PostgreSQL

en.wikibooks.org

June 29, 2024

Contents

1 Major Features

1.1 An outstanding Goal

The very first concept paper[1] - published in 1986 - defines a goal that distinguishes PostgreSQL from many other systems until today: *"provide user extendibility for data types, operators, and access methods"*. This goal is reached. And it is not only available for users, even the internal implementation utilizes those interfaces to create system components.

1.2 Architecture

PostgreSQL implements a client/server model. Each *client process* connects to one *backend process* at the server site. Such backend processes are part of the *instance*, a group of many processes which act closely together and handle the data access. PostgreSQL does not use threading in the backend processes or elsewhere.

1.3 Features

1.3.1 Security

- Authentication methods: SCRAM-SHA-256, GSSAPI, SSPI, LDAP, RADIUS, Certificate, PAM
- Roles (users and groups) authorize access to data and execution of functions

1.3.2 Reliability

- Transactions with full ACID support and diverse isolation levels
- Savepoints (Sub-transactions)
- Multi-Version Concurrency Control (MVCC)
- Point-in-Time Recovery
- Partitioning of tables and indexes
- Synchronous, asynchronous, and logical replication
- Bi-Directional replication
- Publish/subscribe mechanism
- Parallel execution of single queries at multiple CPUs

1 `https://dsf.berkeley.edu/papers/ERL-M85-95.pdf`

1.3.3 Application Aspects

- Rich set of predefined data-types, i.a. JSON
- Support for arrays
- Composite type (similar to a *record* in some programming languages), constructor for rows via *row* keyword
- Check constraints
- Referential integrity with foreign keys
- Table inheritance
- Views, materialized views, updateable views

1.3.4 Extendability

- User defined data-types, operators, and index access methods
- User-defined functions, procedures, triggers, and procedural languages
- *Create extension* interface to create user-defined packages. Some publicly available examples:
  - *Foreign data wrappers* to other - PostgreSQL or non-PostgreSQL - databases or to the file-system
  - *PostGIS*: an extention for Spatial and Geographic Objects
  - *hstore*: a key/value storage

1.3.5 SQL Support

- High degree of conformance to the SQL standard: 170 out of 177 features
- Outer join, union, intersect, except
- Group by, grouping set, cube, rollup
- Common table expressions (CTE)
- Recursive queries, graph queries
- Window functions, analytic functions

2 Supported Platforms

PostgreSQL is available at diverse CPU architectures: x86, x86_64, IA64, PowerPC, PowerPC 64, S/390, S/390x, Sparc, Sparc 64, ARM, MIPS, MIPSEL, PA-RISC and runs on all major operating systems. [1]

- Linux
  - Red Hat family Linux (including CentOS/Fedora/Scientific/Oracle variants)
  - Debian GNU/Linux and derivatives
  - Ubuntu Linux and derivatives
  - SuSE and OpenSuSE
- macOS
- Windows (XP+)
- Solaris
- BSD
  - FreeBSD
  - OpenBSD
  - NetBSD
- AIX
- HP/UX

2.1 References

1 Operating Systems `https://www.postgresql.org/docs/current/static/supported-platforms.html`

3 Download, Installation

Before you download PostgreSQL you must make two crucial decisions. First, decide whether to compile and install PostgreSQL from source code or to install from prebuilt binaries. Second (if you want to use any binary), you must know for which operating system you need the software. PostgreSQL supports most UNIX-based systems (including macOS) as well as Windows.

After you have made those decisions you can download and use the complete source code, an installer, a Bitnami Infrastructure Stack, or the pure binaries.

3.1 Start at the Source Code Level

The source code is available as a single packed file [1] or in a git repository [2]. To install from source you must download it to your local computer and compile it with a C compiler (at least C99-compliant, in most cases people use GCC[3]) to the binary format of your computer. Details of the requirements [4], the download process, and the compilation steps [5] are available in the PostgreSQL documentation.

The advantages of working with the source code are that you can read and study it, modify it, or compile it on an exotic platform. But you must have some pre-knowledge and experience in handling specific tasks of your operating system, e.g.: working in a shell, installing additional programs, ...

The PostgreSQL documentation describes all details of the installation from source in the chapters:

- Installation from Source Code on Unix[6]
- Installation from Source Code on Windows[7]

3.2 Start with the Help of a Prebuild Program

In opposite to start at the source code level, it is relatively easy to use one of the pre-build programs or scripts. This is the preferred way for beginners. You can choose from several options:

1 Source code in a single packed file (via FTP) `https://www.postgresql.org/ftp/source/`
2 Source code in a git repository `https://www.postgresql.org/docs/current/git.html`
3 `https://gcc.gnu.org/`
4 Requirements for compilation (Unix)`https://www.postgresql.org/docs/current/static/install-requirements.html`
5 Installing from Source `https://www.postgresql.org/docs/current/installation.html`
6 `https://www.postgresql.org/docs/current/installation.html`
7 `https://www.postgresql.org/docs/current/install-windows.html`

- Installer [8]: This is the most comfortable way to download and install PostgreSQL on your local computer. The installer guides you not only through the installation steps, but also offers the option to install helpful additional tools and drivers. Installers are not available for all versions of all operating systems.
- Bitnami infrastructure stack [9]: Such stacks (WAPP, MAPP, LAPP, and others) offer the complete infrastructure (PostgreSQL, Apache Web Server, PHP) to run Web applications on Windows, macOS, or Linux.
- Pure binaries [10]: This is a listing of operating-specific commands which leads you thru the download and installation process of binaries.

3.3 Examples

Install binaries for Linux (Ubuntu) PostgreSQL Apt Repository [11]. . Retrieved

```
# Create the file repository configuration:
sudo sh -c 'echo "deb http://apt.postgresql.org/pub/repos/apt $(lsb_release
 -cs)-pgdg main" > /etc/apt/sources.list.d/pgdg.list'

# Import the repository signing key:
wget --quiet -O - https://www.postgresql.org/media/keys/ACCC4CF8.asc | sudo
 apt-key add -

# Update the package lists:
sudo apt-get update

# Install the latest version of PostgreSQL.
# If you want a specific version, use 'postgresql-12' or similar instead of
 'postgresql':
sudo apt-get -y install PostgreSQL
```

Starting and stopping

```
sudo /etc/init.d/postgresql start
sudo /etc/init.d/postgresql stop
```

Windows

By default, PostgreSQL launches at each reboot so it may consume many resources. To avoid that, just execute *services.msc* and change the PostgreSQL service to start manually. Then, create a file *postgresql.cmd* containing:

```
net start postgresql-x64-9.5
pause
net stop postgresql-x64-9.5
```

As long as this script is launched as an administrator, the cluster with all its databases is available. Just press a key to shutdown the service.

8 Installer https://www.enterprisedb.com/downloads/postgres-postgresql-downloads
9 Bitnami stacks https://bitnami.com/tag/postgresql
10 Download binaries https://www.postgresql.org/download/
11

3.4 More Information

The PostgreSQL wiki[12] offers a lot more information and hints about the installation steps.

After a successful installation, you will have

- The PostgreSQL binaries on your disc.
- A first `cluster` called *data* on your disc. The cluster consists of an empty `database` called *postgres* (plus two template databases) and a `user` resp. `role` called *postgres* as well.
- A set of Unix programs or a Windows `service` running on your computer. These programs/services handle the cluster with all its databases.

By default, PostgreSQL listens to `port` 5432. Possibly you must configure your firewall to reflect this situation.

3.5 Connect to the Database

After a successful installation, you have a cluster *data*, a database *postgres*, the database superuser *postgres*, and a new operating system user *postgres*. Login at the operating system level with the new operating system user. In a shell you can connect to the new database via the often used program `psql`. psql is a line-mode program similar to a shell and allows you to send SQL commands to the database.

```
$ # Example in Unix syntax
$ su - postgres
Password:
$
$ # psql --help      to see a detailed explanation of psql's options
$ # psql [OPTION]... [DBNAME [USERNAME]]
$ psql postgres postgres
psql (14.1 (Ubuntu 14.1-2.pgdg20.04+1))
Type "help" for help.

postgres=#
postgres=# \q  -- terminate psql with backslash q  or  ctrl-d
$
```

The default prompt (prefix of every new line) of psql is 'postgres=#'. After you have successfully started it, you can use SQL commands to communicate with the database. Here is an example that creates a new database user with the name 'nancy' - and deletes it afterward.

```
postgres=# CREATE USER nancy WITH ENCRYPTED PASSWORD 'ab8sxx5F4';
CREATE ROLE
postgres=#
postgres=# DROP USER nancy;  -- delete the user
DROP ROLE
postgres=#
```

The database responds to every SQL command indicating its successful execution or an error. In the previous example `CREATE ROLE` means that the user is created.

12 `https://wiki.postgresql.org/wiki/Detailed_installation_guides`

3.6 Separation of Concerns

Please recap what you have so far: a `cluster` *data*, a `database` *postgres*, a `user` *postgres*. Furthermore, PostgreSQL divides every database into logical units which are called `schema` . Most objects reside in such a schema. The default schema is named *public* and exists in every database. The same applies to some special schemas where system information is stored. As far as you don't explicitly use schema-names, the schema *public* is utilized by default. This means that a `CREATE TABLE t (column_1 INTEGER);` command will create the table *t* in schema *public*.

We recommend avoiding the schema *public* for your data. Because *public* exists in every database, some tools use it to store their data there. Create and work in your own schema to have a clear distinction between system-, tools-, and user-data.Second, avoid working with user *postgres*. This user account has very strong privileges and you should rarely use it. Create a user who acts as the stakeholder for your data, views, functions, trigger, etc. .

The following script creates a new user and its schema.

```
$ # start 'psql' as the original 'postgres' user with its strong priviledges
$ psql postgres postgres
postgres=# -- the owner of the new schema shall be 'finance_master'
postgres=# CREATE USER finance_master WITH CREATEROLE LOGIN ENCRYPTED PASSWORD
 'xxx';
CREATE ROLE
postgres=# -- the new schema 'finance' for your data
postgres=# CREATE SCHEMA finance AUTHORIZATION finance_master;
CREATE SCHEMA
postgres=# -- change 'search_path' (description of search_path: see below)
postgres=# ALTER ROLE finance_master SET search_path = finance, public;
ALTER ROLE
postgres=# \q
```

Start psql with the new user *finance_ master*. We want him to work in schema *finance*, but every connection between psql and PostgreSQL acts at the database-level. It's not possible to specify an individual schema for a connection. Therefore PostgreSQL has implemented a mechanism called `search_path`. It simplifies the switching between schemas. `search_path` contains a list of schema names. Whenever you omit a schema name, this list is consulted to decide which schema to use. For our user *finance_ manager* we have defined in the above `ALTER ROLE` command that he shall work in schema *finance* and - if there is no hit for his SQL command e.g. for a `SELECT` - the schema *public* is consulted next.

```
$ # -- first parameter of psql: database   second parameter: user   nothing for
 schema
$ psql postgres finance_master
postgres=# -- create a table
postgres=# CREATE TABLE t1 (column_1 INTEGER);  -- table will be in schema
 'finance' because of the 'search_path' definition
CREATE TABLE
postgres=# -- you can use the schema name explicitly
postgres=# CREATE TABLE finance.t2 (column_1 INTEGER);  -- table will be in
 schema 'finance' as well
CREATE TABLE
postgres=# -- it's possible to overwrite 'search_path' by using the schema name
 explicitly
postgres=# CREATE TABLE public.t3 (column_1 INTEGER);  -- table will be in
 schema 'public'
CREATE TABLE
postgres=#
postgres=# \d  -- this command lists schema, table, and owner names
```

```
             List of relations
 Schema  |  Name   | Type  |     Owner
---------+---------+-------+----------------
 finance | t1      | table | finance_master
 finance | t2      | table | finance_master
 public  | t3      | table | finance_master
postgres=#
```

3.7 References

4 Managing the Instance

The PostgreSQL **instance** consists of several processes that run continuously on the server. They work together in a coordinated manner using common configuration files and RAM. Thus all are running or none of them.

The process *postmaster* is one of them. It starts, stops, and controls the other processes. *postmaster* himself can be started directly or with the help of the wrapper program `pg_ctl` . Its simplified syntax is:

```
pg_ctl [ status | start | stop | restart | reload | init ] [-U username] [-P
 password] [--help]
```

The instance must be run by the operating system user *postgres* and not by *root*.

4.1 status

When pg_ctl runs in the `status` mode, it lists the actual status of the instance.

```
$ pg_ctl status
pg_ctl: server is running (PID: 16244)
/usr/lib/postgresql/14/bin/postgres
$
```

You can observe whether the instance is running or not, and the process id (PID) of the *postmaster* process.

4.2 start

When pg_ctl runs in the `start` mode, it tries to start the instance.

```
$ pg_ctl start
...
...
 done
server started
$
```

When you see the above message, everything works fine.

4.3 stop

When pg_ctl runs in the `stop` mode, it tries to stop the instance.

```
$ pg_ctl stop
...
...
 done
server stopped
$
```

When you see the above message, the instance is shut down, all connections to client applications are closed and no new applications can reach the database. The `stop` mode knows three different sub-modes for shutting down the instance:

- *Smart* mode waits for all active clients to disconnect.
- *Fast* mode (the default) does not wait for clients to disconnect. All active transactions are rolled back and clients are forcibly disconnected.
- *Immediate* mode aborts all server processes immediately, without a clean shutdown.

Syntax: `pg_ctl stop [-m s[mart] | f[ast] | i[mmediate] ]`

4.4 restart

When pg_ctl runs in the `restart` mode, it performs the same actions as in a sequence of `stop` and `start`.

4.5 reload

In the `reload` mode the instance reads and reloads its configuration file.

4.6 init

In the `init` mode the instance creates a complete new cluster with the 3 databases *template0* , *template1*, and *postgres*. This command needs the additional parameter `-D datadir` to know at which place in the file system it shall create the new cluster.

4.7 Automated start at boot time

In most cases, it is desired that PostgreSQL starts immediately after the server boots. Whether this happens - or not - may be configured in the file `start.conf`. Depending on the operation system, the file is located in different directories[1].

There is only one entry and its allowed values are:

- auto: automatically start/stop at server boot/shutdown time

1 `https://www.postgresql.org/docs/current/server-start.html`

- manual: do not start/stop automatically, but allow manually managing as described above
- disabled: do not allow manual startup with pg_ctlcluster (this can be easily circumvented and is only meant to be a small protection for accidents)

5 DBA Tools: psql, pgAdmin, phpAdmin, ...

Tools for database administration (DBA) tasks such as backups, restores, and cleanups are mostly not part of the SQL standard. Vendor specific database products usually include a combination of database specific tools and SQL extensions for administration purposes. PostgreSQL provides a set of both PostgreSQL specific tools and SQL extensions. The main ones are described here as well as some reliable external tools.

5.1 psql

psql is a client program which is delivered as an integral part of the PostgreSQL downloads. Similar to a bash shell it is a line-mode program and may run on the server hardware or a client. *psql* knows two kinds of commands:

- Commands starting with a backslash, eg: `\dt` to list tables. Those commands are interpreted by *psql* itself.
- All other commands are sent to the instance and interpreted there, e.g.: `SELECT * FROM mytable;`.

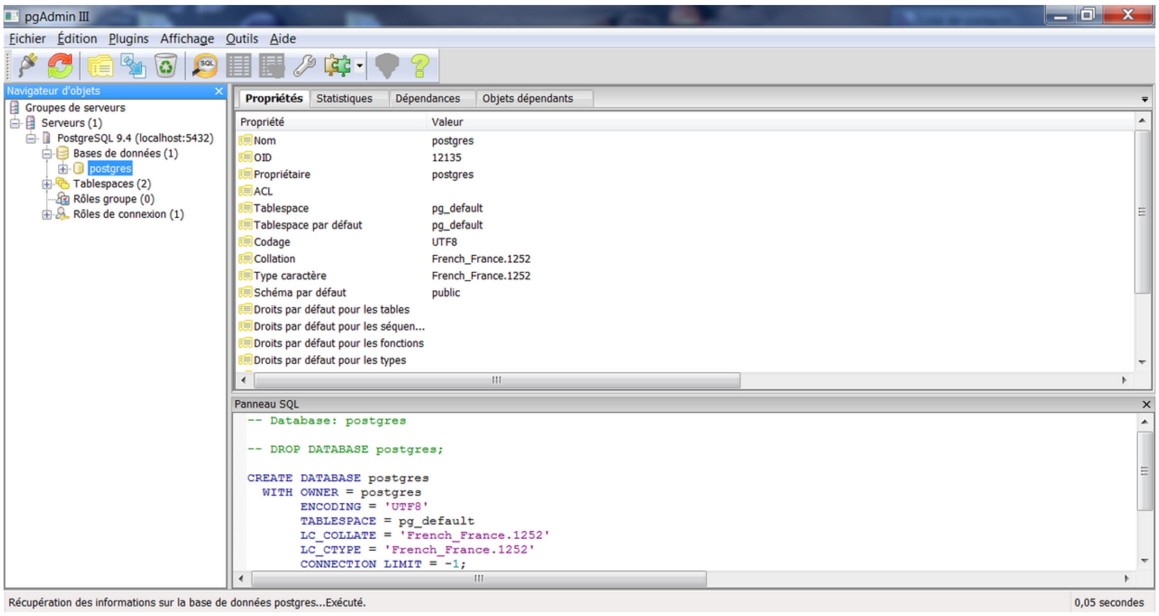
Thus it is an ideal tool for interactive and batch SQL processing. The whole range of PostgreSQL SQL syntax can be used to perform everything that can be expressed in SQL.

```
$ # start psql from a bash shell for database 'postgres' and user 'postgres'
$ psql postgres postgres
postgres=#
postgres=# -- a standard SQL command
postgres=# CREATE TABLE t1 (id integer, col_1 text);
CREATE TABLE
postgres=# -- display information about the new table
postgres=# \dt t1
        List of relations
 Schema | Name | Type  |  Owner
--------+------+-------+---------
 public | t1   | table | postgres
(1 row)
postgres=#
postgres=# -- perform a PostgreSQL specific task - as an example of a typically
 DBA action
postgres=# SELECT pg_start_backup('pitr');
 pg_start_backup
-----------------
 0/2000028
(1 row)
postgres=#
postgres=# -- terminate psql
postgres=#\q
$
```

Here are some more examples of *psql* 'backslash'-commands

- `\h` lists syntax of SQL commands
- `\h SQL-command` lists syntax of the named SQL-command
- `\?` help to all 'backslash' commands
- `\l` lists all databases in the current cluster
- `\echo :DBNAME` lists the current database (consider upper case letters). In most cases the name is part of the psql prompt.
- `\dn` lists all schemas in the current database
- `\d` lists all tables, views, sequences, materialized views, and foreign tables in the current schema
- `\dt` lists all tables in the current schema
- `\d+ TABLENAME` lists all columns and indexes in table TABLENAME
- `\du` lists all users in the current cluster
- `\dp` lists access rights (privileges) to tables, ...
- `\dx` lists installed extensions
- `\o FILENAME` redirects following output to FILENAME
- `\t` changes output to 'pure' data (no header, ...)
- `\! COMMAND` executes COMMAND in a shell (outside psql)
- `\q` terminates *psql*

5.2 pgAdmin



Propriété	Valeur
Nom	postgres
OID	12135
Propriétaire	postgres
ACL	
Tablespace	pg_default
Tablespace par défaut	pg_default
Codage	UTF8
Collation	French_France.1252
Type caractère	French_France.1252
Schéma par défaut	public
Droits par défaut pour les tables	
Droits par défaut pour les séquen...	
Droits par défaut pour les fonctions	
Droits par défaut pour les types	

```
-- Database: postgres

-- DROP DATABASE postgres;

CREATE DATABASE postgres
  WITH OWNER = postgres
       ENCODING = 'UTF8'
       TABLESPACE = pg_default
       LC_COLLATE = 'French_France.1252'
       LC_CTYPE = 'French_France.1252'
       CONNECTION LIMIT = -1;
```

Récupération des informations sur la base de données postgres...Exécuté. 0,05 secondes

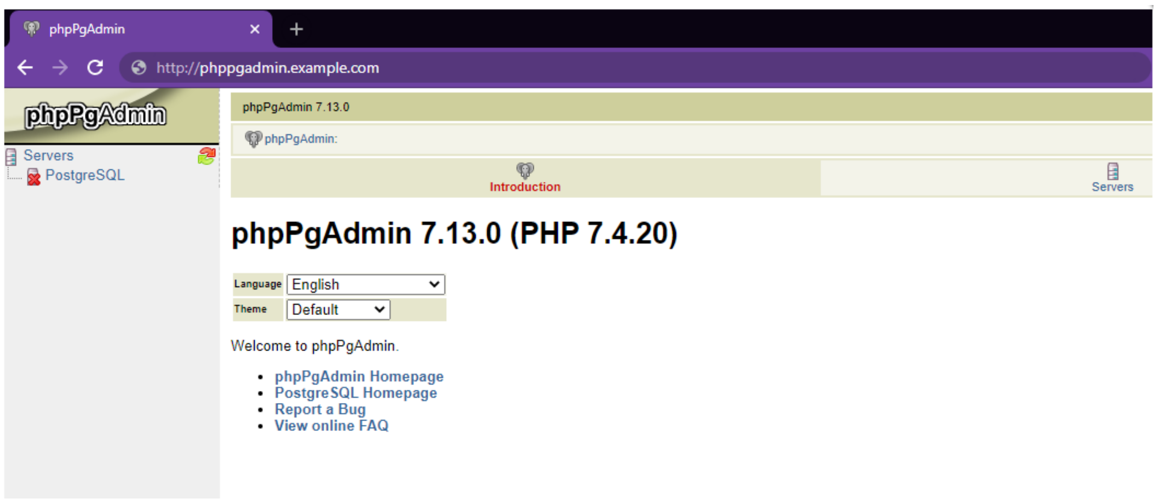
Figure 1 pgAdmin

pgAdmin is a tool with a graphical user interface for Unix, Mac OSX and Windows operating systems. In most cases it runs on a different hardware than the instance. For the major operating systems it is an integral part of the download, but it is possible to download the tool separately[1].

pgAdmin significantly extends the functionalities of *psql* with intuitive, graphical representations of database objects, eg. schemas, tables, columns, users, result lists, query execution plans, dependencies between database objects, and much more. To give you a first impression of the surface, some screenshots[2] are online.

Since 2016 *pgAdmin 3* is superseded by *pgAdmin 4*. *pgAdmin 4* is a complete re-implementation - written as a web application in Python. You can run it either on a web server using a browser, or standalone on a workstation.

5.3 phpPgAdmin



phpPgAdmin 7.13.0 (PHP 7.4.20)

Language: English
Theme: Default

Welcome to phpPgAdmin.

- phpPgAdmin Homepage
- PostgreSQL Homepage
- Report a Bug
- View online FAQ

Figure 2 PhpPgAdmin

phpPgAdmin is a graphical tool that offers features that are similar to those of *pgAdmin*. It is written in PHP, therefore you additionally need Apache and PHP packages.

phpPgAdmin is not part of the standard PostgreSQL downloads. It is distributed via GitHub[3]. The project was largely dormant over many years, but as of 2019 has been updated with support for PHP7 and PostgreSQL 13.

1 `http://www.pgadmin.org/download/`
2 `http://www.pgadmin.org/screenshots/`
3 `https://github.com/phppgadmin/phppgadmin/`

5.4 Other Tools

There are a lot of other general tools[4] and monitoring tools[5] with a GUI interface. Their functionality varies greatly from pure SQL support up to entity-relationship and UML support. Some of the tools are open/free source, others proprietary.

5.5 References

4 https://wiki.postgresql.org/wiki/Community_Guide_to_PostgreSQL_GUI_Tools

5 https://wiki.postgresql.org/wiki/Monitoring#Postgres-centric_monitoring_solutions

6 Configuration

The main configuration file is *postgresql.conf*. It is divided into several sections according to different tasks. The second important configuration file is *pg_hba.conf*, where authentication definitions are stored.

Both files reside in the special directory $PGDATA (Debian/Ubuntu) or in the main directory of the cluster (RedHat).

Some configuration items have a dynamic nature, and will take effect with a simple `pg_ctl reload`. Others require a restart of the instance `pg_ctl restart`. The comments in the default configuration files state which of the two actions has to be taken.

6.1 postgresql.conf

6.1.1 File Locations

The value of *data_directory* defines the location of the cluster's main directory. In the same way the value of *hba_file* defines the location and the name of the above mentioned *pg_hba.conf* file (host based authentication file), where rules for authentication are stored - some more details are shown below[1].

6.1.2 Connections

In the connections section you define the port number (default: 5432), with which client applications can reach the instance. In addition the maximum number of connections is defined as well as SSL, IP and TCP settings.

6.1.3 Resources

The main definition in the resources section is the size of shared buffers. It determines, how much space is reserved to "mirror" the content of data files within PostgeSQL's buffers in RAM. The predefined default value of 128 MB is relatively low.

Secondly, there are definitions for the work and the maintenance memory. They determine the RAM sizes for sorts, create index commands, ... These two RAM areas exist per connection and are used individually by them whereas the shared buffers exist only once for the whole instance and are used concurrently by multiple processes.

Additionally there are some definitions concerning *vacuum* and *background writer* processes.

1 Chapter 6.2 on page 23

6.1.4 WAL

In the WAL section there are definitions for the behaviour of the WAL mechanism.

First, you define a WAL level out of the four possibilities *minimal*, *archive*, *hot_ standby*, and *logical*. Depending on the decision, which kind of archiving or replication you want to use, the WAL mechanism will either write only basic information to the WAL files or include additional information. *minimal* is the basic method which is always required for every crash recovery. *archive* is necessary for any archiving action, which includes the point-in-time-recovery (PITR) mechanism. *hot_ standby* adds information required to run read-only queries on a standby server. *logical* adds information necessary to support logical decoding.

Additionally and in correlation to the WAL level *archive* there are definitions which describe the archive behaviour. Especially the 'archive_command' is essential. It contains a command which copies WAL files to an archive location.

6.1.5 Replication

If you use replication to a different server, you can define the necessary values for master and standby server in this section. The master reads and pays attention only on the master-definitions and the standby only to the standby-definitions (you can copy this section of 'postgres.conv' directly from master to standby). You must define the WAL level to an appropriate value.

6.1.6 Tuning

The tuning section defines the relative costs of different operations: sequential disc I/O, random disc I/O, process one row, process one index entry, process one function-call or arithmetic operation, size of effective RAM pages (PostgreSQL + OS) per process which will be available at runtime. These values are used by the query planner during its search for an optimal query execution plan. The values are not real values in sense of milliseconds or number of CPU cycles. They are only a rough guideline for the query planer and relative to each other. The real values are calculated during the query execution may differ significantly.

There is also a subsection concerning costs for the genetic query optimizer, which - in opposite to the standard query optimizer - implements a heuristic searching for optimal plans.

6.1.7 Error Logging

The error logging section defines the amount, location and format of log messages which are reported in error situations or for debugging purposes.

6.1.8 Statistics

In the statistics section you can define - among other things - the amount of statistic collection for parsing, planing and execution of queries.

6.2 pg_hba.conf

The *pg_hba.conf* file (host-based authentication) contains rules for client access to the instance. All connection attempts of clients which do not satisfy these rules are rejected. The rules restrict the connection type, client IP address, database within the cluster, user-name, and authentication method.

There are two main connection types: local connections (*local*) via sockets and connections via TCP/IP (*host*). The term *local* refers to the situation, where a client program resides on the same machine as the instance. The client may override the *local* connection and use the *host* connection type by using the TCP/IP address syntax (e.g.: 'localhost:5432') of the cluster.

The client IP address is a single IPv4 or IPv6 address or a masking of a net-segment via a CIDR mask.

The database and the client user name must be given explicitly or may be abbreviated by the key word ”ALL”.

There are different authentication methods

- *trust*: don't ask for any password
- *reject*: don't allow any access
- *password*: ask for a password
- *md5*: same as 'password', but the transfer of the password occurs MD5-encrypted
- *peer*: trust the client, if he uses the same database username as his operation system username (only applicable for local connections)

Since the pg_hba.conf records are examined sequentially for each connection attempt, the order of the records is significant. The first match between defined criteria and properties of incoming connection requests hits.

7 Backup & Recovery

7.1 Overview

Creating backups is an essential task for every database administrator. If the hardware crashes or any form of data corruption occurs, the DBA must ensure that a database can be restored with minimal data loss. PostgreSQL offers multiple strategies to support the DBA in achieving this goal.

In principle, backup technology can be divided into two classes: **cold** backups and **hot** backups. A cold backup is a backup taken when no database file is open. In the case of PostgreSQL this means that the instance must be stopped during the complete time interval of taking the backup. A hot backup is a backup taken during normal working hours. Clients can perform read and write actions in parallel to this form of backup creation.

PostgreSQL supports different types of backups:

- Cold backups are called *File System Level Backup.*
- There are two types of hot backups
  - *SQL Dump* produces SQL commands, e.g.: `INSERT`, which can re-create the database.
  - *Continuous Archiving and Point-in-Time Recovery (PITR)* uses the combination of a special backup plus all data-changes since then.

7.2 File System Level Backup

A cold backup is a backup taken when the PostgreSQL instance is **not** running. It consists of **all** files of all databases of a cluster.

