this case.

- `-? --help`

Show help about dropdb command line arguments, and exit.

dropdb also accepts the following command-line arguments for connection parameters:

- `-h host --host=host`

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket.

- `-p port --port=port`

Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections.

- `-U username --username=username`

User name to connect as.

- `-w --no-password`

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

- `-W --password`

Force dropdb to prompt for a password before connecting to a database.This option is never essential, since dropdb will automatically prompt for a password if the server demands password authentication. However, dropdb will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing `-W` to avoid the extra connection attempt.

- `--maintenance-db=dbname`

Specifies the name of the database to connect to in order to drop the target database. If not specified, the `postgres` database will be used; if that does not exist (or is the database being dropped), `template1` will be used. This can be a connection string. If so, connection string parameters will override any conflicting command line options.

Environment

- `PGHOST PGPORT PGUSER`

Default connection parameters

- `PG_COLOR`

Specifies whether to use color in diagnostic messages. Possible values are `always`, `auto` and `never`.

This utility, like most other IvorySQL utilities, also uses the environment variables supported by libpq .

Diagnostics

In case of difficulty, see DROP DATABASE and psql for discussions of potential problems and error messages. The database server must be running at the targeted host. Also, any default connection settings and environment variables used by the libpq front-end library will apply.

Examples

To destroy the database **demo** on the default database server:

```
$ dropdb demo
```

To destroy the database **demo** using the server on host **eden**, port 5000, with verification and a peek at the underlying command:

```
$ dropdb -p 5000 -h eden -i -e demo
Database "demo" will be permanently deleted.
Are you sure? (y/n) y
DROP DATABASE demo;
```

dropuser

dropuser — remove a IvorySQL user account

Synopsis

dropuser [**connection-option**⋯] [**option**⋯] [**username**]

Options

dropuser accepts the following command-line arguments:

- **username**

Specifies the name of the IvorySQL user to be removed. You will be prompted for a name if none is specified on the command line and the **-i** / **--interactive** option is used.

- **-e --echo**

Echo the commands that dropuser generates and sends to the server.

- **-i --interactive**

Prompt for confirmation before actually removing the user, and prompt for the user name if none is specified on the command line.

- **-V --version**

Print the dropuser version and exit.

- **--if-exists**

Do not throw an error if the user does not exist. A notice is issued in this case.

- **-? --help**

Show help about dropuser command line arguments, and exit.

dropuser also accepts the following command-line arguments for connection parameters:

- **-h host --host=host**

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket.

- `-p port --port=port`

Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections.

- `-U username --username=username`

User name to connect as (not the user name to drop).

- `-w --no-password`

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

- `-W --password`

Force dropuser to prompt for a password before connecting to a database.This option is never essential, since dropuser will automatically prompt for a password if the server demands password authentication. However, dropuser will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing `-W` to avoid the extra connection attempt.

Environment

- `PGHOST PGPORT PGUSER`

Default connection parameters

- `PG_COLOR`

Specifies whether to use color in diagnostic messages. Possible values are `always`, `auto` and `never`.

This utility, like most other IvorySQL utilities, also uses the environment variables supported by libpq

Diagnostics

In case of difficulty, see DROP ROLE and psql for discussions of potential problems and error messages. The database server must be running at the targeted host. Also, any default connection settings and environment variables used by the libpq front-end library will apply.

Examples

To remove user `joe` from the default database server:

```
$ dropuser joe
```

To remove user `joe` using the server on host `eden`, port 5000, with verification and a peek at the underlying command:

```
$ dropuser -p 5000 -h eden -i -e joe
Role "joe" will be permanently removed.
Are you sure? (y/n) y
DROP ROLE joe;
```

ecpg

ecpg — embedded SQL C preprocessor

Synopsis

ecpg [**option**…] **file**…

Options

ecpg accepts the following command-line arguments:

- **-c**

Automatically generate certain C code from SQL code. Currently, this works for **EXEC SQL TYPE**.

- **-C mode**

Set a compatibility mode. **mode** can be **INFORMIX**, **INFORMIX_SE**, or **ORACLE**.

- **-D `symbol`**

Define a C preprocessor symbol.

- **-h**

Process header files. When this option is specified, the output file extension becomes **.h** not **.c**, and the default input file extension is **.pgh** not **.pgc**. Also, the **-c** option is forced on.

- **-i**

Parse system include files as well.

- **-I `directory`**

Specify an additional include path, used to find files included via **EXEC SQL INCLUDE**. Defaults are **.** (current directory), **/usr/local/include**, the IvorySQL include directory which is defined at compile time (default: **/usr/local/pgsql/include**), and **/usr/include**, in that order.

- **-o `filename`**

Specifies that **ecpg** should write all its output to the given **filename**. Write **-o -** to send all output to standard output.

- **-r `option`**

Selects run-time behavior. **Option** can be one of the following:`no_indicator`Do not use indicators but instead use special values to represent null values. Historically there have been databases using this approach.**prepare`Prepare all statements before using them. Libecpg will keep a cache of prepared statements and reuse a statement if it gets executed again. If the cache runs full, libecpg will free the least used statement.`questionmarks** Allow question mark as placeholder for compatibility reasons. This used to be the default long ago.

- **-t**

Turn on autocommit of transactions. In this mode, each SQL command is automatically committed unless it is inside an explicit transaction block. In the default mode, commands are committed only when **EXEC SQL COMMIT** is issued.

- **-v**

Print additional information including the version and the "include" path.

- `--version`

Print the ecpg version and exit.

- `-? --help`

Show help about ecpg command line arguments, and exit.

Notes

When compiling the preprocessed C code files, the compiler needs to be able to find the ECPG header files in the IvorySQL include directory. Therefore, you might have to use the `-I` option when invoking the compiler (e.g., `-I/usr/local/pgsql/include`).

Programs using C code with embedded SQL have to be linked against the `libecpg` library, for example using the linker options `-L/usr/local/pgsql/lib -lecpg`.

The value of either of these directories that is appropriate for the installation can be found out using pg_config.

Examples

If you have an embedded SQL C source file named `prog1.pgc`, you can create an executable program using the following sequence of commands:

```
ecpg prog1.pgc
cc -I/usr/local/pgsql/include -c prog1.c
cc -o prog1 prog1.o -L/usr/local/pgsql/lib -lecpg
```

pg_amcheck

pg_amcheck — checks for corruption in one or more IvorySQL databases

Synopsis

`pg_amcheck` [`option`⋯] [`dbname`]

Options

The following command-line options control what is checked:

- `-a --all`

Check all databases, except for any excluded via `--exclude-database`.

- `-d pattern --database=pattern`

Check databases matching the specified `pattern`, except for any excluded by `--exclude-database`. This option can be specified more than once.

- `-D pattern --exclude-database=pattern`

Exclude databases matching the given `pattern`. This option can be specified more than once.

- `-i pattern --index=pattern`

Check indexes matching the specified `pattern`, unless they are otherwise excluded. This option can be specified more than once.This is similar to the `--relation` option, except that it applies only to indexes, not

to other relation types.

- `-I pattern --exclude-index=pattern`

Exclude indexes matching the specified `pattern`. This option can be specified more than once.This is similar to the `--exclude-relation` option, except that it applies only to indexes, not other relation types.

- `-r `pattern` --relation=`pattern``

Check relations matching the specified `pattern`, unless they are otherwise excluded. This option can be specified more than once.Patterns may be unqualified, e.g. `myrel*`, or they may be schema-qualified, e.g. `myschema*.myrel*` or database-qualified and schema-qualified, e.g. `mydb*.myscheam*.myrel*`. A database-qualified pattern will add matching databases to the list of databases to be checked.

- `-R pattern --exclude-relation=pattern`

Exclude relations matching the specified `pattern`. This option can be specified more than once.As with `--relation`, the `pattern` may be unqualified, schema-qualified, or database- and schema-qualified.

- `-s pattern --schema=pattern`

Check tables and indexes in schemas matching the specified `pattern`, unless they are otherwise excluded. This option can be specified more than once.To select only tables in schemas matching a particular pattern, consider using something like `--table=SCHEMAPAT.* --no-dependent-indexes`. To select only indexes, consider using something like `--index=SCHEMAPAT..A schema pattern may be database-qualified. For example, you may write --schema=mydb.myschema*` to select schemas matching `myschema*` in databases matching `mydb*`.

- `-S pattern --exclude-schema=pattern`

Exclude tables and indexes in schemas matching the specified `pattern`. This option can be specified more than once.As with `--schema`, the pattern may be database-qualified.

- `-t pattern --table=pattern`

Check tables matching the specified `pattern`, unless they are otherwise excluded. This option can be specified more than once.This is similar to the `--relation` option, except that it applies only to tables, materialized views, and sequences, not to indexes.

- `-T pattern --exclude-table=pattern`

Exclude tables matching the specified `pattern`. This option can be specified more than once.This is similar to the `--exclude-relation` option, except that it applies only to tables, materialized views, and sequences, not to indexes.

- `--no-dependent-indexes`

By default, if a table is checked, any btree indexes of that table will also be checked, even if they are not explicitly selected by an option such as `--index` or `--relation`. This option suppresses that behavior.

- `--no-dependent-toast`

By default, if a table is checked, its toast table, if any, will also be checked, even if it is not explicitly selected by an option such as `--table` or `--relation`. This option suppresses that behavior.

- `--no-strict-names`

By default, if an argument to `--database`, `--table`, `--index`, or `--relation` matches no objects, it is a fatal error. This option downgrades that error to a warning.

The following command-line options control checking of tables:

- `--exclude-toast-pointers`

By default, whenever a toast pointer is encountered in a table, a lookup is performed to ensure that it references apparently-valid entries in the toast table. These checks can be quite slow, and this option can be used to skip them.

- `--on-error-stop`

After reporting all corruptions on the first page of a table where corruption is found, stop processing that table relation and move on to the next table or index.Note that index checking always stops after the first corrupt page. This option only has meaning relative to table relations.

- `--skip=`option``

If `all-frozen` is given, table corruption checks will skip over pages in all tables that are marked as all frozen.If `all-visible` is given, table corruption checks will skip over pages in all tables that are marked as all visible.By default, no pages are skipped. This can be specified as `none`, but since this is the default, it need not be mentioned.

- `--startblock=`block``

Start checking at the specified block number. An error will occur if the table relation being checked has fewer than this number of blocks. This option does not apply to indexes, and is probably only useful when checking a single table relation. See `--endblock` for further caveats.

- `--endblock=`block``

End checking at the specified block number. An error will occur if the table relation being checked has fewer than this number of blocks. This option does not apply to indexes, and is probably only useful when checking a single table relation. If both a regular table and a toast table are checked, this option will apply to both, but higher-numbered toast blocks may still be accessed while validating toast pointers, unless that is suppressed using `--exclude-toast-pointers`.

The following command-line options control checking of B-tree indexes:

- `--heapallindexed`

For each index checked, verify the presence of all heap tuples as index tuples in the index using amcheck's `heapallindexed` option.

- `--parent-check`

For each btree index checked, use amcheck's `bt_index_parent_check` function, which performs additional checks of parent/child relationships during index checking.The default is to use amcheck’ s `bt_index_check` function, but note that use of the `--rootdescend` option implicitly selects `bt_index_parent_check`.

- `--rootdescend`

For each index checked, re-find tuples on the leaf level by performing a new search from the root page for each tuple using amcheck's `rootdescend` option.Use of this option implicitly also selects the `--parent-check` option.This form of verification was originally written to help in the development of btree index features. It may be of limited use or even of no use in helping detect the kinds of corruption that occur in practice. It may also cause corruption checking to take considerably longer and consume considerably more resources on the server.

Warning

The extra checks performed against B-tree indexes when the `--parent-check` option or the `--rootdescend` option is specified require relatively strong relation-level locks. These checks are the only checks that will block concurrent data modification from `INSERT`, `UPDATE`, and `DELETE` commands.

The following command-line options control the connection to the server:

- `-h hostname --host=hostname`

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket.

- `-p port --port=port`

Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections.

- `-U --username=username`

User name to connect as.

- `-w --no-password`

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

- `-W --password`

Force pg_amcheck to prompt for a password before connecting to a database.This option is never essential, since pg_amcheck will automatically prompt for a password if the server demands password authentication. However, pg_amcheck will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing `-W` to avoid the extra connection attempt.

- `--maintenance-db=dbname`

Specifies a database or connection string to be used to discover the list of databases to be checked. If neither `--all` nor any option including a database pattern is used, no such connection is required and this option does nothing. Otherwise, any connection string parameters other than the database name which are included in the value for this option will also be used when connecting to the databases being checked. If this option is omitted, the default is `postgres` or, if that fails, `template1`.

Other options are also available:

- `-e --echo`

Echo to stdout all SQL sent to the server.

- `-j \`num\` --jobs=\`num\``

Use `num` concurrent connections to the server, or one per object to be checked, whichever is less.The default is to use a single connection.

- `-P --progress`

Show progress information. Progress information includes the number of relations for which checking has been completed, and the total size of those relations. It also includes the total number of relations that will eventually be checked, and the estimated size of those relations.

- `-v --verbose`

Print more messages. In particular, this will print a message for each relation being checked, and will increase the level of detail shown for server errors.

- `-V --version`

Print the pg_amcheck version and exit.

- `--install-missing --install-missing=\`schema\``

Install any missing extensions that are required to check the database(s). If not yet installed, each extension’ s objects will be installed into the given **schema**, or if not specified into schema **pg_catalog**.At present, the only required extension is amcheck.

- **-? --help**

Show help about pg_amcheck command line arguments, and exit.

pg_basebackup

pg_basebackup — take a base backup of a IvorySQL cluster

Synopsis

pg_basebackup [**option**…]

Options

The following command-line options control the location and format of the output:

- **-D directory --pgdata=directory**

Sets the target directory to write the output to. pg_basebackup will create this directory (and any missing parent directories) if it does not exist. If it already exists, it must be empty.When the backup is in tar format, the target directory may be specified as **-** (dash), causing the tar file to be written to **stdout**.This option is required.

- **-F format --format=format**

Selects the format for the output. **format** can be one of the following: **p plain** Write the output as plain files, with the same layout as the source server’ s data directory and tablespaces. When the cluster has no additional tablespaces, the whole database will be placed in the target directory. If the cluster contains additional tablespaces, the main data directory will be placed in the target directory, but all other tablespaces will be placed in the same absolute path as they have on the source server. (See **--tablespace -mapping** to change that.)This is the default format. **t tar** Write the output as tar files in the target directory. The main data directory’ s contents will be written to a file named **base.tar**, and each other tablespace will be written to a separate tar file named after that tablespace’ s OID.If the target directory is specified as **-** (dash), the tar contents will be written to standard output, suitable for piping to (for example) gzip. This is only allowed if the cluster has no additional tablespaces and WAL streaming is not used.

- **-R --write-recovery-conf**

Creates a **standby.signal** file and appends connection settings to the **IvorySQL.auto.conf** file in the target directory (or within the base archive file when using tar format). This eases setting up a standby server using the results of the backup.The **IvorySQL.auto.conf** file will record the connection settings and, if specified, the replication slot that pg_basebackup is using, so that streaming replication will use the same settings later on.

- **-t target --target=target**

Instructs the server where to place the base backup. The default target is **client**, which specifies that the backup should be sent to the machine where pg_basebackup is running. If the target is instead set to **server:/some/path**, the backup will be stored on the machine where the server is running in the **/some/path** directory. Storing a backup on the server requires superuser privileges or having privileges of the **pg_write_server_files** role. If the target is set to **blackhole**, the contents are discarded and not stored anywhere. This should only be used for testing purposes, as you will not end up with an actual backup.Since WAL streaming is implemented by pg_basebackup rather than by the server, this option cannot be used together with **-Xstream**. Since that is the default, when this option is specified, you must also specify either **-Xfetch** or **-Xnone**.

- **-T olddir=newdir --tablespace-mapping=olddir=newdir**

Relocates the tablespace in directory `olddir` to `newdir` during the backup. To be effective, `olddir` must exactly match the path specification of the tablespace as it is defined on the source server. (But it is not an error if there is no tablespace in `olddir` on the source server.) Meanwhile `newdir` is a directory in the receiving host's filesystem. As with the main target directory, `newdir` need not exist already, but if it does exist it must be empty. Both `olddir` and `newdir` must be absolute paths. If either path needs to contain an equal sign (=), precede that with a backslash. This option can be specified multiple times for multiple tablespaces.If a tablespace is relocated in this way, the symbolic links inside the main data directory are updated to point to the new location. So the new data directory is ready to be used for a new server instance with all tablespaces in the updated locations.Currently, this option only works with plain output format; it is ignored if tar format is selected.

- `--waldir=waldir`

Sets the directory to write WAL (write-ahead log) files to. By default WAL files will be placed in the `pg_wal` subdirectory of the target directory, but this option can be used to place them elsewhere. `waldir` must be an absolute path. As with the main target directory, `waldir` need not exist already, but if it does exist it must be empty. This option can only be specified when the backup is in plain format.

- `-X method --wal-method=method`

Includes the required WAL (write-ahead log) files in the backup. This will include all write-ahead logs generated during the backup. Unless the method `none` is specified, it is possible to start a postmaster in the target directory without the need to consult the log archive, thus making the output a completely standalone backup.The following `methods` for collecting the write-ahead logs are supported: `n none` Don't include write-ahead logs in the backup. `f fetch`The write-ahead log files are collected at the end of the backup. Therefore, it is necessary for the source server's wal_keep_size parameter to be set high enough that the required log data is not removed before the end of the backup. If the required log data has been recycled before it's time to transfer it, the backup will fail and be unusable.When tar format is used, the write-ahead log files will be included in the `base.tar` file.`s stream`Stream write-ahead log data while the backup is being taken. This method will open a second connection to the server and start streaming the write-ahead log in parallel while running the backup. Therefore, it will require two replication connections not just one. As long as the client can keep up with the write-ahead log data, using this method requires no extra write-ahead logs to be saved on the source server.When tar format is used, the write-ahead log files will be written to a separate file named `pg_wal.tar` (if the server is a version earlier than 10, the file will be named `pg_xlog.tar`).This value is the default.

- `-z --gzip`

Enables gzip compression of tar file output, with the default compression level. Compression is only available when using the tar format, and the suffix `.gz` will automatically be added to all tar filenames.

- `-Z level -Z method[:*detail*] --compress=level --compress=method[:*detail*]`

Requests compression of the backup. If `client` or `server` is included, it specifies where the compression is to be performed. Compressing on the server will reduce transfer bandwidth but will increase server CPU consumption. The default is `client` except when `--target` is used. In that case, the backup is not being sent to the client, so only server compression is sensible. When `-Xstream`, which is the default, is used, server-side compression will not be applied to the WAL. To compress the WAL, use client-side compression, or specify `-Xfetch`.The compression method can be set to `gzip`, `lz4`, `zstd`, or `none` for no compression. A compression detail string can optionally be specified. If the detail string is an integer, it specifies the compression level. Otherwise, it should be a comma-separated list of items, each of the form `keyword` or `keyword=value`. Currently, the supported keywords are `level` and `workers`.If no compression level is specified, the default compression level will be used. If only a level is specified without mentioning an algorithm, `gzip` compression will be used if the level is greater than 0, and no compression will be used if the level is 0.When the tar format is used with `gzip`, `lz4`, or `zstd`, the suffix `.gz`, `.lz4`, or `.zst`, respectively, will be automatically added to all tar filenames. When the plain format is used, client-side compression may not be specified, but it is still possible to request server-side compression. If this is done, the server will compress the backup for transmission, and the client will decompress and extract it.When this option is used in combination with `-Xstream`, `pg_wal.tar` will be compressed using `gzip` if client-side gzip compression is selected, but will not be compressed if any other compression algorithm is selected, or if server-side compression is selected.

The following command-line options control the generation of the backup and the invocation of the

program:

- `-c {fast|spread} --checkpoint={fast|spread}`

Sets checkpoint mode to fast (immediate) or spread (the default) .

- `-C --create-slot`

Specifies that the replication slot named by the `--slot` option should be created before starting the backup. An error is raised if the slot already exists.

- `-l label --label=label`

Sets the label for the backup. If none is specified, a default value of "`pg_basebackup base backup`" will be used.

- `-n --no-clean`

By default, when `pg_basebackup` aborts with an error, it removes any directories it might have created before discovering that it cannot finish the job (for example, the target directory and write-ahead log directory). This option inhibits tidying-up and is thus useful for debugging.Note that tablespace directories are not cleaned up either way.

- `-N --no-sync`

By default, `pg_basebackup` will wait for all files to be written safely to disk. This option causes `pg_basebackup` to return without waiting, which is faster, but means that a subsequent operating system crash can leave the base backup corrupt. Generally, this option is useful for testing but should not be used when creating a production installation.

- `-P --progress`

Enables progress reporting. Turning this on will deliver an approximate progress report during the backup. Since the database may change during the backup, this is only an approximation and may not end at exactly `100%`. In particular, when WAL log is included in the backup, the total amount of data cannot be estimated in advance, and in this case the estimated target size will increase once it passes the total estimate without WAL.

- `-r rate --max-rate=rate`

Sets the maximum transfer rate at which data is collected from the source server. This can be useful to limit the impact of pg_basebackup on the server. Values are in kilobytes per second. Use a suffix of `M` to indicate megabytes per second. A suffix of `k` is also accepted, and has no effect. Valid values are between 32 kilobytes per second and 1024 megabytes per second.This option always affects transfer of the data directory. Transfer of WAL files is only affected if the collection method is `fetch`.

- `-S slotname --slot=slotname`

This option can only be used together with `-X stream`. It causes WAL streaming to use the specified replication slot. If the base backup is intended to be used as a streaming-replication standby using a replication slot, the standby should then use the same replication slot name as primary_slot_name. This ensures that the primary server does not remove any necessary WAL data in the time between the end of the base backup and the start of streaming replication on the new standby.The specified replication slot has to exist unless the option `-C` is also used.If this option is not specified and the server supports temporary replication slots (version 10 and later), then a temporary replication slot is automatically used for WAL streaming.

- `-v --verbose`

Enables verbose mode. Will output some extra steps during startup and shutdown, as well as show the exact file name that is currently being processed if progress reporting is also enabled.

- `--manifest-checksums=algorithm`

Specifies the checksum algorithm that should be applied to each file included in the backup manifest. Currently, the available algorithms are `NONE`, `CRC32C`, `SHA224`, `SHA256`, `SHA384`, and `SHA512`. The default is `CRC32C`.If `NONE` is selected, the backup manifest will not contain any checksums. Otherwise, it will contain a checksum of each file in the backup using the specified algorithm. In addition, the manifest will always contain a `SHA256` checksum of its own contents. The `SHA` algorithms are significantly more CPU-intensive than `CRC32C`, so selecting one of them may increase the time required to complete the backup.Using a SHA hash function provides a cryptographically secure digest of each file for users who wish to verify that the backup has not been tampered with, while the CRC32C algorithm provides a checksum that is much faster to calculate; it is good at catching errors due to accidental changes but is not resistant to malicious modifications. Note that, to be useful against an adversary who has access to the backup, the backup manifest would need to be stored securely elsewhere or otherwise verified not to have been modified since the backup was taken. pg_verifybackup can be used to check the integrity of a backup against the backup manifest.

- `--manifest-force-encode`

Forces all filenames in the backup manifest to be hex-encoded. If this option is not specified, only non-UTF8 filenames are hex-encoded. This option is mostly intended to test that tools which read a backup manifest file properly handle this case.

- `--no-estimate-size`

Prevents the server from estimating the total amount of backup data that will be streamed, resulting in the `backup_total` column in the `pg_stat_progress_basebackup` view always being `NULL`.Without this option, the backup will start by enumerating the size of the entire database, and then go back and send the actual contents. This may make the backup take slightly longer, and in particular it will take longer before the first data is sent. This option is useful to avoid such estimation time if it’ s too long.This option is not allowed when using `--progress`.

- `--no-manifest`

Disables generation of a backup manifest. If this option is not specified, the server will generate and send a backup manifest which can be verified using pg_verifybackup. The manifest is a list of every file present in the backup with the exception of any WAL files that may be included. It also stores the size, last modification time, and an optional checksum for each file.

- `--no-slot`

Prevents the creation of a temporary replication slot for the backup.By default, if log streaming is selected but no slot name is given with the `-S` option, then a temporary replication slot is created (if supported by the source server).The main purpose of this option is to allow taking a base backup when the server has no free replication slots. Using a replication slot is almost always preferred, because it prevents needed WAL from being removed by the server during the backup.

