PostgreSQL/Print version

PostgreSQL

Introduction to PostgreSQL

PostgreSQL (or **Postgres** for short) is an open source relational database management systems (RDBMS). In nearly all cases the main application interface to an RDBMS is SQL, the standardised language for accessing relational databases. But beyond SQL every RDBMS needs a lot of additional tools to handle the installation process as well as maintenance and optimization tasks. As such tasks act very closely to the implementation of each RDBMS, they greatly differ from RDBMS to RDBMS - unlike SQL.

The purpose of this book is an introduction to the PostgreSQL specific aspects like its architecture, installing procedure, maintenance and optimization tasks. Thus it is primarily aimed to database administrators. But it is only a starting point in these objectives and many topics are omitted. For a complete treatment we recommend to get through professional training courses offered by many consulting companies. And don't forget: there is a great and complete documentation (http://www.postgresql.org/docs/current/static/index.html) of all the stuff we try to explain in the Wikibook on hand. Additionally some special topics are summarized in the PostgreSQL wiki (https://wiki.postgresql.org/wiki/).

The SQL interface is out of the scope of the Wikibook on hand. If you are interested in this specific topic, you may want to refer to the Wikibook SQL, which describes the SQL standard independent from any RDBMS implementation. It's a good starting point for PostgreSQL as its implementation of SQL is very close to the standard.

The Wikibook on hand is based on PostgreSQL version 10.

Characteristic features of PostgreSQL

- You can define your own types. Just as with tables, types consist of a collection of named attributes so you could define an "employee" type consisting of a text name, an integer employee number and a decimal salary. Unlike tables, you also define input and output functions so that values of your new type can be read and written. It

makes sense to ensure that the input function can interpret the values as represented by the output function so that your application can read its own handwriting, but that is not a formal requirement.
- Inheritance is supported. Suppose you have a table of employees, none of whom currently earn commission. You want to add data about sales representatives to your table, and they will need a commission column. If you add a commission column to your existing employees table, it will always contain NULL in almost every row. Alternatively, you could create a new table just for the sales representatives. It inherits all the columns from the employees table and adds a commission column. PostgreSQL keeps track of the inheritance so you can query all employees (including sales reps), just the standard employees, or just the reps.
- You can extend it with new functions (including aggregates) and operators. In fact this is essential if you want to support any new types you create beyond simply inputting, storing and retrieving them. User-defined functions are also required if you create your own indexing scheme, another possible consequence of creating new data types.

History of PostgreSQL

- Originated from the Ingres project.
- Post-Ingres project was an attempt to improve the type system.

Features

PostgreSQL offers a wide range of functionalities:

General

- Conformance with the SQL standard
- Guarantee for the ACID paradigm
- Full support for parallel running transactions in multiuser environments by an implementation of a multi version controll system (MVCC). All isolation levels, including *Serializable*
- Rich data type and operator system; strong constraints
- Extensibility mechanism for data types and operators, e.g.: spacial data, special indices, ... (used by PostgreSQL itself)

Reliability, Replication

- Online backup
- Point-in-Time Recovery
- Physical replication (complete cluster) as well as logical replication (parts of cluster or database) with synchron or asynchron behaviour
- Hot standby as fallback strategy or to distribute read-load accross multiple server

Advanced Features

- Table partitioning
- Query optimizer; parallel execution of queries
- Access to external data (Foreign Data Wrapper): other database systems, CSV file, ... [1]
- Trigger, procedural language, stored procedures
- A Notification System
- Full-text search

Soft Skills

- Free and Open Source [2]
- Running on major platforms: Linux/Unix/BSD, macOS, Windows [3]
- Excellent documentation [4]
- Continous development of new features
- Huge and agil community [5]
  - Chats, Mailing Lists, Blogs
  - Additional (free and commercial) tools: bidirectional replication, managing of a huge number of clusters, pooling [6]
  - Wiki [7]