There is only one way to create a consistent and therefore useful cold backup: the PostgreSQL instance must be stopped, e.g. by issuing the `pg_ctl stop` command. This will disconnect all clients from all databases of the cluster, shut down the instance, and close all files. After that, the backup can be taken by using one of the usual operating system copy-utilities (cp, tar, dd, rsync, etc.) to create a copy of all files at a secure location, e.g. at disks on a different server. Especially the following files must be copied:

- All files under the directory node where the cluster resides. The environment variable $PGDATA points to this directory and resolves to something like `.../postgres/14/data` . Use `echo $PGDATA` on the command-line, or `show data_directory;` in psql to find the directory.
- All configuration files. They may be in $PGDATA, but can also be located elsewhere. The main configuration files are: postgresql.conf, pg_hba.conf, and pg_ident.conf. Their locations can be found by running the following commands from the psql utility:

```
show config_file;
show hba_file;
show ident_file;
```

- All tablespace files. These files are located elsewhere on the file-system. Their locations can be found by looking at the symlinks in the `$PGDATA/pg_tblspc` directory:

```
cd $PGDATA/pg_tblspc
ls -lt
```

Caution One may try to backup only special parts of a cluster, eg. a huge file that represents a table, or one of the tablespaces - or the opposite: everything except the huge file. Even if the instance is shut down during the generation of such a partial copy, copies of this kind are useless. The recovery of a cold backup needs really **all** data files and meta-information files of the cluster to re-create the cluster.

Caution It is strongly recommended to verify every backup/recovery strategy on a test system to verify their reliability before implementing them on a production server. In particular, it's necessary to test the recovery steps!

►Advantages

- A cold backup is easy to generate and restore.

►Disadvantages

- A continuous 7x24 operation mode of any of the databases in the cluster is not possible.
- It is not possible to backup smaller parts of a cluster like a single database or table.
- Partial restores are not possible. Restores must include all cluster files.
- After a crash, any data changes that occur after the most recent cold backup get lost. Only the data in the backup will be restored.

►How to Recover

- Stop the instance.
- Backup the original files of the crashed cluster. They may be useful for forensic actions.
- Delete all original files of the crashed cluster.
- Copy the files of the cold backup to their original places.
- Start the instance. It should start in the normal way, without any special message.

7.3 Hot Backup

In contrast to cold backups, hot backups are taken while the instance is running and applications may change data during the backup is taken. Hot backups are sometimes called *online backups*. PostgreSQL supports two very different kinds of hot backups: First, a pure

SQL-based version, and second, a product-specific version. They are explained in the next two chapters.

7.4 SQL Dump (or: Logical Backup)

A logical backup is one of the two forms of a hot backup. It consists of data and/or metadata within the cluster, a single database, or some parts of a database. They are created by the utilities `pg_dump` or `pg_dumpall`.

The instance must run for those utilities to operate. Even though they run in parallel with other clients - possibly over a longer period of time -, they create an exact copy of the data as of the moment of their start time. For example, if an application changes some data during this period, the backup takes the old value whereas all other applications operate on the new value. This is possible because of PostgreSQL's MVCC[1] (Multi-version concurrency control) implementation which allows the existence of multiple versions of a row at the same time.

7.4.1 pg_dump

`pg_dump` works at the database level and can backup the complete database as well as some of its parts such as individual tables. It is able to dump data, schema definitions, or both. The parameters `--data-only` and `--schema-only` select the intended part.

`pg_dump` supports two output formats: *plain* (readable plain-text format) and *custom* (a binary format). The format type is chosen by the parameter `--format`. The plain-text format contains SQL commands like CREATE and INSERT. Files created in this format may be used by `psql` to restore the backed-up data. The custom format is sometimes called the *archive format.* To restore files created in this format you must use `pg_restore`.

The following diagram visualizes the cooperation of `pg_dump`, `psql` and `pg_restore`.

1 Chapter 14 on page 53

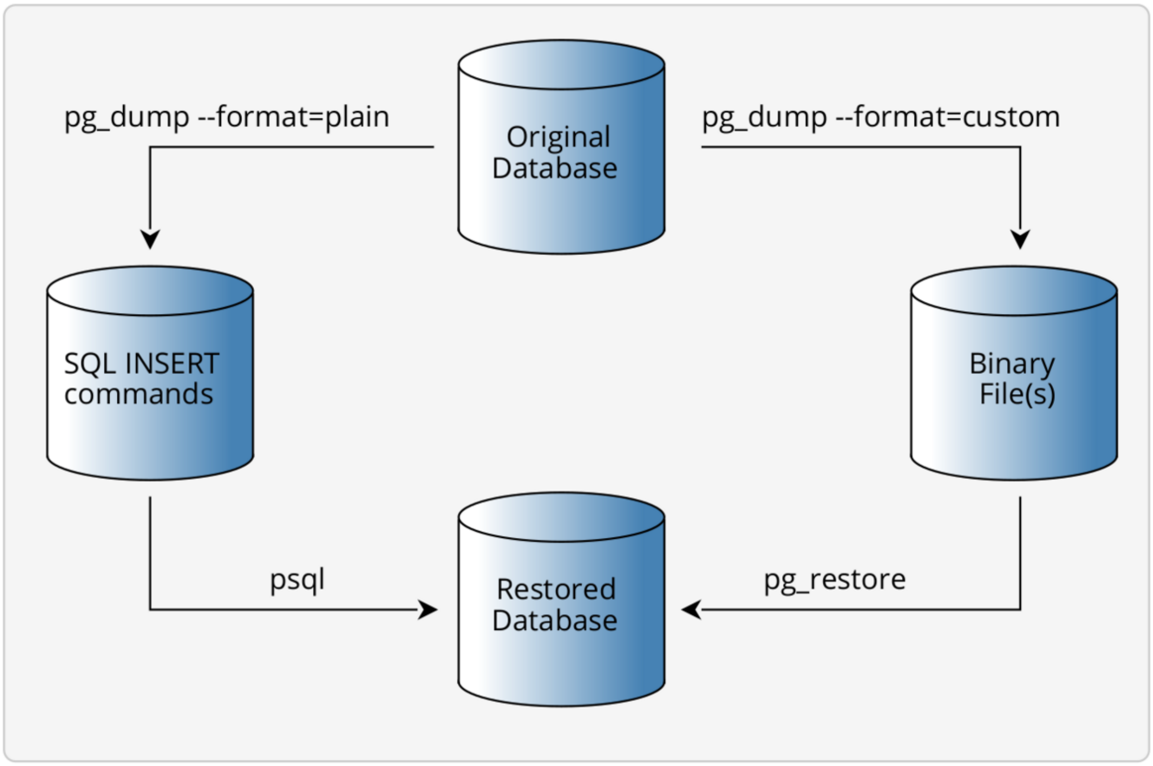


Figure 3

Some Examples:

```
$ # dump complete database 'finance' in plain-text format to a file
$ pg_dump --dbname=finance --username=boss --format=plain --file=finance.sql
$
$ # restore database content (to a different or an empty database)
$ psql --dbname=finance_x --username=boss <finance.sql
$
$
$
$ # dump table 'person' of database 'finance' in binary format to a file
$ pg_dump --dbname=finance --username=boss --table=person  \
        --format=custom --file=finance_person.archive
$
$ restore table 'person' from binary file
$ pg_restore --dbname=finance_x --username=boss        \
           --format=custom <finance_person.archive
$
```

7.4.2 pg_dumpall

The `pg_dumpall` utility works at the cluster level and calls `pg_dump` internally to dump each database of the cluster. Additionally, it dumps cluster level objects ('globals') like user/roles and their rights. If it is started without detailed parameters, it dumps the complete content of the cluster: all data and metadata of all databases plus all cluster level objects. The parameter `--globals-only` can be used to restrict its behavior to dump cluster objects only. `pg_dumpall` output is in plain-text format.

►Advantages

- Continuous 7x24 operation mode is possible.
- Small parts of the cluster or database may be backup-ed or restored.
- When you use the text format, you can switch from one PostgreSQL version to another or from one hardware platform to another.

►Disadvantages

- The text format uses much space, but it compresses well.

►How to Recover

As shown in the above diagram, the recovery process depends on the format of the dump. Text files are in standard SQL syntax. To recreate data from such files you must use `psql`. Files with the custom format have a PostgreSQL-specific binary structure and can only be used by the utility `pg_restore`.

7.5 Continuous Archiving and Point-in-Time Recovery (PITR)

This is the second form of hot backups. Such backups consist of two parts. The first one is the so-called *base backup*, which consists of a copy of all files of a cluster (similar to *File System Level Backup*). The second one consists of all data-changes since the start of the backup command. Such data-changes keep occurring with further online activities (during and after the backup generation), are stored in WAL files, and must be continuously saved ('archived') in the same way as the first part.

To understand the purpose and the technique of such backups, it's helpful to know PostgreSQL's recover-from-crash strategy. At all times and independent from any backup/recovery action, PostgreSQL maintains *Write Ahead Log (WAL)* files - primarily for crash-safety purposes. Such *WAL files* contain *log records*, which reflect all changes made to the data and the schema. Prior to transfers of changes to data files, *log records* are stored in (sequentially written) WAL files. In the case of a system crash, those *log records* are used to recover the cluster to a consistent state during the restart of the instance. The recovery process searches the timestamp of the last *checkpoint*, which is stored in the WAL files, and replays all subsequent *log records* in chronological order against the cluster. Through that action, the cluster gets recovered to a consistent state and contains all changes up to the last COMMIT.

When recovering from a backup, the overall strategy is similar to the recover-from-crash strategy: remove the files of the crashed cluster, restore them from the *base backup*, inform the recovery process (which is an integral part of the instance) how to access the archived WAL files via an operating system command, and restart the instance. The recovery part of the instance replays all *log records* from the archived *WAL files* against the (restored) database files and transfers the cluster to a consistent state. Thereafter the cluster contains all changes up to the last COMMIT before the crash.

To implement this backup strategy, three actions must be taken:

- Define all necessary parameters in *postgres.conf*.

- Generate a *base backup* with the utility `pg_basebackup`.
- Archive all arising WAL files.

If a recovery becomes necessary, you have to delete all files in the cluster, recreate the old state of the cluster by copying the backup to its original location, create a special file (*recovery.signal* or *recovery.conf*, see below: step 3) with some recovery-information (especially to what location WAL files have been archived) and restart the instance. The instance will recreate the cluster according to its parameters in *postgres.conf* and *recovery.conf* to a consistent state including all data changes up to the last COMMIT.

►Advantages

- Continuous 7x24 operation mode is possible.
- Recover with minimal data loss.
- The generated WAL files can be used for additional features like *replication*.

► Disadvantages

- *Base backups* work only on the cluster level, not on any finer granularity like database or table.
- If your database is very busy and clients change a lot of data, many WAL files may arise.

7.5.1 How to Take the Backup

Step 1

You have to define some parameters in *postgres.conf* so that WAL files contain enough data, archiving of WAL files is activated, and a copy command is defined to transfer WAL files to a fail-safe location.

```
# collect enough information in WAL files
wal_level = 'replica'
# activate ARCHIVE mode so that WAL files will be archived by the instance
archive_mode = on
# supply a system command to transfer WAL files to a failsafe location (cp, scp,
 rsync, ...)
# %p represents the pathname including filename. %f represents the filename
 only.
archive_command = 'scp %p dba@archive_server:/postgres/wal_archive/%f'
```

After the parameters are defined, you must restart the cluster: `pg_ctl restart`. The cluster will continuously generate WAL files in its subdirectory *pg_ wal* (*pg_ xlog* in Postgres version 9.x and older) in concordance with data changes in the database. When it has filled a WAL file and must switch to the next one, it will copy the old one to the defined archive location.

Step 2

You must create the so-called *base backup* with the utility `bg_basebackup`.

```
$ # take a copy (base backup) of the files of the cluster with the pg_basebackup
 utility
$ pg_basebackup --pgdata=/safe_drive/backup/
$
```

Step 3

That's all. All other activities are taken by the instance, especially the continuous copy of completely filled WAL files to the archive location.

7.5.2 How to Recover

To perform a recovery the original *base backup* is copied back and the instance is configured to perform recovery during its start.

- Stop the instance - if it is still running.
- Create a copy of the crashed cluster - if you have enough disc space. Maybe, you will need it at a later stage.
- Delete all files of the crashed cluster.
- Recreate the cluster files from the *base backup*.
- Create a special file in $PGDATA:

  - PostgreSQL prior to version 12: Create a file *recovery.conf* in $PGDATA. It must contain a command similar to: restore_command = 'scp dba@archive_server:/postgres/wal_archive/%f %p'. This copy command is the reverse of the command in *postgres.conf*, which saved the WAL files to the archive location.
  - PostgreSQL since version 12: Create an empty file *recovery.signal* in $PGDATA. Add a command similar to: restore_command = 'scp dba@archive_server:/postgres/wal_archive/%f %p' within *postgres.conf*. This copy command is the reverse of the command in *postgres.conf*, which saved the WAL files to the archive location.

- Start the instance. During startup, the instance will copy and process all WAL files found in the archive location.

The fact, that *recovery.signal* respective *recovery.conf* exists, signals the instance to perform a recovery. After a successful recovery, this file is renamed.

If you want to recover to some previous point in time prior to the occurrence of the crash (but behind the creation of the backup), you can do so by specifying this point in time. In this case, the recovery process will stop before processing all archived WAL files. This feature is the origin of the term *Point-In-Time-Recovery*.

In summary the two crucial commands for recovery (in *recovery.conf* resp. *postgres.conf*) may look like this:

```
restore_command      = 'scp dba@archive_server:/postgres/wal_archive/%f %p'
recovery_target_time = '2021-01-31 06:00:00 CET'
```

7.6 Additional Tools

There is an open-source project Barman[2], which simplifies backup and recovery steps. If you have to manage a lot of servers and instances and it becomes complicated to configure

2 https://www.pgbarman.org/

and remember all the details about the server landscape, *Barman* stores the configuration details and automates processes.

7.7 External link

PostgreSQL documentation: Backup[3]

3 https://www.postgresql.org/docs/current/backup.html

8 Architecture: Collaboration of Processes, RAM, and Files

PostgreSQL implements a client-server architecture[1]. Each `Client process` connects to one `Backend process` at the server site.

Clients do not have direct access to database files and the data stored in them. Instead, they send requests to the `Server` and receive the requested data from there. The server launches a single process for each client connection. Such a Backend process handles the client's requests by acting on the `Shared Memory`. This leads to other activities (file access, WAL, vacuum, ...) of the `Instance`. The Instance is a group of server-side processes acting on the common Shared Memory. PostgreSQL does not use threading.

At startup time, the Instance is launched by a single process denoted the `Postmaster` . It loads configuration files, allocates Shared Memory, and starts the other collaborating processes of the Instance: `Background Writer`, `Checkpointer`, `WAL Writer`, `WAL Archiver` , `Autovacuum`, `Statistics Collector`, `Logger`, and more. Subsequently, the Postmaster listens to its configured system port. In response to new client connection attempts he launches new Backend processes and delegates authentication, communication, and the handling of all further requests to them. The next figure visualizes the main aspects of RAM, processes, files, and their collaboration.

1 `https://en.wikipedia.org/wiki/client-server%20architecture`

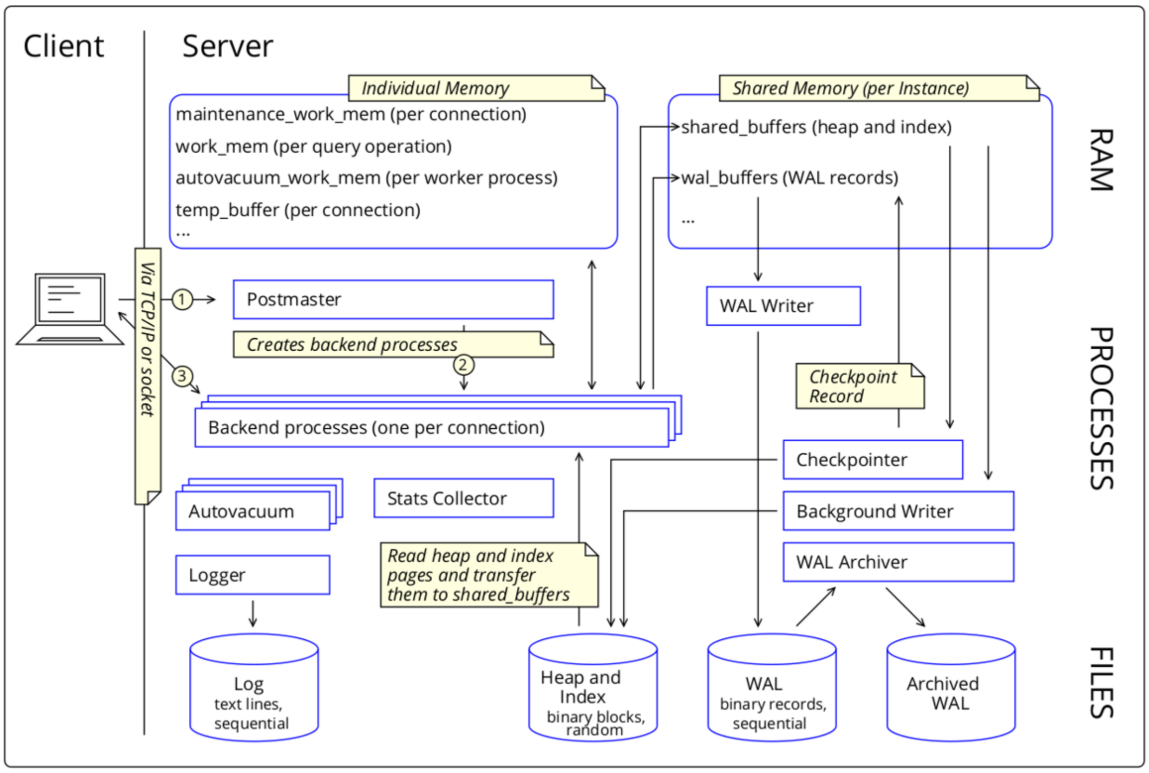

Client
Server
Individual Memory
maintenance_work_mem (per connection)
work_mem (per query operation)
autovacuum_work_mem (per worker process)
temp_buffer (per connection)
...
Shared Memory (per Instance)
shared_buffers (heap and index)
wal_buffers (WAL records)
...
RAM
Via TCP/IP or socket
1
Postmaster
Creates backend processes
2
3
Backend processes (one per connection)
WAL Writer
Checkpoint Record
Checkpointer
Background Writer
WAL Archiver
PROCESSES
Autovacuum
Stats Collector
Logger
Read heap and index pages and transfer them to shared_buffers
Log text lines, sequential
Heap and Index binary blocks, random
WAL binary records, sequential
Archived WAL
FILES

Figure 4 Internal architecture of PostgreSQL

Client requests like SELECT or UPDATE usually lead to the necessity to read or write data. This is carried out by the client's Backend process. Such I/O-activities are **not done directly on the disks**. Instead, they are done in a cache in the Shared Memory that mirrors the file pages. Accesses to such caches are much faster than accesses to disk. Read accesses affect only the cache whereas write accesses are accomplished by writing to a log, the so-called write-ahead-log or WAL.

Shared Memory is limited in size and it can become necessary to evict pages. As long as the content of such pages hasn't changed, this is not a problem. But they may be modified. Modified pages are called dirty pages (or dirty buffers) and before they can be evicted they must be written back to disk. The Background Writer processes and the Checkpointer takes care of that. They ensure that the cache is - after a short time delay - in sync with files. The synchronization from RAM to disk consists of two steps.

First, whenever the content of a page changes, a `WAL record` is created containing the delta-information (difference between the old and new content) and stored in another area of Shared Memory. During a `COMMIT` or earlier the WAL Writer process reads them and appends them to the end of the current `WAL file`. Such sequential writes are faster than writes to random positions of heap and index files. All WAL records created from one dirty page must be transferred to disk before the dirty page itself can be transferred to disk in the second step.

Second, the Background Writer process transfers dirty buffers from Shared Memory to files. Because I/O activities can block other processes, it starts periodically and acts only for a short period. Doing so, its extensive - and expensive - I/O activities are spread over time, avoiding debilitating I/O peaks. The Checkpointer process also transfers dirty buffers to file.

The Checkpointer process creates a `Checkpoint` by writing and flushing all older dirty buffers, all older WAL records, and finally a special Checkpoint record to disk. Therefore a Checkpoint is a point in the sequence of transactions at which it is guaranteed that the heap and index files have been updated with all dirty pages before that Checkpoint.

WAL files contain the changes made to the data. Such 'delta information' is used in the case of a system crash for recovery (database backup + WAL files –> database immediately before the crash). Hence, WAL files shall be duplicated and preserved at a safe place until the next database backup is taken. This is the duty of the WAL Archiver process. He can be configured to run a script that copies WAL files, as soon as they are full and a switch to the next one takes place. Of course, such copies should be done to a separate disk or server for security reasons. But it's also a good idea to store the original WAL files on a different disk than heap and index files. Such a separation boosts performance. It can be done using a symbolic link pointing from the original WAL directory to a directory at a different disk.

The Autovacuum process marks old versions of records in the heap and index files that are no longer used by any transaction as 'finally deleted'. Hence it releases the space occupied by them for reuse. The need for such a process results from the MVCC architecture.

The Statistics Collector collects counters about accesses to SQL objects like tables, rows, indexes, pages, ... and stores them in system tables.

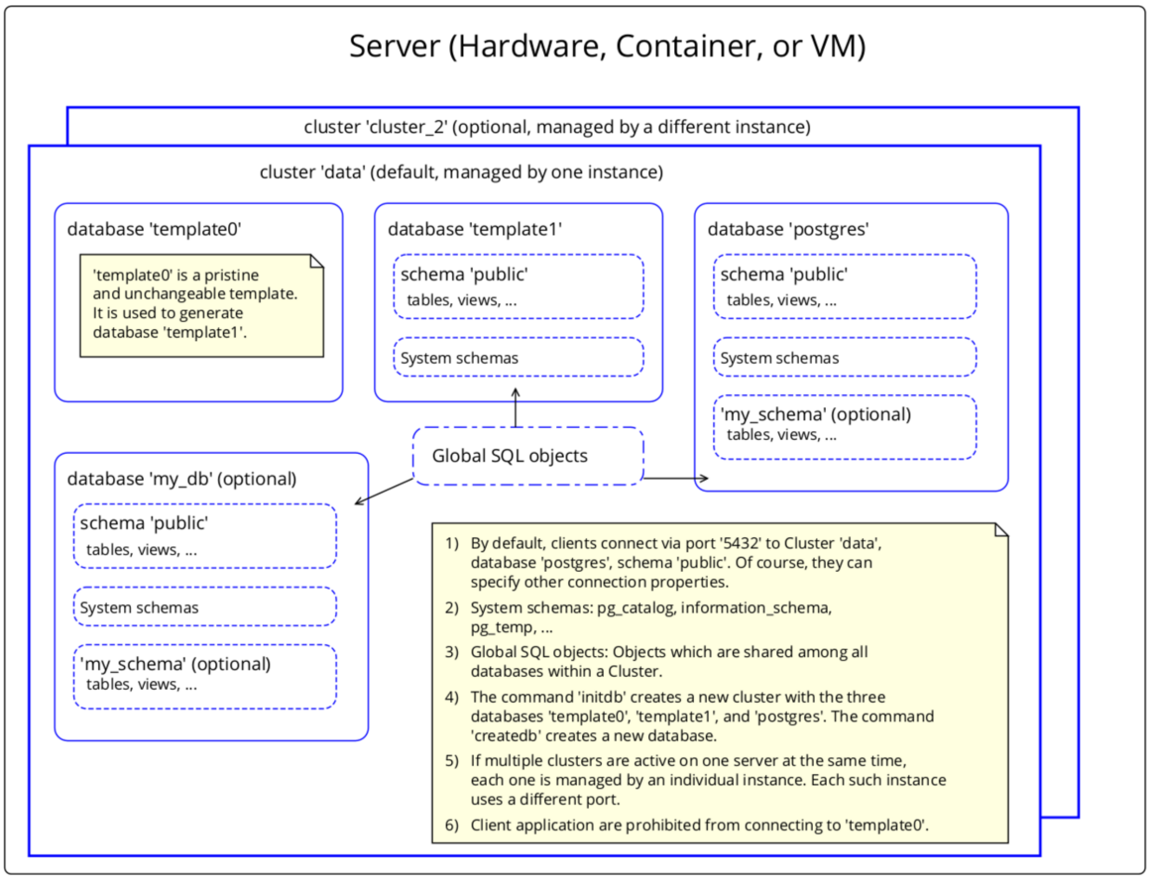
The Logger writes text lines about more or less serious events that may happen during database accesses, e.g., wrong password, no permission, long-running queries, etc. to a sequential file.

9 The logical Perspective: Cluster, Database, Schema

9.1 Overview

A `Server`, which is some hardware, a container, or a VM, contains one or more Database Clusters (`Cluster` for short). Every cluster is controlled by exactly one instance. If there are many clusters and instances on the same server, the ports of the instances must differ from each other as well as the root directories of the clusters.

Each newly created cluster contains the three `databases` *template0*, *template1*, and *postgres* , each of the three databases contain the schema *public* as well as the system schemas *pg_catalog*, *information_schema*, *pg_temp*, and some more. Tables, views, and most other SQL objects reside in such schemas. DBAs can create more clusters, databases, schemas, or SQL objects.



'template0' is a pristine and unchangeable template. It is used to generate database 'template1'.

1) By default, clients connect via port '5432' to Cluster 'data', database 'postgres', schema 'public'. Of course, they can specify other connection properties.
2) System schemas: pg_catalog, information_schema, pg_temp, ...
3) Global SQL objects: Objects which are shared among all databases within a Cluster.
4) The command 'initdb' creates a new cluster with the three databases 'template0', 'template1', and 'postgres'. The command 'createdb' creates a new database.
5) If multiple clusters are active on one server at the same time, each one is managed by an individual instance. Each such instance uses a different port.
6) Client application are prohibited from connecting to 'template0'.

Figure 5

9.2 Initialization Phase

Clusters are created with the command `initdb`. *template0* is the very first database during the creation phase of any cluster. In a second step, database *template1* is generated as a copy of *template0*, and finally database *postgres* is generated as a copy of *template1*. Later, the DBA can create more databases within that cluster, e.g.: *my_db*, with the command `createdb`. Just like at the beginning, the new database will be a copy of *template1*. Due to the unique role of *template0* as the pristine original of all other databases, no client is allowed to connect to it and modify it. But the DBA can change *template1*.

9.3 Connections

Client connections act at the database level and can access data and SQL objects within any schema of the connected database, as far as they are permitted to do so. If they need access to any object of a different database within the same or another cluster, special techniques like `foreign-data wrapper` (FDW) or `dblink` are required (or they use multiple connections and synchronize them at the client-side).

9.4 SQL objects

We use the term *SQL object* for all objects which you can create with the SQL command `CREATE ...`, e.g.: database, schema, table, view, materialized view, index, constraint, sequence, function, procedure, trigger, role, data type, domain, operator, tablespace, extension, foreign-data wrapper, and much more. Such SQL objects are arranged in a hierarchical manner:

- Database names, tablespaces, and roles (users) are known at the cluster level. E.g.: As mentioned above, a connection works at the database level. Nevertheless, when you create a new role with such a connection, the role is also known by all other databases of the same cluster.
- Extensions, e.g.: PostGIS, reside at the database level. After installing an extension, all schemas of this database can use it. But within the other databases of the same cluster, the extension is not known.
- Schemas are part of a database. Some of them are predefined.
  - *pg_catalog* is a schema with tables that describe most of the SQL objects of that database, especially all tables and views. They even describe themself. *information_schema* is a similar schema. It contains several tables and views of pg_catalog in a way that conforms to the SQL standard.
  - *public* acts as the default schema. It should not contain user-defined SQL objects. Instead, it is recommended to create one or more additional schemas to manage application-specific objects like tables or triggers. To access objects in such additional schemas they can be fully qualified, e.g. my_schema.my_table, or by changing the `search_path`.

- There are different types of SQL objects within a schema: 'relation'-like objects (table, view, materialized view, index, sequence, foreign-table), function, procedure, trigger, constraint, data type, domain, operator, and more.
  - SQL objects in one schema are different from SQL objects in different schemas, even if they use the same name, e.g.: table *t1* in *my_schema1* is different from *t1* in *my_schema2*.
  - The names of 'relation'-like objects, data types, and domains are unique within their schema: e.g.: you cannot have a table *emplyee* and a view *employee* in the same schema.