- `--no-verify-checksums`

Disables verification of checksums, if they are enabled on the server the base backup is taken from.By default, checksums are verified and checksum failures will result in a non-zero exit status. However, the base backup will not be removed in such a case, as if the `--no-clean` option had been used. Checksum verification failures will also be reported in the `pg_stat_database` view.

The following command-line options control the connection to the source server:

- `-d connstr --dbname=connstr`

Specifies parameters used to connect to the server, as a connection string; these will override any conflicting command line options.The option is called `--dbname` for consistency with other client applications, but because pg_basebackup doesn’ t connect to any particular database in the cluster, any database name in the connection string will be ignored.

- `-h host --host=host`

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for a Unix domain socket. The default is taken from the `PGHOST` environment variable, if set, else a Unix domain socket connection is attempted.

- `-p port --port=port`

Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections. Defaults to the `PGPORT` environment variable, if set, or a compiled-in default.

- `-s interval --status-interval=interval`

Specifies the number of seconds between status packets sent back to the source server. Smaller values allow more accurate monitoring of backup progress from the server. A value of zero disables periodic status updates completely, although an update will still be sent when requested by the server, to avoid timeout-based disconnects. The default value is 10 seconds.

- `-U username --username=username`

Specifies the user name to connect as.

- `-w --no-password`

Prevents issuing a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

- `-W --password`

Forces pg_basebackup to prompt for a password before connecting to the source server.This option is never essential, since pg_basebackup will automatically prompt for a password if the server demands password authentication. However, pg_basebackup will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing `-W` to avoid the extra connection attempt.

Other options are also available:

- `-V --version`

Prints the pg_basebackup version and exits.

- `-? --help`

Shows help about pg_basebackup command line arguments, and exits.

Environment

This utility, like most other IvorySQL utilities, uses the environment variables supported by libpq .

The environment variable `PG_COLOR` specifies whether to use color in diagnostic messages. Possible values are `always`, `auto` and `never`.

Notes

At the beginning of the backup, a checkpoint needs to be performed on the source server. This can take some time (especially if the option `--checkpoint=fast` is not used), during which pg_basebackup will appear to be idle.

The backup will include all files in the data directory and tablespaces, including the configuration files and any additional files placed in the directory by third parties, except certain temporary files managed by IvorySQL. But only regular files and directories are copied, except that symbolic links used for tablespaces are preserved. Symbolic links pointing to certain directories known to IvorySQL are copied as empty

directories. Other symbolic links and special device files are skipped.

In plain format, tablespaces will be backed up to the same path they have on the source server, unless the option `--tablespace-mapping` is used. Without this option, running a plain format base backup on the same host as the server will not work if tablespaces are in use, because the backup would have to be written to the same directory locations as the original tablespaces.

When tar format is used, it is the user’ s responsibility to unpack each tar file before starting a IvorySQL server that uses the data. If there are additional tablespaces, the tar files for them need to be unpacked in the correct locations. In this case the symbolic links for those tablespaces will be created by the server according to the contents of the `tablespace_map` file that is included in the `base.tar` file.

pg_basebackup works with servers of the same or an older major version.

pg_basebackup will preserve group permissions for data files if group permissions are enabled on the source cluster.

Examples

To create a base backup of the server at `mydbserver` and store it in the local directory `/usr/local/pgsql/data`:

```
$ pg_basebackup -h mydbserver -D /usr/local/pgsql/data
```

To create a backup of the local server with one compressed tar file for each tablespace, and store it in the directory `backup`, showing a progress report while running:

```
$ pg_basebackup -D backup -Ft -z -P
```

To create a backup of a single-tablespace local database and compress this with bzip2:

```
$ pg_basebackup -D - -Ft -X fetch | bzip2 > backup.tar.bz2
```

(This command will fail if there are multiple tablespaces in the database.)

To create a backup of a local database where the tablespace in `/opt/ts` is relocated to `./backup/ts`:

```
$ pg_basebackup -D backup/data -T /opt/ts=$(pwd)/backup/ts
```

To create a backup of a local server with one tar file for each tablespace compressed with gzip at level 9, stored in the directory `backup`:

```
$ pg_basebackup -D backup -Ft --compress=gzip:9
```

pgbench

pgbench — run a benchmark test on IvorySQL

Synopsis

`pgbench -i` [`option`…] [`dbname`]

pgbench [option…] [dbname]

Caution

pgbench -i creates four tables pgbench_accounts, pgbench_branches, pgbench_history, and pgbench_tellers, destroying any existing tables of these names. Be very careful to use another database if you have tables having these names!

At the default "scale factor" of 1, the tables initially contain this many rows:

```
table                   # of rows
---------------------------------
pgbench_branches        1
pgbench_tellers         10
pgbench_accounts        100000
pgbench_history         0
```

You can (and, for most purposes, probably should) increase the number of rows by using the -s (scale factor) option. The -F (fillfactor) option might also be used at this point.

Once you have done the necessary setup, you can run your benchmark with a command that doesn't include -i, that is

```
pgbench [ options ] dbname
```

In nearly all cases, you'll need some options to make a useful test. The most important options are -c (number of clients), -t (number of transactions), -T (time limit), and -f (specify a custom script file). See below for a full list.

Options

The following is divided into three subsections. Different options are used during database initialization and while running benchmarks, but some options are useful in both cases.

Initialization Options

pgbench accepts the following command-line initialization arguments:

- dbname

Specifies the name of the database to test in. If this is not specified, the environment variable PGDATABASE is used. If that is not set, the user name specified for the connection is used.

- -i --initialize

Required to invoke initialization mode.

- -I init_steps --init-steps=init_steps

Perform just a selected set of the normal initialization steps. init_steps specifies the initialization steps to be performed, using one character per step. Each step is invoked in the specified order. The default is dtgvp. The available steps are:`d` (Drop)Drop any existing pgbench tables.t (create Tables)Create the tables used by the standard pgbench scenario, namely pgbench_accounts, pgbench_branches, pgbench_history, and pgbench_tellers.g or G (Generate data, client-side or server-side)Generate data and load it into the standard tables, replacing any data already present.With g (client-side data generation), data is generated in pgbench

client and then sent to the server. This uses the client/server bandwidth extensively through a COPY. pgbench uses the FREEZE option with version 14 or later of IvorySQL to speed up subsequent VACUUM, unless partitions are enabled. Using g causes logging to print one message every 100,000 rows while generating data for the pgbench_accounts table.With G (server-side data generation), only small queries are sent from the pgbench client and then data is actually generated in the server. No significant bandwidth is required for this variant, but the server will do more work. Using G causes logging not to print any progress message while generating data.The default initialization behavior uses client-side data generation (equivalent to g).v (Vacuum)Invoke VACUUM on the standard tables.p (create Primary keys)Create primary key indexes on the standard tables.f (create Foreign keys)Create foreign key constraints between the standard tables. (Note that this step is not performed by default.)

- `-F fillfactor --fillfactor= fillfactor`

Create the pgbench_accounts, pgbench_tellers and pgbench_branches tables with the given fillfactor. Default is 100.

- `-n --no-vacuum`

Perform no vacuuming during initialization. (This option suppresses the v initialization step, even if it was specified in -I.)

- `-q --quiet`

Switch logging to quiet mode, producing only one progress message per 5 seconds. The default logging prints one message each 100,000 rows, which often outputs many lines per second (especially on good hardware).This setting has no effect if G is specified in -I.

- `-s scale_factor --scale=scale_factor`

Multiply the number of rows generated by the scale factor. For example, -s 100 will create 10,000,000 rows in the pgbench_accounts table. Default is 1. When the scale is 20,000 or larger, the columns used to hold account identifiers (aid columns) will switch to using larger integers (bigint), in order to be big enough to hold the range of account identifiers.

- `--foreign-keys`

Create foreign key constraints between the standard tables. (This option adds the f step to the initialization step sequence, if it is not already present.)

- `--index-tablespace=index_tablespace`

Create indexes in the specified tablespace, rather than the default tablespace.

- `--partition-method=`NAME`

Create a partitioned pgbench_accounts table with NAME method. Expected values are range or hash. This option requires that --partitions is set to non-zero. If unspecified, default is range.

- `--partitions=NUM`

Create a partitioned pgbench_accounts table with NUM partitions of nearly equal size for the scaled number of accounts. Default is 0, meaning no partitioning.

- `--tablespace=`tablespace`

Create tables in the specified tablespace, rather than the default tablespace.

- `--unlogged-tables`

Create all tables as unlogged tables, rather than permanent tables.

Benchmarking Options

pgbench accepts the following command-line benchmarking arguments:

- `-b scriptname[@weight] --builtin=scriptname[@weight]`

Add the specified built-in script to the list of scripts to be executed. Available built-in scripts are: `tpcb-like`, `simple-update` and `select-only`. Unambiguous prefixes of built-in names are accepted. With the special name `list`, show the list of built-in scripts and exit immediately.Optionally, write an integer weight after `@` to adjust the probability of selecting this script versus other ones. The default weight is 1. See below for details.

- `-c clients --client=clients`

Number of clients simulated, that is, number of concurrent database sessions. Default is 1.

- `-C --connect`

Establish a new connection for each transaction, rather than doing it just once per client session. This is useful to measure the connection overhead.

- `-d --debug`

Print debugging output.

- `-D varname=value --define=varname=value`

Define a variable for use by a custom script (see below). Multiple `-D` options are allowed.

- `-f filename[@weight] --file=filename[@weight]`

Add a transaction script read from `filename` to the list of scripts to be executed.Optionally, write an integer weight after `@` to adjust the probability of selecting this script versus other ones. The default weight is 1. (To use a script file name that includes an `@` character, append a weight so that there is no ambiguity, for example `filen@me@1`.) See below for details.

- `-j threads --jobs=threads`

Number of worker threads within pgbench. Using more than one thread can be helpful on multi-CPU machines. Clients are distributed as evenly as possible among available threads. Default is 1.

- `-l --log`

Write information about each transaction to a log file. See below for details.

- `-L limit --latency-limit=limit`

Transactions that last more than `limit` milliseconds are counted and reported separately, as late.When throttling is used (`--rate=…`), transactions that lag behind schedule by more than `limit` ms, and thus have no hope of meeting the latency limit, are not sent to the server at all. They are counted and reported separately as skipped.When the `--max-tries` option is used, a transaction which fails due to a serialization anomaly or from a deadlock will not be retried if the total time of all its tries is greater than `limit` ms. To limit only the time of tries and not their number, use `--max-tries=0`. By default, the option `--max-tries` is set to 1 and transactions with serialization/deadlock errors are not retried.

- `-M querymode --protocol=querymode`

Protocol to use for submitting queries to the server: `simple` : use simple query protocol. `extended` : use extended query protocol. `prepared` : use extended query protocol with prepared statements.In the `prepared` mode, pgbench reuses the parse analysis result starting from the second query iteration, so pgbench runs faster than in other modes.The default is simple query protocol.

- `-n --no-vacuum`

Perform no vacuuming before running the test. This option is necessary if you are running a custom test scenario that does not include the standard tables `pgbench_accounts`, `pgbench_branches`, `pgbench_history`, and `pgbench_tellers`.

- `-N --skip-some-updates`

Run built-in simple-update script. Shorthand for `-b simple-update`.

- `-P sec --progress=sec`

Show progress report every `sec` seconds. The report includes the time since the beginning of the run, the TPS since the last report, and the transaction latency average, standard deviation, and the number of failed transactions since the last report. Under throttling (`-R`), the latency is computed with respect to the transaction scheduled start time, not the actual transaction beginning time, thus it also includes the average schedule lag time. When `--max-tries` is used to enable transaction retries after serialization/deadlock errors, the report includes the number of retried transactions and the sum of all retries.

- `-r --report-per-command`

Report the following statistics for each command after the benchmark finishes: the average per-statement latency (execution time from the perspective of the client), the number of failures, and the number of retries after serialization or deadlock errors in this command. The report displays retry statistics only if the `--max-tries` option is not equal to 1.

- `-R rate --rate=rate`

Execute transactions targeting the specified rate instead of running as fast as possible (the default). The rate is given in transactions per second. If the targeted rate is above the maximum possible rate, the rate limit won't impact the results.The rate is targeted by starting transactions along a Poisson-distributed schedule time line. The expected start time schedule moves forward based on when the client first started, not when the previous transaction ended. That approach means that when transactions go past their original scheduled end time, it is possible for later ones to catch up again.When throttling is active, the transaction latency reported at the end of the run is calculated from the scheduled start times, so it includes the time each transaction had to wait for the previous transaction to finish. The wait time is called the schedule lag time, and its average and maximum are also reported separately. The transaction latency with respect to the actual transaction start time, i.e., the time spent executing the transaction in the database, can be computed by subtracting the schedule lag time from the reported latency.If `--latency-limit` is used together with `--rate`, a transaction can lag behind so much that it is already over the latency limit when the previous transaction ends, because the latency is calculated from the scheduled start time. Such transactions are not sent to the server, but are skipped altogether and counted separately.A high schedule lag time is an indication that the system cannot process transactions at the specified rate, with the chosen number of clients and threads. When the average transaction execution time is longer than the scheduled interval between each transaction, each successive transaction will fall further behind, and the schedule lag time will keep increasing the longer the test run is. When that happens, you will have to reduce the specified transaction rate.

- `-s scale_factor --scale=scale_factor`

Report the specified scale factor in pgbench's output. With the built-in tests, this is not necessary; the correct scale factor will be detected by counting the number of rows in the `pgbench_branches` table. However, when testing only custom benchmarks (`-f` option), the scale factor will be reported as 1 unless this option is used.

- `-S --select-only`

Run built-in select-only script. Shorthand for `-b select-only`.

- `-t transactions --transactions=transactions`

Number of transactions each client runs. Default is 10.

- `-T seconds --time=seconds`

Run the test for this many seconds, rather than a fixed number of transactions per client. **-t** and **-T** are mutually exclusive.

- **-v --vacuum-all**

Vacuum all four standard tables before running the test. With neither **-n** nor **-v**, pgbench will vacuum the **pgbench_tellers** and **pgbench_branches** tables, and will truncate **pgbench_history**.

- **--aggregate-interval=`seconds`**

Length of aggregation interval (in seconds). May be used only with **-l** option. With this option, the log contains per-interval summary data, as described below.

- **--failures-detailed**

Report failures in per-transaction and aggregation logs, as well as in the main and per-script reports, grouped by the following types:serialization failures;deadlock failures.

- **--log-prefix=`prefix`**

Set the filename prefix for the log files created by **--log**. The default is **pgbench_log**.

- **--max-tries=`number_of_tries`**

Enable retries for transactions with serialization/deadlock errors and set the maximum number of these tries. This option can be combined with the **--latency-limit** option which limits the total time of all transaction tries; moreover, you cannot use an unlimited number of tries (**--max-tries=0**) without **--latency-limit** or **--time**. The default value is 1 and transactions with serialization/deadlock errors are not retried.

- **--progress-timestamp**

When showing progress (option **-P**), use a timestamp (Unix epoch) instead of the number of seconds since the beginning of the run. The unit is in seconds, with millisecond precision after the dot. This helps compare logs generated by various tools.

- **--random-seed=seed**

Set random generator seed. Seeds the system random number generator, which then produces a sequence of initial generator states, one for each thread. Values for **seed** may be: **time** (the default, the seed is based on the current time), **rand** (use a strong random source, failing if none is available), or an unsigned decimal integer value. The random generator is invoked explicitly from a pgbench script (**random···** functions) or implicitly (for instance option **--rate** uses it to schedule transactions). When explicitly set, the value used for seeding is shown on the terminal. Any value allowed for **seed** may also be provided through the environment variable **PGBENCH_RANDOM_SEED**. To ensure that the provided seed impacts all possible uses, put this option first or use the environment variable.Setting the seed explicitly allows to reproduce a **pgbench** run exactly, as far as random numbers are concerned. As the random state is managed per thread, this means the exact same **pgbench** run for an identical invocation if there is one client per thread and there are no external or data dependencies. From a statistical viewpoint reproducing runs exactly is a bad idea because it can hide the performance variability or improve performance unduly, e.g., by hitting the same pages as a previous run. However, it may also be of great help for debugging, for instance re-running a tricky case which leads to an error. Use wisely.

- **--sampling-rate=`rate`**

Sampling rate, used when writing data into the log, to reduce the amount of log generated. If this option is given, only the specified fraction of transactions are logged. 1.0 means all transactions will be logged, 0.05 means only 5% of the transactions will be logged.Remember to take the sampling rate into account when processing the log file. For example, when computing TPS values, you need to multiply the numbers accordingly (e.g., with 0.01 sample rate, you’ ll only get 1/100 of the actual TPS).

- **--show-script=scriptname**

Show the actual code of builtin script **scriptname** on stderr, and exit immediately.

- **--verbose-errors**

Print messages about all errors and failures (errors without retrying) including which limit for retries was exceeded and how far it was exceeded for the serialization/deadlock failures. (Note that in this case the output can be significantly increased.).

Common Options

pgbench also accepts the following common command-line arguments for connection parameters:

- **-h hostname --host=hostname**

The database server’ s host name

- **-p port --port=port**

The database server’ s port number

- **-U login --username=login**

The user name to connect as

- **-V --version**

Print the pgbench version and exit.

- **-? --help**

Show help about pgbench command line arguments, and exit.

Exit Status

A successful run will exit with status 0. Exit status 1 indicates static problems such as invalid command-line options or internal errors which are supposed to never occur. Early errors that occur when starting benchmark such as initial connection failures also exit with status 1. Errors during the run such as database errors or problems in the script will result in exit status 2. In the latter case, pgbench will print partial results.

Environment

- **PGDATABASE PGHOST PGPORT PGUSER**

Default connection parameters.

This utility, like most other IvorySQL utilities, uses the environment variables supported by libpq .

The environment variable **PG_COLOR** specifies whether to use color in diagnostic messages. Possible values are **always**, **auto** and **never**.

pg_config

pg_config — retrieve information about the installed version of IvorySQL

Synopsis

pg_config [**option**⋯]

Options

To use pg_config, supply one or more of the following options:

- `--bindir`

Print the location of user executables. Use this, for example, to find the `psql` program. This is normally also the location where the `pg_config` program resides.

- `--docdir`

Print the location of documentation files.

- `--htmldir`

Print the location of HTML documentation files.

- `--includedir`

Print the location of C header files of the client interfaces.

- `--pkgincludedir`

Print the location of other C header files.

- `--includedir-server`

Print the location of C header files for server programming.

- `--libdir`

Print the location of object code libraries.

- `--pkglibdir`

Print the location of dynamically loadable modules, or where the server would search for them. (Other architecture-dependent data files might also be installed in this directory.)

- `--localedir`

Print the location of locale support files. (This will be an empty string if locale support was not configured when IvorySQL was built.)

- `--mandir`

Print the location of manual pages.

- `--sharedir`

Print the location of architecture-independent support files.

- `--sysconfdir`

Print the location of system-wide configuration files.

- `--pgxs`

Print the location of extension makefiles.

- `--configure`

Print the options that were given to the `configure` script when IvorySQL was configured for building. This can be used to reproduce the identical configuration, or to find out with what options a binary package was

built. (Note however that binary packages often contain vendor-specific custom patches.) See also the examples below.

- `--cc`

Print the value of the `CC` variable that was used for building IvorySQL. This shows the C compiler used.

- `--cppflags`

Print the value of the `CPPFLAGS` variable that was used for building IvorySQL. This shows C compiler switches needed at preprocessing time (typically, `-I` switches).

- `--cflags`

Print the value of the `CFLAGS` variable that was used for building IvorySQL. This shows C compiler switches.

- `--cflags_sl`

Print the value of the `CFLAGS_SL` variable that was used for building IvorySQL. This shows extra C compiler switches used for building shared libraries.

- `--ldflags`

Print the value of the `LDFLAGS` variable that was used for building IvorySQL. This shows linker switches.

- `--ldflags_ex`

Print the value of the `LDFLAGS_EX` variable that was used for building IvorySQL. This shows linker switches used for building executables only.

- `--ldflags_sl`

Print the value of the `LDFLAGS_SL` variable that was used for building IvorySQL. This shows linker switches used for building shared libraries only.

- `--libs`

Print the value of the `LIBS` variable that was used for building IvorySQL. This normally contains `-l` switches for external libraries linked into IvorySQL.

- `--version`

Print the version of IvorySQL.

- `-? --help`

Show help about pg_config command line arguments, and exit.

If more than one option is given, the information is printed in that order, one item per line. If no options are given, all available information is printed, with labels.

Example

To reproduce the build configuration of the current IvorySQL installation, run the following command:

```
eval ./configure `pg_config --configure`
```

The output of `pg_config --configure` contains shell quotation marks so arguments with spaces are represented correctly. Therefore, using `eval` is required for proper results.

pg_dump

pg_dump — extract a IvorySQL database into a script file or other archive file

Synopsis

pg_dump [connection-option···] [option···] [dbname]

Options

The following command-line options control the content and format of the output.

- dbname

  ```
  Specifies the name of the database to be dumped. If this is not specified, the
  environment variable `PGDATABASE` is used. If that is not set, the user name
  specified for the connection is used.
  ```

- -a --data-only

  ```
  Dump only the data, not the schema (data definitions). Table data, large objects,
  and sequence values are dumped.This option is similar to, but for historical
  reasons not identical to, specifying `--section=data`.
  ```

- -b --blobs

  ```
  Include large objects in the dump. This is the default behavior except when `--
  schema`, `--table`, or `--schema-only` is specified. The `-b` switch is therefore
  only useful to add large objects to dumps where a specific schema or table has been
  requested. Note that blobs are considered data and therefore will be included when
  `--data-only` is used, but not when `--schema-only` is.
  ```

- -B --no-blobs

  ```
  Exclude large objects in the dump.When both `-b` and `-B` are given, the behavior
  is to output large objects, when data is being dumped, see the `-b` documentation.
  ```

- -c --clean

  ```
  Output commands to clean (drop) database objects prior to outputting the commands
  for creating them. (Unless `--if-exists` is also specified, restore might generate
  some harmless error messages, if any objects were not present in the destination
  database.)This option is ignored when emitting an archive (non-text) output file.
  For the archive formats, you can specify the option when you call `pg_restore`.
  ```

- -C --create

  ```
  Begin the output with a command to create the database itself and reconnect to the
  created database. (With a script of this form, it doesn't matter which database in
  ```

the destination installation you connect to before running the script.) If `--clean` is also specified, the script drops and recreates the target database before reconnecting to it.With `--create`, the output also includes the database's comment if any, and any configuration variable settings that are specific to this database, that is, any `ALTER DATABASE ... SET ...` and `ALTER ROLE ... IN DATABASE ... SET ...` commands that mention this database. Access privileges for the database itself are also dumped, unless `--no-acl` is specified.This option is ignored when emitting an archive (non-text) output file. For the archive formats, you can specify the option when you call `pg_restore`.

- -e `pattern` --extension=`pattern`

  Dump only extensions matching *`pattern`*. When this option is not specified, all non-system extensions in the target database will be dumped. Multiple extensions can be selected by writing multiple `-e` switches. The *`pattern`* parameter is interpreted as a pattern according to the same rules used by psql's `\d` commands , so multiple extensions can also be selected by writing wildcard characters in the pattern. When using wildcards, be careful to quote the pattern if needed to prevent the shell from expanding the wildcards.Any configuration relation registered by `pg_extension_config_dump` is included in the dump if its extension is specified by `--extension`.NoteWhen `-e` is specified, pg_dump makes no attempt to dump any other database objects that the selected extension(s) might depend upon. Therefore, there is no guarantee that the results of a specific-extension dump can be successfully restored by themselves into a clean database.

- -E `encoding` --encoding=`encoding`

  Create the dump in the specified character set encoding. By default, the dump is created in the database encoding. (Another way to get the same result is to set the `PGCLIENTENCODING` environment variable to the desired dump encoding.)

- -f `file` --file=`file`

  Send output to the specified file. This parameter can be omitted for file based output formats, in which case the standard output is used. It must be given for the directory output format however, where it specifies the target directory instead of a file. In this case the directory is created by `pg_dump` and must not exist before.