References

1. https://wiki.postgresql.org/wiki/Foreign_data_wrappers
2. https://www.postgresql.org/about/licence/
3. https://www.postgresql.org/download/
4. https://www.postgresql.org/docs/current/static/index.html
5. https://www.postgresql.org/community/
6. https://www.postgresql.org/download/product-categories/
7. https://wiki.postgresql.org/

Featured Platforms

PostgreSQL supports 32- and 64-bit hardware. It is available on the following CPU architectures: x86, x86_64, IA64, PowerPC, PowerPC 64, S/390, S/390x, Sparc, Sparc 64, ARM, MIPS, MIPSEL, M68K, PA-RISC [1] and operating systems [2]

- BSD
  - FreeBSD
  - OpenBSD
- Linux
  - Red Hat family Linux (including CentOS/Fedora/Scientific/Oracle variants)
  - Debian GNU/Linux and derivatives
  - Ubuntu Linux and derivatives
  - SuSE and OpenSuSE
- Solaris (Sparc and i386)
- Other Unix systems
- macOS
- Windows (Win 2008 +)

References

1. CPU Architectures [1] (http://www.postgresql.org/docs/current/static/supported-platforms.html)
2. Operating Systems [2] (http://www.postgresql.org/download)

Download and Installation

Before you download PostgreSQL you must make some crucial decisions. First, there are two principle ways to get the system running: either you pick up the complete source code or the prebuild binaries. If you take the source code you must compile them with a C compiler (at least C89-compliant, in most cases people use GCC) [1] to the binary format of your computer. If you get the binaries directly the compilation step is superfluous. Next, you must know for which operating system you need the software. PostgreSQL supports a lot of UNIX-based systems (including macOS) as well as Windows - both in 32- and 64-bit versions.

Download

After you have made the above decisions you can download the source code and/or the binaries from this page (http://www.postgresql.org/download/) and its subpages. For some operating systems you will find a graphical installer which leads you through the subsequent installation steps. For the same or other operating systems not only the PostgreSQL DBMS will be downloaded but also the DBA tool `pgAdmin`, which helps you doing your daily work thru its graphical interface.

There are different versions available: the actual release, old releases and the upcomming release.

Installation

Installation steps vary depending on the chosen operating system. In the simplest case the above mentioned graphical installer hides the details. The PostgreSQL wiki (https://wiki.postgresql.org/wiki/Detailed_installation_guides) and documentation (http://www.postgresql.org/download/) leads you thru all necessary steps for your operating system.

If you install from source code, details are explained for Unix systems (http://www.postgresql.org/docs/current/static/installation.html) and Windows (http://www.postgresql.org/docs/current/static/install-windows.html).

After a successful installation you will have

- The PostgreSQL binaries on your disc.
- A first database cluster called 'main' on your disc. The database cluster consists of an empty database called 'postgres' (plus two template databases) and an user/role called 'postgres' as well.
- A set of programs (Unix) repectively a service (Windows) running on your computer. These programs/service handle the database cluster as an entire.

Linux

Installing from packages

```
sudo apt-get install postgresql
```

Installing from source

Download from http://www.postgresql.org/download/.

Starting and stopping

```
$ /etc/init.d/postgresql start
```

Windows

By default, PostgreSQL launches at each reboot so it can consume too many resources. To avoid that, just execute *services.msc* and set the

PostgreSQL service in manual start.

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 database will work. Just press a key to shutdown it after utilization.

Firewall

Once installed, PostgreSQL listen to the port 5432. So the firewall might need an exception to allow connections, even from localhost.

Creating a user

The following command creates a new user with superuser privileges:

```
$ createuser -U postgres -s <some username>
```

Creating a new user is a database operation, so it can only be done by an existing database user. You need to specify which database user to use (otherwise it will default to using a database user with the same name as your current UNIX user, which is unlikely to be correct). Default installs typically have a user called `postgres` that can be used for this.