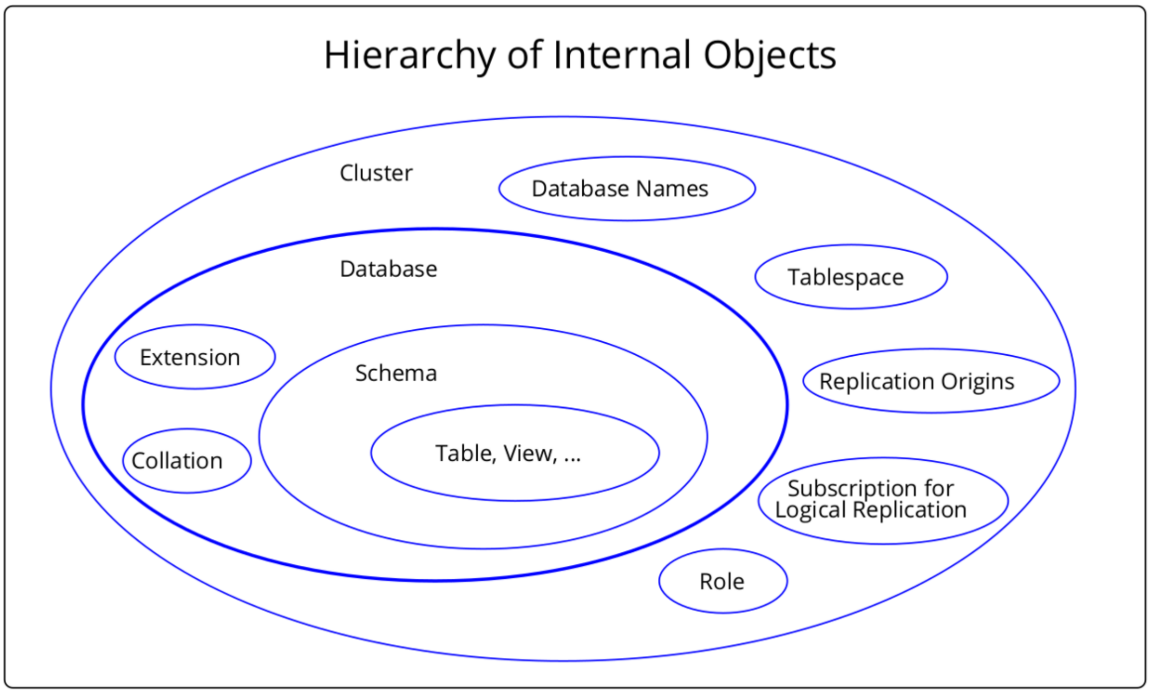


Figure 6 centre

10 The physical Perspective: Directories and Files

PostgreSQL organizes durable (persistent) data as well as volatile state information about transactions or replication actions in the file system. Every cluster has its root directory somewhere in the file system. In many cases, the environment variable *PGDATA* points to this directory. The following graphic uses *data*, which is the default, as the name of the cluster's root directory.

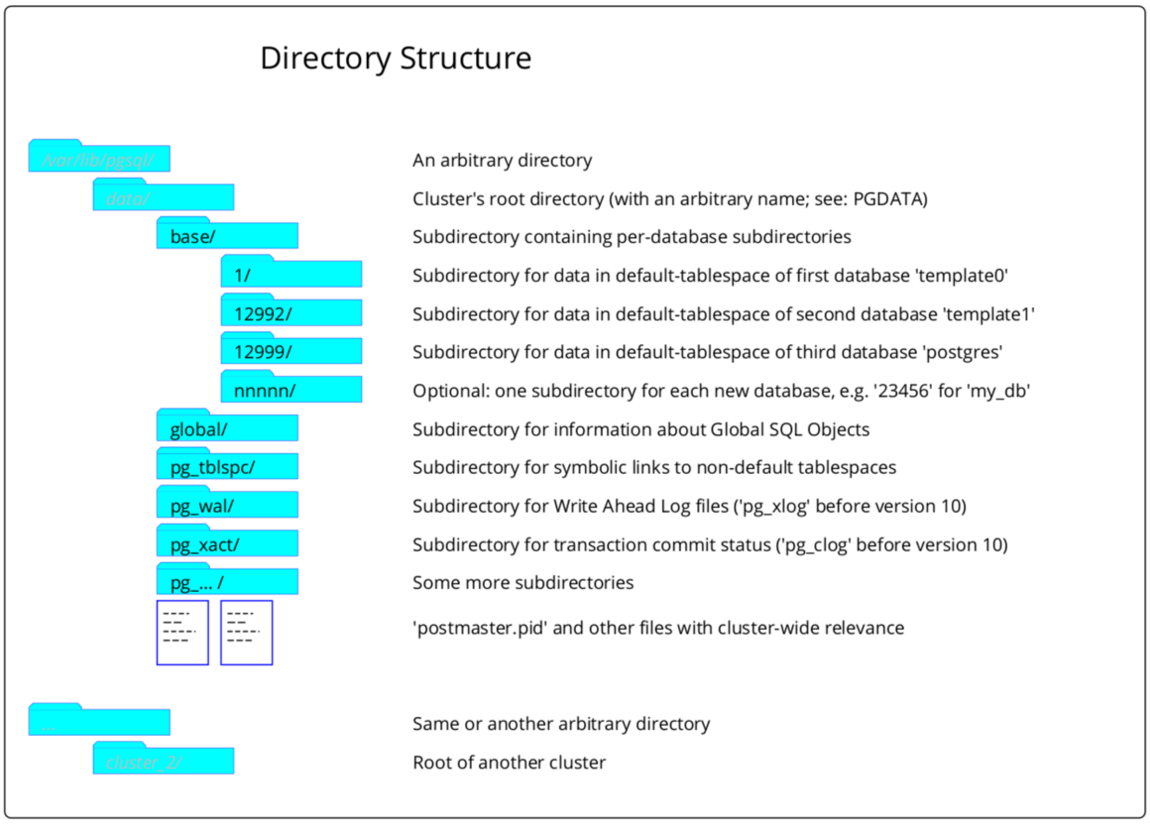

Directory Structure
/var/lib/pgsql/
An arbitrary directory
data/
Cluster's root directory (with an arbitrary name; see: PGDATA)
base/
Subdirectory containing per-database subdirectories
1/
Subdirectory for data in default-tablespace of first database 'template0'
12992/
Subdirectory for data in default-tablespace of second database 'template1'
12999/
Subdirectory for data in default-tablespace of third database 'postgres'
nnnnn/
Optional: one subdirectory for each new database, e.g. '23456' for 'my_db'
global/
Subdirectory for information about Global SQL Objects
pg_tblspc/
Subdirectory for symbolic links to non-default tablespaces
pg_wal/
Subdirectory for Write Ahead Log files ('pg_xlog' before version 10)
pg_xact/
Subdirectory for transaction commit status ('pg_clog' before version 10)
pg_... /
Some more subdirectories
'postmaster.pid' and other files with cluster-wide relevance
...
Same or another arbitrary directory
cluster_2/
Root of another cluster

Figure 7

The cluster's root directory contains many subdirectories and some files, all of which are necessary to store durable as well as temporary information. The root's name can be selected as desired, but the names of its subdirectories and files are constant respectively determined by PostgreSQL. The following paragraphs describe the most important subdirectories and files.

base contains one subdirectory per database. The names of those subdirectories consist of numbers. These are the internal Object Identifiers (OID), which are numbers to identify their definition in the System Catalog.

Within the database-specific subdirectories of *base*, there are many files: one or more for every heap and index. Again, the filenames consist of numbers. Those files are accompanied by files for the Free Space Maps (suffixed _fsm) and Visibility Maps (suffixed _vm), which contain optimization information. An example for filenames is: *3083*, *3083_fsm*, *3083_vm* .

Another subdirectory is *global*. It contains files with information about SQL Objects which are not restricted to a certain schema, but known and relevant at the schema level.

In *pg_tblspc*, there are symbolic links that point to directories that are outside of the root directory tree, e.g. at a different disk. Heap and index files of non-default tablespaces reside there. Those defined within the default tablespace reside in the database-specific subdirectories.

The subdirectory *pg_wal* contains the WAL files. They arise and grow in parallel with data changes in the cluster and remain as long as they are required for recovery, archiving, or replication.

The subdirectory *pg_xact* contains information about the status of each transaction: in_progress, committed, aborted, or sub_committed.

In the root directory, there are some files. In many cases, the configuration files of the cluster are stored here. Also, if the instance is up and running, the file *postmaster.pid* exists here (by default, but other locations are possible). It contains the process ID (pid) of the Postmaster process which has started the instance and controls it.

11 Transactions

All data-changing operations like `INSERT`, `UPDATE`, or `DELETE` must run within a surrounding construct which is called a TRANSACTION. Transactions are created with the SQL command `BEGIN` and finished with either `COMMIT` or `ROLLBACK`. During the lifetime of the transaction the changes to the database are written only preliminarily. At the end, `COMMIT` finishes the transaction regularily and commits all intended data changes, or `ROLLBACK` aborts the transaction and reverts all those preliminary changes.

In addition to this explicit usage of SQL keywords to manage transactions, some of PostgreSQL's client libraries create implicitly a new transaction if one of the data-changing operations doesn't run in an explicitly created transaction. In this case, the operation is automatically committed immediately after its execution.

```
BEGIN; -- establish a new transaction
UPDATE accounts SET balance = balance - 100.00 WHERE name = 'Alice';
UPDATE accounts SET balance = balance + 100.00 WHERE name = 'Bob';
COMMIT; -- finish the transaction

-- this UPDATE runs as the only command of an implicitly created transaction ...
UPDATE accounts SET balance = balance - 100.00 WHERE name = 'Alice';

-- ... and this one runs in another transaction
UPDATE accounts SET balance = balance + 100.00 WHERE name = 'Bob';
```

Hint: When working within a procedure or function, there is a `DECLARE ... BEGIN ... END;` construct to define 'blocks'. In this context the meaning of `BEGIN` (no semikolon after BEGIN!) differs from what is explained here. You can avoid the ambiguity by using the keywords `START TRANSACTION;` as an alternative for `BEGIN;` in the context of transactions. Besides that, `START TRANSACTION;` conforms to the SQL standard.

Transactions generate a great relief for applications. Especially for business logic that must execute many statements as a consistent unit - like the above money transfer from one bank account to another -, there is no need to take individual actions after an error occurred in the middle of a transaction. In many cases it's enough to restart the transaction or to handle errors in a unified way.

Transactions in PostgreSQL guarantee that all requirements of the ACID paradigm are fulfilled, see next chapter[1].

11.1 Sub-Transactions

Within a transaction the keyword `SAVEPOINT` defines and denotes a position, to which the transaction may be rolled-back.

1 Chapter 12 on page 45

```
-- The transaction will insert the values 1 and 3, but not 2.
BEGIN;
INSERT INTO my_table VALUES (1);
SAVEPOINT my_savepoint;
INSERT INTO my_table VALUES (2);
ROLLBACK TO SAVEPOINT my_savepoint;
INSERT INTO my_table VALUES (3);
COMMIT;
```

12 The ACID Paradigm

The ACID paradigm[1] is a cornerstone of database management systems. With respect to data modifications, the paradigm demands that transactions must fulfill certain requirements and have to guarantee that they are satisfied not only during regular operations but also in all cases of minor and major problems like mutual-locking, connection-loss, server-down, disk-full, disk-crash, ...

Please note especially that the requirements are defined at the **transaction level**. They are named:

- Atomicity
- Consistency
- Isolation
- Durability

As shown in the previous chapter every single write operation or a collection of write operations is embedded in a transaction[2]. Read operations may also be part of a transaction.

12.0.1 Atomicity

All writing operations within a transaction create a single, undividable unity. Either all of them succeed or none. Writing operations to different tables are an example of such a situation. Another example is the decrease of one person's bank account and the associated increase of another bank account during a money transfer.

12.0.2 Consistency

At the end of a transaction, the database is in a consistent state. All defined integrity rules like uniqueness, check constraints, foreign-key and primary-key definitions are fulfilled. Furthermore, all involved triggers have been successfully executed. It's possible that during the lifetime of a transaction those rules may be broken, e.g. the foreign key relationship of two nodes in a doubly-linked list[3].

In essence, a transaction transfers the database from one consistent state to another consistent state.

1 `https://en.wikipedia.org/wiki/ACID`
2 `https://en.wikibooks.org/wiki/PostgreSQL%2FTransaction`
3 `https://en.wikipedia.org/wiki/doubly%20linked%20list`

12.0.3 Isolation

In many cases, transactions run in parallel. But the database system gives them the illusion that they act one after the other. Depending on the chosen isolation level, the exact behavior of competing read and write operations may differ. Nevertheless, PostgreSQL guarantees in all cases, that read operations never block write operations and write operations never block read operations.

12.0.4 Durability

Durability guarantees that after a successful termination (COMMIT) of a transaction, the carried-out changes keep in the database, even if a significant problem like a disk crash occurs. PostgreSQL implements this by saving the data changes not only in the data files but - redundant - also in Write-Ahead-Log (WAL) files. Therefore it is recommended that data and WAL files shall be stored on different disks.

13 Visibility of Rows: Isolation Levels

13.1 Some exemplary Problems

It's obvious that every transaction 'sees' all data changes, it has been carried out during its lifetime, without problems. But there are situations where more than one process wants to read or write the same data during an overlapping time interval of their transactions or even at the same point in time, which is possible on servers with multiple CPUs or a disk array. In such cases, different types of conflicts and suspicious effects may occur.

Applications may or may not accept the effects resulting from such competing situations. They can choose different levels of *isolation* against the activities of other transactions depending on their needs. The level defines which effects they are willing to accept and which not. Higher levels mean that fewer effects can occur but the database system must work harder and that the overall throughput decreases.

Here are some examples with two transactions T_A and T_B. Both don't perform a `COMMIT` if not explicitly noted.

- T_A reads the row with `id = 1`. T_B reads the same row. T_A increases column X by 1. T_B increases the same column by 1. What will be the result? There is the danger of a 'Lost update'.
- T_A changes a value of the row with `id = 1`. What shall T_B see if it reads the same row? T_A may perform a `ROLLBACK`. (Uncommitted read[1])
- T_A reads the row with `id = 1`. T_B reads the same row, changes a value and performs a `COMMIT`. T_A reads the row again. In comparison to its first read, it will see a different value. (Non-repeatable read[2])
- T_A reads all rows with `status = 'ok'`. T_B inserts an additional row with `status = 'ok'` and performs a `COMMIT`. T_A reads all rows with `status = 'ok'` again and receives a different number of rows. (Phantom read[3])
- T_A reads and changes the row with `id = 1`. T_B reads and changes the row with `id = 2`. T_B wants to read and change the row with `id = 1`. Because T_A has not yet committed its changes, T_B must wait for T_A. T_A wants to read and change the row with `id = 2`. Because T_B has not yet committed its changes, T_A must wait for T_B. (Deadlock[4])

1 https://en.wikipedia.org/wiki/Isolation_%28database_systems%29%23Dirty_reads
2 https://en.wikipedia.org/wiki/Isolation_%28database_systems%29%23Non-repeatable_reads
3 https://en.wikipedia.org/wiki/Isolation_%28database_systems%29%23Phantom_reads
4 https://en.wikipedia.org/wiki/Deadlock

13.2 PostgreSQL's Solutions

The SQL standard describes the 3 effects (or problematic situations) 'Uncommitted read', 'Non-repeatable read', and 'Phantom read' and defines 4 levels of isolation between transactions: READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, and SERIALIZABLE. Every level is stricter than its predecessor and prevents more effects, which means e.g. that a 'Non-repeatable read' is possible in level READ COMMITTED but not in REPEATABLE READ or SERIALIZABLE.

PostgreSQL implements those levels[5]. But, as a consequence of its MVCC model, it implements some aspects a little stricter than they are demanded by the standard. If a transaction requests the level READ UNCOMMITTED, PostgreSQL handles it always as a READ COMMITTED, which leads to the overall behavior that all uncommitted changes are invisible to all other transactions at any level - only committed changes can be seen by other transactions.

13.3 Examples

The following examples act on a table *t1* with the two columns *id* and *col* and a single row.

```
DROP TABLE IF EXISTS t1;
CREATE TABLE t1 (id INTEGER, col INTEGER);
INSERT INTO t1 VALUES (1, 100);
SELECT * FROM t1;
id | col
----+-----
  1 | 100
(1 row)
```

13.3.1 Uncommitted read

The example shows that PostgreSQL solely shows committed rows to other transactions.

Transaction A

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
-- you can shorten the two commands into one:
-- BEGIN TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
UPDATE t1 SET col=101 WHERE id=1;
SELECT col FROM t1 WHERE id=1;
-- 101
```

Transaction B

```
-- 'READ UNCOMMITTED' acts equal to 'READ COMMITTED'
-- other transactions solely sees committed rows!
BEGIN TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT col FROM t1 WHERE id=1;
-- 100 (the committed one!)
```

5 https://www.postgresql.org/docs/current/transaction-iso.html

Transaction A

```
COMMIT;
SELECT col FROM t1 WHERE id=1;
-- 101
```

Transaction B

```
SELECT col FROM t1 WHERE id=1;
-- 101 (again: the committed one!)
COMMIT; -- no real effect
SELECT col FROM t1 WHERE id=1;
-- 101
```

13.3.2 Lost update

The example shows that PostgreSQL prevents 'lost update' in the lowest level of isolation - as well as in all other levels. (The table *t1* contains its original values.)

Transaction A

```
BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT col FROM t1 WHERE id=1;
-- 100
UPDATE t1 SET col=col+1 WHERE id=1;
SELECT col FROM t1 WHERE id=1;
-- 101
```

Transaction B

```
BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT col FROM t1 WHERE id=1;
-- 100
UPDATE t1 SET col=col+1 WHERE id=1;
-- UPDATE is queued and must wait for the
-- COMMIT of transaction A
.
.
```

Transaction A

```
COMMIT;
```

Transaction B

```
-- the above UPDATE executes after (!) the COMMIT
-- of transaction A
SELECT col FROM t1 WHERE id=1;
-- 102
```

Both `UPDATE` statements are executed, nothing gets lost.

Please note that transaction B is an example for a 'non-repeatable read' (see below) because the isolation level is '(UN)COMMITTED READ'. First, it reads the value '100' with its `SELECT` command. Next, it reads '101' with its `UPDATE` command - after COMMIT of transaction A - and increases it to '102'. If the isolation level would be 'REPEATABLE

READ', transaction B would receive the error message 'could not serialize access due to concurrent update' as PostgreSQL's reaction to the `UPDATE` request.

13.3.3 Non-repeatable read

The example shows a non-repeatable read. (The table *t1* contains its original values.)

Transaction A

```
BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT col FROM t1 WHERE id=1;
-- 100
```

Transaction B

```
BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;
UPDATE t1 SET col=101 WHERE id=1;
SELECT col FROM t1 WHERE id=1;
-- 101
COMMIT;
```

```
SELECT col FROM t1 WHERE id=1;
-- 101 (same transaction, but different value)
-- ' ISOLATION LEVEL REPEATABLE READ' or
-- 'SERIALIZATION' will avoid such an effect
```

13.3.4 Phantom read

The example shows a phantom read. (The table *t1* contains its original values.)

Transaction A

```
BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT col FROM t1 WHERE id>0;
-- 1 row: 100
```

Transaction B

```
BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;
INSERT INTO t1 VALUES (2, 200);
COMMIT;
SELECT col FROM t1 WHERE id>0;
-- 2 rows: 100 and 200
```

```
SELECT col FROM t1 WHERE id>0;
-- 2 rows: 100 and 200
-- (same transaction, same query, but different rows)
-- ' ISOLATION LEVEL SERIALIZABLE'
-- will avoid such an effect
```

13.3.5 Dead lock

The example shows a dead lock. (The table *t1* contains two rows.)

```
DELETE FROM t1;
INSERT INTO t1 VALUES (1, 100);
INSERT INTO t1 VALUES (2, 200);
```

Transaction A

```
BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;
UPDATE t1 SET col=col+1 WHERE id=1;
SELECT col FROM t1 WHERE id=1;
-- 101
```

Transaction B

```
BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;
UPDATE t1 SET col=col+1 WHERE id=2;
SELECT col FROM t1 WHERE id=2;
-- 201

UPDATE t1 SET col=col+1 WHERE id=1;
.
.
-- must wait for COMMIT/ROLLBACK of transaction A
```

Transaction A

```
UPDATE t1 SET col=col+1 WHERE id=2;
-- must wait for COMMIT/ROLLBACK of transaction B.
--
-- PostgreSQL detects the deadlock and performs a
-- ROLLBACK to overcome the circular situation.
-- message: "ERROR:  deadlock detected ..."
```

Transaction B

```
-- processing goes on with a 'success message'
SELECT col FROM t1 WHERE id>0;
-- 101
-- 201
-- no UPDATEs from transaction A. They were
-- ROLLBACK-ed by PostgreSQL.
```

14 Multiversion Concurrency Control (MVCC): Implementation of the ACID Paradigm

In nearly all cases, PostgreSQL databases must support many clients, which want to add or change data, at the same time. This makes it necessary to protect concurrently running requests from each other - preferably without blocking them. Situations may occur where two clients want to change the same row at the same time or that one client wants to revoke (rollback) his changes while another client may still have tried to read the newest version.

Imagine an Online shop offering the last copy of an article. Two clients display the article at their user interface. After a while, but at the same time, both clients decide to put the article into their shopping cart or even to buy it. Both have seen the article, but only one can be allowed to buy it. The database must enforce an order of the requests, permit the write access to one of them, block the other from writing, and inform the blocked client that the data has been changed by a different process and shall be re-read.

PostgreSQL implements a sophisticated technique to handle concurrent accesses that avoids locking: Multiversion Concurrency Control (MVCC). **Instead of locking a row, the MVCC technique creates a new version of that row when a data change takes place.** "The main advantage of using the MVCC ... rather than locking is that in MVCC locks acquired for querying (reading) data do not conflict with locks acquired for writing data, and so reading never blocks writing and writing never blocks reading. PostgreSQL maintains this guarantee even when providing the strictest level of transaction isolation through the use of ... Serializable Snapshot Isolation (SSI)[1]". [2]

The implementation of MVCC is based on transaction IDs (XID). Every transaction in a cluster gets a unique sequential number as its ID. Every `INSERT`, `UPDATE`, or `DELETE` command stores the XID in `xmin` or `xmax` within the affected rows. `xmin`, `xmax`, and some more are system columns contained in every row. Both are not visible with the usual `SELECT * FROM ...` command. But you can read them with commands like `SELECT xmin, xmax, * FROM ...`. The column `xmin` contains the XID of the transaction which has created this version of the row and `xmax` contains the XID of the transaction which has deleted this version, or zero if the version is not deleted.

So, what's going on in detail when write accesses take place? The following graphic shows details concerning `xmin`, `xmax`, and the regular application data.

1 `https://en.wikipedia.org/wiki/Snapshot_isolation`

2 MVCC in PostgreSQL [`https://www.postgresql.org/docs/current/mvcc-intro.html`

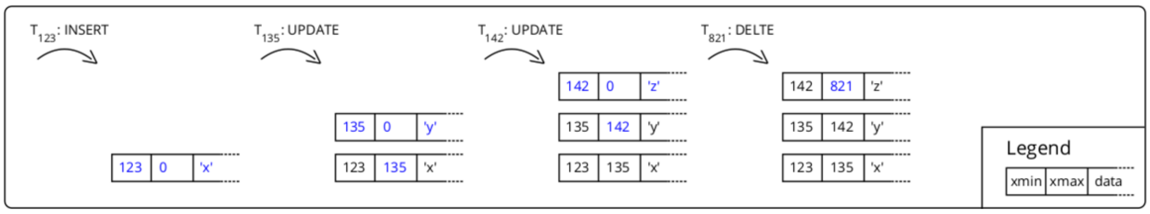


Figure 8 centre

An `INSERT` command creates the very first version of a row. Besides its application data 'x', this version contains the ID of the creating transaction 123 in `xmin` and 0 in `xmax`. `xmin` indicates that the version exists since transaction 123 and the value 0 in `xmax`indicates that it is currently not deleted.

Somewhat later, transaction 135 executes an `UPDATE` of this row by changing the application data from 'x' to 'y'. According to the MVCC principles, the data in the old version of the row is not changed. The value 'x' remains as it was before. Only `xmax` changes to 135. Now, this version is treated as valid exclusively for transactions with XIDs from 123 to 134. In addition to preserve the data in the old version, the `UPDATE` creates a new version of the complete row with its XID in `xmin`, 0 in `xmax`, and 'y' in the application data (plus all other application data from the old version). This new row version is visible to all future transactions. (Internally, an `UPDATE` command acts as a `DELETE` command followed by an `INSERT` command.)

All subsequent `UPDATE` commands behave in the same way as the first one: they put their XID in `xmax` of the current version, create a new version with their XID in `xmin` and 0 in `xmax`.

Finally, a row may be deleted by a `DELETE` command. Even in this case, all versions of the row including the newest one remain in the database - nothing is thrown away. Only `xmax` of the last version is set to the XID of the `DELETE` transaction, which indicates that it is only visible to transactions with older XIDs - in this example from 142 to 820.

In summary, the MVCC technology creates more and more versions of the same row in the table's heap file and leaves them there, even after a `DELETE` command. Only the youngest version is relevant for all future transactions. But the system must also preserve some of the older ones for a short time because they could still be requested by transactions that had started before the deleting transaction and hence have a smaller XID. Over time, also the older ones goes out of scope for ALL transactions and therefore become ultimately unnecessary. Nevertheless, they do exist physically on the disk and occupy space. They are called *dead rows*[3] and are part of the so-called *bloat.*

Please keep in mind:

- `xmin` and `xmax` indicate the range in which row versions are visible for transactions. This range doesn't imply any direct temporal meaning. The sequence of XIDs reflects only the sequence of transactions' begin events.
- Internally, an `UPDATE` command acts in the same way as a `DELETE` command followed by an `INSERT` command.

3 Chapter 15 on page 57

- Nothing is removed - with the consequence that the database occupies more and more disk space. It is obvious that this behavior has to be corrected in some way. The next chapter explains how `vacuum` and `autovacuum` fulfill this task.

So far this is only a raw description of the principles of MVCC. The implementation considers more problems, e.g.:

- Changes may be revoked by a `ROLLBACK` command.
- After some time the sequence of XIDs may start from zero (*wrap-around*). In this case `xmax` can be smaller than `xmin`.
-

14.1 Note

XIDs are sequences (with a reserved value to handle *wrap-around* in pre-9.4 PostgreSQL versions). PostgreSQL knows some configuration parameters concerning transactions and their XIDs with names like *xxx_ age*, e.g.: *vacuum_freeze_ min_ age*. For such parameters, the 'age' doesn't specify a period of time but represents a certain number of transactions, e.g., 100 millions.

14.2 References

15 Vacuum

Eliminating Bloat

As we have seen in the MVCC[1] chapter, the database tends to occupy more and more disk space caused by *bloat*: over time more and more logically deleted but physically existing old versions of rows arise within heap and index files. This chapter explains how the SQL command **VACUUM** and the automatically running **Autovacuum processes** clean up files and thereby prevent their endless growth.

One process of the Instance is the Autovacuum daemon. It continuously monitors the state of all databases based on values that are collected by the Statistics Collector and starts Autovacuum processes whenever it detects certain situations, e.g.: a huge number of modifications within a table. This leads to the intended dynamic behavior of PostgreSQL: Only when it is necessary, Autovauum cleans up the files. In addition, client processes can issue the SQL command VACUUM at any time. DBAs do this interactively when they recognize critical situations, or they start it in periodically running batch jobs. In most cases, this is not necessary because of the constantly running Autovacuum daemon.

The central value to determine which of the physically existing row versions are no longer needed is `xmax`, which shows what transaction has deleted the row. The elimination operation must evaluate it against several criteria which must all apply:

- `xmax` must be different from zero because a value of zero indicates that the row version is not deleted.
- `xmax` must contain an XID which is older than the oldest XID of all currently running transactions. That guarantees that no existing or upcoming transaction will have read or write access to this row version.
- The transaction of `xmax` must be committed. If it is still running or was rollback-ed, this row version is treated as valid (not deleted).
- If there is a situation that the row version is part of multiple transactions, more actions must be taken.

When the vacuum operation detects such an outdated row version, it marks its space as free for future use of write actions, optimizes the physical arrangement of the remaining versions on the page, and removes index tuples pointing to the removed row version. But only in rare situations (or in the case of VACUUM FULL), this space is released to the operating system. In most cases, it remains occupied by the database and will be used by future INSERT, UPDATE or DELETE commands. Hence, even with successful running Autovacuum, the size of files does not shrink; but they have more respectively huger 'holes' where coming data can be stored. Only after this 'hole-space' is exhausted (per page), it gets necessary to claim new disc space from the operating system to store new data.

1 Chapter 14 on page 53

An exception to this conservative behavior is the SQL command VACUUM FULL. It creates a new file at the operating system level, copies all valid row versions to it with no extra space by ignoring the 'holes', and deletes the old file. But it is slower and requires an exclusive lock on the affected tables.

Because vacuum operations typically are I/O intensive, which can hinder other activities, Autovacuum avoids performing many vacuum operations in bulk. Instead, it carries out many small actions with time delays in between. The SQL command VACUUM runs immediately and without any time delay.

More Actions

VACUUM, as well as Autovacuum, don't just eliminate *bloat*. They perform additional tasks for minimizing future I/O activities of themselves as well as of other processes. This extra work can be done in a very efficient way since in most cases the expensive physical access to pages has taken place anyway. The additional operations are:

- *Freeze*: It marks certain row versions as frozen. This means that they are treated as 'valid' (visible) forever, independent from the wraparound problem (see later).
- *Visibility Map and Free Space Map*: It logs information about the state of the handled pages in two additional files, the *Visibility Map* and the *Free Space Map*.
- *Statistics*: Similar to the *Statistics Collector* it collects statistics about the number of rows per table, the distribution of values, and so on, as the basis for decisions of the query planner.

15.1 External links

PostgreSQL Documentation concerning VACUUM[2]

2 `https://www.postgresql.org/docs/current/routine-vacuuming.html`

16 Wraparound and Freeze

16.1 Two Fundamental Problems

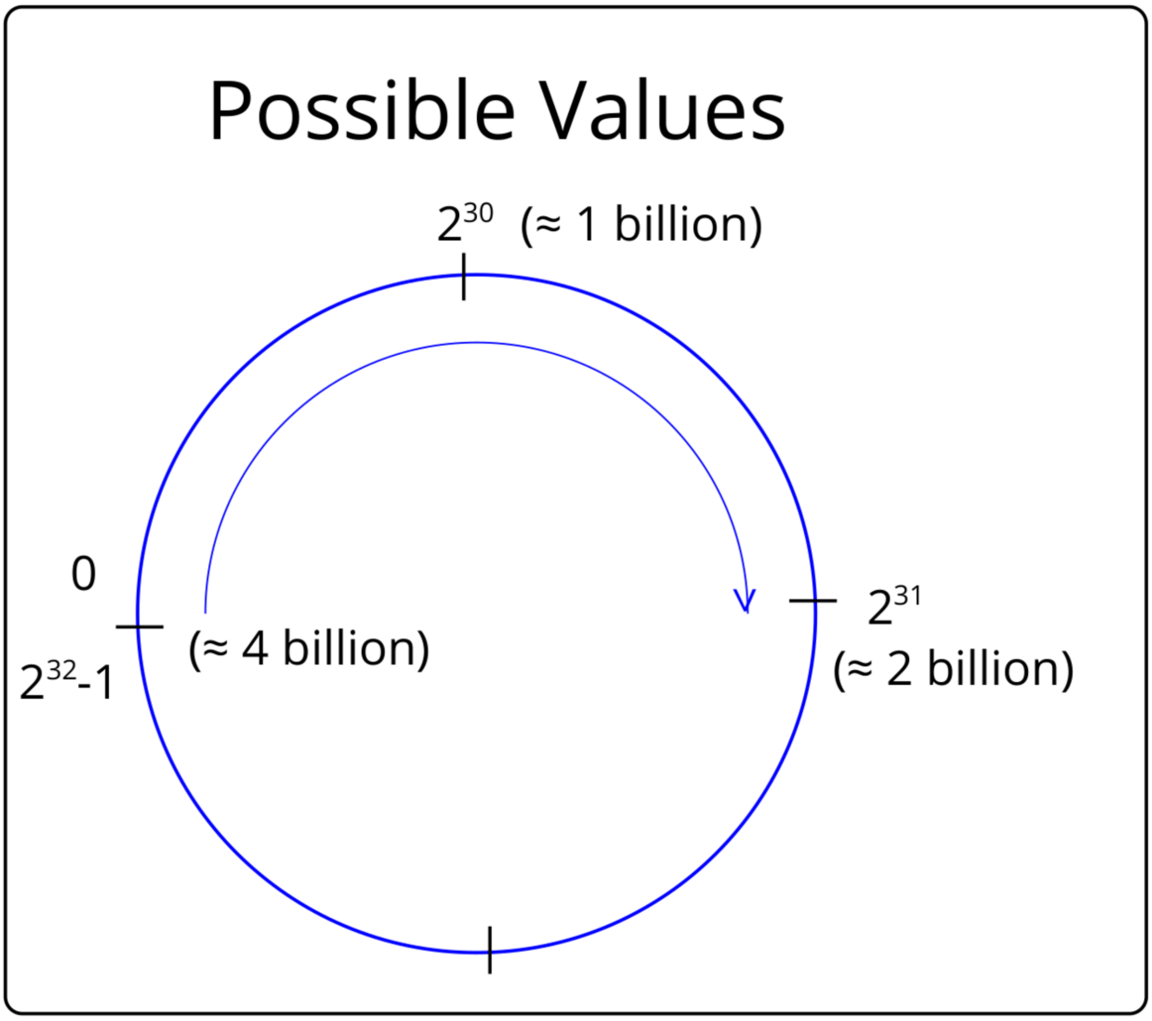


Figure 9

Transactions are identified by ids which are realized as unsigned 32-bit integers and named XID. Transactions and their XIDs are known at the cluster level, covering all databases. Similar to sequences, XIDs are incremented by +1 for every new transaction. Sooner or later, this limited space of 2^{32} numbers is exhausted, and it becomes necessary to restart the sequence from the beginning (the values 0, 1, and 2 are skipped because they are reserved for particular purposes). This restart of XIDs is called a *wraparound* and each cycle an *epoch*.

It's unlikely that more than 2^{32} transactions exist in a cluster at the same time or that a single transaction lasts for such a long time that its XID collides with the same value of the next cycle. At first glance, this cyclic usage of the '2^{32} Universe' seems to be safe and easy to implement. Nevertheless, huge problems will arise with this simple strategy. The reason is that XIDs are stored in system columns within every row (see `xmin, xmax` in the MVCC[1] chapter). And rows stay for a very long time in the database, in many cases forever.

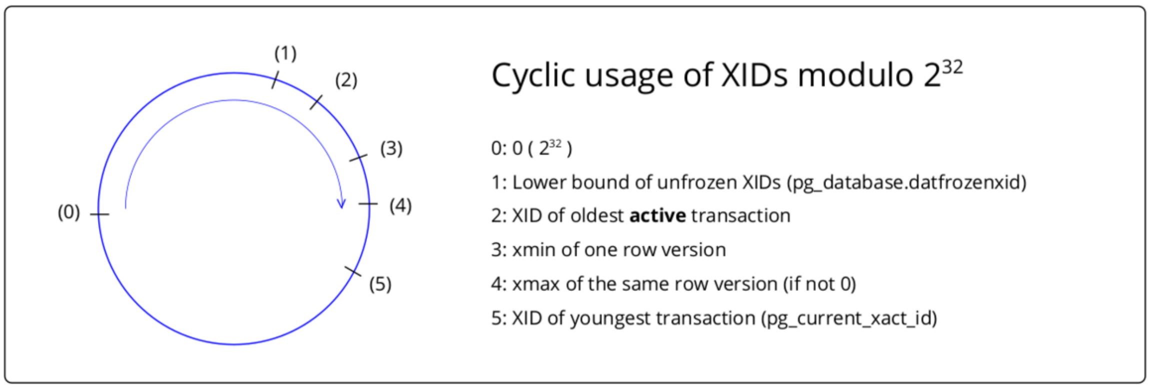

(0)
(1)
(2)
(3)
(4)
(5)
Cyclic usage of XIDs modulo 2^32
0: 0 (2^32)
1: Lower bound of unfrozen XIDs (pg_database.datfrozenxid)
2: XID of oldest **active** transaction
3: xmin of one row version
4: xmax of the same row version (if not 0)
5: XID of youngest transaction (pg_current_xact_id)

Figure 10

XID Collision

The first problem is that after a wraparound, the next XIDs (3, 4, 5, ...) may collide with XIDs of the previous epoch. They are no longer unique because system columns of old rows may contain the same values. But transactions must be able to decide whether retrieved rows are modified (by other transactions) after their own start-time or a long time ago. We call this first problem **XID Collision**.

Sudden Death

The second problem correlates with MVCC[2] and the timeline of transactions. Rows may exist in multiple versions. When a transaction modifies a row and stays alive a little longer - no COMMIT or ROLLBACK because of more activities -, other processes shall 'see' the version of the row as of the start-time of the transaction and not the uncommited modification. Hence, a mechanism must hide the ongoing changes and give other transactions the feeling of a stable data environment. The system realizes this by considering additional criteria with every SQL command, especially - but not only - the system column `xmin`.

You can imagine of those additional criteria that the system silently supplements every query with the predicate `xmin < my_xid`. (This is only an illustration in pseudo-code, the real implementation is different.) It guarantees that changes happening after the start of the requesting transaction are invisible to it.

So far, so good.

1 Chapter 14 on page 53
2 Chapter 14 on page 53

But what will happen after a wraparound? The next transactions will have very small XIDs, e.g., '5'. And what will be the result of an `xmin < 5`? Near nothing. All rows with `xmin` between 5 and $2^{32}-1$ are no longer part of any query. In contrast to the situation a few moments ago, all data suddenly disappeared. It is still in the database, but it is unreachable. We call this second problem **Sudden Death**.

16.2 Solution

Step 1: At the conceptual level, the full '2^{32} Universe' is divided into two halves of 2^{31} numbers. One split point is the current transaction id `pg_current_xact_id` (it was called `txid_current` before PostgreSQL version 13) and the other is the opposite side of the circle `pg_current_xact_id +` 2^{31} (or `pg_current_xact_id -` 2^{31} what is the same). So, the split points are not fixed values but follow dynamically the ongoing of new transactions. One halve represents the previously used and therefore exhausted XIDs; the other halve such XIDs, which are - per definition - free. They will be allocated in the future. Please note the dynamic aspect: with every new transaction in the cluster `pg_current_xact_id` and the border between 'past / future' moves forward. This is a metaphor of an endless walk through time where after some time the old problems will be forgotten or at least idealized.

The idea can be realized by a modification of the above `xmin < my_xid` predicate to an `if/else` block:

```
if (my_xid < $2^{31}$)
  return rows with: xmin < my_xid OR  xmin > my_xid + $2^{31}$
else
  return rows with: xmin < my_xid AND xmin > my_xid + $2^{31}$
```

Of course, this is a simplification and many other criteria like COMMIT status, 'is deleted', and other things must be considered. It focuses purely on the aspect of the 'past / future' metaphor.

Note: With this algorithm the 'critical point' changes from 0 resp. 2^{32} to `pg_current_xact_id +` 2^{31}. It is called the *wraparound point* and the line between `pg_current_xact_id +` 2^{31} and `pg_current_xact_id` the *wraparound horizon*.

Step 2: The outlined algorithm ensures the visibility of 50% of all possible XIDs. But what's going on with the others? As mentioned, rows may stay in the database forever holding very old XIDs in `xmin`. This halve must also be considered. The idea of how to access the complete range of possible XIDs is to complement the previous algorithm with the introduction of a flag that marks certain rows as 'visible forever' (respectively visible until the next write operation to them). This marking is not possible as long as one or more transactions potentially get write access to them. Fortunately, the sequence of new XIDs goes strictly forward, and overtime transactions with old XIDs finish. PostgreSQL does not only know, which is the current XID `pg_current_xact_id`, but also which is the **oldest active** XID per connection (*pg_stat_activity.backend_xmin*), and which is the lower bound of all unfrozen XIDs per table (*pg_class.relfrozenxid*) and per database (*pg_database.datfrozenxid*). Rows with `xmin` older than `oldest(pg_stat_activity.backend_xmin)` are candidates for such a flag. No running transaction has or will get write access to them, only newer ones. According to MVCC[3], they will create a new row version, this one keeps as it is.

It is one of the two main duties of VACUUM to perform this freezing. It marks the identified rows with a flag in their header `t_infomask` as 'visible forever'. From this point on no comparisons with `xmin` take place. The rows are always treated as visible, even if they are part of the 'future'. This marking is called FREEZE and the status of the row FROZEN.

Now the algorithm for retrieving rows changes to:

```
-- 'frozen' rows will always be returned
if (my_xid < $2^{31}$)
  return rows with: frozen OR
                    (xmin < my_xid OR  xmin > my_xid + $2^{31}$)
else
  return rows with: frozen OR
                    (xmin < my_xid AND xmin > my_xid + $2^{31}$)
```

With this extension, the two mentioned problems are solved. The system can generate XIDs even after a wraparound without risk of collision with old XIDs. The old ones may exist, but they are not touched in any way. Second, the algorithm finds all relevant XIDs whether there was a wraparound or not.

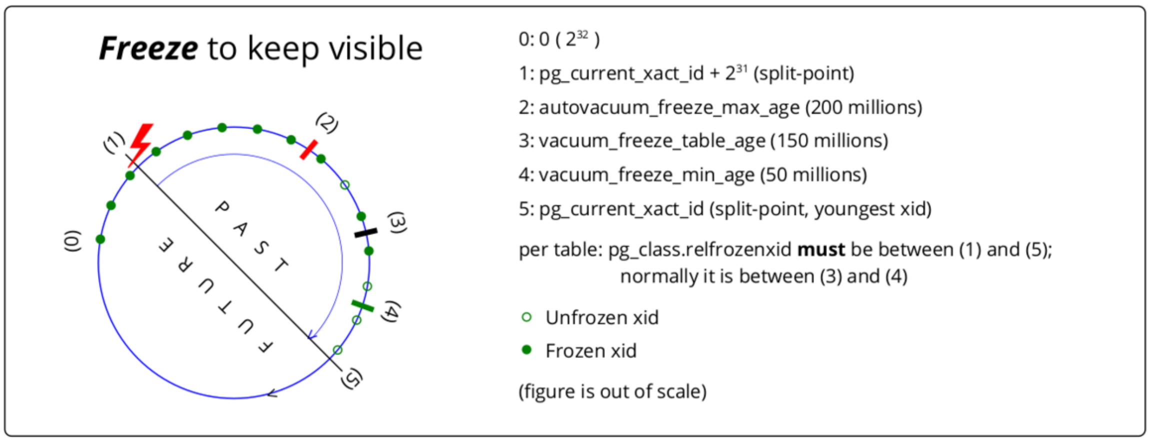

Freeze to keep visible
(0)
(1)
(2)
(3)
(4)
(5)
P A S T
F U T U R E
0: 0 (2^{32})
1: pg_current_xact_id + 2^{31} (split-point)
2: autovacuum_freeze_max_age (200 millions)
3: vacuum_freeze_table_age (150 millions)
4: vacuum_freeze_min_age (50 millions)
5: pg_current_xact_id (split-point, youngest xid)
per table: pg_class.relfrozenxid **must** be between (1) and (5); normally it is between (3) and (4)
○ Unfrozen xid
● Frozen xid
(figure is out of scale)

Figure 11

16.3 Wraparound Failure

It is possible that a transaction - intentionally or by an error in an application - stays alive for a long time. Over time its XID becomes the oldest one in the complete cluster and can be retrieved from *pg_ stat_ activity.backend_ xmin*. As long as this situation continues, the gap between the ongoing *wrapping point* `pg_current_xact_id + ` 2^{31} and *pg_ stat_ activity.backend_ xmin* gets smaller and smaller. If the gap would close completely, we would see all the problems described at the beginning of the chapter. This is called *wraparound failure* and must be avoided under all circumstances. VACUUM is doing its

3 Chapter 14 on page 53

best to freeze as many rows as possible. But if a long-living transaction prevents freezing and the size of the gap falls below a certain limit, VACUUM runs in an 'aggressive mode' and works on all pages of affected tables, independent from the above-mentioned values; if also this fails, the cluster stops the creation of new transactions and prevents further write actions.

16.4 Details

Note: These details can be skipped by a novice reader without losing the context of the ongoing chapters.

To freeze any row version, VACUUM must check several criteria:

- `xmax` must be zero because only non-deleted rows can be visible forever.
- `xmin` must be older than all currently existing transactions `oldest(pg_stat_activity.backend_xmin)`. This guarantees that no existing transaction can modify or delete the version.
- The transactions of `xmin` and `xmax` must be committed.

At what point in time does the freeze operation take place? Please note that there are configuration parameters with names like `xxx_age`. They define distances - mostly to `pg_current_xact_id` -, where the actions of VACUUM shall start. 'age' in this context doesn't imply a certain period of time, it's always a pure number that counts transactions, e.g., 50 million. Please also note, that VACUUM always reads complete physical pages and works on the row versions found there.

- When a client issues the SQL command VACUUM with its FREEZE option. In this case, all pages of the affected tables are processed that are marked in the Visibility Map[4] as potentially having unfrozen rows.
- When a client issues the SQL command VACUUM without any option and there are XIDs older than *vacuum_freeze_table_age* (default: 150 million) minus *vacuum_freeze_min_age* (default: 50 million). As before, all pages are processed that are marked in the Visibility Map to potentially have unfrozen rows.
- When an Autovacuum process runs. Such a process acts in one of two modes: In the *normal mode* it skips pages with row versions that are younger than *vacuum_freeze_min_age* (default: 50 million) and works only on pages where all XIDs are older. The skipping of young XIDs prevents work on such pages, which are likely to be changed by one of the future SQL commands. The process switches to an *aggressive mode* if it recognizes that for the processed table the oldest XID exceeds *vacuum_freeze_table_age* (default: 150 million). In this *aggressive mode*, Autovacuum processes all pages of the affected table.

VACUUM and Autovacuum know to which value the oldest unfrozen XID has moved forward per table and logs the value in *pg_class.relfrozenxid*. The distance between this value and the `pg_current_xact_id` split point becomes smaller (there are potentially unfrozen rows), and the distance to the *wraparound point* `pg_current_xact_id + 2`^{31}

4 Chapter 17 on page 65

becomes larger (there are only frozen rows). That is how the freezing follows the moving 'past' / 'future' horizon.

Note: Before version 9.4 of PostgreSQL, the freeze algorithm stored the value '2' (FrozenTransactionId) in `xmin` instead of setting a flag in `t_infomask`.

17 Visibility Map and Free Space Map

Every table is stored in a separate disk file. The names of such files consist of numbers which are the internal Object Identifiers (OID) of the table used in the System Catalog. Each such file is accompanied by a file for its *Visibility Map* and another one for its *Free Space Map*. They have the same names expanded by the suffix '_vm' respectively '_fsm'. An example for such a triplet of filenames is: 3083, 3083_vm, and 3083_fsm.

The two additional files contain metainformation to optimize I/O activities to the original file - especially for vacuuming and freezing, but also for other write activities. Because I/O works per physical page, the metainformation addresses complete physical pages, not any part of pages like rows or versions.

Visibility Map

The Visibility Map contains two flags — stored as two bits — for each page of the original file. The first bit indicates that the associated page contains only valid row versions, i.e. there is no bloat to be vacuumed. The second bit indicates that the page only contains already frozen row versions.

Please consider two details. First, in most cases a page contains many rows or row versions. However, both flags are associated with the page, not with an individual row or row version. The flags are set only under the condition that they are valid for ALL row versions stored on the page. Second, since there are only two bits per page (2 bits correlate to 8 kiloBytes, which is a relation of about 1 : 32.000), the Visibility Map is considerably smaller than the original file.

VACUUM and Autovacuum set the flags. Every write operation on any row version of the page clears the flags.

The Visibility Map helps VACUUM and Autovacuum to save unnecessary I/O. When the first bit is set, it's unnecessary to read the original page and check its content for removing bloat. It's clear that there is no bloat. Correspondingly, if VACUUM or Autovacuum have to perform freezing of rows, they can skip pages where the second bit signals that the page only contains already frozen row versions - no freeze is necessary.

Free Space Map

The Free Space Map tracks the amount of free, unused space per page. It is organized as a highly condensed B-tree of (rounded) free space size per page.

VACUUM and Autovacuum change the Free Space Map according to their write operations (marking of row versions as 'obsolete' and rearranging the physical layout of pages). Other write operations consult the Free Space Map to locate pages with enough free space for the intended write operations and change the Free Space Map afterward.

18 Write-Ahead-Log (WAL)

WAL (Write Ahead Logging) files are files, where PostgreSQL stores changed data values in a binary format. It is additional information and in this respect it is redundant to the information in the database files. WAL files can be interpreted as a specific kind of 'diff' files.

WAL files are used for:

- Recreation of a consistent state of the database after a system crash
- Backup and restore with the technique Continuous archiving[1]
- Replication

18.1 Usage

Writing to WAL files is very fast as they are written always sequentially. This is particularly true for traditional discs with rotating panes and moving read/write heads. In contrast to WAL files database files are organized in special structures like trees, which possibly must be reorganized during write operations or which contain pointers to other blocks at far positions. Thus writes to database files are much slower.

For the mentioned performance reasons, when a client requests a write operation like `UPDATE` or `DELETE` the modifications to the data are done in a special sequence and - in some parts - asynchronously to the client requests. First, data is written and flushed to WAL files. Second, it is stored in shared buffers in RAM. Finally, it is written from shared buffers to database files. The client doesn't wait until the end of all operations. After the first two very fast actions, he is informed that his request is completed. The third operation is performed asynchronously at a later (or prior) point in time.

18.2 Removal

WAL files are collected in the directory *pg_wal* (*pg_xlog* in PostgreSQL versions prior to version 10). Depending on the write activities on the database the total size of all WAL files may increase dramatically. Therefore the system must delete them when they are no longer needed. WAL files are available for deletion after the changes in the shared buffers (which correlate to the content of the WAL files) are flushed to the database files. As it is guaranteed that this criterion is met after a `CHECKPOINT`, there are some dependencies between WAL file delete operations and `CHECKPOINT`s:

1 Chapter 7 on page 25

- You can define a limit for the total size of all files in the directory: `max_wal_size`. If it is reached, PostgreSQL performs an automatic `CHECKPOINT` operation.
- You can define a `checkpoint_timeout` in seconds. No later than this number of seconds, PostgreSQL performs an automatic `CHECKPOINT` operation.

In both cases the shared buffers get written to disc, a checkpoint-record is written to the actual WAL file and all older WAL files are ready to be deleted.

The deletion of WAL files may be prevented by other criteria, especially by failing archive commands. `max_wal_size` is a soft limit and can silently be exceeded by the system in such situations.

19 Client-Server Communication: The Client Side

Before a client program like createdb, psql, pg_dump, vacuumdb, ... can perform any action on a database, it must establish a connection to that database. To do so, it must provide concrete values for the essential boundary conditions.

- The IP address or DNS name of the server, where the instance is running.
- The port on this server, to whom the instance is listening. (The combination of IP address and port identifies the instance. Multiple instances at the same IP address are possible as long as the port is different.)
- The name of the database within the instance (respective within the cluster).
- The name of the user (= role) with which the client program wants to work.
- The password of this user.

You can specify these values in three different ways:

- As explicit parameters of the client program.
- As environment variables.
- As a fixed line of text in the special file *pgpass*.

19.1 Parameters

You can specify the parameters in the usual short (-) or long (–) format of createdb, psql, pg_dump, vacuumdb, and other standard PostgreSQL command-line tools.

```
$ # Example
$ psql -h www.dbserver.com --port=5432   ....
```

The parameter names and their meanings are:

Short Form	**Long Form**	**Meaning**
-h	–host	IP or DNS
-p	–port	port number (default: 5432)
-d	–dbname	database within the cluster
-U	–username	name of the user

If necessary, the client program will prompt for the password.

19.2 Environment Variables

As an alternative to the parameter passing, you can define environment variables within your shell.

Environment Variable	**Meaning**
PGHOST	IP or DNS
PGPORT	port number (default: 5432)
PGDATABASE	database within the cluster
PGUSER	name of the user
PGPASSWORD	password of this user (not recommended)
PGPASSFILE	name of a file where those values are stored as plain text, see below (default: .pgpass)

19.3 File 'pgpass'

Instead of using parameters or environment variables as shown above you can store those values in a file. Use one line per definition in the form:

```
hostname:port:database:username:password
```

The default filename on UNIX systems is `~/.pgpass` and on Windows: `C:\Users\MyUser\AppData\Roaming\postgresql\pgpass.conf`. On UNIX systems the file protections must disallow any access of world or group: `chmod 0600 ~/.pgpass`.

You can create the file with any text editor. This is not necessary if you use pgAdmin. pgAdmin creates the file automatically after a successful connection and stores the actual connection values.

20 Client-Server Communication: The Server Side

20.1 Protocol

All access to data is done by server (or backend) processes, to which client (or frontend) processes must connect to. In most cases instances of the two process classes reside on different hardware, but it's also possible that they run on the same computer. The communication between them uses a PostgreSQL-specific protocol, which runs over TCP/IP or over UNIX sockets. It is implemented in the C library *libpq*. For every incoming new connection the backend process (sometimes called the *postmaster*- process) creates a new *postgres* backend process. This backend process gets part of the *PostgeSQL instance*, which is responsible for data accesses and database consistency.

The protocol handles the authentication process, client request, server responses, exceptions, special situations like a NOTIFY, and the final regular or irregular termination of the connection.

20.2 Driver

Most client programs - like *psql* - use this protocol directly. Drivers like ODBC, JDBC (type 4), Perl DBI, and those for Python, C, C++, and much more are also based on *libpq*.

You can find an extensive list of drivers at the postgres wiki [1] and some more commercial and open source implementations at the 'products' site [2].

20.3 Authentication

Clients must authenticate themselves before they get access to any data. This process has one or two stages. During the first - optional - step the client gets access to the server by satisfying the operating system hurdles. This is often realized by delivering a public ssh key. The authentication with PostgeSQL is a separate, independent step using a database-username, which may or may not correlate to an operating system username. PostgreSQL stores all rules for this second step in the file *pg_hba.conf*.

pg_hba.conf stores every rule in one line, one rule per line. The lines are evaluated from the top of ph_hba.conf to bottom and the first matching line applies. The main layout of these lines is as follows

1 Driver Wiki `https://wiki.postgresql.org/wiki/List_of_drivers`
2 Commercial and open source driver `http://www.postgresql.org/download/products/2/`

```
local  DATABASE  USER          METHOD  [OPTIONS]
host   DATABASE  USER  ADDRESS  METHOD  [OPTIONS]
```

Words in upper case must be replaced by specific values. Lower case words like *local* and *host* are key words. They decide, for which kind of connection the rule shall apply: *local* for clients residing at the same computer as the backend (they use UNIX sockets for the communication) and *host* for clients at different computers (they use TCP/IP). There is one notable exception. In the former case clients can use the usual TCP/IP syntax `--host=localhost --port=5432` to switch over to use TCP/IP. Thus the *host* syntax applies for them.

DATABASE and USER have to be replaced by the name of the database and the name of the database-user, for which the rule will apply. In both cases the key word ALL is possible to define, that the rule shall apply to all databases and respectively all database-users.

ADDRESS must be replaced by the hostname or the IP adress plus CIDR mask[3] of the client, for which the rule will apply. IPv6 notation is supported.

METHOD is one of the following. The thereby defined rule (=line) applies, if database/user/address combination is the first matching combination in pg_hba.conf.

- trust: The connection is allowed without a password.
- reject: The connection is rejected.
- password: The client must send a valid user/password combination.
- md5: Same as 'password', but the password is encrypted.
- ldap: It uses LDAP as the password verification method.
- peer: The connection is allowed, if the client is authorized against the operation system with the same username as the given database username. This method is only supported on local connections.

There are some more techniques in respect to the METHOD.

Some examples:

```
# joe cannot connect to mydb - eg. with psql -, when he is logged in to the
 backend.
local  mydb  joe  reject

# bill (and all other persons) can connect to mydb when they are logged in to
 the
# backend  without specifying any further password.  joe will never reach this
 rule, he
# is rejected by the rule in the line before. The rule sequence is important!
local  mydb  all  trust

# joe can connect to mydb from his workstation '192.168.178.10', if he sends
# the valid md5 encrypted password
host  mydb  joe  192.168.178.10/32 md5

# every connection to mydb coming from the IP range 192.168.178.0 -
 192.168.178.255
# is accepted, if they send the valid md5 encrypted password
host  mydb  all  192.168.178.0/24 md5
```

3 https://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing%23CIDR_notation

For the DATABASE specification there is the special keyword REPLICATION. It denotes the streaming replication process. REPLICATION is not part of ALL and must be specified separately.

20.4 References

21 Security within the Database: Roles, Users, Privileges

21.1 Roles

PostgreSQL supports the concept of `roles` [1] to handle security issues within the database. Roles are independent from operating system user accounts (with the exception of the special case peer authentication[2] which is defined in the pg_hba.conf file).

The concept of roles subsumes the concepts of individual users and groups of users with similar rights. A role can be thought of as either an individual database user, or a group of database users, depending on how the role is set up. Thus the outdated SQL command `CREATE USER ...` is only an alias for `CREATE ROLE ...`. Roles have certain privileges on database objects like tables or functions and can assign those privileges to other roles. Roles are global across a cluster - not per individual database.

Often individual users, which shall have identical privileges, are grouped together to a user group and the privileges are granted to that group.