- -F `format` --format=`format`

  Selects the format of the output. *`format`* can be one of the following:`p` `plain`Output a plain-text SQL script file (the default).`c` `custom`Output a custom-format archive suitable for input into pg_restore. Together with the directory output format, this is the most flexible output format in that it allows manual selection and reordering of archived items during restore. This format is also compressed by default.`d` `directory`Output a directory-format archive suitable for input into pg_restore. This will create a directory with one file for each table and blob being dumped, plus a so-called Table of Contents file describing the dumped objects in a machine-readable format that pg_restore can

read. A directory format archive can be manipulated with standard Unix tools; for example, files in an uncompressed archive can be compressed with the gzip tool. This format is compressed by default and also supports parallel dumps.`t` `tar`Output a `tar`-format archive suitable for input into pg_restore. The tar format is compatible with the directory format: extracting a tar-format archive produces a valid directory-format archive. However, the tar format does not support compression. Also, when using tar format the relative order of table data items cannot be changed during restore.

- -j `njobs` --jobs=`njobs`

  Run the dump in parallel by dumping *`njobs`* tables simultaneously. This option may reduce the time needed to perform the dump but it also increases the load on the database server. You can only use this option with the directory output format because this is the only output format where multiple processes can write their data at the same time.pg_dump will open *`njobs`* + 1 connections to the database, so make sure your http://www.postgresql.org/docs/17/runtime-config-connection.html#GUC-MAX-CONNECTIONS[max_connections] setting is high enough to accommodate all connections.Requesting exclusive locks on database objects while running a parallel dump could cause the dump to fail. The reason is that the pg_dump leader process requests shared locks on the objects that the worker processes are going to dump later in order to make sure that nobody deletes them and makes them go away while the dump is running. If another client then requests an exclusive lock on a table, that lock will not be granted but will be queued waiting for the shared lock of the leader process to be released. Consequently any other access to the table will not be granted either and will queue after the exclusive lock request. This includes the worker process trying to dump the table. Without any precautions this would be a classic deadlock situation. To detect this conflict, the pg_dump worker process requests another shared lock using the `NOWAIT` option. If the worker process is not granted this shared lock, somebody else must have requested an exclusive lock in the meantime and there is no way to continue with the dump, so pg_dump has no choice but to abort the dump.To perform a parallel dump, the database server needs to support synchronized snapshots, a feature that was introduced in IvorySQL for primary servers and 10 for standbys. With this feature, database clients can ensure they see the same data set even though they use different connections. `pg_dump -j` uses multiple database connections; it connects to the database once with the leader process and once again for each worker job. Without the synchronized snapshot feature, the different worker jobs wouldn't be guaranteed to see the same data in each connection, which could lead to an inconsistent backup.

- -n `pattern` --schema=`pattern`

  Dump only schemas matching *`pattern`*; this selects both the schema itself, and all its contained objects. When this option is not specified, all non-system schemas in the target database will be dumped. Multiple schemas can be selected by writing multiple `-n` switches. The *`pattern`* parameter is interpreted as a pattern according to the same rules used by psql's `\d` commands, so multiple schemas can also be selected by writing wildcard characters in the pattern. When using wildcards, be careful to quote the pattern if needed to prevent the shell from expanding the wildcards. NoteWhen `-n` is specified, pg_dump makes no attempt to dump any other database objects that the selected schema(s) might depend upon.

```
Therefore, there is no guarantee that the results of a specific-schema dump can be
successfully restored by themselves into a clean database.NoteNon-schema objects
such as blobs are not dumped when `-n` is specified. You can add blobs back to the
dump with the `--blobs` switch.
```

- -N `pattern` --exclude-schema=`pattern`

```
Do not dump any schemas matching *`pattern`*. The pattern is interpreted according
to the same rules as for `-n`. `-N` can be given more than once to exclude schemas
matching any of several patterns.When both `-n` and `-N` are given, the behavior is
to dump just the schemas that match at least one `-n` switch but no `-N` switches.
If `-N` appears without `-n`, then schemas matching `-N` are excluded from what is
otherwise a normal dump.
```

- -O --no-owner

```
Do not output commands to set ownership of objects to match the original database.
By default, pg_dump issues `ALTER OWNER` or `SET SESSION AUTHORIZATION` statements
to set ownership of created database objects. These statements will fail when the
script is run unless it is started by a superuser (or the same user that owns all
of the objects in the script). To make a script that can be restored by any user,
but will give that user ownership of all the objects, specify `-O`.This option is
ignored when emitting an archive (non-text) output file. For the archive formats,
you can specify the option when you call `pg_restore`.
```

- -R --no-reconnect

```
This option is obsolete but still accepted for backwards compatibility.
```

- -s --schema-only

```
Dump only the object definitions (schema), not data.This option is the inverse of
`--data-only`. It is similar to, but for historical reasons not identical to,
specifying `--section=pre-data --section=post-data`.(Do not confuse this with the
`--schema` option, which uses the word “schema” in a different meaning.)To
exclude table data for only a subset of tables in the database, see `--exclude-
table-data`.
```

- -S `username` --superuser=`username`

```
Specify the superuser user name to use when disabling triggers. This is relevant
only if `--disable-triggers` is used. (Usually, it's better to leave this out, and
instead start the resulting script as superuser.)
```

- -t `pattern` --table=`pattern`

```
Dump only tables with names matching *`pattern`*. Multiple tables can be selected
```

by writing multiple `-t` switches. The *`pattern`* parameter is interpreted as a pattern according to the same rules used by psql's `\d` commands , so multiple tables can also be selected by writing wildcard characters in the pattern. When using wildcards, be careful to quote the pattern if needed to prevent the shell from expanding the wildcards; As well as tables, this option can be used to dump the definition of matching views, materialized views, foreign tables, and sequences. It will not dump the contents of views or materialized views, and the contents of foreign tables will only be dumped if the corresponding foreign server is specified with `--include-foreign-data`.The `-n` and `-N` switches have no effect when `-t` is used, because tables selected by `-t` will be dumped regardless of those switches, and non-table objects will not be dumped.NoteWhen `-t` is specified, pg_dump makes no attempt to dump any other database objects that the selected table(s) might depend upon. Therefore, there is no guarantee that the results of a specific-table dump can be successfully restored by themselves into a clean database.

- `-T `pattern` --exclude-table=`pattern``

  Do not dump any tables matching *`pattern`*. The pattern is interpreted according to the same rules as for `-t`. `-T` can be given more than once to exclude tables matching any of several patterns.When both `-t` and `-T` are given, the behavior is to dump just the tables that match at least one `-t` switch but no `-T` switches. If `-T` appears without `-t`, then tables matching `-T` are excluded from what is otherwise a normal dump.

- `-v --verbose`

  Specifies verbose mode. This will cause pg_dump to output detailed object comments and start/stop times to the dump file, and progress messages to standard error. Repeating the option causes additional debug-level messages to appear on standard error.

- `-V --version`

  Print the pg_dump version and exit.

- `-x --no-privileges --no-acl`

  Prevent dumping of access privileges (grant/revoke commands).

- `-Z `0..9` --compress=`0..9``

  Specify the compression level to use. Zero means no compression. For the custom and directory archive formats, this specifies compression of individual table-data segments, and the default is to compress at a moderate level. For plain text output, setting a nonzero compression level causes the entire output file to be compressed, as though it had been fed through gzip; but the default is not to compress. The tar archive format currently does not support compression at all.

- --binary-upgrade

  This option is for use by in-place upgrade utilities. Its use for other purposes is not recommended or supported. The behavior of the option may change in future releases without notice.

- --column-inserts --attribute-inserts

  Dump data as `INSERT` commands with explicit column names (`INSERT INTO *`table`* (*`column`*, ...) VALUES ...`). This will make restoration very slow; it is mainly useful for making dumps that can be loaded into non-IvorySQL databases. Any error during restoring will cause only rows that are part of the problematic `INSERT` to be lost, rather than the entire table contents.

- --disable-dollar-quoting

  This option disables the use of dollar quoting for function bodies, and forces them to be quoted using SQL standard string syntax.

- --disable-triggers

  This option is relevant only when creating a data-only dump. It instructs pg_dump to include commands to temporarily disable triggers on the target tables while the data is restored. Use this if you have referential integrity checks or other triggers on the tables that you do not want to invoke during data restore.Presently, the commands emitted for `--disable-triggers` must be done as superuser. So, you should also specify a superuser name with `-S`, or preferably be careful to start the resulting script as a superuser.This option is ignored when emitting an archive (non-text) output file. For the archive formats, you can specify the option when you call `pg_restore`.

- --enable-row-security

  This option is relevant only when dumping the contents of a table which has row security. By default, pg_dump will set http://www.postgresql.org/docs/17/runtime-config-client.html#GUC-ROW-SECURITY[row_security] to off, to ensure that all data is dumped from the table. If the user does not have sufficient privileges to bypass row security, then an error is thrown. This parameter instructs pg_dump to set http://www.postgresql.org/docs/17/runtime-config-client.html#GUC-ROW-SECURITY[row_security] to on instead, allowing the user to dump the parts of the contents of the table that they have access to.Note that if you use this option currently, you probably also want the dump be in `INSERT` format, as the `COPY FROM` during restore does not support row security.

- --exclude-table-data=`pattern`

  Do not dump data for any tables matching *`pattern`*. The pattern is interpreted according to the same rules as for `-t`. `--exclude-table-data` can be given more

than once to exclude tables matching any of several patterns. This option is useful when you need the definition of a particular table even though you do not need the data in it.To exclude data for all tables in the database, see `--schema-only`.

- --extra-float-digits=`ndigits`

  Use the specified value of `extra_float_digits` when dumping floating-point data, instead of the maximum available precision. Routine dumps made for backup purposes should not use this option.

- --if-exists

  Use conditional commands (i.e., add an `IF EXISTS` clause) when cleaning database objects. This option is not valid unless `--clean` is also specified.

- --include-foreign-data=`foreignserver`

  Dump the data for any foreign table with a foreign server matching *`foreignserver`* pattern. Multiple foreign servers can be selected by writing multiple `--include-foreign-data` switches. Also, the *`foreignserver`* parameter is interpreted as a pattern according to the same rules used by psql's `\d` commands, so multiple foreign servers can also be selected by writing wildcard characters in the pattern. When using wildcards, be careful to quote the pattern if needed to prevent the shell from expanding the wildcards; The only exception is that an empty pattern is disallowed.NoteWhen `--include-foreign-data` is specified, pg_dump does not check that the foreign table is writable. Therefore, there is no guarantee that the results of a foreign table dump can be successfully restored.

- --inserts

  Dump data as `INSERT` commands (rather than `COPY`). This will make restoration very slow; it is mainly useful for making dumps that can be loaded into non-IvorySQL databases. Any error during restoring will cause only rows that are part of the problematic `INSERT` to be lost, rather than the entire table contents. Note that the restore might fail altogether if you have rearranged column order. The `--column-inserts` option is safe against column order changes, though even slower.

- --load-via-partition-root

  When dumping data for a table partition, make the `COPY` or `INSERT` statements target the root of the partitioning hierarchy that contains it, rather than the partition itself. This causes the appropriate partition to be re-determined for each row when the data is loaded. This may be useful when restoring data on a server where rows do not always fall into the same partitions as they did on the original server. That could happen, for example, if the partitioning column is of type text and the two systems have different definitions of the collation used to sort the partitioning column.It is best not to use parallelism when restoring from an archive made with this option, because pg_restore will not know exactly which partition(s) a given archive data item will load data into. This could result in

inefficiency due to lock conflicts between parallel jobs, or perhaps even restore failures due to foreign key constraints being set up before all the relevant data is loaded.

- --lock-wait-timeout=`timeout`

  Do not wait forever to acquire shared table locks at the beginning of the dump. Instead fail if unable to lock a table within the specified *`timeout`*. The timeout may be specified in any of the formats accepted by `SET statement_timeout`

- --no-comments

  Do not dump comments.

- --no-publications

  Do not dump publications.

- --no-security-labels

  Do not dump security labels.

- --no-subscriptions

  Do not dump subscriptions.

- --no-sync

  By default, `pg_dump` will wait for all files to be written safely to disk. This option causes `pg_dump` to return without waiting, which is faster, but means that a subsequent operating system crash can leave the dump corrupt. Generally, this option is useful for testing but should not be used when dumping data from production installation.

- --no-tablespaces

  Do not output commands to select tablespaces. With this option, all objects will be created in whichever tablespace is the default during restore.This option is ignored when emitting an archive (non-text) output file. For the archive formats, you can specify the option when you call `pg_restore`.

- --no-toast-compression

  Do not output commands to set TOAST compression methods. With this option, all

columns will be restored with the default compression setting.

- --no-unlogged-table-data

  Do not dump the contents of unlogged tables and sequences. This option has no effect on whether or not the table and sequence definitions (schema) are dumped; it only suppresses dumping the table and sequence data. Data in unlogged tables and sequences is always excluded when dumping from a standby server.

- --on-conflict-do-nothing

  Add `ON CONFLICT DO NOTHING` to `INSERT` commands. This option is not valid unless `--inserts`, `--column-inserts` or `--rows-per-insert` is also specified.

- --quote-all-identifiers

  Force quoting of all identifiers. This option is recommended when dumping a database from a server whose IvorySQL major version is different from pg_dump's, or when the output is intended to be loaded into a server of a different major version. By default, pg_dump quotes only identifiers that are reserved words in its own major version. This sometimes results in compatibility issues when dealing with servers of other versions that may have slightly different sets of reserved words. Using `--quote-all-identifiers` prevents such issues, at the price of a harder-to-read dump script.

- --rows-per-insert=`nrows`

  Dump data as `INSERT` commands (rather than `COPY`). Controls the maximum number of rows per `INSERT` command. The value specified must be a number greater than zero. Any error during restoring will cause only rows that are part of the problematic `INSERT` to be lost, rather than the entire table contents.

- --section=`sectionname`

  Only dump the named section. The section name can be `pre-data`, `data`, or `post-data`. This option can be specified more than once to select multiple sections. The default is to dump all sections.The data section contains actual table data, large-object contents, and sequence values. Post-data items include definitions of indexes, triggers, rules, and constraints other than validated check constraints. Pre-data items include all other data definition items.

- --serializable-deferrable

  Use a `serializable` transaction for the dump, to ensure that the snapshot used is consistent with later database states; but do this by waiting for a point in the transaction stream at which no anomalies can be present, so that there isn't a risk of the dump failing or causing other transactions to roll back with a

`serialization_failure`.This option is not beneficial for a dump which is intended only for disaster recovery. It could be useful for a dump used to load a copy of the database for reporting or other read-only load sharing while the original database continues to be updated. Without it the dump may reflect a state which is not consistent with any serial execution of the transactions eventually committed. For example, if batch processing techniques are used, a batch may show as closed in the dump without all of the items which are in the batch appearing.This option will make no difference if there are no read-write transactions active when pg_dump is started. If read-write transactions are active, the start of the dump may be delayed for an indeterminate length of time. Once running, performance with or without the switch is the same.

- `--snapshot=`snapshotname``

  Use the specified synchronized snapshot when making a dump of the database.This option is useful when needing to synchronize the dump with a logical replication slot or with a concurrent session.In the case of a parallel dump, the snapshot name defined by this option is used rather than taking a new snapshot.

- `--strict-names`

  Require that each extension (`-e`/`--extension`), schema (`-n`/`--schema`) and table (`-t`/`--table`) qualifier match at least one extension/schema/table in the database to be dumped. Note that if none of the extension/schema/table qualifiers find matches, pg_dump will generate an error even without `--strict-names`.This option has no effect on `-N`/`--exclude-schema`, `-T`/`--exclude-table`, or `--exclude-table-data`. An exclude pattern failing to match any objects is not considered an error.

- `--use-set-session-authorization`

  Output SQL-standard `SET SESSION AUTHORIZATION` commands instead of `ALTER OWNER` commands to determine object ownership. This makes the dump more standards-compatible, but depending on the history of the objects in the dump, might not restore properly. Also, a dump using `SET SESSION AUTHORIZATION` will certainly require superuser privileges to restore correctly, whereas `ALTER OWNER` requires lesser privileges.

- `-? --help`

  Show help about pg_dump command line arguments, and exit.

The following command-line options control the database connection parameters.

- `-d `dbname` --dbname=`dbname``

  Specifies the name of the database to connect to. This is equivalent to specifying *`dbname`* as the first non-option argument on the command line. The *`dbname`* can be a http://www.postgresql.org/docs/17/libpq-connect.html#LIBPQ-

CONNSTRING[connection string]. If so, connection string parameters will override any conflicting command line options.

- -h `host` --host=`host`

  Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket. The default is taken from the `PGHOST` environment variable, if set, else a Unix domain socket connection is attempted.

- -p `port` --port=`port`

  Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections. Defaults to the `PGPORT` environment variable, if set, or a compiled-in default.

- -U `username` --username=`username`

  User name to connect as.

- -w --no-password

  Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

- -W --password

  Force pg_dump to prompt for a password before connecting to a database.This option is never essential, since pg_dump will automatically prompt for a password if the server demands password authentication. However, pg_dump will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing `-W` to avoid the extra connection attempt.

- --role=`rolename`

  Specifies a role name to be used to create the dump. This option causes pg_dump to issue a `SET ROLE` *`rolename`* command after connecting to the database. It is useful when the authenticated user (specified by `-U`) lacks privileges needed by pg_dump, but can switch to a role with the required rights. Some installations have a policy against logging in directly as a superuser, and use of this option allows dumps to be made without violating the policy.

Environment

- `PGDATABASE PGHOST PGOPTIONS PGPORT PGUSER`

  ```
  Default connection parameters.
  ```

- `PG_COLOR`

  ```
  Specifies whether to use color in diagnostic messages. Possible values are
  `always`, `auto` and `never`.
  ```

This utility, like most other IvorySQL utilities, also uses the environment variables supported by libpq.

Diagnostics

pg_dump internally executes `SELECT` statements. If you have problems running pg_dump, make sure you are able to select information from the database using, for example, psql. Also, any default connection settings and environment variables used by the libpq front-end library will apply.

The database activity of pg_dump is normally collected by the cumulative statistics system. If this is undesirable, you can set parameter `track_counts` to false via `PGOPTIONS` or the `ALTER USER` command.

Notes

If your database cluster has any local additions to the `template1` database, be careful to restore the output of pg_dump into a truly empty database; otherwise you are likely to get errors due to duplicate definitions of the added objects. To make an empty database without any local additions, copy from `template0` not `template1`, for example:

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

When a data-only dump is chosen and the option `--disable-triggers` is used, pg_dump emits commands to disable triggers on user tables before inserting the data, and then commands to re-enable them after the data has been inserted. If the restore is stopped in the middle, the system catalogs might be left in the wrong state.

The dump file produced by pg_dump does not contain the statistics used by the optimizer to make query planning decisions. Therefore, it is wise to run `ANALYZE` after restoring from a dump file to ensure optimal performance.

When dumping logical replication subscriptions, pg_dump will generate `CREATE SUBSCRIPTION` commands that use the `connect = false` option, so that restoring the subscription does not make remote connections for creating a replication slot or for initial table copy. That way, the dump can be restored without requiring network access to the remote servers. It is then up to the user to reactivate the subscriptions in a suitable way. If the involved hosts have changed, the connection information might have to be changed. It might also be appropriate to truncate the target tables before initiating a new full table copy. If users intend to copy initial data during refresh they must create the slot with `two_phase = false`. After the initial sync, the `two_phase` option will be automatically enabled by the subscriber if the subscription had been originally created with `two_phase = true` option.

Examples

To dump a database called `mydb` into an SQL-script file:

```
$ pg_dump mydb > db.sql
```

To reload such a script into a (freshly created) database named `newdb`:

```
$ psql -d newdb -f db.sql
```

To dump a database into a custom-format archive file:

```
$ pg_dump -Fc mydb > db.dump
```

To dump a database into a directory-format archive:

```
$ pg_dump -Fd mydb -f dumpdir
```

To dump a database into a directory-format archive in parallel with 5 worker jobs:

```
$ pg_dump -Fd mydb -j 5 -f dumpdir
```

To reload an archive file into a (freshly created) database named `newdb`:

```
$ pg_restore -d newdb db.dump
```

To reload an archive file into the same database it was dumped from, discarding the current contents of that database:

```
$ pg_restore -d postgres --clean --create db.dump
```

To dump a single table named `mytab`:

```
$ pg_dump -t mytab mydb > db.sql
```

To dump all tables whose names start with `emp` in the `detroit` schema, except for the table named `employee_log`:

```
$ pg_dump -t 'detroit.emp*' -T detroit.employee_log mydb > db.sql
```

To dump all schemas whose names start with `east` or `west` and end in `gsm`, excluding any schemas whose names contain the word `test`:

```
$ pg_dump -n 'east*gsm' -n 'west*gsm' -N '*test*' mydb > db.sql
```

The same, using regular expression notation to consolidate the switches:

```
$ pg_dump -n '(east|west)*gsm' -N '*test*' mydb > db.sql
```

To dump all database objects except for tables whose names begin with `ts_`:

```
$ pg_dump -T 'ts_*' mydb > db.sql
```

To specify an upper-case or mixed-case name in `-t` and related switches, you need to double-quote the name; else it will be folded to lower case.But double quotes are special to the shell, so in turn they must be quoted. Thus, to dump a single table with a mixed-case name, you need something like

```
$ pg_dump -t "\"MixedCaseName\"" mydb > mytab.sql
```

pg_dumpall

pg_dumpall — extract a IvorySQL database cluster into a script file

Synopsis

`pg_dumpall` [`connection-option`⋯] [`option`⋯]

Options

The following command-line options control the content and format of the output.

- `-a --data-only`

Dump only the data, not the schema (data definitions).

- `-c --clean`

Include SQL commands to clean (drop) databases before recreating them. `DROP` commands for roles and tablespaces are added as well.

- `-E encoding --encoding=encoding`

Create the dump in the specified character set encoding. By default, the dump is created in the database encoding. (Another way to get the same result is to set the `PGCLIENTENCODING` environment variable to the desired dump encoding.)

- `-f filename --file=filename`

Send output to the specified file. If this is omitted, the standard output is used.

- `-g --globals-only`

Dump only global objects (roles and tablespaces), no databases.

- `-O --no-owner`

Do not output commands to set ownership of objects to match the original database. By default, pg_dumpall issues `ALTER OWNER` or `SET SESSION AUTHORIZATION` statements to set ownership of created schema elements. These statements will fail when the script is run unless it is started by a superuser (or the same user that owns all of the objects in the script). To make a script that can be restored by any user, but will give that user ownership of all the objects, specify `-O`.

- `-r --roles-only`

Dump only roles, no databases or tablespaces.

- `-s --schema-only`

Dump only the object definitions (schema), not data.

- `-S username --superuser=username`

Specify the superuser user name to use when disabling triggers. This is relevant only if `--disable-triggers` is used. (Usually, it's better to leave this out, and instead start the resulting script as superuser.)

- `-t --tablespaces-only`

Dump only tablespaces, no databases or roles.

- `-v --verbose`

Specifies verbose mode. This will cause pg_dumpall to output start/stop times to the dump file, and progress messages to standard error. Repeating the option causes additional debug-level messages to appear on standard error. The option is also passed down to pg_dump.

- `-V --version`

Print the pg_dumpall version and exit.

- `-x --no-privileges --no-acl`

Prevent dumping of access privileges (grant/revoke commands).

- `--binary-upgrade`

This option is for use by in-place upgrade utilities. Its use for other purposes is not recommended or supported. The behavior of the option may change in future releases without notice.

- `--column-inserts --attribute-inserts`

Dump data as `INSERT` commands with explicit column names (`INSERT INTO table (column, ⋯) VALUES ⋯`). This will make restoration very slow; it is mainly useful for making dumps that can be loaded into non-IvorySQL databases.

- `--disable-dollar-quoting`

This option disables the use of dollar quoting for function bodies, and forces them to be quoted using SQL standard string syntax.

- `--disable-triggers`

This option is relevant only when creating a data-only dump. It instructs pg_dumpall to include commands to temporarily disable triggers on the target tables while the data is restored. Use this if you have referential integrity checks or other triggers on the tables that you do not want to invoke during data restore.Presently, the commands emitted for `--disable-triggers` must be done as superuser. So, you should also specify a superuser name with `-S`, or preferably be careful to start the resulting script as a superuser.

- `--exclude-database=`pattern`

Do not dump databases whose name matches `pattern`. Multiple patterns can be excluded by writing multiple `--exclude-database` switches. The `pattern` parameter is interpreted as a pattern according to the same rules used by psql's `\d` commands, so multiple databases can also be excluded by writing wildcard characters in the pattern. When using wildcards, be careful to quote the pattern if needed to prevent shell wildcard expansion.