To set a password for the newly created user, log in to the database using an account with superuser privileges, and issue the command:

```
ALTER USER <username> WITH ENCRYPTED PASSWORD 'secret';
```

First steps

You can create your tables, views, functions etc. in this database 'postgres' and work with the user 'postgres'. But this approach is not recommended. The user 'postgres' has very high privileges by default and the database 'postgres' is sometimes used by tools and third party programs as a container for temporary data. You are encouraged to define your own database, one user who acts as the owner of the database and some application users.

As a first step start psql with user 'postgres' and create your own users. Please notice, that 'users' respective 'roles' are global objects which are known by all databases within the cluster, not only within a certain database. But users/roles have specific rights within each database.

```
$ psql
postgres=#
postgres=# -- the future owner of the new database shall be 'finance_master' with DDL and DML rights
postgres=# CREATE ROLE finance_master;
CREATE ROLE
postgres=# ALTER  ROLE finance_master WITH NOSUPERUSER INHERIT CREATEROLE CREATEDB LOGIN NOREPLICATION ENCRYPTED PASSWORD
'xxx';
ALTER ROLE
postgres=# -- one user for read/write and one for read-only access (no DDL rights)
postgres=# CREATE ROLE rw_user;
CREATE ROLE
postgres=# ALTER  ROLE rw_user WITH NOSUPERUSER INHERIT NOCREATEROLE NOCREATEDB LOGIN NOREPLICATION ENCRYPTED PASSWORD
'xxx';
ALTER ROLE
postgres=# CREATE ROLE ro_user;
CREATE ROLE
postgres=# ALTER  ROLE ro_user WITH NOSUPERUSER INHERIT NOCREATEROLE NOCREATEDB LOGIN NOREPLICATION ENCRYPTED PASSWORD
'xxx';
ALTER ROLE
postgres=#
```

Next, create a new database 'finance_db'. You can do this as user 'postgres' or as the previously created 'finance_master'.

```
postgres=#
postgres=# CREATE DATABASE finance_db
postgres=#    WITH OWNER = finance_master
postgres=#    ENCODING   = 'UTF8'
postgres=#    LC_COLLATE = 'en_US.UTF-8'
postgres=#    LC_CTYPE   = 'en_US.UTF-8';
CREATE DATABASE
```

```
postgres=#
```

As the last step you have to delegate the intended rights to the users/roles. This is a little tricky because PostgreSQL uses an elaborated role system where every role inherits rights from the implicit 'public' role.

```
postgres=#
postgres=# \connect finance_db
finance_db=# -- revoke schema creation from role 'public' because all roles inherit her rights from 'public'
finance_db=# REVOKE CREATE ON DATABASE finance_db FROM public;
REVOKE
finance_db=# -- same: revoke table creation
finance_db=# REVOKE CREATE ON SCHEMA public FROM public;
REVOKE
finance_db=# -- grant only DML rights to 'rw_user', no DDL rights like 'CREATE TABLE'
finance_db=# GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES    IN SCHEMA public TO rw_user;
GRANT
finance_db=# GRANT USAGE                          ON ALL SEQUENCES IN SCHEMA public TO rw_user;
GRANT
finance_db=# -- grant read rights to the read-only user
finance_db=# GRANT SELECT                         ON ALL TABLES    IN SCHEMA public TO ro_user;
GRANT
postgres=#
```