```
-- ROLE, in the sense of a group of individual users or other roles
CREATE ROLE group_1 ENCRYPTED PASSWORD 'xyz';
-- assign some rights to the role
GRANT SELECT ON table_1 TO group_1;
-- ROLE, in the sense of some individual users
CREATE ROLE adam LOGIN ENCRYPTED PASSWORD 'xyz';  -- Default is NOLOGIN
CREATE ROLE anne LOGIN ENCRYPTED PASSWORD 'xyz';
-- the link between user group and individual users
GRANT group_1 TO adam, anne;
```

With the `CREATE ROLE` command you can assign the privileges *SUPERUSER, CREATEDB, CREATEROLE, REPLICATION* and *LOGIN* to that role. With the `GRANT` command you can assign access privileges to database objects like tables. The second purpose of the `GRANT` command is the definition of the group membership.

In addition to the roles created by the database administrator there is always the special role PUBLIC, which can be thought of as a role which is a member of all other roles. Thus, privileges assigned to PUBLIC are implicitly given to all roles, even if those roles are created at a later stage.

21.1.1 List existing roles

Roles can be listed with the following commands.

1 Concept of roles `http://www.postgresql.org/docs/current/static/user-manag.html`
2 Chapter 20.3 on page 71

With SQL, this will display an additional set of postgreSQL default roles that group together sets of common access levels:

```
SELECT rolname FROM pg_roles;
```

or the psql command:

```
\du
```

21.2 Users

```
select * from postgres.pg_catalog.pg_user
```

21.3 References

22 Replication

Replication is the process of transferring data changes from one or many databases (master) to one or many other databases (standby) running on one or many other nodes. The purpose of replication is

- High Availability: If one node fails, another node replaces him and applications can work continuously.
- Scaling: The workload demand may be too high for one single node. Therefore, it is spread over several nodes.

22.1 Concepts

PostgreSQL offers a bunch of largely mutually independent concepts for use in replication solutions. They can be picked up and combined - with only few restrictions - depending on the use case.

Events

- With *Trigger Based Replication* a trigger (per table) starts the transfer of changed data. This technique is outdated and not used.
- With *Log Based Replication* such information is transferred, which describes data changes and is created and stored in WAL files anyway.

Shipping

- *WAL-File-Shipping Replication* (or *File-based Replication*) denotes the transfer of completely filled WAL files (16 MB) from master to standby. This technique is not very elegant and will be replaced by *Streaming Replication* over time.
- *Streaming Replication* denotes the transfer of log records (single change information) from master to standby over a TCP connection.

Primary parameter: 'primary_conninfo' in recovery.conf on standby server.

Format

- In *Physical Format* the transferred WAL records have the same structure as they are used in WAL files. They reflect the structure of database files including block numbers, VACUUM information and more.
- The *Logical Format* is a decoding of WAL records into an abstract format, which is independent from PostgreSQL versions and hardware platforms.

Primary parameter: 'wal_level=logical' in postgres.conf on master server.

Synchronism

- In *Asynchronous Replication* data is transferred to a different node without waiting for a confirmation of its receiving.
- In *Synchronous Replication* the data transfer waits - in the case of a COMMIT - for a confirmation of its successful processing on the standby.

Primary parameter: 'synchronous_standby_names' in postgres.conf on master server.

Standby Mode

- *Hot*: In *Hot Standby Mode* the standby server runs in 'recovery mode', accepts client connections, and processes their read-only queries.
- *Warm*: In *Warm Standby Mode* the standby server runs in 'recovery mode' and doesn't allow clients to connect.
- *Cold*: Although it is not an official PostgreSQL term, *Cold Standby Mode* can be associated with a not running standby server with log-shipping technique. The WAL files are transferred to the standby but not processed until the standby starts up.

Primary parameter: 'hot_standby=on/off' in recovery.conf on standby server.

Architecture

In contrast to the above categories, the two different architectures (*Master/Standby* and *Multi-Master*) are not strictly distinct from each other. For example, if you focus on atomic replication channels of a *Multi-Master* architecture, you will also see a *Master/Standby* replication.

- The *Master/Standby* architecture denotes a situation, where one or many standby nodes receive change data from one master node. In such situations standby nodes may replicate the received data to other nodes, so they are master and standby at the same time.
- The *Multi-Master* architecture denotes a situation, where one or many standby nodes receive change data from many master nodes.

22.2 Configuration

There are 3 main configuration files:

- 'postgres.conf'
- 'pg_hba.conf'
- 'recovery.conf'

The 'postgres.conf' is the main configuration file. It is used to configure the master site. Additional instances can exist on standby sites.

The 'pg_hba.conf' is the security and authentication configuration file.

The 'recovery.conf' was optional and contained restore and recovery configurations. As of PostgreSQL-12 it is no longer used and its existence will prevent the server from starting. The recovery.conf settings as of PostgreSQL-12 can be set in postgres.conf.

Because the great number of possible combinations of concepts and correlating configuration values may be confusing at the beginning, this book will focus on a minimal set of initial configuration values.

Shipping: WAL-File-Shipping vs. Streaming

WAL files are generated anyway because they are necessary for recovery after a crash. If they are - additionally - used to shipp information to a standby server, it is necessary to add some more information to the files. This is activated by choosing 'replica' or 'logical' as a value for wal_level.

```
# WAL parameters on MASTER's postgres.conf
wal_level=replica                  # 'archive' | 'hot_standby' in versions
prior to PG 9.6
archive_mode=on                    # activate the feature
archive_command='scp ...'          # the transfer-to-standby command (or to an
archive location, which is the original purpose of this command)
```

If you switch the shipping technique to streaming instead of WAL-file you must not deactivate WAL-file generating and transferring. For safety reasons you may want to transfer WAL files anyway (to a platform different from the standby server). Therefore, you can retain the above parameters in addition to streaming replication parameters.

The streaming activities are initiated by the standby server. When he finds the file 'recovery.conf' during its start up, he assumes that it is necessary to perform a recovery. In our case of replication he uses nearly the same techniques as in the recovery-from-crash situation. The parameters in 'recovery.conf' advice him to start a so-called WAL receiver process within its instance. This process connects to the master server and initiates a WAL sender process over there. Both exchange information in an endless loop whereas the standby server keeps in 'recovery mode'.

The authorization at the operating system level shall be done by exchanging ssh keys.

```
#  Parameters in the STANDBY's recovery.conf
standby_mode=on   # activates standby mode
# How to reach the master:
primary_conninfo='user=<replication_dbuser_at_master>
host=<IP_of_master_server> port=<port_of_master_server>
                 sslmode=prefer sslcompression=1 krbsrvname=...'
# This file can be created by the pg_basebackup utility, see below
```

On the master site there must be a privileged database user with the special role REPLICATION:

```
CREATE ROLE <replication_dbuser_at_master> REPLICATION ...;
```

And the master must accept connections from the standby in general and with a certain number of processes.

```
# Allow connections from standby to master in MASTER's postgres.conf
listen_addresses ='<ip_of_standby_server>'         # what IP address(es) to
listen on
max_wal_senders = 5   # no more replication processes/connections than this
number
```

Additionally, authentication of the replication database user must be possible. Please notice that the key word ALL for the database name does not include the authentication of the replication activities. 'Replication' is a key word of its own and must be noted explicitly.

```
# One additional line in MASTER's pg_hba.conf
# Allow the <replication_dbuser> to connect from standby to master
host  replication   <replication_dbuser>   <IP_of_standby_server>/32    trust
```

Now you are ready to start. First, you must start the master. Second, you must transfer the complete databases from the master to the standby. And at last you can start the standby. Just as the replication, the transfer of the databases is initiated at the standby site.

```
pg_basebackup -h <IP_of_master_server> -D main --wal-methode=stream
--checkpoint=fast -R
```

The utility *pg_ basebackup* transfers everythink to the directory 'main' (shall be empty), in this case it uses the streaming methode, it initiates a checkpoint at the master site to enforce consistency of database files and WAL files, and due to the -R flag it generates previous mentioned recovery.conf file.

Format: Physical vs. Logical

The decoding of WAL records from their physical format to a logical format was introduced in PostgreSQL 9.4. The physical format contains - among others - block numbers, VACUUM information and it depends on the used character encoding of the databases. In contrast, the logical format is independent from all these details - conceptually even from the PostgreSQL version. Decoded records are offered to registered streams for consuming.

This logical format offers some great advantages: transfer to databases at different major release levels, at different hardware architectures, and even to other writing master. Thus multi-master-architectures are possible. And additionally it's not necessary to replicate the complete cluster: you can pick single database objects.

In release 9.5 the feature is not delivered with core PostgreSQL. You must install some extensions:

```
CREATE EXTENTION btreee_gist;
CREATE EXTENSION bdr;
```

As the feature is relative new, we don't offer details and refer to the documentation[1]. And there is an important project Bi-Directional Replication[2], which is based on this technique.

1 http://www.postgresql.org/docs/current/static/logicaldecoding.html
2 http://bdr-project.org/docs/stable/index.html

Synchronism: synchron vs. asynchron

The default behaviour is asynchronous replication. This means that transferred data is processed at the standby server without any synchronization with the master, even in the case of a COMMIT. In opposite to this behaviour the master of a synchronous replication waits for a successful processing of COMMIT statements at the standby before he confirms it to its client.

The synchronous replication is activated by the parameter 'synchronous_standby_names'. Its values identify such standby servers, for which the synchronicity shall take place. A '*' indicates all standby server.

```
# master's postgres.conf file
synchronous_standby_names = '*'
```

Standby Mode: hot vs. warm

As long as the standby server is running, he will continuously handle incoming change information and store it in its databases. If there is no necessity to process requests from applications, he shall run in warm standby mode. This behaviour is enforced in the recovery.conf file.

```
# recovery.conf on standby server
hot_standby = off
```

If he shall allow client connections, he must start in hot standby mode. In this mode read-only access from clients are possible - write actions are denied.

```
# recovery.conf on standby server
hot_standby = on
```

To generate enough information on the master site for the standby's hot standby mode, its WAL level must also be replica or higher.

```
# postgres.conf on master server
wal_level = replica
```

22.3 Typical Use Cases

We offer some typical combinations of the above-mentioned concepts and show its advantages and disadvantages.

22.3.1 Warm Standby with Log-Shipping

In this situation a master sends information about changed data to a standby using completely filled WAL files (16 MB). The standby continuously processes the incoming information, which means that the changes made on the master are seen at the standby over time.

To build this scenario, you must perform steps, which are very similar to Backup with PITR[3]:

- Take a physical backup exactly as described in Backup with PITR[4] and transfer it to the standby.
- At the master site `postgres.conf` must specify `wal_level=replica; archive_mode=on` and a copy command to transfer WAL files to the standby site.
- At the standby site the central step is the creation of a `recovery.conf` file with the line `standby_mode='on'`. This is a sign to the standby to perform an 'endless recovery process' after its start.
- `recovery.conf` must contain some more definitions: `restore_command, archive_cleanup_command`

With this parametrisation the master will copy its completely filled WAL files to the standby. The standby processes the received WAL files by copying the change information into its database files. This behaviour is nearly the same as a recovery after a crash. The difference is, that the recovery mode is not finish after processing the last WAL file, the standby waits for the arrival of the next WAL file.

You can copy the arising WAL files to a lot of servers and activate warm standby on each of them. Doing so, you get a lot of standbys.

22.3.2 Hot Standby with Log-Shipping

This variant offers a very valuable feature in comparison with the warm standby scenario: applications can connect to the standby and send read requests to him while he runs in standby mode.

To achieve this situation, you must increase `wal_level` to `hot_standby` at the master site. This leads to some additional information in the WAL files. And on the standby site you must add `hot_standby=on` in `postgres.conf`. After its start the standby will not only process the WAL files but also accept and response to read-requests from clients.

The main use case for hot standby is load-balancing. If there is a huge number of read-requests, you can reduce the masters load by delegating them to one or more standby servers. This solution scales very good across a great number of parallel working standby servers.

Both scenarios *cold/hot with log-shipping* have a common shortage: The amount of transferred data is always 16 MB. Depending on the frequency of changes at the master site it

3 Chapter 7.4.2 on page 28
4 Chapter 7.4.2 on page 28

can take a long time until the transfer is started. The next chapter shows a technique which does not have this deficiency.

22.3.3 Hot Standby with Streaming Replication

The use of files to transfer information from one server to another - as it is shown in the above log-shipping scenarios - has a lot of shortages and is therefore a little outdated. Direct communication between programs running on different nodes is more complex but offers significant advantages: the speed of communication is incredible higher and in much cases the size of transferred data is smaller. In order to gain these benefits, PostgreSQL has implemented the streaming replication technique, which connects master and standby servers via TCP. This technique adds two additional processes: the *WAL sender* process at the master site and the *WAL receiver* process at the standby site. They exchange information about data changes in the master's database.

The communication is initiated by the standby site and must run with a database user with REPLICATION privileges. This user must be created at the master site and authorized in the master's pg_hba.conf file. The master must accept connections from the standby in general and with a certain number of processes. The authorization at the operating system level shall be done by exchanging ssh keys.

```
Master site:
============

-- SQL
CREATE ROLE <replication_dbuser_at_master> REPLICATION ...;

# postgresql.conf: allow connections from standby to master
listen_addresses ='<ip_of_standby_server>'        # what IP address(es) to
 listen on
max_wal_senders = 5   # no more replication processes/connections than this
 number
# make hot standby possible
wal_level = replica   # 'hot_standby' in versions prior to PG 9.6

# pg_hba.conf: one additional line (the 'all' entry doesn't apply to
 replication)
# Allow the <replication_dbuser> to connect from standby to master
host   replication    <replication_dbuser>    <IP_of_standby_server>/32    trust


Standby site:
=============

# recovery.conf (this file can be created by the pg_basebackup utility, see
 below)
standby_mode=on   # activates standby mode
# How to reach the master:
primary_conninfo='user=<replication_dbuser_at_master_server>
 host=<IP_of_master_server> port=<port_of_master_server>
                  sslmode=prefer sslcompression=1 krbsrvname=...'

# postgres.conf: activate hot standby
hot_standby = on
```

Now you are ready to start. First, you must start the master. Second, you must transfer the complete databases from the master to the standby. And at last you start the standby.

Just as the replication activities, the transfer of the databases is initiated at the standby site.

```
pg_basebackup -h <IP_of_master_server> -D main --wal-method=stream
--checkpoint=fast -R
```

The utility *pg_ basebackup* transfers everythink to the directory 'main' (shall be empty), in this case it uses the streaming methode, it initiates a checkpoint at the master site to enforce consistency of database files and WAL files, and due to the -R flag it generates the previous mentioned recovery.conf file.

The activation of the 'hot' standby is done exactly as in the previous use case.

22.4 An Additional Tool

If you have to manage a complex replication use case, you may want to check the open source project 'repmgr'[5]. It supports you to monitor the cluster of nodes or perform a failover.

5 `https://repmgr.org/`

23 Partitioning

If you have a table with a very huge amount of data, it may be helpful to scatter the data to different physical tables which share a common data structure. In such use cases, where DML statements concern only one of those physical tables, you can get great performance benefits from partitioning. Typically this is the case, if there is any timeline or a geographical distribution of the values of a column.

23.1 Declarative-partitioning-syntax: since version 10

Postgres 10 introduced a declarative partition-defining-syntax in addition to the previous table-inheritance-syntax. With this syntax the necessity to define an additional trigger disappears, but in comparision to the previous solution the functionality stays unchanged.

First, you define a master table containing a partitioning methode which is `PARTITION BY RANGE (column_name)` in this example:

```
CREATE TABLE log (
  id       int  not null,
  logdate  date not null,
  message  varchar(500)
) PARTITION BY RANGE (logdate);
```

Next, you create partitions with the same structure as the master and ensure, that only rows within the expected data range can be stored there. Those partitions are conventional, physical tables.

```
CREATE TABLE log_2015_01 PARTITION OF log FOR VALUES FROM ('2015-01-01') TO
 ('2015-02-01');
CREATE TABLE log_2015_02 PARTITION OF log FOR VALUES FROM ('2015-02-01') TO
 ('2015-03-01');
...
CREATE TABLE log_2015_12 PARTITION OF log FOR VALUES FROM ('2015-12-01') TO
 ('2016-01-01');
CREATE TABLE log_2016_01 PARTITION OF log FOR VALUES FROM ('2016-01-01') TO
 ('2016-02-01');
...
```

23.2 Table-inheritance-syntax

First, you define a master table, which is a conventional table.

```
CREATE TABLE log (
  id       int  not null,
  logdate  date not null,
  message  varchar(500)
);
```

Next, you create partitions with the same structure as the master table by using the table-inheritance mechanism `INHERITS (table_name)`. Additionally you must ensure that only rows within the expected data range can be stored in the derived tables.

```
CREATE TABLE log_2015_01 (CHECK (logdate >= DATE '2015-01-01' AND logdate < DATE
 '2015-02-01')) INHERITS (log);
CREATE TABLE log_2015_02 (CHECK (logdate >= DATE '2015-02-01' AND logdate < DATE
 '2015-03-01')) INHERITS (log);
...
CREATE TABLE log_2015_12 (CHECK (logdate >= DATE '2015-12-01' AND logdate < DATE
 '2016-01-01')) INHERITS (log);
CREATE TABLE log_2016_01 (CHECK (logdate >= DATE '2016-01-01' AND logdate < DATE
 '2016-02-01')) INHERITS (log);
...
```

You need a function, which transfers rows into the appropriate partition.

```
CREATE OR REPLACE FUNCTION log_ins_function() RETURNS TRIGGER AS $$
BEGIN
  IF (NEW.logdate >= DATE '2015-01-01' AND NEW.logdate < DATE '2015-02-01' )
 THEN
        INSERT INTO log_2015_01 VALUES (NEW.*);
  ELSIF (NEW.logdate >= DATE '2015-02-01' AND NEW.logdate < DATE '2015-03-01' )
 THEN
        INSERT INTO log_2015_02 VALUES (NEW.*);
  ELSIF ...
    ...
  END IF;
  RETURN NULL;
END;
$$
LANGUAGE plpgsql;
```

The function is called by a trigger.

```
CREATE TRIGGER log_ins_trigger
  BEFORE INSERT ON log
  FOR EACH ROW EXECUTE PROCEDURE log_ins_function();
```

23.3 Further Steps

It's a good idea to create an index.

```
CREATE INDEX log_2015_01_idx ON log_2015_01 (logdate);
CREATE INDEX log_2015_02_idx ON log_2015_02 (logdate);
...
```

Many DML statements like `SELECT * FROM log WHERE logdate = '2015-01-15';` act only on one partition and can ignore all the others. This is very helpfull especially in such cases where a full table scan becomes necessary. The query optimizer has the chance to generate execution plans which avoid scanning unnecessary partitions.

In the shown example new rows will mainly go to the newest partition. After some years you can drop old partitions as a whole. This shall be done with the command `DROP TABLE` - not with a `DELETE` command. The `DROP TABLE` command is much faster than the `DELETE` command as it removes the complete partition in one single step instead of touching every single row.

24 Tablespace

The default behaviour of PostgreSQL is, that all data, indices, and management information is stored in subdirectories of a single directory. But this approach is not always suitable. In some situation you may want to change the storage area of one or more tables: data grows and may blow up partition limits, you may want to use fast devices like a ssd for heavily used tables, etc. . Therefore you need a technique to become more flexible.

Tablespaces offers the possibility to push data on arbitrary directories within your file system.

```
CREATE TABLESPACE fast LOCATION '/ssd1/postgresql/fastTablespace';
```

After the tablespace is defined it can be used in DDL statements.