- `--extra-float-digits=`ndigits``

Use the specified value of extra_float_digits when dumping floating-point data, instead of the maximum available precision. Routine dumps made for backup purposes should not use this option.

- `--if-exists`

Use conditional commands (i.e., add an `IF EXISTS` clause) to drop databases and other objects. This option is not valid unless `--clean` is also specified.

- `--inserts`

Dump data as `INSERT` commands (rather than `COPY`). This will make restoration very slow; it is mainly useful for making dumps that can be loaded into non-IvorySQL databases. Note that the restore might fail altogether if you have rearranged column order. The `--column-inserts` option is safer, though even slower.

- `--load-via-partition-root`

When dumping data for a table partition, make the `COPY` or `INSERT` statements target the root of the partitioning hierarchy that contains it, rather than the partition itself. This causes the appropriate partition to be re-determined for each row when the data is loaded. This may be useful when restoring data on a server where rows do not always fall into the same partitions as they did on the original server. That could happen, for example, if the partitioning column is of type text and the two systems have different definitions of the collation used to sort the partitioning column.

- `--lock-wait-timeout=`timeout``

Do not wait forever to acquire shared table locks at the beginning of the dump. Instead, fail if unable to lock a table within the specified `timeout`. The timeout may be specified in any of the formats accepted by `SET statement_timeout`.

- `--no-comments`

Do not dump comments.

- `--no-publications`

Do not dump publications.

- `--no-role-passwords`

Do not dump passwords for roles. When restored, roles will have a null password, and password authentication will always fail until the password is set. Since password values aren’ t needed when this option is specified, the role information is read from the catalog view `pg_roles` instead of `pg_authid`. Therefore, this option also helps if access to `pg_authid` is restricted by some security policy.

- `--no-security-labels`

Do not dump security labels.

- `--no-subscriptions`

Do not dump subscriptions.

- `--no-sync`

By default, `pg_dumpall` will wait for all files to be written safely to disk. This option causes `pg_dumpall` to return without waiting, which is faster, but means that a subsequent operating system crash can leave the dump corrupt. Generally, this option is useful for testing but should not be used when dumping data from production installation.

- `--no-table-access-method`

Do not output commands to select table access methods. With this option, all objects will be created with whichever table access method is the default during restore.

- `--no-tablespaces`

Do not output commands to create tablespaces nor select tablespaces for objects. With this option, all objects will be created in whichever tablespace is the default during restore.

- `--no-toast-compression`

Do not output commands to set TOAST compression methods. With this option, all columns will be restored with the default compression setting.

- `--no-unlogged-table-data`

Do not dump the contents of unlogged tables. This option has no effect on whether or not the table definitions (schema) are dumped; it only suppresses dumping the table data.

- `--on-conflict-do-nothing`

Add `ON CONFLICT DO NOTHING` to `INSERT` commands. This option is not valid unless `--inserts` or `--column-inserts` is also specified.

- `--quote-all-identifiers`

Force quoting of all identifiers. This option is recommended when dumping a database from a server whose IvorySQL major version is different from pg_dumpall's, or when the output is intended to be loaded into a server of a different major version. By default, pg_dumpall quotes only identifiers that are reserved words in its own major version. This sometimes results in compatibility issues when dealing with servers of other versions that may have slightly different sets of reserved words. Using `--quote-all-identifiers` prevents such issues, at the price of a harder-to-read dump script.

- `--rows-per-insert=`nrows`

Dump data as `INSERT` commands (rather than `COPY`). Controls the maximum number of rows per `INSERT` command. The value specified must be a number greater than zero. Any error during restoring will cause only rows that are part of the problematic `INSERT` to be lost, rather than the entire table contents.

- `--use-set-session-authorization`

Output SQL-standard `SET SESSION AUTHORIZATION` commands instead of `ALTER OWNER` commands to determine object ownership. This makes the dump more standards compatible, but depending on the history of the objects in the dump, might not restore properly.

- `-? --help`

Show help about pg_dumpall command line arguments, and exit.

The following command-line options control the database connection parameters.

- `-d connstr --dbname=connstr`

Specifies parameters used to connect to the server, as a connection string; these will override any conflicting command line options.The option is called `--dbname` for consistency with other client applications, but because pg_dumpall needs to connect to many databases, the database name in the connection string will be ignored. Use the `-l` option to specify the name of the database used for the initial connection, which will dump global objects and discover what other databases should be dumped.

- `-h host --host=host`

Specifies the host name of the machine on which the database server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket. The default is taken from the `PGHOST` environment variable, if set, else a Unix domain socket connection is attempted.

- `-l dbname --database=dbname`

Specifies the name of the database to connect to for dumping global objects and discovering what other databases should be dumped. If not specified, the `postgres` database will be used, and if that does not exist, `template1` will be used.

- `-p `port` --port=`port``

Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections. Defaults to the `PGPORT` environment variable, if set, or a compiled-in default.

- `-U username --username=username`

User name to connect as.

- `-w --no-password`

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

- `-W --password`

Force pg_dumpall to prompt for a password before connecting to a database.This option is never essential, since pg_dumpall will automatically prompt for a password if the server demands password authentication. However, pg_dumpall will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing `-W` to avoid the extra connection attempt.Note that the password prompt will occur again for each database to be dumped. Usually, it’ s better to set up a `~/.pgpass` file than to rely on manual password entry.

- `--role=`rolename``

Specifies a role name to be used to create the dump. This option causes pg_dumpall to issue a `SET ROLE rolename` command after connecting to the database. It is useful when the authenticated user (specified by `-U`) lacks privileges needed by pg_dumpall, but can switch to a role with the required rights. Some installations have a policy against logging in directly as a superuser, and use of this option allows dumps to be made without violating the policy.

Environment

- `PGHOST PGOPTIONS PGPORT PGUSER`

Default connection parameters

- `PG_COLOR`

Specifies whether to use color in diagnostic messages. Possible values are `always`, `auto` and `never`.

This utility, like most other IvorySQL utilities, also uses the environment variables supported by libpq

Notes

Since pg_dumpall calls pg_dump internally, some diagnostic messages will refer to pg_dump.

The `--clean` option can be useful even when your intention is to restore the dump script into a fresh cluster. Use of `--clean` authorizes the script to drop and re-create the built-in `postgres` and `template1` databases, ensuring that those databases will retain the same properties (for instance, locale and encoding) that they had in the source cluster. Without the option, those databases will retain their existing database-level properties, as well as any pre-existing contents.

Once restored, it is wise to run `ANALYZE` on each database so the optimizer has useful statistics. You can also run `vacuumdb -a -z` to analyze all databases.

The dump script should not be expected to run completely without errors. In particular, because the script will issue `CREATE ROLE` for every role existing in the source cluster, it is certain to get a "role already exists" error for the bootstrap superuser, unless the destination cluster was initialized with a different bootstrap superuser name. This error is harmless and should be ignored. Use of the `--clean` option is likely to produce additional harmless error messages about non-existent objects, although you can minimize those by adding `--if-exists`.

pg_dumpall requires all needed tablespace directories to exist before the restore; otherwise, database creation will fail for databases in non-default locations.

Examples

To dump all databases:

```
$ pg_dumpall > db.out
```

To restore database(s) from this file, you can use:

```
$ psql -f db.out postgres
```

It is not important to which database you connect here since the script file created by pg_dumpall will contain the appropriate commands to create and connect to the saved databases. An exception is that if you specified `--clean`, you must connect to the `postgres` database initially; the script will attempt to drop other databases immediately, and that will fail for the database you are connected to.

pg_isready

pg_isready — check the connection status of a IvorySQL server

Synopsis

`pg_isready` [`connection-option`⋯] [`option`⋯]

Options

- `-d dbname --dbname=dbname`

Specifies the name of the database to connect to. The `dbname` can be a connection string. If so, connection string parameters will override any conflicting command line options.

- `-h hostname --host=hostname`

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix-domain socket.

- `-p port --port=port`

Specifies the TCP port or the local Unix-domain socket file extension on which the server is listening for connections. Defaults to the value of the `PGPORT` environment variable or, if not set, to the port specified at compile time, usually 5432.

- `-q --quiet`

Do not display status message. This is useful when scripting.

- `-t seconds --timeout=seconds`

The maximum number of seconds to wait when attempting connection before returning that the server is not responding. Setting to 0 disables. The default is 3 seconds.

- `-U username --username=username`

Connect to the database as the user `username` instead of the default.

- `-V --version`

Print the pg_isready version and exit.

- `-? --help`

Show help about pg_isready command line arguments, and exit.

Exit Status

pg_isready returns `0` to the shell if the server is accepting connections normally, `1` if the server is rejecting connections (for example during startup), `2` if there was no response to the connection attempt, and `3` if no attempt was made (for example due to invalid parameters).

Environment

`pg_isready`, like most other IvorySQL utilities, also uses the environment variables supported by libpq .

The environment variable `PG_COLOR` specifies whether to use color in diagnostic messages. Possible values are `always`, `auto` and `never`.

Notes

It is not necessary to supply correct user name, password, or database name values to obtain the server status; however, if incorrect values are provided, the server will log a failed connection attempt.

Examples

Standard Usage:

```
$ pg_isready
/tmp:5432 - accepting connections
$ echo $?
0
```

Running with connection parameters to a IvorySQL cluster in startup:

```
$ pg_isready -h localhost -p 5433
localhost:5433 - rejecting connections
$ echo $?
1
```

Running with connection parameters to a non-responsive IvorySQL cluster:

```
$ pg_isready -h someremotehost
someremotehost:5432 - no response
```

```
$ echo $?
2
```

pg_receivewal

pg_receivewal — stream write-ahead logs from a IvorySQL server

Synopsis

pg_receivewal [**option**…]

Options

- **-D directory --directory=directory**

Directory to write the output to.This parameter is required.

- **-E lsn --endpos=lsn**

Automatically stop replication and exit with normal exit status 0 when receiving reaches the specified LSN.If there is a record with LSN exactly equal to **lsn**, the record will be processed.

- **--if-not-exists**

Do not error out when **--create-slot** is specified and a slot with the specified name already exists.

- **-n --no-loop**

Don’t loop on connection errors. Instead, exit right away with an error.

- **--no-sync**

This option causes **pg_receivewal** to not force WAL data to be flushed to disk. This is faster, but means that a subsequent operating system crash can leave the WAL segments corrupt. Generally, this option is useful for testing but should not be used when doing WAL archiving on a production deployment.This option is incompatible with **--synchronous**.

- **-s interval --status-interval=interval**

Specifies the number of seconds between status packets sent back to the server. This allows for easier monitoring of the progress from server. A value of zero disables the periodic status updates completely, although an update will still be sent when requested by the server, to avoid timeout disconnect. The default value is 10 seconds.

- **-S slotname --slot=slotname**

Require pg_receivewal to use an existing replication slot, When this option is used, pg_receivewal will report a flush position to the server, indicating when each segment has been synchronized to disk so that the server can remove that segment if it is not otherwise needed.When the replication client of pg_receivewal is configured on the server as a synchronous standby, then using a replication slot will report the flush position to the server, but only when a WAL file is closed. Therefore, that configuration will cause transactions on the primary to wait for a long time and effectively not work satisfactorily. The option **--synchronous** (see below) must be specified in addition to make this work correctly.

- **--synchronous**

Flush the WAL data to disk immediately after it has been received. Also send a status packet back to the server immediately after flushing, regardless of **--status-interval**.This option should be specified if the replication client of pg_receivewal is configured on the server as a synchronous standby, to ensure that timely feedback is sent to the server.

- `-v --verbose`

Enables verbose mode.

- `-Z level -Z method[:*detail*] --compress=level --compress=method[:*detail*]`

Enables compression of write-ahead logs.The compression method can be set to `gzip`, `lz4` (if IvorySQL was compiled with `--with-lz4`) or `none` for no compression. A compression detail string can optionally be specified. If the detail string is an integer, it specifies the compression level. Otherwise, it should be a comma-separated list of items, each of the form `keyword` or `keyword=value`. Currently, the only supported keyword is `level`.If no compression level is specified, the default compression level will be used. If only a level is specified without mentioning an algorithm, `gzip` compression will be used if the level is greater than 0, and no compression will be used if the level is 0.The suffix `.gz` will automatically be added to all filenames when using `gzip`, and the suffix `.lz4` is added when using `lz4`.

The following command-line options control the database connection parameters.

- `-d connstr --dbname=connstr`

Specifies parameters used to connect to the server, as a connection string; these will override any conflicting command line options.The option is called `--dbname` for consistency with other client applications, but because pg_receivewal doesn’ t connect to any particular database in the cluster, database name in the connection string will be ignored.

- `-h `host` --host=`host``

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket. The default is taken from the `PGHOST` environment variable, if set, else a Unix domain socket connection is attempted.

- `-p `port` --port=`port``

Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections. Defaults to the `PGPORT` environment variable, if set, or a compiled-in default.

- `-U `username` --username=`username``

User name to connect as.

- `-w --no-password`

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

- `-W --password`

Force pg_receivewal to prompt for a password before connecting to a database.This option is never essential, since pg_receivewal will automatically prompt for a password if the server demands password authentication. However, pg_receivewal will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing `-W` to avoid the extra connection attempt.

pg_receivewal can perform one of the two following actions in order to control physical replication slots:

- `--create-slot`

Create a new physical replication slot with the name specified in `--slot`, then exit.

- `--drop-slot`

Drop the replication slot with the name specified in `--slot`, then exit.

Other options are also available:

- `-V --version`

Print the pg_receivewal version and exit.

- `-? --help`

Show help about pg_receivewal command line arguments, and exit.

Exit Status

pg_receivewal will exit with status 0 when terminated by the SIGINT signal. (That is the normal way to end it. Hence it is not an error.) For fatal errors or other signals, the exit status will be nonzero.

Environment

This utility, like most other IvorySQL utilities, uses the environment variables supported by libpq

The environment variable `PG_COLOR` specifies whether to use color in diagnostic messages. Possible values are `always`, `auto` and `never`.

Notes

When using pg_receivewal instead of archive_command as the main WAL backup method, it is strongly recommended to use replication slots. Otherwise, the server is free to recycle or remove write-ahead log files before they are backed up, because it does not have any information, either from archive_command or the replication slots, about how far the WAL stream has been archived. Note, however, that a replication slot will fill up the server’ s disk space if the receiver does not keep up with fetching the WAL data.

pg_receivewal will preserve group permissions on the received WAL files if group permissions are enabled on the source cluster.

Examples

To stream the write-ahead log from the server at `mydbserver` and store it in the local directory `/usr/local/pgsql/archive`:

```
$ pg_receivewal -h mydbserver -D /usr/local/pgsql/archive
```

pg_recvlogical

pg_recvlogical — control IvorySQL logical decoding streams

Synopsis

`pg_recvlogical` [`option`⋯]

Options

At least one of the following options must be specified to select an action:

- `--create-slot`

Create a new logical replication slot with the name specified by `--slot`, using the output plugin specified by `--plugin`, for the database specified by `--dbname`.The `--two-phase` can be specified with `--create-slot` to enable decoding of prepared transactions.

- `--drop-slot`

Drop the replication slot with the name specified by `--slot`, then exit.

- `--start`

Begin streaming changes from the logical replication slot specified by `--slot`, continuing until terminated by a signal. If the server side change stream ends with a server shutdown or disconnect, retry in a loop unless `--no-loop` is specified.The stream format is determined by the output plugin specified when the slot was created.The connection must be to the same database used to create the slot.

`--create-slot` and `--start` can be specified together. `--drop-slot` cannot be combined with another action.

The following command-line options control the location and format of the output and other replication behavior:

- `-E lsn --endpos=lsn`

In `--start` mode, automatically stop replication and exit with normal exit status 0 when receiving reaches the specified LSN. If specified when not in `--start` mode, an error is raised.If there's a record with LSN exactly equal to `lsn`, the record will be output.The `--endpos` option is not aware of transaction boundaries and may truncate output partway through a transaction. Any partially output transaction will not be consumed and will be replayed again when the slot is next read from. Individual messages are never truncated.

- `-f filename --file=filename`

Write received and decoded transaction data into this file. Use `-` for stdout.

- `-F interval_seconds --fsync-interval=interval_seconds`

Specifies how often pg_recvlogical should issue `fsync()` calls to ensure the output file is safely flushed to disk.The server will occasionally request the client to perform a flush and report the flush position to the server. This setting is in addition to that, to perform flushes more frequently.Specifying an interval of `0` disables issuing `fsync()` calls altogether, while still reporting progress to the server. In this case, data could be lost in the event of a crash.

- `-I lsn --startpos=lsn`

In `--start` mode, start replication from the given LSN. For details on the effect of this.

- `--if-not-exists`

Do not error out when `--create-slot` is specified and a slot with the specified name already exists.

- `-n --no-loop`

When the connection to the server is lost, do not retry in a loop, just exit.

- `-o name[=value] --option=name[=value]`

Pass the option `name` to the output plugin with, if specified, the option value `value`. Which options exist and their effects depends on the used output plugin.

- `-P plugin --plugin=plugin`

When creating a slot, use the specified logical decoding output plugin. This option has no effect if the slot already exists.

- `-s interval_seconds --status-interval=interval_seconds`

This option has the same effect as the option of the same name in pg_receivewal. See the description there.

- `-S slot_name --slot=slot_name`

In `--start` mode, use the existing logical replication slot named `slot_name`. In `--create-slot` mode, create the slot with this name. In `--drop-slot` mode, delete the slot with this name.

- `-t --two-phase`

Enables decoding of prepared transactions. This option may only be specified with `--create-slot`

- `-v --verbose`

Enables verbose mode.

The following command-line options control the database connection parameters.

- `-d dbname --dbname=dbname`

The database to connect to. See the description of the actions for what this means in detail. The `dbname` can be a connection string. If so, connection string parameters will override any conflicting command line options. Defaults to the user name.

- `-h hostname-or-ip --host=hostname-or-ip`

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket. The default is taken from the `PGHOST` environment variable, if set, else a Unix domain socket connection is attempted.

- `-p port --port=port`

Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections. Defaults to the `PGPORT` environment variable, if set, or a compiled-in default.

- `-U user --username=user`

User name to connect as. Defaults to current operating system user name.

- `-w --no-password`

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

- `-W --password`

Force pg_recvlogical to prompt for a password before connecting to a database.This option is never essential, since pg_recvlogical will automatically prompt for a password if the server demands password authentication. However, pg_recvlogical will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing `-W` to avoid the extra connection attempt.

The following additional options are available:

- `-V --version`

Print the pg_recvlogical version and exit.

- `-? --help`

Show help about pg_recvlogical command line arguments, and exit.

Environment

This utility, like most other IvorySQL utilities, uses the environment variables supported by libpq .

The environment variable **PG_COLOR** specifies whether to use color in diagnostic messages. Possible values are **always**, **auto** and **never**.

Notes

pg_recvlogical will preserve group permissions on the received WAL files if group permissions are enabled on the source cluster.

pg_restore

pg_restore — restore a IvorySQL database from an archive file created by pg_dump

Synopsis

pg_restore [**connection-option**⋯] [**option**⋯] [**filename**]

Options

pg_restore accepts the following command line arguments.

- **filename**

Specifies the location of the archive file (or directory, for a directory-format archive) to be restored. If not specified, the standard input is used.

- **-a --data-only**

Restore only the data, not the schema (data definitions). Table data, large objects, and sequence values are restored, if present in the archive.This option is similar to, but for historical reasons not identical to, specifying **--section=data**.

- **-c --clean**

Clean (drop) database objects before recreating them. (Unless **--if-exists** is used, this might generate some harmless error messages, if any objects were not present in the destination database.)

- **-C --create**

Create the database before restoring into it. If **--clean** is also specified, drop and recreate the target database before connecting to it.With **--create**, pg_restore also restores the database' s comment if any, and any configuration variable settings that are specific to this database, that is, any **ALTER DATABASE ⋯ SET ⋯** and **ALTER ROLE ⋯ IN DATABASE ⋯ SET ⋯** commands that mention this database. Access privileges for the database itself are also restored, unless **--no-acl** is specified.When this option is used, the database named with **-d** is used only to issue the initial **DROP DATABASE** and **CREATE DATABASE** commands. All data is restored into the database name that appears in the archive.

- **-d dbname --dbname=dbname**

Connect to database **dbname** and restore directly into the database. The **dbname** can be a connection string. If so, connection string parameters will override any conflicting command line options.

- **-e --exit-on-error**

Exit if an error is encountered while sending SQL commands to the database. The default is to continue and to display a count of errors at the end of the restoration.

- **-f `filename` --file=`filename`**

Specify output file for generated script, or for the listing when used with **-l**. Use **-** for stdout.

- `-F format --format=format`

Specify format of the archive. It is not necessary to specify the format, since pg_restore will determine the format automatically. If specified, it can be one of the following:`c` `custom`The archive is in the custom format of pg_dump.`d directory`The archive is a directory archive.`t tar`The archive is a `tar` archive.

- `-I index --index=index`

Restore definition of named index only. Multiple indexes may be specified with multiple `-I` switches.

- `-j number-of-jobs --jobs=number-of-jobs`

Run the most time-consuming steps of pg_restore — those that load data, create indexes, or create constraints — concurrently, using up to `number-of-jobs` concurrent sessions. This option can dramatically reduce the time to restore a large database to a server running on a multiprocessor machine. This option is ignored when emitting a script rather than connecting directly to a database server.Each job is one process or one thread, depending on the operating system, and uses a separate connection to the server.The optimal value for this option depends on the hardware setup of the server, of the client, and of the network. Factors include the number of CPU cores and the disk setup. A good place to start is the number of CPU cores on the server, but values larger than that can also lead to faster restore times in many cases. Of course, values that are too high will lead to decreased performance because of thrashing.Only the custom and directory archive formats are supported with this option. The input must be a regular file or directory (not, for example, a pipe or standard input). Also, multiple jobs cannot be used together with the option `--single -transaction`.

- `-l --list`

List the table of contents of the archive. The output of this operation can be used as input to the `-L` option. Note that if filtering switches such as `-n` or `-t` are used with `-l`, they will restrict the items listed.

- `-L list-file --use-list=list-file`

Restore only those archive elements that are listed in `list-file`, and restore them in the order they appear in the file. Note that if filtering switches such as `-n` or `-t` are used with `-L`, they will further restrict the items restored.`list-file` is normally created by editing the output of a previous `-l` operation. Lines can be moved or removed, and can also be commented out by placing a semicolon (`;`) at the start of the line. See below for examples.

- `-n schema --schema=schema`

Restore only objects that are in the named schema. Multiple schemas may be specified with multiple `-n` switches. This can be combined with the `-t` option to restore just a specific table.

- `-N schema --exclude-schema=schema`

Do not restore objects that are in the named schema. Multiple schemas to be excluded may be specified with multiple `-N` switches.When both `-n` and `-N` are given for the same schema name, the `-N` switch wins and the schema is excluded.

- `-O --no-owner`

Do not output commands to set ownership of objects to match the original database. By default, pg_restore issues `ALTER OWNER` or `SET SESSION AUTHORIZATION` statements to set ownership of created schema elements. These statements will fail unless the initial connection to the database is made by a superuser (or the same user that owns all of the objects in the script). With `-O`, any user name can be used for the initial connection, and this user will own all the created objects.

- `-P function-name(argtype [, ⋯]) --function=function-name(argtype [, ⋯])`

Restore the named function only. Be careful to spell the function name and arguments exactly as they appear in the dump file’ s table of contents. Multiple functions may be specified with multiple `-P` switches.

- `-R --no-reconnect`

This option is obsolete but still accepted for backwards compatibility.

- `-s --schema-only`

Restore only the schema (data definitions), not data, to the extent that schema entries are present in the archive.This option is the inverse of `--data-only`. It is similar to, but for historical reasons not identical to, specifying `--section=pre-data --section=post-data`.(Do not confuse this with the `--schema` option, which uses the word “schema” in a different meaning.)

- `-S username --superuser=username`

Specify the superuser user name to use when disabling triggers. This is relevant only if `--disable-triggers` is used.

- `-t table --table=table`

Restore definition and/or data of only the named table. For this purpose, “table” includes views, materialized views, sequences, and foreign tables. Multiple tables can be selected by writing multiple `-t` switches. This option can be combined with the `-n` option to specify table(s) in a particular schema.NoteWhen `-t` is specified, pg_restore makes no attempt to restore any other database objects that the selected table(s) might depend upon. Therefore, there is no guarantee that a specific-table restore into a clean database will succeed.NoteThis flag does not behave identically to the `-t` flag of pg_dump. There is not currently any provision for wild-card matching in pg_restore, nor can you include a schema name within its `-t`. And, while pg_dump’s `-t` flag will also dump subsidiary objects (such as indexes) of the selected table(s), pg_restore’s `-t` flag does not include such subsidiary objects.