References

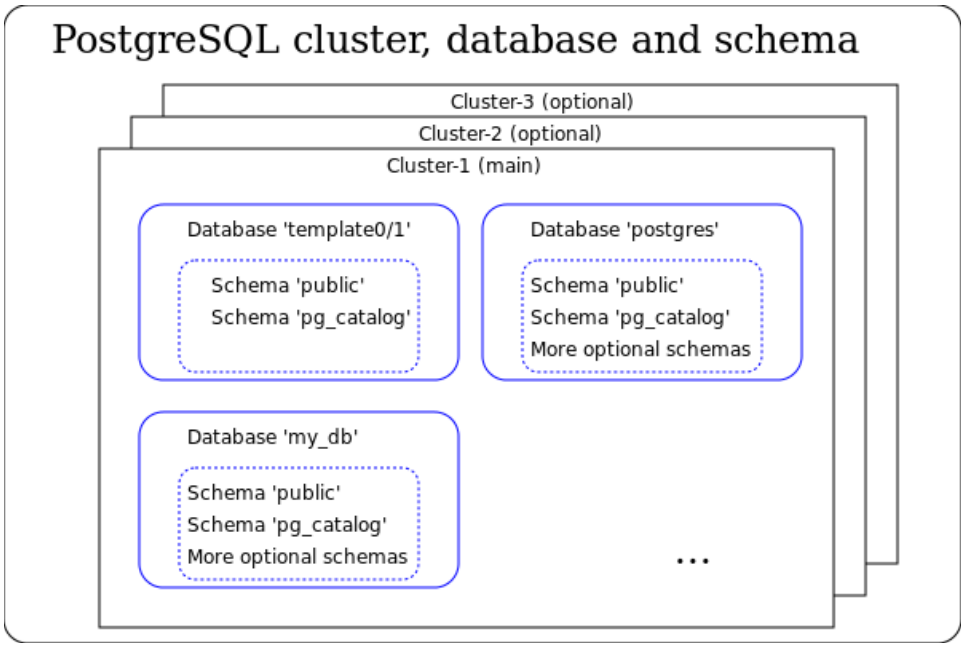
1. Requirements for Compilation [3] (http://www.postgresql.org/docs/current/static/install-requirements.html)

Terms

To promote a consistent use and understandig of important terms we list and define them herinafter. In some cases there are some short annotations to give a first introduction to the subject.

Database Cluster

Overview



Server (or Node)

A server is some (real or virtual) hardware where PostgreSQL is installed. Please don't exchange it with the term *instance*.

Cluster of Nodes

A set of nodes, which interchange information via replication.

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`.

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*.

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, thus it is defined that the user did 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.

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

An instance is a group of processes (on a UNIX server) respectively 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.
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 refere to real or virtual hardware, we do not use *server* as a synonym for *instance*.

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 keeps empty and application data is stored in separate databases like 'finance' or 'engineering'. Nevertheless 'postgres' shall not be droped because some tools try to store temporary data within this database.

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.

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.

initdb (OS command)

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

createdb (OS command)

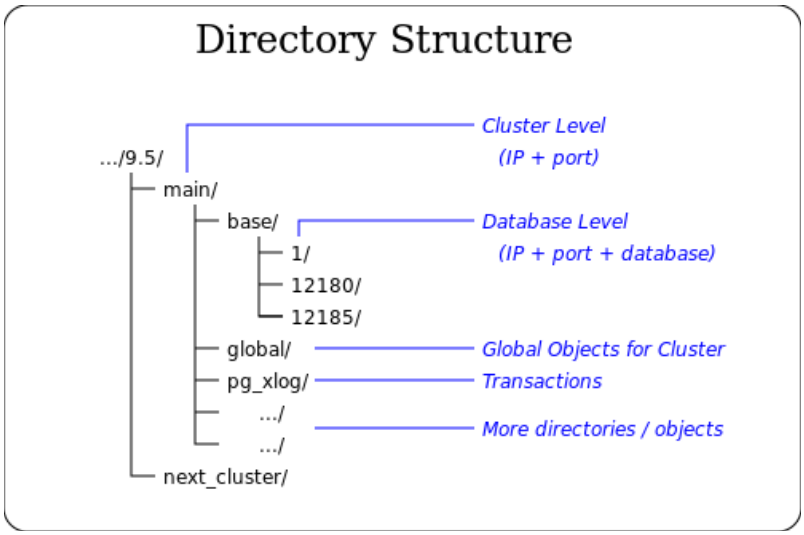
The utility `createdb` creates a new *database* within the actual *cluster*.