```
CREATE TABLE t1(col_1 int) TABLESPACE fast;
```

25 Upgrade

When upgrading the PostgreSQL software, you must take care of the data in the cluster - depending on the question whether it is an upgrade of a major or a minor version. The PostgreSQL version number consists of two or three groups of digits, divided by colons. The first two groups denotes the major version and the third group (if present) denotes the minor version.

Upgrades within minor versions are simple. The internal data format does not change, so you only need to install the new software while the instance is down.

Upgrades of major versions may lead to incompatibilities of internal data structures. Therefore special actions may become necessary. There are several strategies to overcome the situation. In many cases upgrades of major versions additionally introduce some user-visible incompatibilities, so application programming changes might be required. You should read the release notes carefully.

25.1 pg_upgrade

`pg_upgrade` is a utility which modifies data files and system catalogs according to the needs of the new version. It has two major behaviors: In –link mode files are modified in place, otherwise the files are copied to a new location.

25.2 pg_dumpall

`pg_dumpall` is a standard utility to generate **logical** backups of the cluster. Files generated by `pg_dumpall` are plain text files and thus independent from all internal structures. When modifications of the data's internal structure become necessary (upgrade, different hardware architecture, different operating system, ...), such logical backups can be used for the data transfer from the old to the new system.

25.3 Replication

The Slony replication system offers the possiblity to transfer data over different major versions. Using this, you can switch a replication slave to the new master within a very short time frame.

PostgreSQL offers replication in *logical streaming* format. With the actual version 9.5 this feature is restricted to the same versions of master and standby server, but it is planned to extend it for use in a heterogenuous server landscape.

26 Glossary / Prominent Terms

To promote a consistent use and understanding of important terms we list and define them here. For some terms we include short annotations to give a first introduction to the subject.

26.1 Database Cluster

26.1.1 Overview

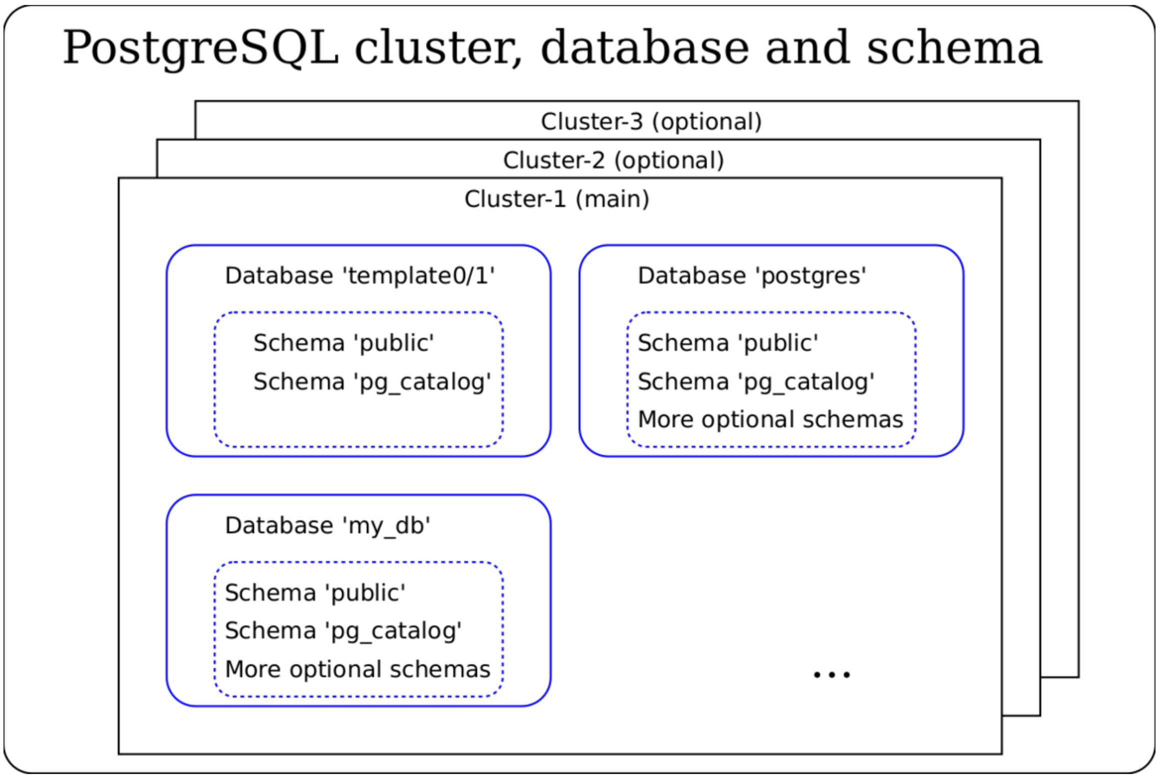


Figure 12

26.1.2 Server (or Node)

A server is some (real or virtual) hardware where PostgreSQL is installed. In this document, the word *instance* is a different concept from *server*. See the definition of *instance* later in this document.

26.1.3 Cluster of Nodes

A set of nodes, which interchange information via replication.

26.1.4 Installation

After you have downloaded and installed PostgreSQL, you have a set of programs, scripts, configuration- and other files on a *server*. This set is called the 'Installation'. It includes all *instance* programs as well as some client programs like `psql`.

26.1.5 Server Database

The term *server database* is often used in the context of client/server connections to refer to an *instance* or a single *database*.

26.1.6 Cluster (or 'Database Cluster')

A cluster is a storage area (directory, subdirectories and files) in the file system, where a collection of databases plus meta-information resides. Within the database cluster there are also the definitions of global objects like users and their rights. They are known across the entire database cluster. (Access rights for an user may be limited to individual objects like a certain table or a certain schema. In that case, the user will not have this access rights to the other objects of the cluster.)

Within a database cluster there are at least three databases: 'template0', 'template1', 'postgres' and possibly more.

- 'template0': A template database, which may be used by the command `CREATE DATABASE` (template0 should never be modified)
- 'template1': A template database, which may be used by the command `CREATE DATABASE` (template1 may be modified by DBA)
- 'postgres': An empty database, mainly for maintenance purposes

Most PostgreSQL installations use only one database cluster. Its name is 'main'. But you can create more clusters on the same PostgreSQL installation, see tools `initdb` further down.

26.1.7 Instance (or 'Database Server Instance' or 'Database Server' or 'Backend')

An instance is a group of processes (on a UNIX server) or one service (on a Windows server) plus shared memory, which controls and manages exactly one *cluster*. Using IP terminology one can say that one instance occupies one IP/port combination, eg. the combination `http://localhost:5432`. It is possible that on a different port of the same *server* another instance is running. The processes (in a UNIX server), which build an instance, are called: postmaster (creates one 'postgres'-process per client-connection), logger, checkpointer, background writer, WAL writer, autovacuum launcher, archiver, stats collector. The role of each process is explained in the chapter architecture[1].

If you have many *clusters* on your *server*, you can run many instances at the same machine - one per *cluster*.

Hint: Other publications sometimes use the term *server* to refer to an instance. As the term *server* is widely used to refer to real or virtual hardware, we do not use *server* as a synonym for *instance*.

26.1.8 Database

A database is a storage area in the file system, where a collection of objects is stored in files. The objects consist of data, metadata (table definitions, data types, constraints, views, ...) and other data like indices. Those objects are stored in the default database 'postgres' or in a newly created database.

The storage area for one database is organized as one subdirectory tree within the storage area of the database cluster. Thus a database cluster may contain multiple databases.

In a newly created database cluster (see below: initdb) there is an empty database with the name 'postgres'. In most cases this database stays empty and application data is stored in separate databases like 'finance' or 'engineering'. Nevertheless 'postgres' should not be dropped because some tools try to store temporary data within this database.

26.1.9 Schema

A schema is a namespace within a database: it contains named objects (tables, data types, functions, and operators) whose names can duplicate those of other objects existing in other schemas of this database. Every database contains the default schema 'public' and may contain more schemas. All objects of one schema must reside within the same database. Objects of different schemas within the same database may have the same name.

There is another special schema in each database. The schema 'pg_catalog' contains all system tables, built-in data types, functions, and operators. See also 'Search Path' below.

26.1.10 Search Path (or 'Schema Search Path')

A Search Path is a list of schema names. If applications use unqualified object names (e.g.: 'employee_table' for a table name), the search path is used to locate this object in the given sequence of schemas. The schema 'pg_catalog' is always the first part of the search path although it is not explicitly listed in the search path. This behaviour ensures that PostgreSQL finds the system objects.

1 Chapter 8 on page 33

26.1.11 initdb (OS command)

Despite of its name the utility `initdb` creates a new *cluster*, which contains the 3 *databases* 'template0', 'template1' and 'postgres'.

26.1.12 createdb (OS command)

The utility `createdb` creates a new *database* within the actual *cluster*.

26.1.13 CREATE DATABASE (SQL command)

The SQL command `CREATE DATABASE` creates a new *database* within the actual *cluster*.

26.1.14 Directory Structure

A *cluster* and its *databases* consists of files, which hold data, actual status information, modification information and a lot more. Those files are organized in a fixed way under one directory node.

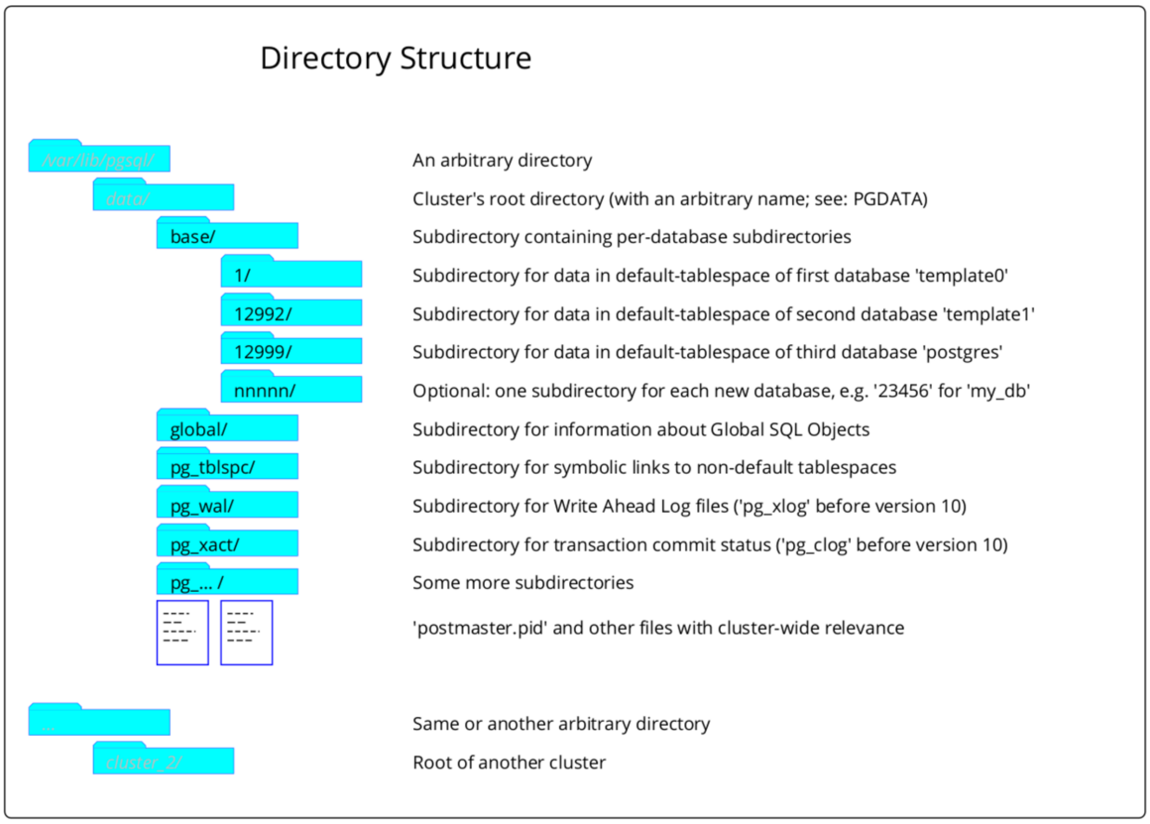

Directory Structure
/var/lib/pgsql/
An arbitrary directory
data/
Cluster's root directory (with an arbitrary name; see: PGDATA)
base/
Subdirectory containing per-database subdirectories
1/
Subdirectory for data in default-tablespace of first database 'template0'
12992/
Subdirectory for data in default-tablespace of second database 'template1'
12999/
Subdirectory for data in default-tablespace of third database 'postgres'
nnnnn/
Optional: one subdirectory for each new database, e.g. '23456' for 'my_db'
global/
Subdirectory for information about Global SQL Objects
pg_tblspc/
Subdirectory for symbolic links to non-default tablespaces
pg_wal/
Subdirectory for Write Ahead Log files ('pg_xlog' before version 10)
pg_xact/
Subdirectory for transaction commit status ('pg_clog' before version 10)
pg_... /
Some more subdirectories
'postmaster.pid' and other files with cluster-wide relevance
...
Same or another arbitrary directory
cluster_2/
Root of another cluster

Figure 13

26.2 Consistent Writes

26.2.1 Shared Buffers

Shared bufferes are RAM pages, which mirror pages of data files on disc. They exist due to performance reasons. The term *shared* results from the fact that a lot of processes read and write to that area.

26.2.2 'Dirty' Page

Pages in the *shared buffers* mirror pages of data files on disc. When clients request changes of data, those pages get changed without - provisionally - a change of the related pages on disc. Until the *background writer* writes those modified pages to disc, they are called 'dirty' pages.

26.2.3 Checkpoint

A checkpoint is a special point in time where it is guaranteed that the database files are in a consistent state. At checkpoint time all change records are flushed to the WAL file, all dirty data pages (in shared buffers) are flushed to disc, and at last a special checkpoint record is written to the WAL file.

The instance's checkpointer process automatically triggers checkpoints on a regular basis. Additionally they can be forced by issuing the command `CHECKPOINT` in a client program. For the database system it takes a lot of time to perform a checkpoint - because of the physical writes to disc.

26.2.4 WAL File

WAL files contain the changes which are applied to the data by modifying commands like `INSERT`, `UPDATE`, `DELETE` or `CREATE TABLE ....` This is redundant information as it is also recorded in the data files (for better performance at a later time). According to the configuration of the instance, there may be more information within WAL files. WAL files reside in the `pg_wal` directory (which was named `pg_xlog` before version 10), have a binary format and a fixed size of 16MB. When they are no longer needed, they get recycled by renaming and reusing their already allocated space.

A single information unit within a WAL file is called a *log record*.

Hint: In the PostgreSQL documentation and in related documents there are a lot of other, similar terms which refer to what we denote as *WAL file* in this Wikibook: segment, WAL segment, logfile (don't mix it with the term *logfile*, see below), WAL log file, ...

26.2.5 Logfile

The *instance* logs and reports warning and error messages about special situations in readable text files. These logfiles can reside at any place in the directory structure of the *server* and are not part of the *cluster*.

Hint: The term 'logfile' does not relate to the other terms of this subchapter. It is mentioned here because the term sometimes is used as a synonym for what we call *WAL file* - see above.

26.2.6 Log Record

A log record is a single information unit within a *WAL file*.

26.2.7 Segment

The term *segment* is sometimes used as a synonym for *WAL file*.

26.2.8 MVCC

Multiversion Concurrency Control[2] (MVCC) is a common database technique to accomplish two goals: First, it allows the management of parallel running transactions on a logical level and second, it ensures high performance for concurrent read and write actions. It is implemented as follows: Whenever some values of an existing row change, PostgreSQL writes a new version of this row to the database without deleting the old one. In such situations the database contains multiple versions of the row. In addition to their regular data the rows contain transaction IDs which allows to decide, which other transactions will see the new or the old row. Hence other transactions sees only those values (of other transactions), which are committed.

Outdated old rows are deleted at a later time by the utility `vacuumdb` respectively the SQL command `vacuum`.

26.3 Backup and Recovery

The term *cold* as an addition to the backup method name indicates that with this method the instance must be stopped to create a useful backup. In contrast, the addition 'hot' denotes methods where the instance MUST run (and hence changes to the data may occur during backup actions).

26.3.1 (Cold) Backup (file system tools)

A cold backup is a consistent copy of all files of the *cluster* with OS tools like cp or tar. During the creation of a cold backup the *instance* must **not** run - otherwise the backup is useless. Hence you need a period of time in which applications do not use any *database* of the *cluster* - a continuous 7×24 operation mode is not possible. And secondly: the cold backup works only on the *cluster* level, not on any finer granularity like database or table.

Hint: A cold backup is sometimes called an ”offline backup”.

2 `https://en.wikipedia.org/wiki/Multiversion_concurrency_control`

26.3.2 (Hot) Logical Backup (pg_dump utility)

A logical backup is a consistent copy of the data within a *database* or some of its parts. It is created with the utility `pg_dump`. Although `pg_dump` may run in parallel to applications (the *instance* must be up), it creates a consistent snapshot as of the time of its start.

`pg_dump` supports two output formats. The first one is a text format containing SQL commands like CREATE and INSERT. Files created in this format may be used by `psql` to restore the backed-up data. The second format is a binary format and is called the 'archive format'. Files with this format can be used to restore its data with the tool `pg_restore`.

As mentioned, `pg_dump` works at the *database* level or smaller parts of *databases* like tables. If you want to refer to the *cluster* level, you must use `pg_dumpall`. Please notice, that important objects like users/roles and their rights are always defined at *cluster* level.

26.3.3 (Hot) Continuous Archiving

Such backups consist of two parts. The first one is the so-called *base backup*, which is a copy of all files of a cluster. The second one consists of all data-changes since the start of the backup command. They are stored in WAL files.

Such backups work only at the cluster level, not on any finer granularity like database or table.

26.3.4 PITR: Point in Time Recovery

When you use the technique of 'Continuous Archiving' (Base Backup) and archive all occurring WAL files, you can restore the database as it was at an arbitrary time. To do so, you must restore the base backup and replay the WAL files against it up to a defined timestamp.

26.3.5 Archiving

Archiving is the process of copying *WAL files* to a failsafe location. When you plan to use *PITR* you must ensure that the sequence of *WAL files* is saved for a longer period. To support the process of copying *WAL files* at the right moment (when they are completely filled and a switch to the next *WAL file* has taken place), PostgreSQL runs the *archiving process* which is part of the *instance*. This process copies *WAL files* to a configurable destination.

26.3.6 Recovering

Recovering is the process of playing *WAL files* against a *physical backup*. One of the involved steps is the copy of the *WAL files* from the failsafe archive location to its original location in '/pg_xlog'. The aim of recovery is to bring the *cluster* into a consistent state at a defined timestamp.

26.3.7 Archive Recovery Mode

When recovering takes place, the *instance* is in *archive recovery mode*.

26.3.8 Restartpoint

A restart point is an action similar to a *checkpoint*. Restart points are only performed when the instance is in *archive recovery mode* or in *standby mode*.

26.3.9 Timeline

After a successful recovery PostgreSQL transfers the *cluster* into a new timeline to avoid problems, which may occur when PITR is reset and *WAL files* reapplied (e.g.: to a different timestamp). Timeline names are sequential numbers: 1, 2, 3, ...

26.4 Replication

Replication is a technique to send data, which was written within a *master server*, to one or more *standby servers* or even another *master server*.

26.4.1 Master Server

The master server is an *instance* on a *server* which sends data to other *instances* in addition to its local processing of data.

26.4.2 Standby Server

The standby server is an *instance* on a *server* which receives information from a *master server* about changes of its data.

26.4.3 Warm Standby Server

A warm standby server is a running *instance*, which is in *standby_ mode* (recovery.conf file). It continuously reads and processes incoming *WAL files* (in the case of *log-shipping*) or *log records* (in the case of *streaming replication*). It does not accept client connections.

26.4.4 Hot Standby Server

A hot standby server is a warm standby server with the additional flag *hot_ standby* in postgres.conf. It accepts client connections and read-only queries.

26.4.5 Synchronous Replication

Replication is called *synchronous*, when the *standby server* processes the received data immediately, sends a confirmation record to the *master server* and the *master server* delays its COMMIT action until he has received the confirmation of the *standby server*.

26.4.6 Asynchronous Replication

Replication is called *asynchronous*, when the *master server* sends data to the *standby server* and does not expect any feedback about this action.

26.4.7 Streaming Replication

The term is used when *log entries* are transfered from *master server* to *standby server* over a TCP connection - in addition to their transfer to the local *WAL file*. Streaming replication is *asynchronous* by default but can also be *synchronous*.

26.4.8 Log-Shipping Replication

Log shipping is the process of transfering *WAL files* from a *master server* to a *standby server* . Log shipping is an asynchronous operation.

27 The Extension Mechanism

PostgreSQL offers an extensibility architecture and implements its internal data types, operators, functions, indexes, and more on top of it. This architecture is open for everybody to implement and add his own functionality to the PostgreSQL system. You can define new datatypes with or without special operators and functions as needed by your use case. After you have added them, you have the best of two worlds: the special functionalities you have created plus the standard functionality of a database system like ACID, SQL, security, standard data types, WAL, client APIs, ... An introduction to extensibility is given in the PostgreSQL documentation[1].

Over time the community has developed a set of extensions that are useful for their own needs and for a great number of applications - sometimes even for the requirements and definitions given by standardization organizations. Some popular examples are

- Data types, operators, and function for the handling of **spatial data** like points, polylines, overlaps(), ... as defined by OSGeo[2] and SQL Multimedia and Application Packages Part 3: Spatial[3].
- Functionality for **full text** search as defined by SQL Multimedia and Application Packages Part 2: Full-Text[4].
- Access to **data outside** the current database (other PostgreSQL instance, other SQL, NoSQL or BigData database system, LDAP, flat files like csv, json, xml) as defined by SQL Part 9: Management of External Data[5].

The lifecycle of such an extension starts with the implementation of its features by a group of persons or a company. After publishing, the extension may be used and further expanded by other persons or companies of the community. Sometimes such extensions keep independent from the PostgreSQL system, e.g.: PostGIS, in other cases they are delivered with the standard download and explicitly listed as an Additional Supplied Module[6] within the documentation with hints how to install them. And in rare cases, extensions are incorporated into the core system so that they become a native part of PostgreSQL.

To activate and use an extension, you must download and install the necessary files (if not delivered with the standard download) and issue the command `CREATE EXTENSION <extension_name>;` within an SQL client like `psql`. To control which extensions are already installed use: `\dx` within `psql`.

1 `https://www.postgresql.org/docs/current/extend.html`
2 `https://en.wikipedia.org/wiki/Open_Source_Geospatial_Foundation`
3 `https://en.wikipedia.org/wiki/SQL%23Interoperability_and_standardization`
4 `https://en.wikipedia.org/wiki/SQL%23Interoperability_and_standardization`
5 `https://en.wikipedia.org/wiki/SQL%2FMED`
6 `https://www.postgresql.org/docs/current/contrib.html`

27.1 PostGIS

PostGIS[7] is a project which extends PostgreSQL with a rich set of 2D and 3D spacial data types plus associated functions, operators and index types as defined by OSGeo[8] and SQL Multimedia and Application Packages Part 3: Spatial[9]. Typically data types are *polygon* or *multipoint*, typical functions are *st_ length()* or *st_ contains()*. The appropriated index type for spatial objects is the GiST index[10].

The PostGIS project has its own representation on the WEB[11] where all its aspects are described, especially the download process and the activation of the extension itself.

27.2 Foreign Data Wrappers

Foreign Data Wrappers (FDW) are PostgreSQL extensions that offer access to data outside of the actual database and instance. There are different types of data wrappers:

- One wrapper to other PostgreSQL instances: *postgres_fdw*
- A lot of wrappers to other relational database systems like Oracle, MySQL, MS SQL Server, ...
- A lot of wrappers to NoSQL database systems: CouchDB, MongoDB, Cassandra, ...
- Generic wrappers to ODBC and JDBC
- A lot of wrapper to files of different formats: csv, xml, json, tar, zip, ... (file_fdw)
- LDAP wrapper
- ... and more.

A comprehensive list[12] gives an overview.

The technique of FDW is defined in the SQL standard Part 9: Management of External Data[13].

Here is an example of how to access another PostgreSQL instance via FDW.

```
-- Install the extension to other PostgreSQL instances
CREATE EXTENSION postgres_fdw;

-- Define the connection to a database/instance at a different server
CREATE SERVER remote_geo_server
  FOREIGN DATA WRAPPER postgres_fdw
  OPTIONS (host '10.10.10.10', port '5432', dbname 'geo_data');

-- Define a user for the connection (The remote user must have access rights at
 the remote database)
CREATE USER MAPPING FOR CURRENT_USER
  SERVER remote_geo_server
  OPTIONS (user 'geo_data_user', password 'xxx');

-- Define two foreign tables via an IMPORT command ...
```

7 https://en.wikipedia.org/wiki/PostGIS
8 https://en.wikipedia.org/wiki/Open_Source_Geospatial_Foundation
9 https://en.wikipedia.org/wiki/SQL%23Interoperability_and_standardization
10 Chapter 32.5 on page 124
11 http://postgis.net/
12 https://wiki.postgresql.org/wiki/Foreign_data_wrappers
13 https://en.wikipedia.org/wiki/SQL%2FMED

```
IMPORT FOREIGN SCHEMA geo_schema
  LIMIT TO (city, point_of_interest)
  FROM SERVER remote_geo_server
  INTO my_schema;

-- .. and another foreign table via an explicit definition
CREATE FOREIGN TABLE remote_person (
  id            SERIAL,
  person_name   TEXT         NOT NULL,
  city_id       INT4         NOT NULL
)
SERVER remote_geo_server
OPTIONS(schema_name 'geo_schema', table_name 'person');
```

After the execution of the above statements you have access to the three tables city, point_of_interest and remote_person with the usual DML commands SELECT, UPDATE, COMMIT, ... Nevertheless the data keeps at the 'remote' server (10.10.10.10), queries are executed there, and only the results of queries are transfered via network to the actual instance and your client application.

```
SELECT count(*) FROM city; -- table 'city' resides on a different server
```

27.3 Bidirectional Replication (BDR)

BDR is an extension that allows replication in both directions between involved (master-) nodes in parallel to their regular read and writes activities of their client applications. So it realizes a multi-master replication. Actually, the project is a standalone project[14]. But multiple technologies emerging from BDR development have already become an integral part of core PostgreSQL, such as Event Triggers[15], Logical Decoding[16], Replication Slots[17], Background Workers[18], and more.

14 http://bdr-project.org/docs/next/index.html
15 https://www.postgresql.org/docs/current/event-triggers.html
16 https://www.postgresql.org/docs/current/logicaldecoding.html
17 https://www.postgresql.org/docs/current/logicaldecoding-explanation.html#LOGICALDECODING-REPLICATION-SLOTS
18 https://www.postgresql.org/docs/current/bgworker.html

28 PostGIS - Spatial Data

PostgreSQL offers an extensibility architecture and implements its internal data types, operators, functions, indexes, and more on top of it. This architecture is open for everybody to implement and add his own functionality to the PostgreSQL system. You can define new datatypes with or without special operators and functions as needed by your use case. After you have added them, you have the best of two worlds: the special functionalities you have created plus the standard functionality of a database system like ACID, SQL, security, standard data types, WAL, client APIs, ... An introduction to extensibility is given in the PostgreSQL documentation[1].

Over time the community has developed a set of extensions that are useful for their own needs and for a great number of applications - sometimes even for the requirements and definitions given by standardization organizations. Some popular examples are

- Data types, operators, and function for the handling of **spatial data** like points, polylines, overlaps(), ... as defined by OSGeo[2] and SQL Multimedia and Application Packages Part 3: Spatial[3].
- Functionality for **full text** search as defined by SQL Multimedia and Application Packages Part 2: Full-Text[4].
- Access to **data outside** the current database (other PostgreSQL instance, other SQL, NoSQL or BigData database system, LDAP, flat files like csv, json, xml) as defined by SQL Part 9: Management of External Data[5].

The lifecycle of such an extension starts with the implementation of its features by a group of persons or a company. After publishing, the extension may be used and further expanded by other persons or companies of the community. Sometimes such extensions keep independent from the PostgreSQL system, e.g.: PostGIS, in other cases they are delivered with the standard download and explicitly listed as an Additional Supplied Module[6] within the documentation with hints how to install them. And in rare cases, extensions are incorporated into the core system so that they become a native part of PostgreSQL.

To activate and use an extension, you must download and install the necessary files (if not delivered with the standard download) and issue the command `CREATE EXTENSION <extension_name>;` within an SQL client like `psql`. To control which extensions are already installed use: `\dx` within `psql`.

1 `https://www.postgresql.org/docs/current/extend.html`
2 `https://en.wikipedia.org/wiki/Open_Source_Geospatial_Foundation`
3 `https://en.wikipedia.org/wiki/SQL%23Interoperability_and_standardization`
4 `https://en.wikipedia.org/wiki/SQL%23Interoperability_and_standardization`
5 `https://en.wikipedia.org/wiki/SQL%2FMED`
6 `https://www.postgresql.org/docs/current/contrib.html`

28.1 PostGIS

PostGIS[7] is a project which extends PostgreSQL with a rich set of 2D and 3D spacial data types plus associated functions, operators and index types as defined by OSGeo[8] and SQL Multimedia and Application Packages Part 3: Spatial[9]. Typically data types are *polygon* or *multipoint*, typical functions are *st_length()* or *st_contains()*. The appropriated index type for spatial objects is the GiST index[10].

The PostGIS project has its own representation on the WEB[11] where all its aspects are described, especially the download process and the activation of the extension itself.

28.2 Foreign Data Wrappers

Foreign Data Wrappers (FDW) are PostgreSQL extensions that offer access to data outside of the actual database and instance. There are different types of data wrappers:

- One wrapper to other PostgreSQL instances: *postgres_fdw*
- A lot of wrappers to other relational database systems like Oracle, MySQL, MS SQL Server, ...
- A lot of wrappers to NoSQL database systems: CouchDB, MongoDB, Cassandra, ...
- Generic wrappers to ODBC and JDBC
- A lot of wrapper to files of different formats: csv, xml, json, tar, zip, ... (file_fdw)
- LDAP wrapper
- ... and more.

A comprehensive list[12] gives an overview.

The technique of FDW is defined in the SQL standard Part 9: Management of External Data[13].

Here is an example of how to access another PostgreSQL instance via FDW.

```
-- Install the extension to other PostgreSQL instances
CREATE EXTENSION postgres_fdw;

-- Define the connection to a database/instance at a different server
CREATE SERVER remote_geo_server
  FOREIGN DATA WRAPPER postgres_fdw
  OPTIONS (host '10.10.10.10', port '5432', dbname 'geo_data');

-- Define a user for the connection (The remote user must have access rights at
 the remote database)
CREATE USER MAPPING FOR CURRENT_USER
  SERVER remote_geo_server
  OPTIONS (user 'geo_data_user', password 'xxx');

-- Define two foreign tables via an IMPORT command ...
```

7 https://en.wikipedia.org/wiki/PostGIS
8 https://en.wikipedia.org/wiki/Open_Source_Geospatial_Foundation
9 https://en.wikipedia.org/wiki/SQL%23Interoperability_and_standardization
10 Chapter 32.5 on page 124
11 http://postgis.net/
12 https://wiki.postgresql.org/wiki/Foreign_data_wrappers
13 https://en.wikipedia.org/wiki/SQL%2FMED

```
IMPORT FOREIGN SCHEMA geo_schema
  LIMIT TO (city, point_of_interest)
  FROM SERVER remote_geo_server
  INTO my_schema;

-- .. and another foreign table via an explicit definition
CREATE FOREIGN TABLE remote_person (
  id           SERIAL,
  person_name  TEXT        NOT NULL,
  city_id      INT4        NOT NULL
)
SERVER remote_geo_server
OPTIONS(schema_name 'geo_schema', table_name 'person');
```

After the execution of the above statements you have access to the three tables city, point_of_interest and remote_person with the usual DML commands SELECT, UPDATE, COMMIT, ... Nevertheless the data keeps at the 'remote' server (10.10.10.10), queries are executed there, and only the results of queries are transfered via network to the actual instance and your client application.

```
SELECT count(*) FROM city; -- table 'city' resides on a different server
```

28.3 Bidirectional Replication (BDR)

BDR is an extension that allows replication in both directions between involved (master-) nodes in parallel to their regular read and writes activities of their client applications. So it realizes a multi-master replication. Actually, the project is a standalone project[14]. But multiple technologies emerging from BDR development have already become an integral part of core PostgreSQL, such as Event Triggers[15], Logical Decoding[16], Replication Slots[17], Background Workers[18], and more.

14 http://bdr-project.org/docs/next/index.html
15 https://www.postgresql.org/docs/current/event-triggers.html
16 https://www.postgresql.org/docs/current/logicaldecoding.html
17 https://www.postgresql.org/docs/current/logicaldecoding-explanation.html#LOGICALDECODING-REPLICATION-SLOTS
18 https://www.postgresql.org/docs/current/bgworker.html

29 Foreign Data Wrappers

PostgreSQL offers an extensibility architecture and implements its internal data types, operators, functions, indexes, and more on top of it. This architecture is open for everybody to implement and add his own functionality to the PostgreSQL system. You can define new datatypes with or without special operators and functions as needed by your use case. After you have added them, you have the best of two worlds: the special functionalities you have created plus the standard functionality of a database system like ACID, SQL, security, standard data types, WAL, client APIs, ... An introduction to extensibility is given in the PostgreSQL documentation[1].

Over time the community has developed a set of extensions that are useful for their own needs and for a great number of applications - sometimes even for the requirements and definitions given by standardization organizations. Some popular examples are

- Data types, operators, and function for the handling of **spatial data** like points, polylines, overlaps(), ... as defined by OSGeo[2] and SQL Multimedia and Application Packages Part 3: Spatial[3].
- Functionality for **full text** search as defined by SQL Multimedia and Application Packages Part 2: Full-Text[4].
- Access to **data outside** the current database (other PostgreSQL instance, other SQL, NoSQL or BigData database system, LDAP, flat files like csv, json, xml) as defined by SQL Part 9: Management of External Data[5].

The lifecycle of such an extension starts with the implementation of its features by a group of persons or a company. After publishing, the extension may be used and further expanded by other persons or companies of the community. Sometimes such extensions keep independent from the PostgreSQL system, e.g.: PostGIS, in other cases they are delivered with the standard download and explicitly listed as an Additional Supplied Module[6] within the documentation with hints how to install them. And in rare cases, extensions are incorporated into the core system so that they become a native part of PostgreSQL.

To activate and use an extension, you must download and install the necessary files (if not delivered with the standard download) and issue the command `CREATE EXTENSION <extension_name>;` within an SQL client like `psql`. To control which extensions are already installed use: `\dx` within `psql`.

1 https://www.postgresql.org/docs/current/extend.html
2 https://en.wikipedia.org/wiki/Open_Source_Geospatial_Foundation
3 https://en.wikipedia.org/wiki/SQL%23Interoperability_and_standardization
4 https://en.wikipedia.org/wiki/SQL%23Interoperability_and_standardization
5 https://en.wikipedia.org/wiki/SQL%2FMED
6 https://www.postgresql.org/docs/current/contrib.html

29.1 PostGIS

PostGIS[7] is a project which extends PostgreSQL with a rich set of 2D and 3D spacial data types plus associated functions, operators and index types as defined by OSGeo[8] and SQL Multimedia and Application Packages Part 3: Spatial[9]. Typically data types are *polygon* or *multipoint*, typical functions are *st_length()* or *st_contains()*. The appropriated index type for spatial objects is the GiST index[10].

The PostGIS project has its own representation on the WEB[11] where all its aspects are described, especially the download process and the activation of the extension itself.

29.2 Foreign Data Wrappers

Foreign Data Wrappers (FDW) are PostgreSQL extensions that offer access to data outside of the actual database and instance. There are different types of data wrappers:

- One wrapper to other PostgreSQL instances: *postgres_fdw*
- A lot of wrappers to other relational database systems like Oracle, MySQL, MS SQL Server, ...
- A lot of wrappers to NoSQL database systems: CouchDB, MongoDB, Cassandra, ...
- Generic wrappers to ODBC and JDBC
- A lot of wrapper to files of different formats: csv, xml, json, tar, zip, ... (file_fdw)
- LDAP wrapper
- ... and more.

A comprehensive list[12] gives an overview.

The technique of FDW is defined in the SQL standard Part 9: Management of External Data[13].

Here is an example of how to access another PostgreSQL instance via FDW.

```
-- Install the extension to other PostgreSQL instances
CREATE EXTENSION postgres_fdw;

-- Define the connection to a database/instance at a different server
CREATE SERVER remote_geo_server
  FOREIGN DATA WRAPPER postgres_fdw
  OPTIONS (host '10.10.10.10', port '5432', dbname 'geo_data');

-- Define a user for the connection (The remote user must have access rights at
 the remote database)
CREATE USER MAPPING FOR CURRENT_USER
  SERVER remote_geo_server
  OPTIONS (user 'geo_data_user', password 'xxx');

-- Define two foreign tables via an IMPORT command ...
```

7 https://en.wikipedia.org/wiki/PostGIS
8 https://en.wikipedia.org/wiki/Open_Source_Geospatial_Foundation
9 https://en.wikipedia.org/wiki/SQL%23Interoperability_and_standardization
10 Chapter 32.5 on page 124
11 http://postgis.net/
12 https://wiki.postgresql.org/wiki/Foreign_data_wrappers
13 https://en.wikipedia.org/wiki/SQL%2FMED

```
IMPORT FOREIGN SCHEMA geo_schema
  LIMIT TO (city, point_of_interest)
  FROM SERVER remote_geo_server
  INTO my_schema;

-- .. and another foreign table via an explicit definition
CREATE FOREIGN TABLE remote_person (
  id           SERIAL,
  person_name  TEXT         NOT NULL,
  city_id      INT4         NOT NULL
)
SERVER remote_geo_server
OPTIONS(schema_name 'geo_schema', table_name 'person');
```

After the execution of the above statements you have access to the three tables city, point_of_interest and remote_person with the usual DML commands SELECT, UPDATE, COMMIT, ... Nevertheless the data keeps at the 'remote' server (10.10.10.10), queries are executed there, and only the results of queries are transfered via network to the actual instance and your client application.

```
SELECT count(*) FROM city; -- table 'city' resides on a different server
```

29.3 Bidirectional Replication (BDR)

BDR is an extension that allows replication in both directions between involved (master-) nodes in parallel to their regular read and writes activities of their client applications. So it realizes a multi-master replication. Actually, the project is a standalone project[14]. But multiple technologies emerging from BDR development have already become an integral part of core PostgreSQL, such as Event Triggers[15], Logical Decoding[16], Replication Slots[17], Background Workers[18], and more.

14 http://bdr-project.org/docs/next/index.html
15 https://www.postgresql.org/docs/current/event-triggers.html
16 https://www.postgresql.org/docs/current/logicaldecoding.html
17 https://www.postgresql.org/docs/current/logicaldecoding-explanation.html#LOGICALDECODING-REPLICATION-SLOTS
18 https://www.postgresql.org/docs/current/bgworker.html

30 BDR: Bidirectional Replication

PostgreSQL offers an extensibility architecture and implements its internal data types, operators, functions, indexes, and more on top of it. This architecture is open for everybody to implement and add his own functionality to the PostgreSQL system. You can define new datatypes with or without special operators and functions as needed by your use case. After you have added them, you have the best of two worlds: the special functionalities you have created plus the standard functionality of a database system like ACID, SQL, security, standard data types, WAL, client APIs, ... An introduction to extensibility is given in the PostgreSQL documentation[1].

Over time the community has developed a set of extensions that are useful for their own needs and for a great number of applications - sometimes even for the requirements and definitions given by standardization organizations. Some popular examples are

- Data types, operators, and function for the handling of **spatial data** like points, polylines, overlaps(), ... as defined by OSGeo[2] and SQL Multimedia and Application Packages Part 3: Spatial[3].
- Functionality for **full text** search as defined by SQL Multimedia and Application Packages Part 2: Full-Text[4].
- Access to **data outside** the current database (other PostgreSQL instance, other SQL, NoSQL or BigData database system, LDAP, flat files like csv, json, xml) as defined by SQL Part 9: Management of External Data[5].

The lifecycle of such an extension starts with the implementation of its features by a group of persons or a company. After publishing, the extension may be used and further expanded by other persons or companies of the community. Sometimes such extensions keep independent from the PostgreSQL system, e.g.: PostGIS, in other cases they are delivered with the standard download and explicitly listed as an Additional Supplied Module[6] within the documentation with hints how to install them. And in rare cases, extensions are incorporated into the core system so that they become a native part of PostgreSQL.

To activate and use an extension, you must download and install the necessary files (if not delivered with the standard download) and issue the command `CREATE EXTENSION <extension_name>;` within an SQL client like `psql`. To control which extensions are already installed use: `\dx` within `psql`.

1 https://www.postgresql.org/docs/current/extend.html
2 https://en.wikipedia.org/wiki/Open_Source_Geospatial_Foundation
3 https://en.wikipedia.org/wiki/SQL%23Interoperability_and_standardization
4 https://en.wikipedia.org/wiki/SQL%23Interoperability_and_standardization
5 https://en.wikipedia.org/wiki/SQL%2FMED
6 https://www.postgresql.org/docs/current/contrib.html

30.1 PostGIS

PostGIS[7] is a project which extends PostgreSQL with a rich set of 2D and 3D spacial data types plus associated functions, operators and index types as defined by OSGeo[8] and SQL Multimedia and Application Packages Part 3: Spatial[9]. Typically data types are *polygon* or *multipoint*, typical functions are *st_length()* or *st_contains()*. The appropriated index type for spatial objects is the GiST index[10].

The PostGIS project has its own representation on the WEB[11] where all its aspects are described, especially the download process and the activation of the extension itself.

30.2 Foreign Data Wrappers

Foreign Data Wrappers (FDW) are PostgreSQL extensions that offer access to data outside of the actual database and instance. There are different types of data wrappers:

- One wrapper to other PostgreSQL instances: *postgres_fdw*
- A lot of wrappers to other relational database systems like Oracle, MySQL, MS SQL Server, ...
- A lot of wrappers to NoSQL database systems: CouchDB, MongoDB, Cassandra, ...
- Generic wrappers to ODBC and JDBC
- A lot of wrapper to files of different formats: csv, xml, json, tar, zip, ... (file_fdw)
- LDAP wrapper
- ... and more.

A comprehensive list[12] gives an overview.

The technique of FDW is defined in the SQL standard Part 9: Management of External Data[13].

Here is an example of how to access another PostgreSQL instance via FDW.