- `-T trigger --trigger=trigger`

Restore named trigger only. Multiple triggers may be specified with multiple `-T` switches.

- `-v --verbose`

Specifies verbose mode. This will cause pg_restore to output detailed object comments and start/stop times to the output file, and progress messages to standard error. Repeating the option causes additional debug-level messages to appear on standard error.

- `-V --version`

Print the pg_restore version and exit.

- `-x --no-privileges --no-acl`

Prevent restoration of access privileges (grant/revoke commands).

- `-1 --single-transaction`

Execute the restore as a single transaction (that is, wrap the emitted commands in `BEGIN` / `COMMIT`). This ensures that either all the commands complete successfully, or no changes are applied. This option implies `--exit-on-error`.

- `--disable-triggers`

This option is relevant only when performing a data-only restore. It instructs pg_restore to execute commands to temporarily disable triggers on the target tables while the data is restored. Use this if you have referential integrity checks or other triggers on the tables that you do not want to invoke during data restore.Presently, the commands emitted for `--disable-triggers` must be done as superuser. So you should also specify a superuser name with `-S` or, preferably, run pg_restore as a IvorySQL superuser.

- `--enable-row-security`

This option is relevant only when restoring the contents of a table which has row security. By default,

pg_restore will set row_security to off, to ensure that all data is restored in to the table. If the user does not have sufficient privileges to bypass row security, then an error is thrown. This parameter instructs pg_restore to set row_security to on instead, allowing the user to attempt to restore the contents of the table with row security enabled. This might still fail if the user does not have the right to insert the rows from the dump into the table.Note that this option currently also requires the dump be in `INSERT` format, as `COPY FROM` does not support row security.

- `--if-exists`

Use conditional commands (i.e., add an `IF EXISTS` clause) to drop database objects. This option is not valid unless `--clean` is also specified.

- `--no-comments`

Do not output commands to restore comments, even if the archive contains them.

- `--no-data-for-failed-tables`

By default, table data is restored even if the creation command for the table failed (e.g., because it already exists). With this option, data for such a table is skipped. This behavior is useful if the target database already contains the desired table contents. For example, auxiliary tables for IvorySQL extensions such as PostGIS might already be loaded in the target database; specifying this option prevents duplicate or obsolete data from being loaded into them.This option is effective only when restoring directly into a database, not when producing SQL script output.

- `--no-publications`

Do not output commands to restore publications, even if the archive contains them.

- `--no-security-labels`

Do not output commands to restore security labels, even if the archive contains them.

- `--no-subscriptions`

Do not output commands to restore subscriptions, even if the archive contains them.

- `--no-table-access-method`

Do not output commands to select table access methods. With this option, all objects will be created with whichever access method is the default during restore.

- `--no-tablespaces`

Do not output commands to select tablespaces. With this option, all objects will be created in whichever tablespace is the default during restore.

- `--section=`sectionname`

Only restore the named section. The section name can be `pre-data`, `data`, or `post-data`. This option can be specified more than once to select multiple sections. The default is to restore all sections.The data section contains actual table data as well as large-object definitions. Post-data items consist of definitions of indexes, triggers, rules and constraints other than validated check constraints. Pre-data items consist of all other data definition items.

- `--strict-names`

Require that each schema (`-n`/`--schema`) and table (`-t`/`--table`) qualifier match at least one schema/table in the backup file.

- `--use-set-session-authorization`

Output SQL-standard `SET SESSION AUTHORIZATION` commands instead of `ALTER OWNER` commands to

determine object ownership. This makes the dump more standards-compatible, but depending on the history of the objects in the dump, might not restore properly.

- `-? --help`

Show help about pg_restore command line arguments, and exit.

pg_restore also accepts the following command line arguments for connection parameters:

- `-h host --host=host`

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket. The default is taken from the `PGHOST` environment variable, if set, else a Unix domain socket connection is attempted.

- `-p port --port=port`

Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections. Defaults to the `PGPORT` environment variable, if set, or a compiled-in default.

- `-U username --username=username`

User name to connect as.

- `-w --no-password`

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

- `-W --password`

Force pg_restore to prompt for a password before connecting to a database.This option is never essential, since pg_restore will automatically prompt for a password if the server demands password authentication. However, pg_restore will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing `-W` to avoid the extra connection attempt.

- `--role=`rolename``

Specifies a role name to be used to perform the restore. This option causes pg_restore to issue a `SET ROLE rolename` command after connecting to the database. It is useful when the authenticated user (specified by `-U`) lacks privileges needed by pg_restore, but can switch to a role with the required rights. Some installations have a policy against logging in directly as a superuser, and use of this option allows restores to be performed without violating the policy.

Environment

- `PGHOST PGOPTIONS PGPORT PGUSER`

Default connection parameters

- `PG_COLOR`

Specifies whether to use color in diagnostic messages. Possible values are `always`, `auto` and `never`.

This utility, like most other IvorySQL utilities, also uses the environment variables supported by libpq. However, it does not read `PGDATABASE` when a database name is not supplied.

Diagnostics

When a direct database connection is specified using the `-d` option, pg_restore internally executes SQL statements. If you have problems running pg_restore, make sure you are able to select information from the

database using, for example, psql. Also, any default connection settings and environment variables used by the libpq front-end library will apply.

Notes

If your installation has any local additions to the `template1` database, be careful to load the output of pg_restore into a truly empty database; otherwise you are likely to get errors due to duplicate definitions of the added objects. To make an empty database without any local additions, copy from `template0` not `template1`, for example:

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

The limitations of pg_restore are detailed below.

- When restoring data to a pre-existing table and the option `--disable-triggers` is used, pg_restore emits commands to disable triggers on user tables before inserting the data, then emits commands to re-enable them after the data has been inserted. If the restore is stopped in the middle, the system catalogs might be left in the wrong state.
- pg_restore cannot restore large objects selectively; for instance, only those for a specific table. If an archive contains large objects, then all large objects will be restored, or none of them if they are excluded via `-L`, `-t`, or other options.

Examples

Assume we have dumped a database called `mydb` into a custom-format dump file:

```
$ pg_dump -Fc mydb > db.dump
```

To drop the database and recreate it from the dump:

```
$ dropdb mydb
$ pg_restore -C -d postgres db.dump
```

The database named in the `-d` switch can be any database existing in the cluster; pg_restore only uses it to issue the `CREATE DATABASE` command for `mydb`. With `-C`, data is always restored into the database name that appears in the dump file.

To restore the dump into a new database called `newdb`:

```
$ createdb -T template0 newdb
$ pg_restore -d newdb db.dump
```

Notice we don’ t use `-C`, and instead connect directly to the database to be restored into. Also note that we clone the new database from `template0` not `template1`, to ensure it is initially empty.

To reorder database items, it is first necessary to dump the table of contents of the archive:

```
$ pg_restore -l db.dump > db.list
```

The listing file consists of a header and one line for each item, e.g.:

```
;
; Archive created at Mon Sep 14 13:55:39 2009
;     dbname: DBDEMOS
;     TOC Entries: 81
;     Compression: 9
;     Dump Version: 1.10-0
;     Format: CUSTOM
;     Integer: 4 bytes
;     Offset: 8 bytes
;     Dumped from database version: 8.3.5
;     Dumped by pg_dump version: 8.3.8
;
;
; Selected TOC Entries:
;
3; 2615 2200 SCHEMA - public pasha
1861; 0 0 COMMENT - SCHEMA public pasha
1862; 0 0 ACL - public pasha
317; 1247 17715 TYPE public composite pasha
319; 1247 25899 DOMAIN public domain0 pasha
```

Semicolons start a comment, and the numbers at the start of lines refer to the internal archive ID assigned to each item.

Lines in the file can be commented out, deleted, and reordered. For example:

```
10; 145433 TABLE map_resolutions postgres
;2; 145344 TABLE species postgres
;4; 145359 TABLE nt_header postgres
6; 145402 TABLE species_records postgres
;8; 145416 TABLE ss_old postgres
```

could be used as input to pg_restore and would only restore items 10 and 6, in that order:

```
$ pg_restore -L db.list db.dump
```

pg_verifybackup

pg_verifybackup — verify the integrity of a base backup of a IvorySQL cluster

Synopsis

pg_verifybackup [*option*⋯]

Options

pg_verifybackup accepts the following command-line arguments:

- `-e --exit-on-error`

Exit as soon as a problem with the backup is detected. If this option is not specified, `pg_verifybackup` will continue checking the backup even after a problem has been detected, and will report all problems detected as errors.

- `-i path --ignore=path`

Ignore the specified file or directory, which should be expressed as a relative path name, when comparing the list of data files actually present in the backup to those listed in the `backup_manifest` file. If a directory is specified, this option affects the entire subtree rooted at that location. Complaints about extra files, missing files, file size differences, or checksum mismatches will be suppressed if the relative path name matches the specified path name. This option can be specified multiple times.

- `-m path --manifest-path=path`

Use the manifest file at the specified path, rather than one located in the root of the backup directory.

- `-n --no-parse-wal`

Don’t attempt to parse write-ahead log data that will be needed to recover from this backup.

- `-q --quiet`

Don’t print anything when a backup is successfully verified.

- `-s --skip-checksums`

Do not verify data file checksums. The presence or absence of files and the sizes of those files will still be checked. This is much faster, because the files themselves do not need to be read.

- `-w path --wal-directory=path`

Try to parse WAL files stored in the specified directory, rather than in `pg_wal`. This may be useful if the backup is stored in a separate location from the WAL archive.

Other options are also available:

- `-V --version`

Print the pg_verifybackup version and exit.

- `-? --help`

Show help about pg_verifybackup command line arguments, and exit.

Examples

To create a base backup of the server at `mydbserver` and verify the integrity of the backup:

```
$ pg_basebackup -h mydbserver -D /usr/local/pgsql/data
$ pg_verifybackup /usr/local/pgsql/data
```

To create a base backup of the server at `mydbserver`, move the manifest somewhere outside the backup directory, and verify the backup:

```
$ pg_basebackup -h mydbserver -D /usr/local/pgsql/backup1234
$ mv /usr/local/pgsql/backup1234/backup_manifest
/my/secure/location/backup_manifest.1234
$ pg_verifybackup -m /my/secure/location/backup_manifest.1234
/usr/local/pgsql/backup1234
```

To verify a backup while ignoring a file that was added manually to the backup directory, and also skipping checksum verification:

```
$ pg_basebackup -h mydbserver -D /usr/local/pgsql/data
$ edit /usr/local/pgsql/data/note.to.self
$ pg_verifybackup --ignore=note.to.self --skip-checksums /usr/local/pgsql/data
```

psql

psql — IvorySQL interactive terminal

Synopsis

psql [**option**⋯] [**dbname** [**username**]]

Options

- **-a --echo-all**

Print all nonempty input lines to standard output as they are read. (This does not apply to lines read interactively.) This is equivalent to setting the variable **ECHO** to **all**.

- **-A --no-align**

Switches to unaligned output mode. (The default output mode is **aligned**.) This is equivalent to **\pset format unaligned**.

- **-b --echo-errors**

Print failed SQL commands to standard error output. This is equivalent to setting the variable **ECHO** to **errors**.

- **-c command --command=command**

Specifies that psql is to execute the given command string, **command**. This option can be repeated and combined in any order with the **-f** option. When either **-c** or **-f** is specified, psql does not read commands from standard input; instead it terminates after processing all the **-c** and **-f** options in sequence.**command** must be either a command string that is completely parsable by the server (i.e., it contains no psql-specific features), or a single backslash command. Thus you cannot mix SQL and psql meta-commands within a **-c** option. To achieve that, you could use repeated **-c** options or pipe the string into psql, for example:` psql -c '\x' -c 'SELECT * FROM foo;'` **or`echo '\x \\ SELECT * FROM foo;' | psql `(**\\` is the separator meta-command.)Each SQL command string passed to **-c** is sent to the server as a single request. Because of this, the server executes it as a single transaction even if the string contains multiple SQL commands, unless there are explicit **BEGIN**/**COMMIT** commands included in the string to divide it into multiple transactions.If having several commands executed in one transaction is not desired, use repeated **-c** commands or feed multiple commands to psql’ s standard input, either using echo as illustrated above, or via a shell here-document, for example:` psql <<EOF \x SELECT * FROM foo; EOF `

- **--csv**

Switches to CSV (Comma-Separated Values) output mode. This is equivalent to **\pset format csv**.

- `-d dbname --dbname=dbname`

Specifies the name of the database to connect to. This is equivalent to specifying **dbname** as the first non-option argument on the command line. The **dbname** can be a connection string. If so, connection string parameters will override any conflicting command line options.

- `-e --echo-queries`

Copy all SQL commands sent to the server to standard output as well. This is equivalent to setting the variable **ECHO** to **queries**.

- `-E --echo-hidden`

Echo the actual queries generated by **\d** and other backslash commands. You can use this to study psql's internal operations. This is equivalent to setting the variable **ECHO_HIDDEN** to **on**.

- `-f filename --file=filename`

Read commands from the file **filename**, rather than standard input. This option can be repeated and combined in any order with the **-c** option. When either **-c** or **-f** is specified, psql does not read commands from standard input; instead it terminates after processing all the **-c** and **-f** options in sequence. Except for that, this option is largely equivalent to the meta-command **\i**.If **filename** is **-** (hyphen), then standard input is read until an EOF indication or **\q** meta-command. This can be used to intersperse interactive input with input from files. Note however that Readline is not used in this case (much as if **-n** had been specified).Using this option is subtly different from writing **psql < `filename. In general, both will do what you expect, but using `-f** enables some nice features such as error messages with line numbers. There is also a slight chance that using this option will reduce the start-up overhead. On the other hand, the variant using the shell's input redirection is (in theory) guaranteed to yield exactly the same output you would have received had you entered everything by hand.

- `-F separator --field-separator=separator`

Use **separator** as the field separator for unaligned output. This is equivalent to **\pset fieldsep** or **\f**.

- `-h hostname --host=hostname`

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix-domain socket.

- `-H --html`

Switches to HTML output mode. This is equivalent to **\pset format html** or the **\H** command.

- `-l --list`

List all available databases, then exit. Other non-connection options are ignored. This is similar to the meta-command **\list**.When this option is used, psql will connect to the database **postgres**, unless a different database is named on the command line (option **-d** or non-option argument, possibly via a service entry, but not via an environment variable).

- `-L filename --log-file=filename`

Write all query output into file **filename**, in addition to the normal output destination.

- `-n --no-readline`

Do not use Readline for line editing and do not use the command history

- `-o filename --output=filename`

Put all query output into file **filename**. This is equivalent to the command **\o**.

- `-p port --port=port`

Specifies the TCP port or the local Unix-domain socket file extension on which the server is listening for connections. Defaults to the value of the `PGPORT` environment variable or, if not set, to the port specified at compile time, usually 5432.

- `-P assignment --pset=assignment`

Specifies printing options, in the style of `\pset`. Note that here you have to separate name and value with an equal sign instead of a space. For example, to set the output format to LaTeX, you could write `-P format=latex`.

- `-q --quiet`

Specifies that psql should do its work quietly. By default, it prints welcome messages and various informational output. If this option is used, none of this happens. This is useful with the `-c` option. This is equivalent to setting the variable `QUIET` to `on`.

- `-R separator --record-separator=separator`

Use `separator` as the record separator for unaligned output. This is equivalent to `\pset recordsep`.

- `-s --single-step`

Run in single-step mode. That means the user is prompted before each command is sent to the server, with the option to cancel execution as well. Use this to debug scripts.

- `-S --single-line`

Runs in single-line mode where a newline terminates an SQL command, as a semicolon does.NoteThis mode is provided for those who insist on it, but you are not necessarily encouraged to use it. In particular, if you mix SQL and meta-commands on a line the order of execution might not always be clear to the inexperienced user.

- `-t --tuples-only`

Turn off printing of column names and result row count footers, etc. This is equivalent to `\t` or `\pset tuples_only`.

- `-T table_options --table-attr=table_options`

Specifies options to be placed within the HTML `table` tag. See `\pset tableattr` for details.

- `-U \`username\` --username=\`username\``

Connect to the database as the user `username` instead of the default. (You must have permission to do so, of course.)

- `-v assignment --set=assignment --variable=assignment`

Perform a variable assignment, like the `\set` meta-command. Note that you must separate name and value, if any, by an equal sign on the command line. To unset a variable, leave off the equal sign. To set a variable with an empty value, use the equal sign but leave off the value. These assignments are done during command line processing, so variables that reflect connection state will get overwritten later.

- `-V --version`

Print the psql version and exit.

- `-w --no-password`

Never issue a password prompt. If the server requires password authentication and a password is not available from other sources such as a `.pgpass` file, the connection attempt will fail. This option can be

useful in batch jobs and scripts where no user is present to enter a password.Note that this option will remain set for the entire session, and so it affects uses of the meta-command `\connect` as well as the initial connection attempt.

- `-W --password`

Force psql to prompt for a password before connecting to a database, even if the password will not be used.If the server requires password authentication and a password is not available from other sources such as a `.pgpass` file, psql will prompt for a password in any case. However, psql will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing `-W` to avoid the extra connection attempt.Note that this option will remain set for the entire session, and so it affects uses of the meta-command `\connect` as well as the initial connection attempt.

- `-x --expanded`

Turn on the expanded table formatting mode. This is equivalent to `\x` or `\pset expanded`.

- `-X, --no-psqlrc`

Do not read the start-up file (neither the system-wide `psqlrc` file nor the user’ s `~/.psqlrc` file).

- `-z --field-separator-zero`

Set the field separator for unaligned output to a zero byte. This is equivalent to `\pset fieldsep_zero`.

- `-0 --record-separator-zero`

Set the record separator for unaligned output to a zero byte. This is useful for interfacing, for example, with `xargs -0`. This is equivalent to `\pset recordsep_zero`.

- `-1 --single-transaction`

This option can only be used in combination with one or more `-c` and/or `-f` options. It causes psql to issue a `BEGIN` command before the first such option and a `COMMIT` command after the last one, thereby wrapping all the commands into a single transaction. If any of the commands fails and the variable `ON_ERROR_STOP` was set, a `ROLLBACK` command is sent instead. This ensures that either all the commands complete successfully, or no changes are applied.If the commands themselves contain `BEGIN`, `COMMIT`, or `ROLLBACK`, this option will not have the desired effects. Also, if an individual command cannot be executed inside a transaction block, specifying this option will cause the whole transaction to fail.

- `-? --help[=`topic]`

Show help about psql and exit. The optional `topic` parameter (defaulting to `options`) selects which part of psql is explained: `commands` describes psql’ s backslash commands; `options` describes the command-line options that can be passed to psql; and `variables` shows help about psql configuration variables.

Exit Status

psql returns 0 to the shell if it finished normally, 1 if a fatal error of its own occurs (e.g., out of memory, file not found), 2 if the connection to the server went bad and the session was not interactive, and 3 if an error occurred in a script and the variable `ON_ERROR_STOP` was set.

reindexdb

reindexdb — reindex a IvorySQL database

Synopsis

`reindexdb` [`connection-option`⋯] [`option`⋯] [`-S` | `--schema schema`] ⋯ [`-t` | `--table table`] ⋯ [`-i` | `--index index`] ⋯ [`dbname`]

```
reindexdb` [*`connection-option`*...] [*`option`*...] `-a` | `--all
```

reindexdb [**connection-option**⋯] [**option**⋯] **-s** | **--system** [**dbname**]

Options

reindexdb accepts the following command-line arguments:

- **-a --all**

Reindex all databases.

- **--concurrently**

Use the **CONCURRENTLY** option. See REINDEX, where all the caveats of this option are explained in detail.

- **[-d] dbname dbname**

Specifies the name of the database to be reindexed, when **-a**/**--all** is not used. If this is not specified, the database name is read from the environment variable **PGDATABASE**. If that is not set, the user name specified for the connection is used. The **dbname** can be a connection string. If so, connection string parameters will override any conflicting command line options.

- **-e --echo**

Echo the commands that reindexdb generates and sends to the server.

- **-i index --index=index**

Recreate **index** only. Multiple indexes can be recreated by writing multiple **-i** switches.

- **-j njobs --jobs=njobs**

Execute the reindex commands in parallel by running **njobs** commands simultaneously. This option may reduce the processing time but it also increases the load on the database server.reindexdb will open **njobs** connections to the database, so make sure your max_connections setting is high enough to accommodate all connections.Note that this option is incompatible with the **--index** and **--system** options.

- **-q --quiet**

Do not display progress messages.

- **-s --system**

Reindex database’ s system catalogs only.

- **-S schema --schema=schema**

Reindex **schema** only. Multiple schemas can be reindexed by writing multiple **-S** switches.

- **-t table --table=table**

Reindex **table** only. Multiple tables can be reindexed by writing multiple **-t** switches.

- **--tablespace=tablespace**

Specifies the tablespace where indexes are rebuilt. (This name is processed as a double-quoted identifier.)

- **-v --verbose**

Print detailed information during processing.

- `-V --version`

Print the reindexdb version and exit.

- `-? --help`

Show help about reindexdb command line arguments, and exit.

reindexdb also accepts the following command-line arguments for connection parameters:

- `-h host --host=host`

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket.

- `-p port --port=port`

Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections.

- `-U username --username=username`

User name to connect as.

- `-w --no-password`

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

- `-W --password`

Force reindexdb to prompt for a password before connecting to a database.This option is never essential, since reindexdb will automatically prompt for a password if the server demands password authentication. However, reindexdb will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing `-W` to avoid the extra connection attempt.

- `--maintenance-db=dbname`

Specifies the name of the database to connect to to discover which databases should be reindexed, when `-a` / `--all` is used. If not specified, the `postgres` database will be used, or if that does not exist, `template1` will be used. This can be a connection string. If so, connection string parameters will override any conflicting command line options. Also, connection string parameters other than the database name itself will be re-used when connecting to other databases.

Environment

- `PGDATABASE PGHOST PGPORT PGUSER`

Default connection parameters

- `PG_COLOR`

Specifies whether to use color in diagnostic messages. Possible values are `always`, `auto` and `never`.

This utility, like most other IvorySQL utilities, also uses the environment variables supported by libpq .

Diagnostics

In case of difficulty, see REINDEX and psql for discussions of potential problems and error messages. The database server must be running at the targeted host. Also, any default connection settings and environment variables used by the libpq front-end library will apply.

Notes

reindexdb might need to connect several times to the IvorySQL server, asking for a password each time. It is convenient to have a `~/.pgpass` file in such cases.

Examples

To reindex the database `test`:

```
$ reindexdb test
```

To reindex the table `foo` and the index `bar` in a database named `abcd`:

```
$ reindexdb --table=foo --index=bar abcd
```

vacuumdb

vacuumdb — garbage-collect and analyze a IvorySQL database

Synopsis

`vacuumdb` [`connection-option`⋯] [`option`⋯] [`-t` | `--table` `table` [(`column` [,⋯])]] ⋯ [`dbname`]

```
vacuumdb` [*`connection-option`*...] [*`option`*...] `-a` | `--all
```

Options

vacuumdb accepts the following command-line arguments:

- `-a --all`

Vacuum all databases.

- `[-d] `dbname` `dbname``

Specifies the name of the database to be cleaned or analyzed, when `-a`/`--all` is not used. If this is not specified, the database name is read from the environment variable `PGDATABASE`. If that is not set, the user name specified for the connection is used. The `dbname` can be a connection string. If so, connection string parameters will override any conflicting command line options.

- `--disable-page-skipping`

Disable skipping pages based on the contents of the visibility map.

- `-e --echo`

Echo the commands that vacuumdb generates and sends to the server.

- `-f --full`

Perform “full” vacuuming.

- `-F --freeze`

Aggressively “freeze” tuples.

- `--force-index-cleanup`

Always remove index entries pointing to dead tuples.

- `-j njobs --jobs=njobs`

Execute the vacuum or analyze commands in parallel by running `njobs` commands simultaneously. This option may reduce the processing time but it also increases the load on the database server.vacuumdb will open `njobs` connections to the database, so make sure your max_connections setting is high enough to accommodate all connections.Note that using this mode together with the `-f` (`FULL`) option might cause deadlock failures if certain system catalogs are processed in parallel.