```
-- Install the extension to other PostgreSQL instances
CREATE EXTENSION postgres_fdw;

-- Define the connection to a database/instance at a different server
CREATE SERVER remote_geo_server
  FOREIGN DATA WRAPPER postgres_fdw
  OPTIONS (host '10.10.10.10', port '5432', dbname 'geo_data');

-- Define a user for the connection (The remote user must have access rights at
 the remote database)
CREATE USER MAPPING FOR CURRENT_USER
  SERVER remote_geo_server
  OPTIONS (user 'geo_data_user', password 'xxx');

-- Define two foreign tables via an IMPORT command ...
```

7 https://en.wikipedia.org/wiki/PostGIS
8 https://en.wikipedia.org/wiki/Open_Source_Geospatial_Foundation
9 https://en.wikipedia.org/wiki/SQL%23Interoperability_and_standardization
10 Chapter 32.5 on page 124
11 http://postgis.net/
12 https://wiki.postgresql.org/wiki/Foreign_data_wrappers
13 https://en.wikipedia.org/wiki/SQL%2FMED

```
IMPORT FOREIGN SCHEMA geo_schema
  LIMIT TO (city, point_of_interest)
  FROM SERVER remote_geo_server
  INTO my_schema;

-- .. and another foreign table via an explicit definition
CREATE FOREIGN TABLE remote_person (
  id            SERIAL,
  person_name   TEXT         NOT NULL,
  city_id       INT4         NOT NULL
)
SERVER remote_geo_server
OPTIONS(schema_name 'geo_schema', table_name 'person');
```

After the execution of the above statements you have access to the three tables city, point_of_interest and remote_person with the usual DML commands SELECT, UPDATE, COMMIT, ... Nevertheless the data keeps at the 'remote' server (10.10.10.10), queries are executed there, and only the results of queries are transfered via network to the actual instance and your client application.

```
SELECT count(*) FROM city; -- table 'city' resides on a different server
```

30.3 Bidirectional Replication (BDR)

BDR is an extension that allows replication in both directions between involved (master-) nodes in parallel to their regular read and writes activities of their client applications. So it realizes a multi-master replication. Actually, the project is a standalone project[14]. But multiple technologies emerging from BDR development have already become an integral part of core PostgreSQL, such as Event Triggers[15], Logical Decoding[16], Replication Slots[17], Background Workers[18], and more.

14 http://bdr-project.org/docs/next/index.html
15 https://www.postgresql.org/docs/current/event-triggers.html
16 https://www.postgresql.org/docs/current/logicaldecoding.html
17 https://www.postgresql.org/docs/current/logicaldecoding-explanation.html#LOGICALDECODING-REPLICATION-SLOTS
18 https://www.postgresql.org/docs/current/bgworker.html

31 Page Layout for Data and Index Files

PostgreSQL organizes data and associated indices in separate files: one file per table plus one file per index. Files are divided into blocks of 8192 bytes (8 KiB). This is the smallest unit of PostgreSQL's disc I/Os. Currently used blocks are mirrored 1:1 in the Shared Memory of the instance. Within PostgreSQL, a 'block' is often called a 'page' though the term 'page' may be differently interpreted by the underlying file system, e.g.: 4 KiB or some other 2^n value.

All pages of data files are logically equivalent, whereas index files use different page types depending on the needs of the index (meta page, root page, internal tree page, leaf page, ...). Nevertheless, the physical layout for pages of data files and of all types of index files is identical.

Each page consists of 5 major components

1. *Page Header*: General information about the page.
2. *ItemIds*: Array of pointers to Items. It grows or shrinks over time.
3. *Free Space*: The unused space of the page.
4. *Items*: A set of actual data respectively rows or a set of index entries. It grows or shrinks over time.
5. *Special Space*: Data files don't use it. Index files structure it depending on the needs of the index type.

Every additional row or index entry creates a new ItemId at the end of the 2. component plus a new Item at the beginning of the 4. component. As a result, the free space shrinks from both its left and right side.

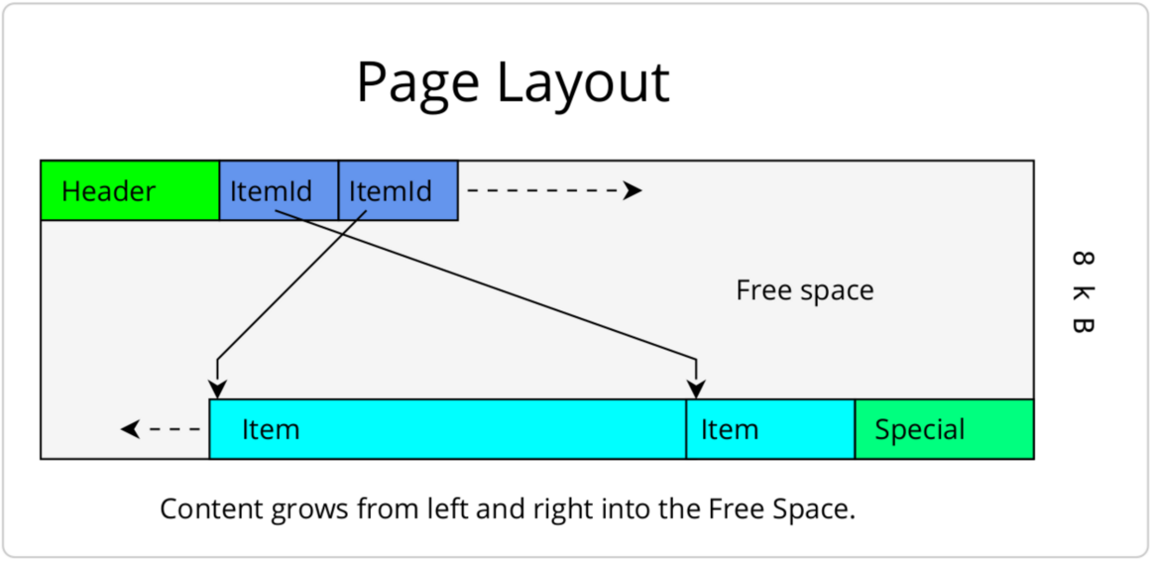


Figure 14 centre

The *Page Header* consists of 24 bytes and contains information like a page checksum, offset to start and end of free space or to special space, information for WAL handling, the layout version number, and some more flags.

Every *ItemId* consists of 4 bytes and contains the offset to and the length of the corresponding *Item*.

In the case of data files, every *Item* (= row) consists of:

- *Item Header*: 23 bytes containing various transaction IDs, current or newer tuple ID, offset to data, and various flags.
- *Null Bit Map*: Marker for such columns which are currently NULL. The map is optional: if a table contains only non-nullable columns, it is superfluous.
- *Data*: The value of every attribute of the row - if not NULL. The sequence and types of attributes are stored in the system schema. In the case of variable-length data types, the currently used length is stored at the beginning of the attribute.

In the case of index files, *Item*s consist of *index entries*, which are differently structured depending on the index type.

The *Special Space* is only used by index files for their individual purpose, e.g.: page number, page type, (double-)linked list within pages of the same level, tree-level, ... Data files don't need such additional information. They are organized as a heap, which means that their pages are logically equivalent without any special order or hierarchy between pages. Additional data is always put to the next free space or to a freshly allocated page.

Often, parts of data files are read and written directly by their page number. But it's not unusual that they are read completely. This is done by a *sequential scan*. Such comprehensive sequential accesses (as an analogy for following a linked list in an index file) are - in most cases - optimized by the underlying file system by its read-ahead technique. This helps especially for files located on a rotating disc (HDD) whereas for SSDs this advantage vanishes a little.

31.1 External links

PostgreSQL Documentation concerning page layout[1]

1 `https://www.postgresql.org/docs/current/storage-page-layout.html`

32 Indices

Relational databases systems store huge amounts of data. Their value only becomes apparent when individual pieces can be retrieved fast enough. E.g., a naive query for a specific telephone number in a phone book with 100 million entries has to read on average 50 million entries. Fortunately, smart algorithms reduce the number of necessary read operations **dramatically**. A binary search[1] will reduce them in the given example to maximal 27. Using smart algorithms is much more efficient than utilizing faster hardware - especially for huge numbers.

In our case of databases, the implementation of such algorithms is based on additional structures which repeat parts of the original data in their specific way. They are called *indices* and, of course, they come with some overhead. They occupy space on disc and in RAM; they generate additional effort, e.g., for sorting, and whenever the original data changes they must be maintained accordingly.

As mentioned, their primary purpose and biggest advantage is the acceleration of queries - with regards to identifying rows as well as sorting the resulting set of rows. Besides this, they support some constraints like *uniqueness*.

If indices exist, it's not sure that the system uses them. Because the system optimizes queries before it executes them, it sometimes decides to ignore an existing index and perform a full table scan instead. This may occur if the table is very small or if the selectiveness of the retrieved value is very low and will return a huge percentage of the existing rows.

PostgreSQL offers some extension mechanisms. Among other things, it is possible to add new data types to the system and integrate them into the existing index types. Beyond that, it's possible to develop application-specific operators to meet the needs of specialized applications, e.g., classification of pictures or music, clustering of arbitrary objects, detection of patterns in stock prices, ... GIN[2], BRIN[3], GiST[4], and SP-GiST[5] offer an interface (some kind of a template) which allows implementing index assisted domain-specific actions. The technique is called *access method*. Only B-Tree and Hash are conventional indices without such an extension mechanism.

32.1 B-Tree

1 `https://en.wikipedia.org/wiki/binary%20search`
2 GIN Extensibility `https://www.postgresql.org/docs/current/gin-extensibility.html`
3 BRIN Extensibility `https://www.postgresql.org/docs/current/brin-extensibility.html`
4 GiST Extensibility `https://www.postgresql.org/docs/current/gist-extensibility.html`
5 SP-GiST Extensibility `https://www.postgresql.org/docs/current/spgist-extensibility.html`

B-Tree[6] (Balanced Tree) is the default index type. It is suitable for use cases where numbers or short strings are often part of the WHERE clause. Possible operators are the usual arithmetic operators: <, <=, =, >, >=.

```
-- create a B-Tree index: key word 'USING' can be omitted
CREATE INDEX test_idx ON table_1 (column_1);
-- equivalent syntax:
CREATE INDEX test_idx ON table_1 USING BTREE(column_1);
-- use it
SELECT * FROM table_1 WHERE column_1 BETWEEN 5 AND 6;
```

Read more[7]

32.2 GIN

GIN (Generalized Inverted Index) supports data types that are divisible into smaller components, e.g., elements of an array, words of a text document, or properties of a JSON object. We call them *compound* data types. In opposite to B-Trees, GIN does not generate a single index entry for the complete value but one index entry for each individual component.

Useable operators in a WHERE clause depend on the data type:

- Arrays: `<@` (is contained in), `@>` (contains), `=` (equal), and `&&` (overlaps / has some common elements).
- Text queries (lexems): `@@` (contains).
- JSON: `->` (JSON object field with the given key), `->>` (JSON object field with the given key, as text).

```
-- create a table with a column that holds an array of integers
CREATE TABLE t2 (id INTEGER, arr INTEGER[]);

-- create a GIN index
CREATE INDEX t2_gin_idx ON t2 USING GIN(arr);

-- use the index
SELECT * FROM t2 WHERE arr @> ARRAY[11];
```

Read more[8]

32.3 BRIN

BRIN (Block-Range Index) is a structure that accelerates queries on tables that contain a huge number of rows (> millions) **and** where the rows occur in a certain physical order within the data file. Typical use cases are such where a column contains a timestamp or a generated sequence number that seldomly or never change over time, e.g.: IoT data, computed values, sensor output, log information.

6 Chapter 33 on page 127
7 Chapter 33 on page 127
8 Chapter 34 on page 131

The correlation between the physical order of rows in the data file and their content in the column of interest arises from the sequence of `INSERT` commands and growing column values: later `INSERT`s have to contain equal or higher values. It is possible that this correlation gets lost over time by later `UPDATE` commands. In this case, the benefit of BRIN may get lost.

The power of BRIN results from the fact that it needs only very little space. Typical BRIN sizes for a table with hundreds of millions of rows are some kB, which easily fits into RAM. All other index types need much more space, 25 - 50% of the table size is not unusual.

```
-- create a table with a timestamp column
CREATE TABLE t3 (id INTEGER, ts TIMESTAMP);

-- create a BRIN index
CREATE INDEX t3_brin_idx ON t3 USING BRIN(ts);

-- use the index in the usual way
SELECT * FROM t3 WHERE ts = '2022-01-01';
```

Read more[9]

32.4 Hash

PostgreSQL uses two fundamental strategies to implement *Hash indices*. First, a *hash function* maps column values of any type and length to a *hash value* of a fixed size of 32 bit. Such hash values together with the TIDs of their originating rows are the basic bricks for the Hash index. Second, an elaborated algorithm ensures that the size of the index file grows smoothly (that is, in a small amount of pages at one point in time) when additional index entries occur. Hence, it's an extendible hash[10].

To save space, the hash index doesn't store the original column value but only the computed hash value. This has some implications. The sort order of the computed hash values haven't any relation to the sort order of the original values. Therefore this index type can support only the `=` operator, but none of the other comparison operators like `<` or `>`. Additionally, there is the danger of duplicates. Two different column values can create the same hash value. This is unavoidable because there are many more possible column values (**any** length) than possible hash values (**fixed** size). Thus, after reading the row according to the found TID, it's necessary to re-evaluate the column value from the heap.

```
DROP TABLE IF EXISTS t5;

-- create a table with a UUID column and some text
CREATE TABLE t5 (id INTEGER, pseudo_id UUID, col TEXT);

-- insert some rows
INSERT INTO t5 VALUES (1, md5('1')::uuid, 'First row.');
INSERT INTO t5 VALUES (2, md5('2')::uuid, 'Second row.');
INSERT INTO t5 VALUES (3, md5('3')::uuid, 'Third row.');
-- ...

-- insert many rows
```

9 Chapter 35 on page 135
10 `https://en.wikipedia.org/wiki/extendible%20hashing`

```
INSERT INTO t5 VALUES
       (generate_series(10, 10000, 1),
    md5(generate_series(10, 10000, 1)::text)::uuid,
   'more text more text more text more text more text more text more text');

-- create a HASH index over UUID column
CREATE INDEX t5_hash_idx ON t5 USING HASH(pseudo_id);

-- use the index
SELECT * FROM t5 WHERE pseudo_id = md5('2')::uuid;

-- show index usage
EXPLAIN SELECT * FROM t5 WHERE pseudo_id = md5('2')::uuid;
```

Read more[11]

32.5 GiST and SP-GiST

GiST[12] stands for *Generalized Search Tree* and implements - similar to B-Tree - a balanced tree structure. This is useful for all kinds of B-Tree and R-Tree structures. Some PostgreSQL extensions use them, e.g.:, hstore (key/value pairs), intarray (array of Integers), ltree ('Labels' like 'World.Countries.Europe.Russia'), pg_trgm (trigram matching), ...

SP-Gist[13] stands for *Space-Partitioned Generalized Search Tree* and implements non-balanced tree structures, mainly for object types that contain similar or equal object types. This is useful for quad-trees, k-d trees, radix trees, ...

32.6 Bloom

The above-mentioned index types and index access methods are an integral part of every PostgreSQL installation.

An additional index access method is *bloom* [14]. *Bloom* must be explicitly installed using PG's extension mechanism[15] (you have to run `CREATE EXTENSION bloom;` once before you can use this index type). This extension implements a Bloom filter[16] that offers an advantage over B-trees in cases where the first columns of a multicolumn index are not specified in the `WHERE` condition of an SQL statement.

32.7 External links

PostgreSQL documentation concerning index types[17]

11 Chapter 36 on page 139
12 https://www.postgresql.org/docs/current/gist.html
13 https://www.postgresql.org/docs/current/spgist.html
14 Bloom filer https://www.postgresql.org/docs/current/bloom.html
15 Chapter 27 on page 103
16 https://en.wikipedia.org/wiki/Bloom_filter
17 https://www.postgresql.org/docs/current/indexes-types.html

32.8 References

33 B-tree

The term *B-tree index* denotes the implementation of a balanced tree[1]. B-trees are characterized by the criterion that the distance from the root node to every leaf node is the same. Such trees support the very fast evaluation of search criteria like `WHERE status=5` . In most cases, such trees have a high branching factor and hence a low depth. If, for example, there is a branching factor of 500, the tree can manage about 125 million entries with 3 page-reads.

In addition, the PostgreSQL implementation optimizes the locking behavior of concurrent write operations to the tree with a strategy that was originally proposed by Lehmann and Yao[2]. The idea is to add additional (redundant as of the perspective of the complete tree) pointers to every page.

33.1 SQL syntax

B-tree is the default index type. It is created by the SQL command CREATE INDEX when the keyword `USING` is omitted.

```
-- create a b-tree index
CREATE INDEX test_idx ON table_1 (column_1);
-- equivalent syntax:
CREATE INDEX test_idx ON table_1 USING BTREE(column_1);
```

33.2 Description

The file containing the B-tree consists of different page types.

- The very first page (#0) of the file holds *meta*-information about the index, e.g., the pointer to the root page, which is not always located on page #1, or the current tree-depth.
- *Internal pages* contain pairs of keys and pointers. Keys hold the values which shall be indexed, and the pointers point to internal pages of the next level or to leaf pages. Those pairs are denoted *index entries.*
- *Leaf pages* contain such pairs as well. But in this case, they point to pages and rows in the data file (heap). Such pointers are called *TupleId*s or *TID*s.
- Internal pages plus leaf pages constitute the B-tree.

1 `https://en.wikipedia.org/wiki/b-tree`
2 `https://dl.acm.org/doi/10.1145/319628.319663`

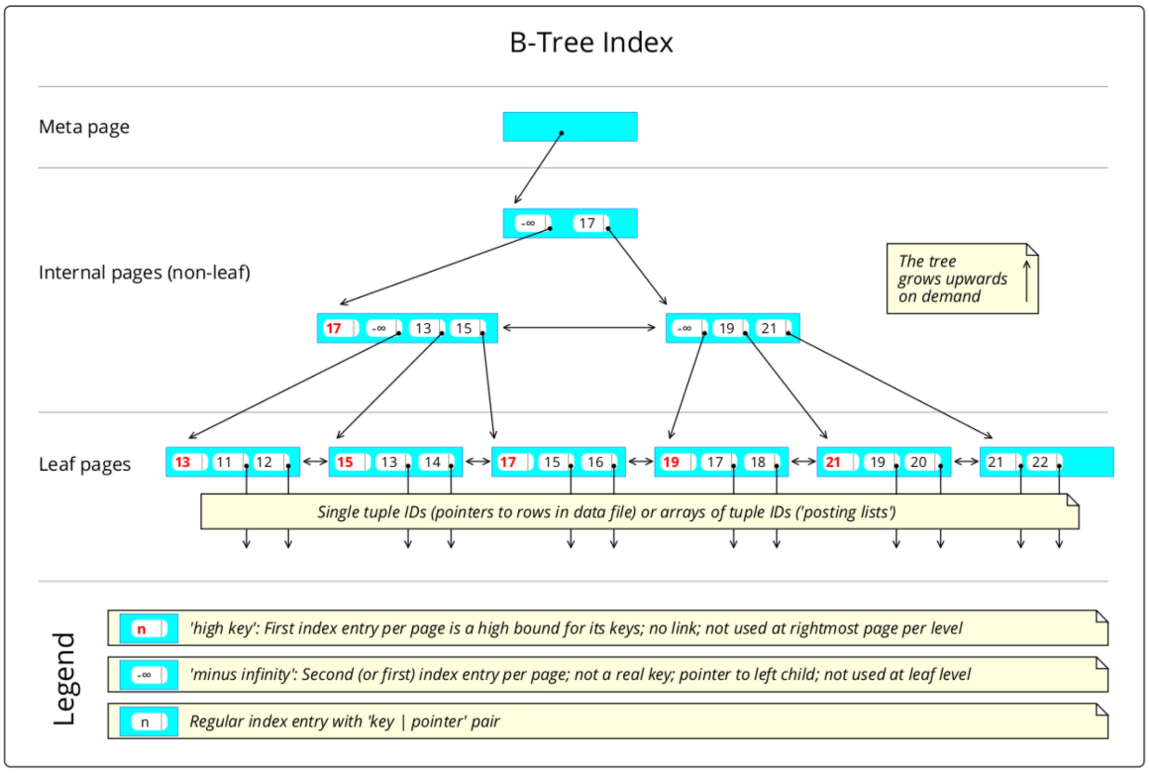

B-Tree Index
Meta page
Internal pages (non-leaf)
Leaf pages
The tree grows upwards on demand
Single tuple IDs (pointers to rows in data file) or arrays of tuple IDs ('posting lists')
Legend
'high key': First index entry per page is a high bound for its keys; no link; not used at rightmost page per level
'minus infinity': Second (or first) index entry per page; not a real key; pointer to left child; not used at leaf level
Regular index entry with 'key | pointer' pair

Figure 15 centre

Over time, pages need to be split because their capacity gets exhausted. First, the tree grows in breadth. In rare cases, it gets necessary that the high of the tree must be extended. In this case, the root page gets split and a new root page is created.

There are some special rules to optimize the access to B-trees, especially to reduce the probability of locks in a multiuser environment. Therefore the pages contain some additional information that enhances a pure B-tree implementation.

- The first index entry of every page contains a value, which is treated as an upper bound for all keys of this page. It does not contain a pointer to another page. It is called the *high key*, and in the above graphic, it is shown in red color. The rightmost page of every level doesn't contain a high key. It's *plus infinity* per definition.
- The second (or first in the case of no high key) index entry points to the left child of the page. It does not contain a real key. Sometimes it's called *minus infinity*. Leaf pages don't use it.
- The pages of every level are connected to each other via a double linked list[3]. This helps to speed up queries like `WHERE status>17` because the need to traverse the tree upwards disappears.

3 `https://en.wikipedia.org/wiki/double%20linked%20list`

33.3 Statements to create the shown B-tree

```
-- PostgreSQL version 14.1 at Ubuntu 20.4

-- a helper to create huge text values via 'gen_random_bytes()'
CREATE EXTENSION IF NOT EXISTS pgcrypto;

-- a helper to inspect physical pages
CREATE EXTENSION IF NOT EXISTS pageinspect;

-- table with a text field
DROP TABLE IF EXISTS t1;
CREATE TABLE t1 (
  key integer,
  val text       -- will be indexed with B-tree
);

-- insert huge text values to enforce page splits in index file
INSERT INTO t1
  (SELECT
    generate_series(11, 22, 1),
    concat(generate_series(11, 22, 1), '  ', gen_random_bytes(1024)::text)
  );
-- same as:
-- INSERT INTO t1 VALUES (11, '11  ' || gen_random_bytes(1024)::text);
-- INSERT INTO t1 VALUES (12, '12  ' || gen_random_bytes(1024)::text);
-- ...

-- create the B-tree
CREATE INDEX t1_btree_idx ON t1 (val);

-- Inspect the pages of the created B-tree
-- read meta page: it shows that page 9 is the root of the B-tree
SELECT * FROM bt_metap('t1_btree_idx');

-- read page 9: root page of B-tree
SELECT itemoffset, ctid, itemlen, nulls, vars, left(data, 23) AS data FROM
 bt_page_items('t1_btree_idx', 9);
-- read pages 3 + 8 (internal pages)
SELECT itemoffset, ctid, itemlen, nulls, vars, left(data, 23) AS data FROM
 bt_page_items('t1_btree_idx', 3);
SELECT itemoffset, ctid, itemlen, nulls, vars, left(data, 23) AS data FROM
 bt_page_items('t1_btree_idx', 8);
-- read pages 1, 2, 4 and 5, 6, 7 (leaf pages)
SELECT itemoffset, ctid, itemlen, nulls, vars, left(data, 23) AS data FROM
 bt_page_items('t1_btree_idx', 1);
SELECT itemoffset, ctid, itemlen, nulls, vars, left(data, 23) AS data FROM
 bt_page_items('t1_btree_idx', 2);
SELECT itemoffset, ctid, itemlen, nulls, vars, left(data, 23) AS data FROM
 bt_page_items('t1_btree_idx', 4);
SELECT itemoffset, ctid, itemlen, nulls, vars, left(data, 23) AS data FROM
 bt_page_items('t1_btree_idx', 5);
SELECT itemoffset, ctid, itemlen, nulls, vars, left(data, 23) AS data FROM
 bt_page_items('t1_btree_idx', 6);
SELECT itemoffset, ctid, itemlen, nulls, vars, left(data, 23) AS data FROM
 bt_page_items('t1_btree_idx', 7);

-- show the TIDs per key (in heap file, NOT in index file)
SELECT key, ctid FROM t1;
```

33.4 External links

PostgreSQL Documentation concerning B-tree implementation[4]

4 https://www.postgresql.org/docs/current/btree-implementation.html

34 GIN

GIN (Generalized Inverted Index) supports data types that are divisible into smaller components, e.g., elements of an array, words of a text document, or properties of a JSON object. We call them *compound* data types. Unlike B-Trees, GIN does not generate a single index entry for the complete value but one index entry for each component.

Every index entry consists of the value of the individual component plus the tuple ID (TID). Please notice that TIDs don't contain the sequence number of the component within the complete value. They contain only the number of the physical page in the data file plus the number of the row within the page.

Useable operators in a `WHERE` clause depend on the data type:

- Arrays: `<@` (is contained in), `@>` (contains), `=` (equal), and `&&` (overlaps / has some common elements).
- Text queries (lexems): `@@` (contains).
- JSON: `->` (JSON object field with the given key), `->>` (JSON object field with the given key, as text).

34.1 SQL syntax

```
-- create a table with a column that holds an array of integers
DROP TABLE IF EXISTS t2;
CREATE TABLE t2 (id INTEGER, arr INTEGER[]);
INSERT INTO t2 VALUES (1,  ARRAY [11, 12, 13]);
INSERT INTO t2 VALUES (2,  ARRAY [21, 22, 23]);

-- insert a lot of other rows to enforce index usage
INSERT INTO t2 (SELECT generate_series(3, 10000, 1), ARRAY[0, 1, 2, 3]);


-- ------------------------------------------
--          create the GIN index
-- ------------------------------------------
CREATE INDEX t2_gin_idx ON t2 USING GIN(arr);