- `--min-mxid-age mxid_age`

Only execute the vacuum or analyze commands on tables with a multixact ID age of at least `mxid_age`. This setting is useful for prioritizing tables to process to prevent multixact ID wraparound .For the purposes of this option, the multixact ID age of a relation is the greatest of the ages of the main relation and its associated TOAST table, if one exists. Since the commands issued by vacuumdb will also process the TOAST table for the relation if necessary, it does not need to be considered separately.

- `--min-xid-age xid_age`

Only execute the vacuum or analyze commands on tables with a transaction ID age of at least `xid_age`. This setting is useful for prioritizing tables to process to prevent transaction ID wraparound.For the purposes of this option, the transaction ID age of a relation is the greatest of the ages of the main relation and its associated TOAST table, if one exists. Since the commands issued by vacuumdb will also process the TOAST table for the relation if necessary, it does not need to be considered separately.

- `--no-index-cleanup`

Do not remove index entries pointing to dead tuples.

- `--no-process-toast`

Skip the TOAST table associated with the table to vacuum, if any.

- `--no-truncate`

Do not truncate empty pages at the end of the table.

- `-P parallel_workers --parallel=parallel_workers`

Specify the number of parallel workers for parallel vacuum. This allows the vacuum to leverage multiple CPUs to process indexes. See VACUUM.

- `-q --quiet`

Do not display progress messages.

- `--skip-locked`

Skip relations that cannot be immediately locked for processing

- `-t` `` `table` [ (`column` [,…]) ] `` `--table=`table` [ (column [,…]) ]` ``

Clean or analyze `table` only. Column names can be specified only in conjunction with the `--analyze` or

`--analyze-only` options. Multiple tables can be vacuumed by writing multiple `-t` switches.TipIf you specify columns, you probably have to escape the parentheses from the shell. (See examples below.)

- `-v --verbose`

Print detailed information during processing.

- `-V --version`

Print the vacuumdb version and exit.

- `-z --analyze`

Also calculate statistics for use by the optimizer.

- `-Z --analyze-only`

Only calculate statistics for use by the optimizer (no vacuum).

- `--analyze-in-stages`

Only calculate statistics for use by the optimizer (no vacuum), like `--analyze-only`. Run three stages of analyze; the first stage uses the lowest possible statistics target and subsequent stages build the full statistics.This option is only useful to analyze a database that currently has no statistics or has wholly incorrect ones, such as if it is newly populated from a restored dump or by `pg_upgrade`. Be aware that running with this option in a database with existing statistics may cause the query optimizer choices to become transiently worse due to the low statistics targets of the early stages.

- `-? --help`

Show help about vacuumdb command line arguments, and exit.

vacuumdb also accepts the following command-line arguments for connection parameters:

- `-h `host` --host=`host``

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket.

- `-p `port` --port=`port``

Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections.

- `-U `username` --username=`username``

User name to connect as.

- `-w --no-password`

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

- `-W --password`

Force vacuumdb to prompt for a password before connecting to a database.This option is never essential, since vacuumdb will automatically prompt for a password if the server demands password authentication. However, vacuumdb will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing `-W` to avoid the extra connection attempt.

- `--maintenance-db=dbname`

Specifies the name of the database to connect to to discover which databases should be vacuumed, when `-a` / `--all` is used. If not specified, the `postgres` database will be used, or if that does not exist, `template1` will be used. This can be a connection string. If so, connection string parameters will override any conflicting command line options. Also, connection string parameters other than the database name itself will be re-used when connecting to other databases.

Environment

- `PGDATABASE PGHOST PGPORT PGUSER`

Default connection parameters

- `PG_COLOR`

Specifies whether to use color in diagnostic messages. Possible values are `always`, `auto` and `never`.

This utility, like most other IvorySQL utilities, also uses the environment variables supported by libpq.

Diagnostics

In case of difficulty, see VACUUM and psql for discussions of potential problems and error messages. The database server must be running at the targeted host. Also, any default connection settings and environment variables used by the libpq front-end library will apply.

Notes

vacuumdb might need to connect several times to the IvorySQL server, asking for a password each time. It is convenient to have a `~/.pgpass` file in such cases.

Examples

To clean the database `test`:

```
$ vacuumdb test
```

To clean and analyze for the optimizer a database named `bigdb`:

```
$ vacuumdb --analyze bigdb
```

To clean a single table `foo` in a database named `xyzzy`, and analyze a single column `bar` of the table for the optimizer:

```
$ vacuumdb --analyze --verbose --table='foo(bar)' xyzzy
```

Server Applications

initdb

initdb — create a new IvorySQL database cluster

Synopsis

`initdb` [`option`⋯] [`--pgdata` | `-D`] `directory`

Options

- `-A authmethod --auth=authmethod`

This option specifies the default authentication method for local users used in `pg_hba.conf` (`host` and `local` lines). Do not use `trust` unless you trust all local users on your system. `trust` is the default for ease of installation.

- `--auth-host=`authmethod`

This option specifies the authentication method for local users via TCP/IP connections used in `pg_hba.conf` (`host` lines).

- `--auth-local=`authmethod`

This option specifies the authentication method for local users via Unix-domain socket connections used in `pg_hba.conf` (`local` lines).

- `-D directory --pgdata=directory`

This option specifies the directory where the database cluster should be stored. This is the only information required by `initdb`, but you can avoid writing it by setting the `PGDATA` environment variable, which can be convenient since the database server (`postgres`) can find the database directory later by the same variable.

- `-E `encoding` --encoding=`encoding``

Selects the encoding of the template databases. This will also be the default encoding of any database you create later, unless you override it then. The default is derived from the locale, if the libc locale provider is used, or `UTF8` if the ICU locale provider is used.

- `-g --allow-group-access`

Allows users in the same group as the cluster owner to read all cluster files created by `initdb`. This option is ignored on Windows as it does not support POSIX-style group permissions.

- `--icu-locale=locale`

Specifies the ICU locale ID, if the ICU locale provider is used.

- `-k --data-checksums`

Use checksums on data pages to help detect corruption by the I/O system that would otherwise be silent. Enabling checksums may incur a noticeable performance penalty. If set, checksums are calculated for all objects, in all databases. All checksum failures will be reported in the `pg_stat_database` view.

- `--locale=locale`

Sets the default locale for the database cluster. If this option is not specified, the locale is inherited from the environment that `initdb` runs in.

- `--lc-collate=locale --lc-ctype=locale --lc-messages=locale --lc-monetary=locale --lc-numeric=locale --lc-time=locale`

Like `--locale`, but only sets the locale in the specified category.

- `--no-locale`

Equivalent to `--locale=C`.

- `--locale-provider={libc|icu}`

This option sets the locale provider for databases created in the new cluster. It can be overridden in the `CREATE DATABASE` command when new databases are subsequently created. The default is `libc`.

- `-N --no-sync`

By default, `initdb` will wait for all files to be written safely to disk. This option causes `initdb` to return without waiting, which is faster, but means that a subsequent operating system crash can leave the data directory corrupt. Generally, this option is useful for testing, but should not be used when creating a production installation.

- `--no-instructions`

By default, `initdb` will write instructions for how to start the cluster at the end of its output. This option causes those instructions to be left out. This is primarily intended for use by tools that wrap `initdb` in platform-specific behavior, where those instructions are likely to be incorrect.

- `--pwfile=filename`

Makes `initdb` read the database superuser's password from a file. The first line of the file is taken as the password.

- `-S --sync-only`

Safely write all database files to disk and exit. This does not perform any of the normal initdb operations.

- `-T config --text-search-config=config`

Sets the default text search configuration.

- `-U username --username=username`

Selects the user name of the database superuser. This defaults to the name of the effective user running `initdb`. It is really not important what the superuser's name is, but one might choose to keep the customary name postgres, even if the operating system user's name is different.

- `-W --pwprompt`

Makes `initdb` prompt for a password to give the database superuser. If you don't plan on using password authentication, this is not important. Otherwise you won't be able to use password authentication until you have a password set up.

- `-X directory --waldir=directory`

This option specifies the directory where the write-ahead log should be stored.

- `--wal-segsize=size`

Set the WAL segment size, in megabytes. This is the size of each individual file in the WAL log. The default size is 16 megabytes. The value must be a power of 2 between 1 and 1024 (megabytes). This option can only be set during initialization, and cannot be changed later.It may be useful to adjust this size to control the granularity of WAL log shipping or archiving. Also, in databases with a high volume of WAL, the sheer number of WAL files per directory can become a performance and management problem. Increasing the WAL file size will reduce the number of WAL files.

Other, less commonly used, options are also available:

- `-d --debug`

Print debugging output from the bootstrap backend and a few other messages of lesser interest for the general public. The bootstrap backend is the program `initdb` uses to create the catalog tables. This option generates a tremendous amount of extremely boring output.

- `--discard-caches`

Run the bootstrap backend with the `debug_discard_caches=1` option. This takes a very long time and is only of use for deep debugging.

- `-L directory`

Specifies where `initdb` should find its input files to initialize the database cluster. This is normally not necessary. You will be told if you need to specify their location explicitly.

- `-n --no-clean`

By default, when `initdb` determines that an error prevented it from completely creating the database cluster, it removes any files it might have created before discovering that it cannot finish the job. This option inhibits tidying-up and is thus useful for debugging.

Other options:

- `-V --version`

Print the initdb version and exit.

- `-? --help`

Show help about initdb command line arguments, and exit.

Environment

- `PGDATA`

Specifies the directory where the database cluster is to be stored; can be overridden using the `-D` option.

- `PG_COLOR`

Specifies whether to use color in diagnostic messages. Possible values are `always`, `auto` and `never`.

- `TZ`

Specifies the default time zone of the created database cluster. The value should be a full time zone name

This utility, like most other IvorySQL utilities, also uses the environment variables supported by libpq

Notes

`initdb` can also be invoked via `pg_ctl initdb`.

pg_archivecleanup

pg_archivecleanup — clean up IvorySQL WAL archive files

Synopsis

`pg_archivecleanup` [`option`⋯] `archivelocation oldestkeptwalfile`

Options

pg_archivecleanup accepts the following command-line arguments:

- `-d`

Print lots of debug logging output on `stderr`.

- `-n`

Print the names of the files that would have been removed on `stdout` (performs a dry run).

- `-V --version`

Print the pg_archivecleanup version and exit.

- `-x extension`

Provide an extension that will be stripped from all file names before deciding if they should be deleted. This is typically useful for cleaning up archives that have been compressed during storage, and therefore have had an extension added by the compression program. For example: `-x .gz`.

- `-? --help`

Show help about pg_archivecleanup command line arguments, and exit.

Environment

The environment variable `PG_COLOR` specifies whether to use color in diagnostic messages. Possible values are `always`, `auto` and `never`.

Examples

On Linux or Unix systems, you might use:

```
archive_cleanup_command = 'pg_archivecleanup -d /mnt/standby/archive %r
2>>cleanup.log'
```

where the archive directory is physically located on the standby server, so that the `archive_command` is accessing it across NFS, but the files are local to the standby. This will:

- produce debugging output in `cleanup.log`
- remove no-longer-needed files from the archive directory

pg_checksums

pg_checksums — enable, disable or check data checksums in a IvorySQL database cluster

Synopsis

`pg_checksums` [`option`⋯] [[`-D` | `--pgdata`] `datadir`]

Options

The following command-line options are available:

- `-D directory --pgdata=directory`

Specifies the directory where the database cluster is stored.

- `-c --check`

Checks checksums. This is the default mode if nothing else is specified.

- `-d --disable`

Disables checksums.

- `-e --enable`

Enables checksums.

- `-f filenode --filenode=filenode`

Only validate checksums in the relation with filenode `filenode`.

- `-N --no-sync`

By default, `pg_checksums` will wait for all files to be written safely to disk. This option causes `pg_checksums` to return without waiting, which is faster, but means that a subsequent operating system crash can leave the updated data directory corrupt. Generally, this option is useful for testing but should not be used on a production installation. This option has no effect when using `--check`.

- `-P --progress`

Enable progress reporting. Turning this on will deliver a progress report while checking or enabling checksums.

- `-v --verbose`

Enable verbose output. Lists all checked files.

- `-V --version`

Print the pg_checksums version and exit.

- `-? --help`

Show help about pg_checksums command line arguments, and exit.

Environment

- `PGDATA`

Specifies the directory where the database cluster is stored; can be overridden using the `-D` option.

- `PG_COLOR`

Specifies whether to use color in diagnostic messages. Possible values are `always`, `auto` and `never`.

Notes

Enabling checksums in a large cluster can potentially take a long time. During this operation, the cluster or other programs that write to the data directory must not be started or else data loss may occur.

When using a replication setup with tools which perform direct copies of relation file blocks (for example pg_rewind), enabling or disabling checksums can lead to page corruptions in the shape of incorrect checksums if the operation is not done consistently across all nodes. When enabling or disabling checksums in a replication setup, it is thus recommended to stop all the clusters before switching them all consistently. Destroying all standbys, performing the operation on the primary and finally recreating the standbys from scratch is also safe.

If pg_checksums is aborted or killed while enabling or disabling checksums, the cluster' s data checksum configuration remains unchanged, and pg_checksums can be re-run to perform the same operation.

pg_controldata

pg_controldata — display control information of a IvorySQL database cluster

Synopsis

pg_controldata [**option**] [[**-D** | **--pgdata**] **datadir**]

Environment

- **PGDATA**

Default data directory location

- **PG_COLOR**

Specifies whether to use color in diagnostic messages. Possible values are **always**, **auto** and **never**.

pg_ctl

pg_ctl — initialize, start, stop, or control a IvorySQL server

Synopsis

pg_ctl init[db] [**-D datadir**] [**-s**] [**-o initdb-options**]

pg_ctl start [**-D datadir**] [**-l filename**] [**-W**] [**-t seconds**] [**-s**] [**-o options**] [**-p path**] [**-c**]

pg_ctl stop [**-D datadir**] [**-m s[mart]** | **f[ast]** | **i[mmediate]**] [**-W**] [**-t seconds**] [**-s**]

pg_ctl restart [**-D datadir**] [**-m s[mart]** | **f[ast]** | **i[mmediate]**] [**-W**] [**-t seconds**] [**-s**] [**-o options**] [**-c**]

pg_ctl reload [**-D datadir**] [**-s**]

pg_ctl status [**-D datadir**]

pg_ctl promote [**-D datadir**] [**-W**] [**-t seconds**] [**-s**]

pg_ctl logrotate [**-D datadir**] [**-s**]

pg_ctl kill signal_name process_id

On Microsoft Windows, also:

pg_ctl register [**-D datadir**] [**-N servicename**] [**-U username**] [**-P password**] [**-S a[uto]** | **d[emand]**] [**-e source**] [**-W**] [**-t seconds**] [**-s**] [**-o options**]

pg_ctl unregister [**-N servicename**]

Options

- **-c --core-files**

Attempt to allow server crashes to produce core files, on platforms where this is possible, by lifting any soft resource limit placed on core files. This is useful in debugging or diagnosing problems by allowing a stack trace to be obtained from a failed server process.

- **-D datadir --pgdata=datadir**

Specifies the file system location of the database configuration files. If this option is omitted, the environment variable **PGDATA** is used.

- **-l filename --log=filename**

Append the server log output to **filename**. If the file does not exist, it is created. The umask is set to 077, so

access to the log file is disallowed to other users by default.

- `-m mode --mode=mode`

Specifies the shutdown mode. `mode` can be `smart`, `fast`, or `immediate`, or the first letter of one of these three. If this option is omitted, `fast` is the default.

- `-o options --options=options`

Specifies options to be passed directly to the `postgres` command. `-o` can be specified multiple times, with all the given options being passed through.The `options` should usually be surrounded by single or double quotes to ensure that they are passed through as a group.

- `-o initdb-options --options=initdb-options`

Specifies options to be passed directly to the `initdb` command. `-o` can be specified multiple times, with all the given options being passed through.The `initdb-options` should usually be surrounded by single or double quotes to ensure that they are passed through as a group.

- `-p path`

Specifies the location of the `postgres` executable. By default the `postgres` executable is taken from the same directory as `pg_ctl`, or failing that, the hard-wired installation directory. It is not necessary to use this option unless you are doing something unusual and get errors that the `postgres` executable was not found.In `init` mode, this option analogously specifies the location of the `initdb` executable.

- `-s --silent`

Print only errors, no informational messages.

- `-t seconds --timeout=seconds`

Specifies the maximum number of seconds to wait when waiting for an operation to complete (see option `-w`). Defaults to the value of the `PGCTLTIMEOUT` environment variable or, if not set, to 60 seconds.

- `-V --version`

Print the pg_ctl version and exit.

- `-w --wait`

Wait for the operation to complete. This is supported for the modes `start`, `stop`, `restart`, `promote`, and `register`, and is the default for those modes.When waiting, `pg_ctl` repeatedly checks the server’ s PID file, sleeping for a short amount of time between checks. Startup is considered complete when the PID file indicates that the server is ready to accept connections. Shutdown is considered complete when the server removes the PID file. `pg_ctl` returns an exit code based on the success of the startup or shutdown.If the operation does not complete within the timeout (see option `-t`), then `pg_ctl` exits with a nonzero exit status. But note that the operation might continue in the background and eventually succeed.

- `-W --no-wait`

Do not wait for the operation to complete. This is the opposite of the option `-w`.If waiting is disabled, the requested action is triggered, but there is no feedback about its success. In that case, the server log file or an external monitoring system would have to be used to check the progress and success of the operation.In prior releases of IvorySQL, this was the default except for the `stop` mode.

- `-? --help`

Show help about pg_ctl command line arguments, and exit.

If an option is specified that is valid, but not relevant to the selected operating mode, pg_ctl ignores it.

Options for Windows

- `-e source`

Name of the event source for pg_ctl to use for logging to the event log when running as a Windows service. The default is `IvorySQL`. Note that this only controls messages sent from pg_ctl itself; once started, the server will use the event source specified by its event_source parameter. Should the server fail very early in startup, before that parameter has been set, it might also log using the default event source name `IvorySQL`.

- `-N \`servicename\``

Name of the system service to register. This name will be used as both the service name and the display name. The default is `IvorySQL`.

- `-P \`password\``

Password for the user to run the service as.

- `-S start-type`

Start type of the system service. `start-type` can be `auto`, or `demand`, or the first letter of one of these two. If this option is omitted, `auto` is the default.

- `-U username`

User name for the user to run the service as. For domain users, use the format `DOMAIN\username`.

Environment

- `PGCTLTIMEOUT`

Default limit on the number of seconds to wait when waiting for startup or shutdown to complete. If not set, the default is 60 seconds.

- `PGDATA`

Default data directory location.

Most `pg_ctl` modes require knowing the data directory location; therefore, the `-D` option is required unless `PGDATA` is set.

`pg_ctl`, like most other IvorySQL utilities, also uses the environment variables supported by libpq.

Files

- `postmaster.pid`

pg_ctl examines this file in the data directory to determine whether the server is currently running.

- `postmaster.opts`

If this file exists in the data directory, pg_ctl (in `restart` mode) will pass the contents of the file as options to postgres, unless overridden by the `-o` option. The contents of this file are also displayed in `status` mode.

Examples

Starting the Server

To start the server, waiting until the server is accepting connections:

```
$ pg_ctl start
```

To start the server using port 5433, and running without `fsync`, use:

```
$ pg_ctl -o "-F -p 5433" start
```

Stopping the Server

To stop the server, use:

```
$ pg_ctl stop
```

The `-m` option allows control over how the server shuts down:

```
$ pg_ctl stop -m smart
```

Restarting the Server

Restarting the server is almost equivalent to stopping the server and starting it again, except that by default, `pg_ctl` saves and reuses the command line options that were passed to the previously-running instance. To restart the server using the same options as before, use:

```
$ pg_ctl restart
```

But if `-o` is specified, that replaces any previous options. To restart using port 5433, disabling `fsync` upon restart:

```
$ pg_ctl -o "-F -p 5433" restart
```

Showing the Server Status

Here is sample status output from pg_ctl:

```
$ pg_ctl status

pg_ctl: server is running (PID: 13718)
/usr/local/pgsql/bin/postgres "-D" "/usr/local/pgsql/data" "-p" "5433" "-B" "128"
```

The second line is the command that would be invoked in restart mode.

pg_resetwal

pg_resetwal — reset the write-ahead log and other control information of a IvorySQL database cluster

Synopsis

`pg_resetwal` [`-f` | `--force`] [`-n` | `--dry-run`] [`option`⋯] [`-D` | `--pgdata`] `datadir`

Options

- `-f --force`

Force `pg_resetwal` to proceed even if it cannot determine valid data for `pg_control`, as explained above.

- `-n --dry-run`

The `-n` / `--dry-run` option instructs `pg_resetwal` to print the values reconstructed from `pg_control` and values about to be changed, and then exit without modifying anything. This is mainly a debugging tool, but can be useful as a sanity check before allowing `pg_resetwal` to proceed for real.

- `-V --version`

Display version information, then exit.

- `-? --help`

Show help, then exit.

The following options are only needed when `pg_resetwal` is unable to determine appropriate values by reading `pg_control`. Safe values can be determined as described below. For values that take numeric arguments, hexadecimal values can be specified by using the prefix `0x`.

- `-c xid,xid --commit-timestamp-ids=xid,xid`

Manually set the oldest and newest transaction IDs for which the commit time can be retrieved.A safe value for the oldest transaction ID for which the commit time can be retrieved (first part) can be determined by looking for the numerically smallest file name in the directory `pg_commit_ts` under the data directory. Conversely, a safe value for the newest transaction ID for which the commit time can be retrieved (second part) can be determined by looking for the numerically greatest file name in the same directory. The file names are in hexadecimal.

- `-e xid_epoch --epoch=xid_epoch`

Manually set the next transaction ID’ s epoch.The transaction ID epoch is not actually stored anywhere in the database except in the field that is set by `pg_resetwal`, so any value will work so far as the database itself is concerned. You might need to adjust this value to ensure that replication systems such as Slony-I and Skytools work correctly — if so, an appropriate value should be obtainable from the state of the downstream replicated database.

- `-l walfile --next-wal-file=walfile`

Manually set the WAL starting location by specifying the name of the next WAL segment file.The name of next WAL segment file should be larger than any WAL segment file name currently existing in the directory `pg_wal` under the data directory. These names are also in hexadecimal and have three parts. The first part is the “timeline ID” and should usually be kept the same. For example, if `000000010000000320000004A` is the largest entry in `pg_wal`, use `-l 000000010000000320000004B` or higher.Note that when using nondefault WAL segment sizes, the numbers in the WAL file names are different from the LSNs that are reported by system functions and system views. This option takes a WAL file name, not an LSN.Note `pg_resetwal` itself looks at the files in `pg_wal` and chooses a default `-l` setting beyond the last existing file name. Therefore, manual adjustment of `-l` should only be needed if you are aware of WAL segment files that are not currently present in `pg_wal`, such as entries in an offline archive; or if the contents of `pg_wal` have been lost entirely.

- `-m mxid,mxid --multixact-ids=mxid,mxid`

Manually set the next and oldest multitransaction ID.A safe value for the next multitransaction ID (first part) can be determined by looking for the numerically largest file name in the directory `pg_multixact/offsets`

under the data directory, adding one, and then multiplying by 65536 (0x10000). Conversely, a safe value for the oldest multitransaction ID (second part of `-m`) can be determined by looking for the numerically smallest file name in the same directory and multiplying by 65536. The file names are in hexadecimal, so the easiest way to do this is to specify the option value in hexadecimal and append four zeroes.

- `-o oid --next-oid=oid`

Manually set the next OID.There is no comparably easy way to determine a next OID that’ s beyond the largest one in the database, but fortunately it is not critical to get the next-OID setting right.

- `-O mxoff --multixact-offset=mxoff`

Manually set the next multitransaction offset.A safe value can be determined by looking for the numerically largest file name in the directory `pg_multixact/members` under the data directory, adding one, and then multiplying by 52352 (0xCC80). The file names are in hexadecimal. There is no simple recipe such as the ones for other options of appending zeroes.

- `--wal-segsize=wal_segment_size`

Set the new WAL segment size, in megabytes. The value must be set to a power of 2 between 1 and 1024 (megabytes). See the same option of initdb for more information.NoteWhile `pg_resetwal` will set the WAL starting address beyond the latest existing WAL segment file, some segment size changes can cause previous WAL file names to be reused. It is recommended to use `-l` together with this option to manually set the WAL starting address if WAL file name overlap will cause problems with your archiving strategy.