-- use the index
ANALYSE t2;
EXPLAIN SELECT * FROM t2 WHERE arr @> ARRAY[11];
```

34.2 Description

A GIN index consists of different sub-structures:

- The *Meta Page*

- One *Entry B-Tree*
- Some *Posting B-trees*
- One *Pending List*

Among others, the *Meta Page* contains pointers to the *Entry B-Tree* and the *Pending List*.

The *Entry B-Tree* implements a tree where the keys consist of the original component's values, e.g., the value of a single array element. At the non-leaf levels, their pointers point to child pages at the next level. At the leaf level, pages consist of two types of entries: First, there are *Posting Lists*. They consist of the key, followed by a list of TIDs. Second, if such a list of TIDs exceeds the capacity of the physical page, the list gets rearranged to a *Posting B-Tree*, which is stored on one or more other pages. The original *Posting List* gets replaced by a pointer to the new *Posting B-Tree*.

A *Posting B-Tree* is part of one or more physical pages and contains a B-Tree over tuple IDs which all point to such rows in the data file where its key can be found as the value of one of its components.

The implementation of the two B-Tree types differs from PostgreSQL's standard B-Tree[1] implementation.

The *Pending List* is a list of pages where keys (component's values) and their dedicated TIDs are stored sequentially. The *Pending List* exists for optimization purposes; see below.

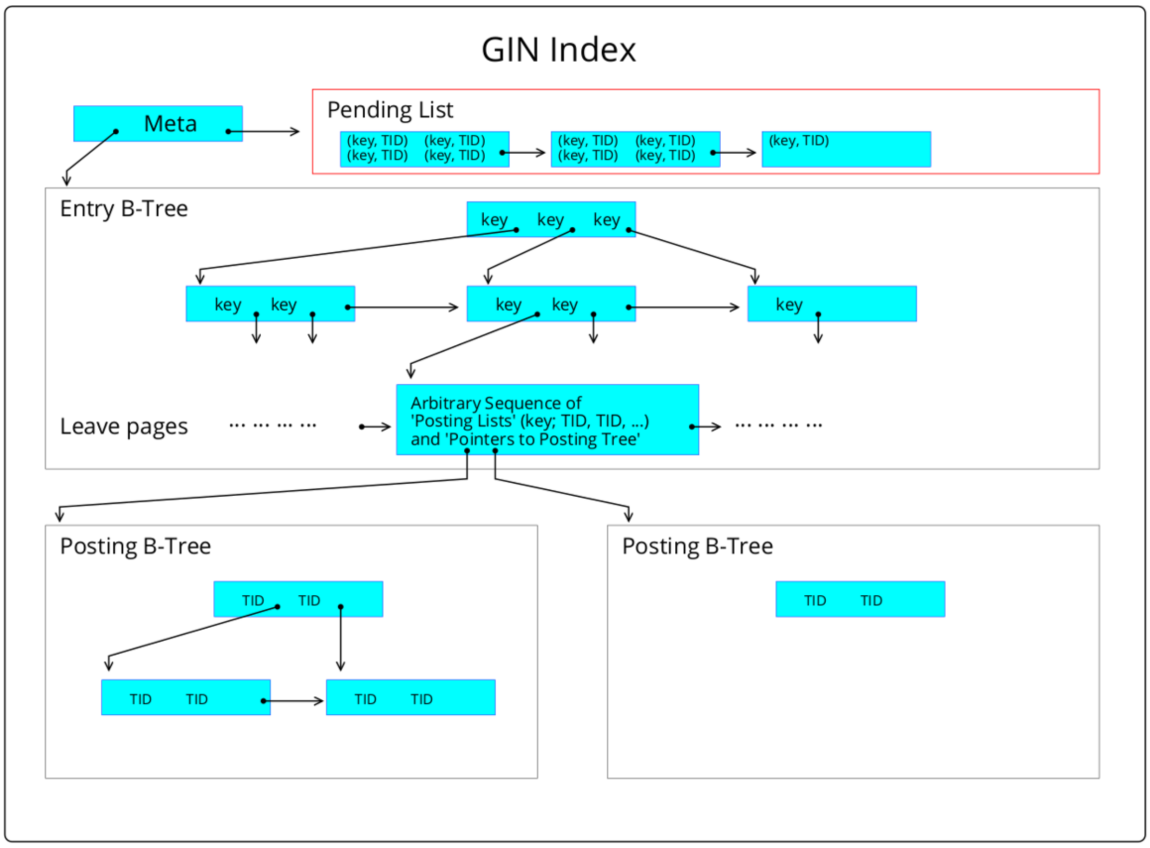

GIN Index
Meta
Pending List
(key, TID) (key, TID) (key, TID) (key, TID)
(key, TID) (key, TID) (key, TID) (key, TID)
(key, TID)
Entry B-Tree
key key key
key key
key key
key
Leave pages
Arbitrary Sequence of 'Posting Lists' (key; TID, TID, ...) and 'Pointers to Posting Tree'
Posting B-Tree
TID TID
Posting B-Tree
TID TID

Figure 16 centre

1 Chapter 33 on page 127

34.3 Optimizations

GIN indices use two special optimizations. First, if the value of different components (possibly in different rows) is used often, the set of TIDs is rearranged to a B-Tree within the GIN. Consider the case of many text documents: Many words will likely be repeatedly used in the same and in different documents. In this case, the list of TIDs may grow to the thousands, and a tree is better suited to manage them than a list.

Second, `INSERT`s or `UPDATE`s of compound data creates many index entries: one per component, e.g., one per word of a text. In the first step, such new index entries are collected in a separate *Pending List* outside of the index tree (in opposite to the original `CREATE INDEX` command). During the next `VACUUM`-run, the entries are moved from the pending list to the GIN tree structure using the same bulk insert technique used during initial index creation. This bulk technique speeds up the process, and - even more helpful - the work is delegated to a background process. Of course, there is a drawback: Every query must scan the pending list in addition to traversing the index tree.

34.4 External links

PostgreSQL Documentation concerning GIN in general[2]

PostgreSQL Documentation concerning GIN implementation[3]

2 `https://www.postgresql.org/docs/current/gin.html`
3 `https://www.postgresql.org/docs/current/gin-implementation.html`

35 BRIN

BRIN (Block-Range Index) is a structure that accelerates queries on tables that contain a huge number of rows (> millions) **and** where the rows occur in a certain physical order within the data file. Typical use cases are such where a column contains a timestamp or a generated sequence number that seldomly or never change over time, e.g.: IoT data, computed values, sensor output, log information.

You can imagine a BRIN as a 'virtual partitioning' of a table. If a query fits into such a virtual partition the number of rows, which must be scanned, decreases significantly.

The power of BRIN results from the fact that it needs only very little space. Typical BRIN sizes for a table with hundreds of millions of rows are some kB, which easily fits into RAM. All other index types need much more space, 25 - 50% of the table size is not unusual.

35.1 SQL syntax

```
DROP TABLE IF EXISTS t3;

-- create a table with a timestamp column
CREATE TABLE t3 (id INTEGER, ts TIMESTAMP);

-- insert data
INSERT INTO t3 VALUES (1, '2022-01-01 00:00:01');
INSERT INTO t3 VALUES (2, '2022-01-01 00:00:02');
-- ...

-- create a BRIN index
CREATE INDEX t3_brin_idx ON t3 USING BRIN(ts);

-- use the index with the usual operators. (As far as there are less than
 100,000 rows, the index is not used.)
SELECT * FROM t3 WHERE ts = '2022-01-01 00:00:02';
```

35.2 Description

Hint: The PostgreSQL terminology knows the term 'block' and uses it synonymously with 'page' (8192 bytes), see here[1]. In the context of BRIN, the term 'block-range' denotes a contiguous sequence of many adjacent pages within a data file.

When a BRIN is created, the sequence of pages of the data file (heap) is virtually divided into slices, called *block-ranges*. E.g., if the file contains 600 pages, the first 128 belongs to block-range #1, the second 128 belongs to block-range #2, ... up to block-range #5, which contains the remaining number of pages. The default size for a block-range is 128 pages; it can be changed within the `CREATE INDEX` command. Next, all rows are scanned and the

1 Chapter 31 on page 119

minimum and maximum values of the indexed column per block-range are saved. Please note that each min/max pair constitutes a numeric *value-range*.

The BRIN structure consists of those block-range numbers and their related value-ranges, e.g., block-range #1: min=11, max=25; block-range #2: min=25, max=31. Hence the name *Block Range Index*.

Ideally, the value-ranges don't overlap, but this is not necessary. The correlation between the order of rows in the data file and their content in the column of interest arises from the sequence of INSERT commands combined with growing values: later INSERTs (to a table without significant free space) should contain equal or higher values. This correlation may get lost over time by later UPDATE commands. In this case, the benefit of BRIN may get lost.

When such a BRIN structure is used during the execution of a query, it is known that all values within the rows of a certain block-range are within its data-range or, vice versa, no value outside its value-range is in any of its columns. But it's unknown which concrete values are really in the data! This has significant consequences for using the BRIN structure; see below.

Complex data types, e.g., geometric objects like rectangles, store more complex data instead of min/max values, e.g., a bounding box.

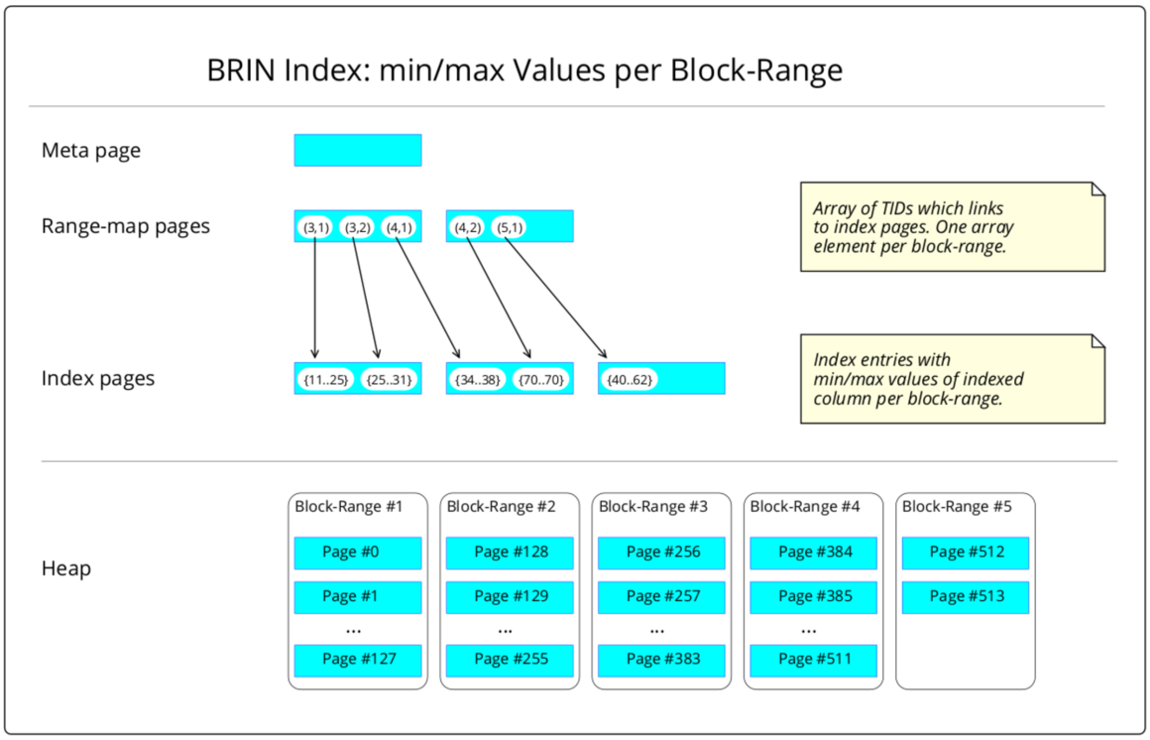


Figure 17 centre

The file containing the BRIN consists of 3 different page types.

- The very first page (#0) of the file holds meta-information, e.g., the number of *Range-map* pages.

- The second (#1) and some more pages contain the so-called *Range-map*. It consists of tuple IDs (TID) which point to pages of the next BRIN level, the *Index Pages*. Because TIDs have a fixed size (of 6 bytes) it's possible to store them like an array: one after the next without any links between them. Their position correlates with the block-range number.
- The rest of the pages contain the *Index Pages*. They contain the minimum and maximum values (value-ranges) per block-range.

35.3 Modus Operandi

35.3.1 Select

If the `WHERE` condition of an SQL command specifies a criterion for a column with a BRIN, the following steps are conducted:

- All range-map pages are entirely scanned.
- The related index pages are read one by one. If the searched value fits into the value-range of any of its index entries, **all** pages of this block-range are considered part of the result set.
- All rows of the identified pages are read from the heap and their columns evaluated. This is necessary because BRIN knows only that the value of every row must be in a certain range, e.g., between 11 and 25. But if the search criterium is `WHERE col = 20`, the identified rows potentially contain other values like 11 or 15.

Summary: BRIN doesn't contain exact pointers to certain rows in the heap file. It contains only information about ranges of blocks and ranges of values. Nevertheless, under certain conditions (huge number of rows, correlation between physical row order and column value, few data changes) this information is enough to reduce the number of rows, which must be read to evaluate a search condition, dramatically. The uncertainness of BRIN correlates with its tiny size.

35.3.2 Update

If a row is added or the BRIN-column of a row changes, the following steps are conducted:

- The page number, where the row is physically stored in the data file, is identified.
- Depending on the page number, the block-range number is computed (page number divided by block-range-size).
- The block-range number determines the position within the range-map.
- The related index entry at the index page is read.
- If the new value of the row fits into the value-range of this index entry, no action is necessary. If the value is outside of the value-range, the value-range is updated (enlarged) on the lower or upper bound.

It is possible that value-ranges overlap. This decreases the efficiency of BRIN.

35.4 SQL syntax - more

```
DROP TABLE IF EXISTS t4;

-- create a table whose rows occupy about 200 byte each
CREATE TABLE t4 (
  id INTEGER,
  ts TIMESTAMP,
  some_space TEXT NOT NULL DEFAULT
              'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ ' ||
              'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ ' ||
              'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ ')
;

-- insert 1 million rows
INSERT INTO t4 (id, ts)
  (SELECT generate_series(1, 1000000, 1),
          generate_series(now(), now() + '1000000 second', '1 second'));

-- size of heap: about 200 MB
SELECT pg_size_pretty(pg_total_relation_size('t4'));

-- create BRIN and show its size: about 48 kB
CREATE INDEX t4_brin_idx ON t4 USING BRIN(ts);
SELECT pg_size_pretty(pg_total_relation_size('t4_brin_idx'));

-- create BTREE and show its size: about 21 MB
CREATE INDEX t4_btree_idx ON t4 USING BTREE(ts);
SELECT pg_size_pretty(pg_total_relation_size('t4_btree_idx'));

-- size of BRIN to BTREE is about 1 : 400

-- ----------------------------------------------------------------

-- show the meta page of BRIN (page #0)
SELECT * FROM brin_metapage_info(get_raw_page('t4_brin_idx', 0));

-- show (the only) revmap page of BRIN (page #1)
SELECT * FROM brin_revmap_data(get_raw_page('t4_brin_idx', 1)) where pages
 !='(0,0)';

-- show index pages (in this example there is only a single one: page #2)
SELECT itemoffset, blknum, value
FROM   brin_page_items(get_raw_page('t4_brin_idx', 2), 't4_brin_idx')
ORDER BY itemoffset;
```

35.5 External links

PostgreSQL Documentation concerning BRIN[2]

2 https://www.postgresql.org/docs/current/brin.html

36 Hash

PostgreSQL uses two fundamental strategies to implement *Hash indices*. First, a *hash function* maps column values of any type and length to a *hash value* of a fixed size of 32 bit. Such hash values together with the TIDs of their originating rows are the basic bricks for the Hash index. Second, an elaborated algorithm ensures that the size of the index file grows smoothly (that is, in a small amount of pages at one point in time) when additional index entries occur. Hence, it's an extendible hash[1].

To save space, the hash index doesn't store the original column value but only the computed hash value. This has some implications. The sort order of the computed hash values haven't any relation to the sort order of the original values. Therefore this index type can support only the = operator, but none of the other comparison operators like < or >. Additionally, there is the danger of duplicates. Two different column values can create the same hash value. This is unavoidable because there are many more possible column values (**any** length) than possible hash values (**fixed** size). Thus, after reading the row according to the found TID, it's necessary to re-evaluate the column value from the heap.

In most cases - especially for long column values - the size of a hash index is smaller than the size of a B-tree index. Also, the execution time of reading and writing commands is often shorter.

The central part of a hash index consists of so-called *buckets*. They are a double-linked list of pages, where *bucket entries* are stored. The first page of a bucket can be accessed very fast because of a correlation between its number and certain bits of the hash value.

36.1 SQL syntax

```
DROP TABLE IF EXISTS t5;

-- create a table with a UUID column and some text
CREATE TABLE t5 (id INTEGER, pseudo_id UUID, col TEXT);

-- insert some rows
INSERT INTO t5 VALUES (1, md5('1')::uuid, 'First row.');
INSERT INTO t5 VALUES (2, md5('2')::uuid, 'Second row.');
INSERT INTO t5 VALUES (3, md5('3')::uuid, 'Third row.');
-- ...

-- insert many rows
INSERT INTO t5 VALUES
       (generate_series(10, 10000, 1),
    md5(generate_series(10, 10000, 1)::text)::uuid,
   'more text more text more text more text more text more text more text');
```

1 https://en.wikipedia.org/wiki/extendible%20hashing

```
-- create a HASH index over UUID column
CREATE INDEX t5_hash_idx ON t5 USING HASH(pseudo_id);

-- use the index
SELECT * FROM t5 WHERE pseudo_id = md5('2')::uuid;

-- show index usage
EXPLAIN SELECT * FROM t5 WHERE pseudo_id = md5('2')::uuid;
```

36.2 Description

During the creation and maintaining of a Hash index, several steps are conducted:

- The column value and the TID of its original row are read.
- A *hash functions* computes a *hash value* out of the original column value. Its size is always 32 bit - independent of its data type and data length.
- The combination of hash value and TID builds a *bucket entry*.
- There are several *buckets*, where the bucket entries are stored. Certain bits of the hash value determine which bucket owns and receives the bucket entry.
- Buckets consist of a *primary bucket page* plus optional *overflow bucket page*s. Within the bucket, pages are linked together via a double-linked list.
- If the primary bucket page and all its overflow pages don't offer a free slot for the new bucket entry, a new overflow page is created.
- The ratio of the number of existing buckets to all bucket entries is computed. Depending on this value and the chosen `fillfactor` of the index, new buckets are created on-demand.

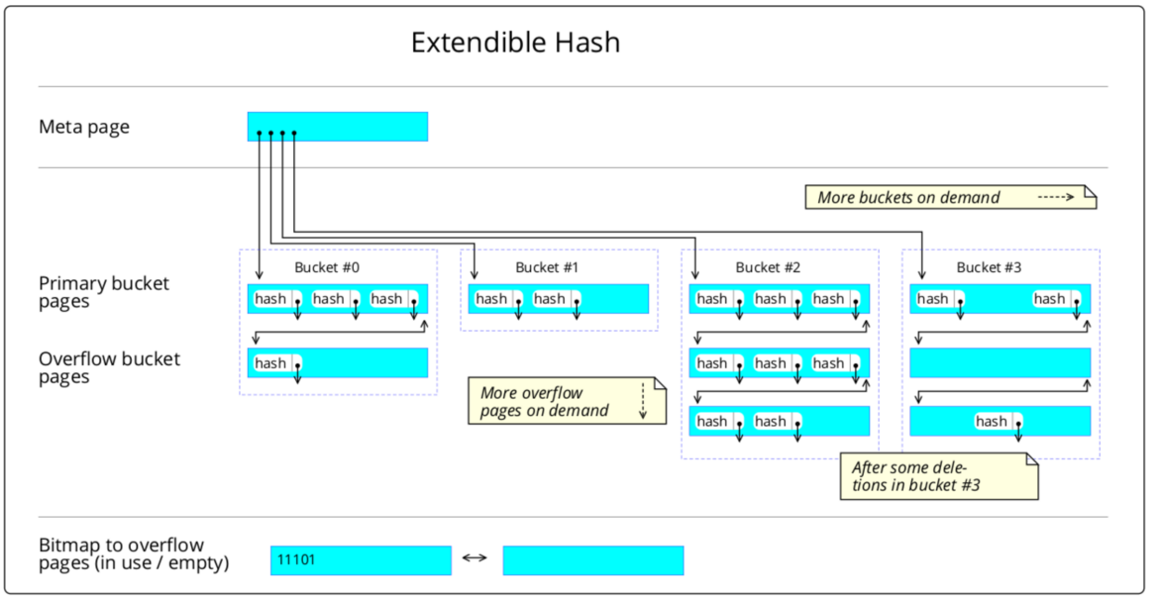


Figure 18 centre

The *meta page* contains statistical data about the index: number of buckets and bucket entries, an array of links to buckets (hashm_spares), and more.

The *primary* and *overflow bucket pages* contain pairs of hash values and TIDs that point to rows of the heap.

The *bitmap pages* contain an array of bits, which indicates that there may be unused overflow pages (after DELETE operations to the according rows). Those pages can be reused by other buckets.

36.3 Inspect Hash Index Pages

```
DROP TABLE IF EXISTS t6;

-- create a table with a UUID column and some text
CREATE TABLE t6 (id INTEGER, pseudo_id UUID, col TEXT);

-- insert data
-- INSERT INTO t6 VALUES (1, md5('1')::uuid, 'First row.');
-- ...

-- insert many rows
INSERT INTO t6 VALUES
       (generate_series(10, 10000, 1),
    md5(generate_series(10, 10000, 1)::text)::uuid,
   'abc abc abc abc abc abc abc abc abc abc abc');

-- create a HASH index
CREATE INDEX t6_hash_idx ON t6 USING HASH(pseudo_id) WITH (fillfactor = 50);

-- inspect size
SELECT pg_size_pretty(pg_total_relation_size('t6_hash_idx'));

-- inspect physical pages
-- page type of any page
SELECT hash_page_type(get_raw_page('t6_hash_idx', 0));
-- infos out of meta page
\x
SELECT * FROM hash_metapage_info(get_raw_page('t6_hash_idx', 0));

-- infos out of primary bucket or overflow bucket pages
SELECT * FROM hash_page_stats(get_raw_page('t6_hash_idx', 1));
SELECT * FROM hash_page_items(get_raw_page('t6_hash_idx', 1)) LIMIT 20;
```

36.4 External links

PostgreSQL Documentation concerning Hash indices[2]

2 https://www.postgresql.org/docs/current/hash-index.html

37 Contributors

Edits	User
2	1234qwer1234qwer4[1]
1	Agusbou2015[2]
2	Alex Schröder[3]
1	BethNaught[4]
1	Bimsarayapa1989[5]
7	Chelnik[6]
48	DC Slagel[7]
26	DannyS712[8]
1	Dirk Hünniger[9]
3	Fishpi[10]
2	Intgr[11]
23	JackBot[12]
18	JackPotte[13]
1	Jellysandwich0[14]
178	Kelti[15]
5	Mild Bill Hiccup[16]
1	Neils51[17]
1	NguoiDungKhongDinhDanh[18]
3	Tech201805[19]

1 https://en.wikibooks.org/wiki/User:1234qwer1234qwer4
2 https://en.wikibooks.org/wiki/User:Agusbou2015
3 https://en.wikibooks.org/w/index.php%3ftitle=User:Alex_Schr%25C3%25B6der&action=edit&redlink=1
4 https://en.wikibooks.org/wiki/User:BethNaught
5 https://en.wikibooks.org/w/index.php%3ftitle=User:Bimsarayapa1989&action=edit&redlink=1
6 https://en.wikibooks.org/wiki/User:Chelnik
7 https://en.wikibooks.org/wiki/User:DC_Slagel
8 https://en.wikibooks.org/wiki/User:DannyS712
9 https://en.wikibooks.org/wiki/User:Dirk_H%25C3%25BCnniger
10 https://en.wikibooks.org/w/index.php%3ftitle=User:Fishpi&action=edit&redlink=1
11 https://en.wikibooks.org/wiki/User:Intgr
12 https://en.wikibooks.org/wiki/User:JackBot
13 https://en.wikibooks.org/wiki/User:JackPotte
14 https://en.wikibooks.org/w/index.php%3ftitle=User:Jellysandwich0&action=edit&redlink=1
15 https://en.wikibooks.org/wiki/User:Kelti
16 https://en.wikibooks.org/w/index.php%3ftitle=User:Mild_Bill_Hiccup&action=edit&redlink=1
17 https://en.wikibooks.org/w/index.php%3ftitle=User:Neils51&action=edit&redlink=1
18 https://en.wikibooks.org/wiki/User:NguoiDungKhongDinhDanh
19 https://en.wikibooks.org/wiki/User:Tech201805

11	Uziel302[20]
1	ZI Jony[21]

20 https://en.wikibooks.org/wiki/User:Uziel302
21 https://en.wikibooks.org/wiki/User:ZI_Jony

List of Figures

is reproduction in a format without relief (drawings, paintings, films) provided they are not detrimental to the image of the euro.

- LFK: Lizenz Freie Kunst. `http://artlibre.org/licence/lal/de`
- CFR: Copyright free use.
- EPL: Eclipse Public License. `http://www.eclipse.org/org/documents/epl-v10.php`

Copies of the GPL, the LGPL as well as a GFDL are included in chapter Licenses[22]. Please note that images in the public domain do not require attribution. You may click on the image numbers in the following table to open the webpage of the images in your webbrower.

22 Chapter 38 on page 149

1	JackPotte[23], JackPotte[24]	CC-BY-SA-4.0
2	Alex J. Ivasyuv	PD
3	Kelti[25], Kelti[26]	CC-BY-SA-4.0
4	Kelti[27], Kelti[28]	CC-BY-SA-4.0
5	Kelti[29], Kelti[30]	CC-BY-SA-4.0
6	Kelti[31], Kelti[32]	CC-BY-SA-4.0
7	Kelti[33], Kelti[34]	CC-BY-SA-4.0
8	Kelti[35], Kelti[36]	CC-BY-SA-4.0
9	Kelti[37], Kelti[38]	CC-BY-SA-4.0
10	Kelti[39], Kelti[40]	CC-BY-SA-4.0
11	Kelti[41], Kelti[42]	CC-BY-SA-4.0
12	Kelti[43], Kelti[44]	CC-BY-SA-4.0
13	Kelti[45], Kelti[46]	CC-BY-SA-4.0
14	Kelti[47], Kelti[48]	CC-BY-SA-4.0
15	Kelti[49], Kelti[50]	CC-BY-SA-4.0
16	Kelti[51], Kelti[52]	CC-BY-SA-4.0
17	Kelti[53], Kelti[54]	CC-BY-SA-4.0
18	Kelti[55], Kelti[56]	CC-BY-SA-4.0

23 http://commons.wikimedia.org/wiki/User:JackPotte
24 https:///wiki/User:JackPotte
25 http://commons.wikimedia.org/w/index.php?title=User:Kelti&action=edit&redlink=1
26 https:///w/index.php?title=User:Kelti&action=edit&redlink=1
27 http://commons.wikimedia.org/w/index.php?title=User:Kelti&action=edit&redlink=1
28 https:///w/index.php?title=User:Kelti&action=edit&redlink=1
29 http://commons.wikimedia.org/w/index.php?title=User:Kelti&action=edit&redlink=1
30 https:///w/index.php?title=User:Kelti&action=edit&redlink=1
31 http://commons.wikimedia.org/w/index.php?title=User:Kelti&action=edit&redlink=1
32 https:///w/index.php?title=User:Kelti&action=edit&redlink=1
33 http://commons.wikimedia.org/w/index.php?title=User:Kelti&action=edit&redlink=1
34 https:///w/index.php?title=User:Kelti&action=edit&redlink=1
35 http://commons.wikimedia.org/w/index.php?title=User:Kelti&action=edit&redlink=1
36 https:///w/index.php?title=User:Kelti&action=edit&redlink=1
37 http://commons.wikimedia.org/w/index.php?title=User:Kelti&action=edit&redlink=1
38 https:///w/index.php?title=User:Kelti&action=edit&redlink=1
39 http://commons.wikimedia.org/w/index.php?title=User:Kelti&action=edit&redlink=1
40 https:///w/index.php?title=User:Kelti&action=edit&redlink=1
41 http://commons.wikimedia.org/w/index.php?title=User:Kelti&action=edit&redlink=1
42 https:///w/index.php?title=User:Kelti&action=edit&redlink=1
43 http://commons.wikimedia.org/w/index.php?title=User:Kelti&action=edit&redlink=1
44 https:///w/index.php?title=User:Kelti&action=edit&redlink=1
45 http://commons.wikimedia.org/w/index.php?title=User:Kelti&action=edit&redlink=1
46 https:///w/index.php?title=User:Kelti&action=edit&redlink=1
47 http://commons.wikimedia.org/w/index.php?title=User:Kelti&action=edit&redlink=1
48 https:///w/index.php?title=User:Kelti&action=edit&redlink=1
49 http://commons.wikimedia.org/w/index.php?title=User:Kelti&action=edit&redlink=1
50 https:///w/index.php?title=User:Kelti&action=edit&redlink=1
51 http://commons.wikimedia.org/w/index.php?title=User:Kelti&action=edit&redlink=1
52 https:///w/index.php?title=User:Kelti&action=edit&redlink=1
53 http://commons.wikimedia.org/w/index.php?title=User:Kelti&action=edit&redlink=1
54 https:///w/index.php?title=User:Kelti&action=edit&redlink=1
55 http://commons.wikimedia.org/w/index.php?title=User:Kelti&action=edit&redlink=1
56 https:///w/index.php?title=User:Kelti&action=edit&redlink=1

38 Licenses

38.1 GNU GENERAL PUBLIC LICENSE

those terms and this License would be to refrain entirely from conveying the Program. 13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such. 14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version. 15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. 16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. 17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.> Copyright (C) <year> <name of author>

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

<program> Copyright (C) <year> <name of author> This program comes with ABSOLUTELY NO WARRANTY; for details type 'show w'. This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see <http://www.gnu.org/licenses/>.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read <http://www.gnu.org/philosophy/why-not-lgpl.html>.

38.2 GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed. 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference. 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License. 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies. 3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document. 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

* A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission. * B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement. * C. State on the Title page the name of the publisher of the Modified Version, as the publisher. * D. Preserve all the copyright notices of the Document. * E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices. * F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below. * G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice. * H. Include an unaltered copy of this License. * I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence. * J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission. * K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein. * L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles. * M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version. * N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section. * O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version. 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements". 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document. 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate. 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title. 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it. 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document. 11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing. ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (C) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

38.3 GNU Lesser General Public License