- `-u xid --oldest-transaction-id=xid`

Manually set the oldest unfrozen transaction ID.A safe value can be determined by looking for the numerically smallest file name in the directory `pg_xact` under the data directory and then multiplying by 1048576 (0x100000). Note that the file names are in hexadecimal. It is usually easiest to specify the option value in hexadecimal too. For example, if `0007` is the smallest entry in `pg_xact`, `-u 0x700000` will work (five trailing zeroes provide the proper multiplier).

- `-x xid --next-transaction-id=xid`

Manually set the next transaction ID.A safe value can be determined by looking for the numerically largest file name in the directory `pg_xact` under the data directory, adding one, and then multiplying by 1048576 (0x100000). Note that the file names are in hexadecimal. It is usually easiest to specify the option value in hexadecimal too. For example, if `0011` is the largest entry in `pg_xact`, `-x 0x1200000` will work (five trailing zeroes provide the proper multiplier).

Environment

- `PG_COLOR`

Specifies whether to use color in diagnostic messages. Possible values are `always`, `auto` and `never`.

Notes

This command must not be used when the server is running. `pg_resetwal` will refuse to start up if it finds a server lock file in the data directory. If the server crashed then a lock file might have been left behind; in that case you can remove the lock file to allow `pg_resetwal` to run. But before you do so, make doubly certain that there is no server process still alive.

`pg_resetwal` works only with servers of the same major version.

pg_rewind

pg_rewind — synchronize a IvorySQL data directory with another data directory that was forked from it

Synopsis

```
pg_rewind [option…] { -D | --target-pgdata } directory { --source-pgdata=`directory` | --source-server=`connstr` }
```

Warning

If pg_rewind fails while processing, then the data folder of the target is likely not in a state that can be recovered. In such a case, taking a new fresh backup is recommended.

As pg_rewind copies configuration files entirely from the source, it may be required to correct the configuration used for recovery before restarting the target server, especially if the target is reintroduced as a standby of the source. If you restart the server after the rewind operation has finished but without configuring recovery, the target may again diverge from the primary.

pg_rewind will fail immediately if it finds files it cannot write directly to. This can happen for example when the source and the target server use the same file mapping for read-only SSL keys and certificates. If such files are present on the target server it is recommended to remove them before running pg_rewind. After doing the rewind, some of those files may have been copied from the source, in which case it may be necessary to remove the data copied and restore back the set of links used before the rewind.

Options

pg_rewind accepts the following command-line arguments:

- `-D directory --target-pgdata=directory`

This option specifies the target data directory that is synchronized with the source. The target server must be shut down cleanly before running pg_rewind

- `--source-pgdata=directory`

Specifies the file system path to the data directory of the source server to synchronize the target with. This option requires the source server to be cleanly shut down.

- `--source-server=connstr`

Specifies a libpq connection string to connect to the source IvorySQL server to synchronize the target with. The connection must be a normal (non-replication) connection with a role having sufficient permissions to execute the functions used by pg_rewind on the source server (see Notes section for details) or a superuser role. This option requires the source server to be running and accepting connections.

- `-R --write-recovery-conf`

Create `standby.signal` and append connection settings to `IvorySQL.auto.conf` in the output directory. `--source-server` is mandatory with this option.

- `-n --dry-run`

Do everything except actually modifying the target directory.

- `-N --no-sync`

By default, `pg_rewind` will wait for all files to be written safely to disk. This option causes `pg_rewind` to return without waiting, which is faster, but means that a subsequent operating system crash can leave the data directory corrupt. Generally, this option is useful for testing but should not be used on a production installation.

- `-P --progress`

Enables progress reporting. Turning this on will deliver an approximate progress report while copying data from the source cluster.

- `-c --restore-target-wal`

Use `restore_command` defined in the target cluster configuration to retrieve WAL files from the WAL archive if these files are no longer available in the `pg_wal` directory.

- `--config-file=filename`

Use the specified main server configuration file for the target cluster. This affects pg_rewind when it uses internally the postgres command for the rewind operation on this cluster (when retrieving `restore_command` with the option `-c/--restore-target-wal` and when forcing a completion of crash recovery).

- `--debug`

Print verbose debugging output that is mostly useful for developers debugging pg_rewind.

- `--no-ensure-shutdown`

pg_rewind requires that the target server is cleanly shut down before rewinding. By default, if the target server is not shut down cleanly, pg_rewind starts the target server in single-user mode to complete crash recovery first, and stops it. By passing this option, pg_rewind skips this and errors out immediately if the server is not cleanly shut down. Users are expected to handle the situation themselves in that case.

- `-V --version`

Display version information, then exit.

- `-? --help`

Show help, then exit.

Environment

When `--source-server` option is used, pg_rewind also uses the environment variables supported by libpq .

The environment variable `PG_COLOR` specifies whether to use color in diagnostic messages. Possible values are `always`, `auto` and `never`.

Notes

When executing pg_rewind using an online cluster as source, a role having sufficient permissions to execute the functions used by pg_rewind on the source cluster can be used instead of a superuser. Here is how to create such a role, named `rewind_user` here:

```
CREATE USER rewind_user LOGIN;
GRANT EXECUTE ON function pg_catalog.pg_ls_dir(text, boolean, boolean) TO rewind_user;
GRANT EXECUTE ON function pg_catalog.pg_stat_file(text, boolean) TO rewind_user;
GRANT EXECUTE ON function pg_catalog.pg_read_binary_file(text) TO rewind_user;
GRANT EXECUTE ON function pg_catalog.pg_read_binary_file(text, bigint, bigint,
boolean) TO rewind_user;
```

When executing pg_rewind using an online cluster as source which has been recently promoted, it is necessary to execute a `CHECKPOINT` after promotion such that its control file reflects up-to-date timeline information, which is used by pg_rewind to check if the target cluster can be rewound using the designated source cluster.

How It Works

The basic idea is to copy all file system-level changes from the source cluster to the target cluster:

1.Scan the WAL log of the target cluster, starting from the last checkpoint before the point where the source cluster’ s timeline history forked off from the target cluster. For each WAL record, record each data block that was touched. This yields a list of all the data blocks that were changed in the target cluster, after the source cluster forked off. If some of the WAL files are no longer available, try re-running pg_rewind with the `-c` option to search for the missing files in the WAL archive. 2.Copy all those changed blocks from the source cluster to the target cluster, either using direct file system access (`--source-pgdata`) or SQL (`--source -server`). Relation files are now in a state equivalent to the moment of the last completed checkpoint prior to the point at which the WAL timelines of the source and target diverged plus the current state on the source of any blocks changed on the target after that divergence. 3.Copy all other files, including new relation files, WAL segments, `pg_xact`, and configuration files from the source cluster to the target cluster. Similarly to base backups, the contents of the directories `pg_dynshmem/`, `pg_notify/`, `pg_replslot/`, `pg_serial/`, `pg_snapshots/`, `pg_stat_tmp/`, and `pg_subtrans/` are omitted from the data copied from the source cluster. The files `backup_label`, `tablespace_map`, `pg_internal.init`, `postmaster.opts`, and `postmaster.pid`, as well as any file or directory beginning with `pgsql_tmp`, are omitted. 4.Create a `backup_label` file to begin WAL replay at the checkpoint created at failover and configure the `pg_control` file with a minimum consistency LSN defined as the result of `pg_current_wal_insert_lsn()` when rewinding from a live source or the last checkpoint LSN when rewinding from a stopped source. 5.When starting the target, IvorySQL replays all the required WAL, resulting in a data directory in a consistent state.

pg_test_fsync

pg_test_fsync — determine fastest `wal_sync_method` for IvorySQL

Synopsis

`pg_test_fsync` [`option`⋯]

Options

pg_test_fsync accepts the following command-line options:

- `-f --filename`

Specifies the file name to write test data in. This file should be in the same file system that the `pg_wal` directory is or will be placed in. (`pg_wal` contains the WAL files.) The default is `pg_test_fsync.out` in the current directory.

- `-s --secs-per-test`

Specifies the number of seconds for each test. The more time per test, the greater the test’ s accuracy, but the longer it takes to run. The default is 5 seconds, which allows the program to complete in under 2 minutes.

- `-V --version`

Print the pg_test_fsync version and exit.

- `-? --help`

Show help about pg_test_fsync command line arguments, and exit.

Environment

The environment variable `PG_COLOR` specifies whether to use color in diagnostic messages. Possible values are `always`, `auto` and `never`.

pg_test_timing

pg_test_timing — measure timing overhead

Synopsis

`pg_test_timing` [`option`⋯]

Options

pg_test_timing accepts the following command-line options:

- `-d \`duration\` --duration=\`duration\``

Specifies the test duration, in seconds. Longer durations give slightly better accuracy, and are more likely to discover problems with the system clock moving backwards. The default test duration is 3 seconds.

- `-V --version`

Print the pg_test_timing version and exit.

- `-? --help`

Show help about pg_test_timing command line arguments, and exit.

Usage

Interpreting Results

Good results will show most (>90%) individual timing calls take less than one microsecond. Average per loop overhead will be even lower, below 100 nanoseconds. This example from an Intel i7-860 system using a TSC clock source shows excellent performance:

```
Testing timing overhead for 3 seconds.
Per loop time including overhead: 35.96 ns
Histogram of timing durations:
  < us   % of total      count
     1     96.40465   80435604
     2      3.59518    2999652
     4      0.00015        126
     8      0.00002         13
    16      0.00000          2
```

Note that different units are used for the per loop time than the histogram. The loop can have resolution within a few nanoseconds (ns), while the individual timing calls can only resolve down to one microsecond (us).

Measuring Executor Timing Overhead

When the query executor is running a statement using `EXPLAIN ANALYZE`, individual operations are timed as well as showing a summary. The overhead of your system can be checked by counting rows with the psql program:

```
CREATE TABLE t AS SELECT * FROM generate_series(1,100000);
```

```
\timing
SELECT COUNT(*) FROM t;
EXPLAIN ANALYZE SELECT COUNT(*) FROM t;
```

The i7-860 system measured runs the count query in 9.8 ms while the `EXPLAIN ANALYZE` version takes 16.6 ms, each processing just over 100,000 rows. That 6.8 ms difference means the timing overhead per row is 68 ns, about twice what pg_test_timing estimated it would be. Even that relatively small amount of overhead is making the fully timed count statement take almost 70% longer. On more substantial queries, the timing overhead would be less problematic.

Changing Time Sources

On some newer Linux systems, it's possible to change the clock source used to collect timing data at any time. A second example shows the slowdown possible from switching to the slower acpi_pm time source, on the same system used for the fast results above:

```
# cat /sys/devices/system/clocksource/clocksource0/available_clocksource
tsc hpet acpi_pm
# echo acpi_pm > /sys/devices/system/clocksource/clocksource0/current_clocksource
# pg_test_timing
Per loop time including overhead: 722.92 ns
Histogram of timing durations:
  < us   % of total      count
     1     27.84870    1155682
     2     72.05956    2990371
     4      0.07810       3241
     8      0.01357        563
    16      0.00007          3
```

In this configuration, the sample `EXPLAIN ANALYZE` above takes 115.9 ms. That's 1061 ns of timing overhead, again a small multiple of what's measured directly by this utility. That much timing overhead means the actual query itself is only taking a tiny fraction of the accounted for time, most of it is being consumed in overhead instead. In this configuration, any `EXPLAIN ANALYZE` totals involving many timed operations would be inflated significantly by timing overhead.

FreeBSD also allows changing the time source on the fly, and it logs information about the timer selected during boot:

```
# dmesg | grep "Timecounter"
Timecounter "ACPI-fast" frequency 3579545 Hz quality 900
Timecounter "i8254" frequency 1193182 Hz quality 0
Timecounters tick every 10.000 msec
Timecounter "TSC" frequency 2531787134 Hz quality 800
# sysctl kern.timecounter.hardware=TSC
kern.timecounter.hardware: ACPI-fast -> TSC
```

Other systems may only allow setting the time source on boot. On older Linux systems the "clock" kernel setting is the only way to make this sort of change. And even on some more recent ones, the only option you'll see for a clock source is "jiffies". Jiffies are the older Linux software clock implementation, which can

have good resolution when it's backed by fast enough timing hardware, as in this example:

```
$ cat /sys/devices/system/clocksource/clocksource0/available_clocksource
jiffies
$ dmesg | grep time.c
time.c: Using 3.579545 MHz WALL PM GTOD PIT/TSC timer.
time.c: Detected 2400.153 MHz processor.
$ pg_test_timing
Testing timing overhead for 3 seconds.
Per timing duration including loop overhead: 97.75 ns
Histogram of timing durations:
  < us   % of total      count
     1     90.23734   27694571
     2      9.75277    2993204
     4      0.00981       3010
     8      0.00007         22
    16      0.00000          1
    32      0.00000          1
```

Clock Hardware and Timing Accuracy

Collecting accurate timing information is normally done on computers using hardware clocks with various levels of accuracy. With some hardware the operating systems can pass the system clock time almost directly to programs. A system clock can also be derived from a chip that simply provides timing interrupts, periodic ticks at some known time interval. In either case, operating system kernels provide a clock source that hides these details. But the accuracy of that clock source and how quickly it can return results varies based on the underlying hardware.

Inaccurate time keeping can result in system instability. Test any change to the clock source very carefully. Operating system defaults are sometimes made to favor reliability over best accuracy. And if you are using a virtual machine, look into the recommended time sources compatible with it. Virtual hardware faces additional difficulties when emulating timers, and there are often per operating system settings suggested by vendors.

The Time Stamp Counter (TSC) clock source is the most accurate one available on current generation CPUs. It's the preferred way to track the system time when it's supported by the operating system and the TSC clock is reliable. There are several ways that TSC can fail to provide an accurate timing source, making it unreliable. Older systems can have a TSC clock that varies based on the CPU temperature, making it unusable for timing. Trying to use TSC on some older multicore CPUs can give a reported time that's inconsistent among multiple cores. This can result in the time going backwards, a problem this program checks for. And even the newest systems can fail to provide accurate TSC timing with very aggressive power saving configurations.

Newer operating systems may check for the known TSC problems and switch to a slower, more stable clock source when they are seen. If your system supports TSC time but doesn't default to that, it may be disabled for a good reason. And some operating systems may not detect all the possible problems correctly, or will allow using TSC even in situations where it's known to be inaccurate.

The High Precision Event Timer (HPET) is the preferred timer on systems where it's available and TSC is not accurate. The timer chip itself is programmable to allow up to 100 nanosecond resolution, but you may not see that much accuracy in your system clock.

Advanced Configuration and Power Interface (ACPI) provides a Power Management (PM) Timer, which Linux refers to as the acpi_pm. The clock derived from acpi_pm will at best provide 300 nanosecond resolution.

Timers used on older PC hardware include the 8254 Programmable Interval Timer (PIT), the real-time clock (RTC), the Advanced Programmable Interrupt Controller (APIC) timer, and the Cyclone timer. These timers aim for millisecond resolution.

pg_upgrade

pg_upgrade — upgrade a IvorySQL server instance

Synopsis

`pg_upgrade -b oldbindir [-B newbindir] -d oldconfigdir -D newconfigdir [option…]`

Options

pg_upgrade accepts the following command-line arguments:

- `-b bindir --old-bindir=bindir`

the old IvorySQL executable directory; environment variable `PGBINOLD`

- `-B bindir --new-bindir=bindir`

the new IvorySQL executable directory; default is the directory where pg_upgrade resides; environment variable `PGBINNEW`

- `-c --check`

check clusters only, don’ t change any data

- `-d configdir --old-datadir=configdir`

the old database cluster configuration directory; environment variable `PGDATAOLD`

- `-D configdir --new-datadir=configdir`

the new database cluster configuration directory; environment variable `PGDATANEW`

- `-j njobs --jobs=njobs`

number of simultaneous processes or threads to use

- `-k --link`

use hard links instead of copying files to the new cluster

- `-N --no-sync`

By default, `pg_upgrade` will wait for all files of the upgraded cluster to be written safely to disk. This option causes `pg_upgrade` to return without waiting, which is faster, but means that a subsequent operating system crash can leave the data directory corrupt. Generally, this option is useful for testing but should not be used on a production installation.

- `-o options --old-options options`

options to be passed directly to the old `postgres` command; multiple option invocations are appended

- `-O options --new-options options`

options to be passed directly to the new `postgres` command; multiple option invocations are appended

- `-p port --old-port=port`

the old cluster port number; environment variable PGPORTOLD

- -P port --new-port=port

the new cluster port number; environment variable PGPORTNEW

- -r --retain

retain SQL and log files even after successful completion

- -s dir --socketdir=dir

directory to use for postmaster sockets during upgrade; default is current working directory; environment variable PGSOCKETDIR

- -U username --username=username

cluster’ s install user name; environment variable PGUSER

- -v --verbose

enable verbose internal logging

- -V --version

display version information, then exit

- --clone

Use efficient file cloning (also known as “reflinks” on some systems) instead of copying files to the new cluster. This can result in near-instantaneous copying of the data files, giving the speed advantages of -k/--link while leaving the old cluster untouched.File cloning is only supported on some operating systems and file systems. If it is selected but not supported, the pg_upgrade run will error. At present, it is supported on Linux (kernel 4.5 or later) with Btrfs and XFS (on file systems created with reflink support), and on macOS with APFS.

- -? --help

show help, then exit

Usage

These are the steps to perform an upgrade with pg_upgrade:

1.Optionally move the old cluster

> If your installation directory is not version-specific, e.g., /usr/local/pgsql, it is necessary to move the current IvorySQL install directory so it does not interfere with the new IvorySQL installation. Once the current IvorySQL server is shut down, it is safe to rename the IvorySQL installation directory; assuming the old directory is /usr/local/pgsql, you can do:
>
> ```
> mv /usr/local/pgsql /usr/local/pgsql.old
> ```
>
> to rename the directory.

2.For source installs, build the new version

Build the new IvorySQL source with `configure` flags that are compatible with the old cluster. pg_upgrade will check `pg_controldata` to make sure all settings are compatible before starting the upgrade.

3.Install the new IvorySQL binaries

Install the new server’ s binaries and support files. pg_upgrade is included in a default installation.

For source installs, if you wish to install the new server in a custom location, use the `prefix` variable:

```
make prefix=/usr/local/pgsql.new install
```

4.Initialize the new IvorySQL cluster

Initialize the new cluster using `initdb`. Again, use compatible `initdb` flags that match the old cluster. Many prebuilt installers do this step automatically. There is no need to start the new cluster.

5.Install extension shared object files

Many extensions and custom modules, whether from `contrib` or another source, use shared object files (or DLLs), e.g., `pgcrypto.so`. If the old cluster used these, shared object files matching the new server binary must be installed in the new cluster, usually via operating system commands. Do not load the schema definitions, e.g., `CREATE EXTENSION pgcrypto`, because these will be duplicated from the old cluster. If extension updates are available, pg_upgrade will report this and create a script that can be run later to update them.

6.Copy custom full-text search files

Copy any custom full text search files (dictionary, synonym, thesaurus, stop words) from the old to the new cluster.

7.Adjust authentication

`pg_upgrade` will connect to the old and new servers several times, so you might want to set authentication to `peer` in `pg_hba.conf` or use a `~/.pgpass` file .

8.Stop both servers

Make sure both database servers are stopped using, on Unix, e.g.:

```
pg_ctl -D /opt/IvorySQL/1.5 stop
pg_ctl -D /opt/IvorySQL/2.1 stop
```

or on Windows, using the proper service names:

```
NET STOP IvorySQL-1.5
NET STOP IvorySQL-2.1
```

Streaming replication and log-shipping standby servers can remain running until a later step.

9.Prepare for standby server upgrades

Verify that the old standby servers are caught up by running pg_controldata against the old primary and standby clusters. Verify that the "Latest checkpoint location" values match in all clusters. (There will be a mismatch if old standby servers were shut down before the old primary or if the old standby servers are still running.) Also, make sure `wal_level` is not set to `minimal` in the `IvorySQL.conf` file on the new primary cluster.

10.Run pg_upgrade

Always run the pg_upgrade binary of the new server, not the old one. pg_upgrade requires the specification of the old and new cluster's data and executable (`bin`) directories. You can also specify user and port values, and whether you want the data files linked or cloned instead of the default copy behavior.

If you use link mode, the upgrade will be much faster (no file copying) and use less disk space, but you will not be able to access your old cluster once you start the new cluster after the upgrade. Link mode also requires that the old and new cluster data directories be in the same file system. (Tablespaces and `pg_wal` can be on different file systems.) Clone mode provides the same speed and disk space advantages but does not cause the old cluster to be unusable once the new cluster is started. Clone mode also requires that the old and new data directories be in the same file system. This mode is only available on certain operating systems and file systems.

The `--jobs` option allows multiple CPU cores to be used for copying/linking of files and to dump and restore database schemas in parallel; a good place to start is the maximum of the number of CPU cores and tablespaces. This option can dramatically reduce the time to upgrade a multi-database server running on a multiprocessor machine.

For Windows users, you must be logged into an administrative account, and then start a shell as the `postgres` user and set the proper path:

```
RUNAS /USER:postgres "CMD.EXE"
SET PATH=%PATH%;C:\Program Files\IvorySQL\15\bin;
```

and then run pg_upgrade with quoted directories, e.g.:

```
pg_upgrade.exe
        --old-datadir "C:/Program Files/IvorySQL/1.5/data"
        --new-datadir "C:/Program Files/IvorySQL/2.1/data"
        --old-bindir "C:/Program Files/IvorySQL/1.5/bin"
        --new-bindir "C:/Program Files/IvorySQL/2.1/bin"
```

Once started, `pg_upgrade` will verify the two clusters are compatible and then do the upgrade. You can use `pg_upgrade --check` to perform only the checks, even if the old server is still running. `pg_upgrade --check` will also outline any manual adjustments you will need to make after the upgrade. If you are

going to be using link or clone mode, you should use the option `--link` or `--clone` with `--check` to enable mode-specific checks. `pg_upgrade` requires write permission in the current directory.

Obviously, no one should be accessing the clusters during the upgrade. pg_upgrade defaults to running servers on port 50432 to avoid unintended client connections. You can use the same port number for both clusters when doing an upgrade because the old and new clusters will not be running at the same time. However, when checking an old running server, the old and new port numbers must be different.

If an error occurs while restoring the database schema, `pg_upgrade` will exit and you will have to revert to the old cluster .To try `pg_upgrade` again, you will need to modify the old cluster so the pg_upgrade schema restore succeeds. If the problem is a `contrib` module, you might need to uninstall the `contrib` module from the old cluster and install it in the new cluster after the upgrade, assuming the module is not being used to store user data.

11.Upgrade streaming replication and log-shipping standby servers

If you used link mode and have Streaming Replication or Log-Shipping standby servers, you can follow these steps to quickly upgrade them. You will not be running pg_upgrade on the standby servers, but rather rsync on the primary. Do not start any servers yet.

If you did not use link mode, do not have or do not want to use rsync, or want an easier solution, skip the instructions in this section and simply recreate the standby servers once pg_upgrade completes and the new primary is running.

1. Install the new IvorySQL binaries on standby servers

```
Make sure the new binaries and support files are installed on all standby
servers.
```

2. Make sure the new standby data directories do *not* exist

```
Make sure the new standby data directories do *not* exist or are empty. If
initdb was run, delete the standby servers' new data directories.
```

3. Install extension shared object files

```
Install the same extension shared object files on the new standbys that you
installed in the new primary cluster.
```

4. Stop standby servers

```
If the standby servers are still running, stop them now using the above
instructions.
```

5. Save configuration files

```
Save any configuration files from the old standbys' configuration directories
you need to keep, e.g., `IvorySQL.conf` (and any files included by it),
`IvorySQL.auto.conf`, `pg_hba.conf`, because these will be overwritten or
```

removed in the next step.

6. Run rsync

When using link mode, standby servers can be quickly upgraded using rsync. To accomplish this, from a directory on the primary server that is above the old and new database cluster directories, run this on the *primary* for each standby server:

```
rsync --archive --delete --hard-links --size-only --no-inc-recursive
old_cluster new_cluster remote_dir
```

where `old_cluster` and `new_cluster` are relative to the current directory on the primary, and `remote_dir` is *above* the old and new cluster directories on the standby. The directory structure under the specified directories on the primary and standbys must match. Consult the rsync manual page for details on specifying the remote directory, e.g.,

```
rsync --archive --delete --hard-links --size-only --no-inc-recursive
/opt/IvorySQL/1.5 \
      /opt/IvorySQL/2.1 standby.example.com:/opt/IvorySQL
```

You can verify what the command will do using rsync's `--dry-run` option. While rsync must be run on the primary for at least one standby, it is possible to run rsync on an upgraded standby to upgrade other standbys, as long as the upgraded standby has not been started.

What this does is to record the links created by pg_upgrade's link mode that connect files in the old and new clusters on the primary server. It then finds matching files in the standby's old cluster and creates links for them in the standby's new cluster. Files that were not linked on the primary are copied from the primary to the standby. (They are usually small.) This provides rapid standby upgrades. Unfortunately, rsync needlessly copies files associated with temporary and unlogged tables because these files don't normally exist on standby servers.

If you have relocated `pg_wal` outside the data directories, rsync must be run on those directories too.

7. Configure streaming replication and log-shipping standby servers

```
Configure the servers for log shipping. (You do not need to run
`pg_backup_start()` and `pg_backup_stop()` or take a file system backup as
the standbys are still synchronized with the primary.) Replication slots are
not copied and must be recreated.
```

12.Restore `pg_hba.conf`

If you modified `pg_hba.conf`, restore its original settings. It might also be necessary to adjust other configuration files in the new cluster to match the old cluster, e.g., `IvorySQL.conf` (and any files included by it), `IvorySQL.auto.conf`.

13.Start the new server

The new server can now be safely started, and then any rsync’ ed standby servers.

14.Post-upgrade processing

If any post-upgrade processing is required, pg_upgrade will issue warnings as it completes. It will also generate script files that must be run by the administrator. The script files will connect to each database that needs post-upgrade processing. Each script should be run using:

```
psql --username=postgres --file=script.sql postgres
```

The scripts can be run in any order and can be deleted once they have been run.

Caution

In general it is unsafe to access tables referenced in rebuild scripts until the rebuild scripts have run to completion; doing so could yield incorrect results or poor performance. Tables not referenced in rebuild scripts can be accessed immediately.

15.Statistics

Because optimizer statistics are not transferred by `pg_upgrade`, you will be instructed to run a command to regenerate that information at the end of the upgrade. You might need to set connection parameters to match your new cluster.

16.Delete old cluster

Once you are satisfied with the upgrade, you can delete the old cluster’ s data directories by running the script mentioned when `pg_upgrade` completes. (Automatic deletion is not possible if you have user-defined tablespaces inside the old data directory.) You can also delete the old installation directories (e.g., `bin`, `share`).

17.Reverting to old cluster

If, after running `pg_upgrade`, you wish to revert to the old cluster, there are several options:

- If the `--check` option was used, the old cluster was unmodified; it can be restarted.
- If the `--link` option was not used, the old cluster was unmodified; it can be restarted.
- If the `--link` option was used, the data files might be shared between the old and new cluster:
  - If `pg_upgrade` aborted before linking started, the old cluster was unmodified; it can be restarted.
  - If you did not start the new cluster, the old cluster was unmodified except that, when linking started, a `.old` suffix was appended to `$PGDATA/global/pg_control`. To reuse the old cluster, remove the `.old` suffix from `$PGDATA/global/pg_control`; you can then restart the old cluster.
  - If you did start the new cluster, it has written to shared files and it is unsafe to use the old cluster. The old cluster will need to be restored from backup in this case.

Notes

pg_upgrade creates various working files, such as schema dumps, stored within `pg_upgrade_output.d` in the directory of the new cluster. Each run creates a new subdirectory named with a timestamp formatted as per ISO 8601 (`%Y%m%dT%H%M%S`), where all its generated files are stored. `pg_upgrade_output.d` and its contained files will be removed automatically if pg_upgrade completes successfully; but in the event of trouble, the files there may provide useful debugging information.

pg_upgrade launches short-lived postmasters in the old and new data directories. Temporary Unix socket files for communication with these postmasters are, by default, made in the current working directory. In some situations the path name for the current directory might be too long to be a valid socket name. In that case you can use the `-s` option to put the socket files in some directory with a shorter path name. For security, be sure that that directory is not readable or writable by any other users. (This is not supported on Windows.)

All failure, rebuild, and reindex cases will be reported by pg_upgrade if they affect your installation; post-upgrade scripts to rebuild tables and indexes will be generated automatically. If you are trying to automate the upgrade of many clusters, you should find that clusters with identical database schemas require the same post-upgrade steps for all cluster upgrades; this is because the post-upgrade steps are based on the database schemas, and not user data.

For deployment testing, create a schema-only copy of the old cluster, insert dummy data, and upgrade that.

pg_upgrade does not support upgrading of databases containing table columns using these `reg*` OID-referencing system data types:

`regcollation`
`regconfig`
`regdictionary`
`regnamespace`
`regoper`
`regoperator`
`regproc`
`regprocedure`

(`regclass`, `regrole`, and `regtype` can be upgraded.)

If you want to use link mode and you do not want your old cluster to be modified when the new cluster is started, consider using the clone mode. If that is not available, make a copy of the old cluster and upgrade that in link mode. To make a valid copy of the old cluster, use `rsync` to create a dirty copy of the old cluster while the server is running, then shut down the old server and run `rsync --checksum` again to update the copy with any changes to make it consistent. (`--checksum` is necessary because `rsync` only has file modification-time granularity of one second.) . If your file system supports file system snapshots or copy-on-write file copies, you can use that to make a backup of the old cluster and tablespaces, though the snapshot and copies must be created simultaneously or while the database server is down.

pg_waldump

pg_waldump — display a human-readable rendering of the write-ahead log of a IvorySQL database cluster

Synopsis

pg_waldump [**option**···] [**startseg** [**endseg**]]

Options

The following command-line options control the location and format of the output:

- **startseg**

Start reading at the specified log segment file. This implicitly determines the path in which files will be searched for, and the timeline to use.

- **endseg**

Stop after reading the specified log segment file.

- **-b --bkp-details**

Output detailed information about backup blocks.

- **-B block --block=block**

Only display records that modify the given block. The relation must also be provided with **--relation** or **-R**.

- **-e end --end=end**

Stop reading at the specified WAL location, instead of reading to the end of the log stream.

- **-f --follow**

After reaching the end of valid WAL, keep polling once per second for new WAL to appear.

- **-F fork --fork=fork**

If provided, only display records that modify blocks in the given fork. The valid values are **main** for the main fork, **fsm** for the free space map, **vm** for the visibility map, and **init** for the init fork.

- **-n limit --limit=limit**

Display the specified number of records, then stop.

- **-p path --path=path**

Specifies a directory to search for log segment files or a directory with a **pg_wal** subdirectory that contains such files. The default is to search in the current directory, the **pg_wal** subdirectory of the current directory, and the **pg_wal** subdirectory of **PGDATA**.

- **-q --quiet**

Do not print any output, except for errors. This option can be useful when you want to know whether a range of WAL records can be successfully parsed but don’ t care about the record contents.

- **-r rmgr --rmgr=rmgr**

Only display records generated by the specified resource manager. You can specify the option multiple times to select multiple resource managers. If **list** is passed as name, print a list of valid resource manager names,

and exit.Extensions may define custom resource managers, but pg_waldump does not load the extension module and therefore does not recognize custom resource managers by name. Instead, you can specify the custom resource managers as `custom where “` is the three-digit resource manager ID. Names of this form will always be considered valid.

- `-R tblspc / db / rel --relation=tblspc / db / rel`

Only display records that modify blocks in the given relation. The relation is specified with tablespace OID, database OID, and relfilenode separated by slashes, for example `1234/12345/12345`. This is the same format used for relations in the program’ s output.

- `-s start --start=start`

WAL location at which to start reading. The default is to start reading the first valid log record found in the earliest file found.

- `-t timeline --timeline=timeline`

Timeline from which to read log records. The default is to use the value in `startseg`, if that is specified; otherwise, the default is 1.

- `-V --version`

Print the pg_waldump version and exit.

- `-w --fullpage`

Only display records that include full page images.

- `-x xid --xid=xid`

Only display records marked with the given transaction ID.

- `-z --stats[=record]`

Display summary statistics (number and size of records and full-page images) instead of individual records. Optionally generate statistics per-record instead of per-rmgr.If pg_waldump is terminated by signal SIGINT (Control + C), the summary of the statistics computed is displayed up to the termination point. This operation is not supported on Windows.

- `-? --help`

Show help about pg_waldump command line arguments, and exit.

Environment

- `PGDATA`

Data directory; see also the `-p` option.

- `PG_COLOR`

Specifies whether to use color in diagnostic messages. Possible values are `always`, `auto` and `never`.

Notes

Can give wrong results when the server is running.

Only the specified timeline is displayed (or the default, if none is specified). Records in other timelines are ignored.

pg_waldump cannot read WAL files with suffix `.partial`. If those files need to be read, `.partial` suffix needs

to be removed from the file name.

postgres

postgres — IvorySQL database server

Synopsis

postgres [**option**···]

Options

postgres accepts the following command-line arguments. You can save typing most of these options by setting up a configuration file. Some (safe) options can also be set from the connecting client in an application-dependent way to apply only for that session. For example, if the environment variable **PGOPTIONS** is set, then libpq-based clients will pass that string to the server, which will interpret it as **postgres** command-line options.

General Purpose

- **-B nbuffers**

Sets the number of shared buffers for use by the server processes. The default value of this parameter is chosen automatically by initdb. Specifying this option is equivalent to setting the shared_buffers configuration parameter.

- **-c name=value**

Sets a named run-time parameter. Most of the other command line options are in fact short forms of such a parameter assignment. **-c** can appear multiple times to set multiple parameters.

- **-C name**

Prints the value of the named run-time parameter, and exits. (See the **-c** option above for details.) This returns values from **postgresql.conf**, modified by any parameters supplied in this invocation. It does not reflect parameters supplied when the cluster was started.This can be used on a running server for most parameters. This option is meant for other programs that interact with a server instance, such as pg_ctl, to query configuration parameter values. User-facing applications should instead use **SHOW** or the **pg_settings** view.

- **-d debug-level**

Sets the debug level. The higher this value is set, the more debugging output is written to the server log. Values are from 1 to 5. It is also possible to pass **-d 0** for a specific session, which will prevent the server log level of the parent **postgres** process from being propagated to this session.

- **-D datadir**

Specifies the file system location of the database configuration files.

- **-e**

Sets the default date style to “European”, that is **DMY** ordering of input date fields. This also causes the day to be printed before the month in certain date output formats.

- **-F**

Disables **fsync** calls for improved performance, at the risk of data corruption in the event of a system crash. Specifying this option is equivalent to disabling the fsync configuration parameter. Read the detailed documentation before using this!

- `-h hostname`

Specifies the IP host name or address on which `postgres` is to listen for TCP/IP connections from client applications. The value can also be a comma-separated list of addresses, or `*` to specify listening on all available interfaces. An empty value specifies not listening on any IP addresses, in which case only Unix-domain sockets can be used to connect to the server. Defaults to listening only on localhost. Specifying this option is equivalent to setting the listen_addresses configuration parameter.

- `-i`

Allows remote clients to connect via TCP/IP (Internet domain) connections. Without this option, only local connections are accepted. This option is equivalent to setting `listen_addresses` to `*` in `IvorySQL.conf` or via `-h`.This option is deprecated since it does not allow access to the full functionality of listen_addresses. It's usually better to set `listen_addresses` directly.

- `-k directory`

Specifies the directory of the Unix-domain socket on which `postgres` is to listen for connections from client applications. The value can also be a comma-separated list of directories. An empty value specifies not listening on any Unix-domain sockets, in which case only TCP/IP sockets can be used to connect to the server. The default value is normally `/tmp`, but that can be changed at build time. Specifying this option is equivalent to setting the unix_socket_directories configuration parameter.

- `-l`

Enables secure connections using SSL. IvorySQL must have been compiled with support for SSL for this option to be available.

- `-N max-connections`

Sets the maximum number of client connections that this server will accept. The default value of this parameter is chosen automatically by initdb. Specifying this option is equivalent to setting the max_connections configuration parameter.

- `-p port`

Specifies the TCP/IP port or local Unix domain socket file extension on which `postgres` is to listen for connections from client applications. Defaults to the value of the `PGPORT` environment variable, or if `PGPORT` is not set, then defaults to the value established during compilation (normally 5432). If you specify a port other than the default port, then all client applications must specify the same port using either command-line options or `PGPORT`.

- `-s`

Print time information and other statistics at the end of each command. This is useful for benchmarking or for use in tuning the number of buffers.

- `-S work-mem`

Specifies the base amount of memory to be used by sorts and hash tables before resorting to temporary disk files.

- `-V --version`

Print the postgres version and exit.

- `--name=value`

Sets a named run-time parameter; a shorter form of `-c`.

- `--describe-config`

This option dumps out the server's internal configuration variables, descriptions, and defaults in tab-

delimited `COPY` format. It is designed primarily for use by administration tools.

- `-? --help`

Show help about postgres command line arguments, and exit.

Semi-Internal Options

The options described here are used mainly for debugging purposes, and in some cases to assist with recovery of severely damaged databases. There should be no reason to use them in a production database setup. They are listed here only for use by IvorySQL system developers. Furthermore, these options might change or be removed in a future release without notice.

- `-f { s | i | o | b | t | n | m | h }`

Forbids the use of particular scan and join methods: `s` and `i` disable sequential and index scans respectively, `o`, `b` and `t` disable index-only scans, bitmap index scans, and TID scans respectively, while `n`, `m`, and `h` disable nested-loop, merge and hash joins respectively.Neither sequential scans nor nested-loop joins can be disabled completely; the `-fs` and `-fn` options simply discourage the optimizer from using those plan types if it has any other alternative.

- `-n`

This option is for debugging problems that cause a server process to die abnormally. The ordinary strategy in this situation is to notify all other server processes that they must terminate and then reinitialize the shared memory and semaphores. This is because an errant server process could have corrupted some shared state before terminating. This option specifies that `postgres` will not reinitialize shared data structures. A knowledgeable system programmer can then use a debugger to examine shared memory and semaphore state.

- `-O`

Allows the structure of system tables to be modified. This is used by `initdb`.

- `-P`

Ignore system indexes when reading system tables, but still update the indexes when modifying the tables. This is useful when recovering from damaged system indexes.

- `-t pa[rser] | pl[anner] | e[xecutor]`

Print timing statistics for each query relating to each of the major system modules. This option cannot be used together with the `-s` option.

- `-T`

This option is for debugging problems that cause a server process to die abnormally. The ordinary strategy in this situation is to notify all other server processes that they must terminate and then reinitialize the shared memory and semaphores. This is because an errant server process could have corrupted some shared state before terminating. This option specifies that `postgres` will stop all other server processes by sending the signal `SIGSTOP`, but will not cause them to terminate. This permits system programmers to collect core dumps from all server processes by hand.

- `-v protocol`

Specifies the version number of the frontend/backend protocol to be used for a particular session. This option is for internal use only.

- `-W seconds`

A delay of this many seconds occurs when a new server process is started, after it conducts the authentication procedure. This is intended to give an opportunity to attach to the server process with a

debugger.

Options for Single-User Mode

The following options only apply to the single-user mode (see Single-User Mode below).

- `--single`

Selects the single-user mode. This must be the first argument on the command line.

- `database`

Specifies the name of the database to be accessed. This must be the last argument on the command line. If it is omitted it defaults to the user name.

- `-E`

Echo all commands to standard output before executing them.

- `-j`

Use semicolon followed by two newlines, rather than just newline, as the command entry terminator.

- `-r filename`

Send all server log output to `filename`. This option is only honored when supplied as a command-line option.

Environment

- `PGCLIENTENCODING`

Default character encoding used by clients. (The clients can override this individually.) This value can also be set in the configuration file.

- `PGDATA`

Default data directory location

- `PGDATESTYLE`

Default value of the DateStyle run-time parameter. (The use of this environment variable is deprecated.)

- `PGPORT`

Default port number (preferably set in the configuration file)

Diagnostics

A failure message mentioning `semget` or `shmget` probably indicates you need to configure your kernel to provide adequate shared memory and semaphores. You might be able to postpone reconfiguring your kernel by decreasing shared_buffers to reduce the shared memory consumption of IvorySQL, and/or by reducing max_connections to reduce the semaphore consumption.

A failure message suggesting that another server is already running should be checked carefully, for example by using the command

```
$ ps ax | grep postgres
```

or

```
$ ps -ef | grep postgres
```

depending on your system. If you are certain that no conflicting server is running, you can remove the lock file mentioned in the message and try again.

A failure message indicating inability to bind to a port might indicate that that port is already in use by some non-IvorySQL process. You might also get this error if you terminate **postgres** and immediately restart it using the same port; in this case, you must simply wait a few seconds until the operating system closes the port before trying again. Finally, you might get this error if you specify a port number that your operating system considers to be reserved. For example, many versions of Unix consider port numbers under 1024 to be "trusted" and only permit the Unix superuser to access them.

Notes

The utility command pg_ctl can be used to start and shut down the **postgres** server safely and comfortably.

If at all possible, do not use **SIGKILL** to kill the main **postgres** server. Doing so will prevent **postgres** from freeing the system resources (e.g., shared memory and semaphores) that it holds before terminating. This might cause problems for starting a fresh **postgres** run.

To terminate the **postgres** server normally, the signals **SIGTERM**, **SIGINT**, or **SIGQUIT** can be used. The first will wait for all clients to terminate before quitting, the second will forcefully disconnect all clients, and the third will quit immediately without proper shutdown, resulting in a recovery run during restart.

The **SIGHUP** signal will reload the server configuration files. It is also possible to send **SIGHUP** to an individual server process, but that is usually not sensible.

To cancel a running query, send the **SIGINT** signal to the process running that command. To terminate a backend process cleanly, send **SIGTERM** to that process. See also **pg_cancel_backend**.

The **postgres** server uses **SIGQUIT** to tell subordinate server processes to terminate without normal cleanup. This signal should not be used by users. It is also unwise to send **SIGKILL** to a server process — the main **postgres** process will interpret this as a crash and will force all the sibling processes to quit as part of its standard crash-recovery procedure.

Bugs

The **--** options will not work on FreeBSD or OpenBSD. Use **-c** instead. This is a bug in the affected operating systems; a future release of IvorySQL will provide a workaround if this is not fixed.

Single-User Mode

To start a single-user mode server, use a command like

```
postgres --single -D /usr/local/pgsql/data other-options my_database
```

Provide the correct path to the database directory with **-D**, or make sure that the environment variable **PGDATA** is set. Also specify the name of the particular database you want to work in.

Normally, the single-user mode server treats newline as the command entry terminator; there is no intelligence about semicolons, as there is in psql. To continue a command across multiple lines, you must type backslash just before each newline except the last one. The backslash and adjacent newline are both dropped from the input command. Note that this will happen even when within a string literal or comment.

But if you use the **-j** command line switch, a single newline does not terminate command entry; instead, the sequence semicolon-newline-newline does. That is, type a semicolon immediately followed by a completely empty line. Backslash-newline is not treated specially in this mode. Again, there is no intelligence about such a sequence appearing within a string literal or comment.

In either input mode, if you type a semicolon that is not just before or part of a command entry terminator, it is considered a command separator. When you do type a command entry terminator, the multiple statements you've entered will be executed as a single transaction.

To quit the session, type EOF (Control+D, usually). If you've entered any text since the last command entry terminator, then EOF will be taken as a command entry terminator, and another EOF will be needed to exit.

Note that the single-user mode server does not provide sophisticated line-editing features (no command history, for example). Single-user mode also does not do any background processing, such as automatic checkpoints or replication.

Examples

To start **postgres** in the background using default values, type:

```
$ nohup postgres >logfile 2>&1 </dev/null &
```

To start **postgres** with a specific port, e.g., 1234:

```
$ postgres -p 1234
```

To connect to this server using psql, specify this port with the -p option:

```
$ psql -p 1234
```

or set the environment variable **PGPORT**:

```
$ export PGPORT=1234
$ psql
```

Named run-time parameters can be set in either of these styles:

```
$ postgres -c work_mem=1234
$ postgres --work-mem=1234
```

Either form overrides whatever setting might exist for **work_mem** in **IvorySQL.conf**. Notice that underscores in parameter names can be written as either underscore or dash on the command line. Except for short-term experiments, it's probably better practice to edit the setting in **IvorySQL.conf** than to rely on a command-line switch to set a parameter.

Chapter 2. FAQ

Contributing

IvorySQL is maintained by a core team of developers with commit rights to the main IvorySQL repository on GitHub. At the same time, we are very eager to receive contributions from anybody in the wider IvorySQL community. This section covers all you need to know if you want to see your code or documentation changes be added to IvorySQL and appear in future releases.

Getting started

IvorySQL is developed on GitHub, and anybody wishing to contribute to it will have to have a GitHub account and be familiar with Git tools and workflow. It is also recommended that you follow the developer’ s mailing list since some of the contributions may generate more detailed discussions there.

Once you have your GitHub account, fork this repository so that you can have your private copy to start hacking on and to use as a source of pull requests.

Licensing of IvorySQL contributions

If the contribution you’ re submitting is original work, you can assume that IvorySQL will release it as part of an overall IvorySQL release available to the downstream consumers under the Apache License, Version 2.0.

If the contribution you’ re submitting is NOT original work you have to indicate the name of the license and also make sure that it is similar in terms to the Apache License 2.0. Apache Software Foundation maintains a list of these licenses under Category A. In addition to that, you may be required to make proper attributions.

Finally, keep in mind that it is NEVER a good idea to remove licensing headers from the work that is not your original one. Even if you are using parts of the file that originally had a licensing header at the top you should err on the side of preserving it. As always, if you are not quite sure about the licensing implications of your contributions, feel free to reach out to us on the developer mailing list.

Coding guidelines

Your chances of getting feedback and seeing your code merged into the project greatly depend on how granular your changes are. If you happen to have a bigger change in mind, we highly recommend engaging on the developer’ s mailing list first and sharing your proposal with us before you spend a lot of time writing code. Even when your proposal gets validated by the community, we still recommend doing the actual work as a series of small, self-contained commits. This makes the reviewer’ s job much easier and increases the timeliness of feedback.

When it comes to C and C++ parts of IvorySQL, we try to follow PostgreSQL Coding Conventions. In addition to that:

For C and Perl code, please run pgindent if necessary. We recommend using git diff --color when reviewing your changes so that you don’ t have any spurious whitespace issues in the code that you submit.

All new functionality that is contributed to IvorySQL should be covered by regression tests that are contributed alongside it. If you are uncertain about how to test or document your work, please raise the question on the ivorysql-hackers mailing list and the developer community will do its best to help you.

At the very minimum, you should always be running make installcheck-world to make sure that you’ re not breaking anything.

Changes applicable to upstream PostgreSQL

If the change you’ re working on touches functionality that is common between PostgreSQL and IvorySQL, you may be asked to forward-port it to PostgreSQL. This is not only so that we keep reducing the delta between the two projects, but also so that any change that is relevant to PostgreSQL can benefit from a much broader review of the upstream PostgreSQL community. In general, it is a good idea to keep both codebases handy so you can be sure whether your changes may need to be forward-ported.

Patch submission

Once you are ready to share your work with the IvorySQL core team and the rest of the IvorySQL community, you should push all the commits to a branch in your own repository forked from the official IvorySQL and send us a pull request.

Patch review

A submitted pull request with passing validation checks is assumed to be available for peer review. Peer review is the process that ensures that contributions to IvorySQL are of high quality and align well with the road map and community expectations. Every member of the IvorySQL community is encouraged to review pull requests and provide feedback. Since you don’ t have to be a core team member to be able to do that, we recommend following a stream of pull reviews to anybody who’ s interested in becoming a long-term contributor to IvorySQL.

One outcome of the peer review could be a consensus that you need to modify your pull request in certain ways. GitHub allows you to push additional commits into a branch from which a pull request was sent. Those additional commits will be then visible to all of the reviewers.

A peer review converges when it receives at least one +1 and no -1s votes from the participants. At that point, you should expect one of the core team members to pull your changes into the project.

At any time during the patch review, you may experience delays based on the availability of reviewers and core team members. Please be patient. That being said, don’ t get discouraged either. If you’ re not getting expected feedback for a few days add a comment asking for updates on the pull request itself or send an email to the mailing list.

Direct commits to the repository

On occasion, you will see core team members committing directly to the repository without going through the pull request workflow. This is reserved for small changes only and the rule of thumb we use is this: if the change touches any functionality that may result in a test failure, then it has to go through a pull request workflow. If, on the other hand, the change is in the non-functional part of the codebase (such as fixing a typo inside of a comment block) core team members can decide to just commit to the repository directly.