CREATE DATABASE (SQL command)

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

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.



Consistent Writes

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.

'Dirty' Page

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

Checkpoint

A checkpoint is a special point in time where it is guarantied 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.

WAL File

WAL files contain the changes which are applied to the data by modifiing 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), has a binary format, and have a fix size of 16MB. When they are no longer needed, they get recycled by renaming and reuseing 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 *WAL file* in our Wikibook: segment, WAL segment, logfile (don't mix it with the term *logfile*, see below), WAL log file, ...

Logfile

The *instance* logs and reports warning and error messages about special situations in readable text files. This logfiles 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. He is mentioned here because the term sometimes is used as a synonym for what we call *WAL file* - see above.

Log Record

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

Segment

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

MVCC

Multiversion Concurrency Control (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 values of an existing row changes, PostgreSQL writes a new version of this row to the database without deleting to 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 commited.
Outdated old rows are deleted at a later time by the utility `vacuumdb` respectively the SQL command `vacuum`.

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).

(Cold) Backup (file system tools) (http://www.postgresql.org/docs/current/static/creating-cluster.html)

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 no application 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".

(Hot) Logical Backup (pg_dump utility) (http://www.postgresql.org/docs/current/static/backup-dump.html)

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.
Hint: A logical backup is one form of an "online backup".

(Hot) Physical Backup or 'Base Backup' (http://www.postgresql.org/docs/current/static/continuous-archiving.html)

A physical backup is a **possibly inconsistent** copy of the files of a *cluster*, created with an operating system utility like cp or tar. At first glance such a backup seems to be useless. To understand its purpose, you must know PostgreSQL's recover-from-crash strategy.

> At all times and independent from any backup/recovery action, PostgreSQL maintains *WAL files* - primarily for crash-safety purposes. *WAL files* contain *log records*, which reflect all changes made to the data. In the case of a system crash those *log records* are used to recover the *cluster* to a consistent state. The recover process searches the timestamp of the last *checkpoint* and replays all subsequent *log records* in chronological order against this version of the *cluster*. Through that actions the *cluster* gets recovered to a consistent state and will contain all changes up to the last COMMIT.

The existence of a physical backup plus *WAL files* in combination with this recovery-from-crash technique can be used for backup/recovery purposes. To implement this, you need a physical backup (which may reflect an inconsistent state of the *cluster*) and which acts as the starting point. Additionally you need all *WAL files* since the point in time when you have created this backup. The recover process uses the described recovery-from-crash technique and replays all *log records* in the *WAL files* against the backup. In the exact same way as before, the *cluster* comes to a consistent state and contains all changes up to the last COMMIT.
Please keep in mind, that physical backups work only on cluster level, not on any finer granularity like database or table.
Hint: A physical backup is one form of an "online backup".

PITR: Point in Time Recovery (http://www.postgresql.org/docs/current/static/continuous-archiving.html)

If the previously mentioned *physical backup* is used during recovery, the recovery process is not forced to run up to the latest available timestamp. Via a parameter you can stop him at a time in the past. This leads to a state of the *cluster* at this moment. Using this technique you can restore your *cluster* to a time, which is between the time of creating the *physical backup* and the end of the last *WAL file*.

Standalone (Hot) Backup

The standalone backup is a special variant of the physical backup. It offers online backup (the *instance* keeps running) but it lacks the possibility of *PITR*. The recovery process recovers always up to the end of the standalone backup process. *WAL files*, which arise after this point in time, cannot be applied to this kind of backup. Details are described here (http://www.postgresql.org/docs/current/static/continuous-archiving.html).

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.

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 bringing the *cluster* into a consistent state at a defined timestamp.

Archive Recovery Mode

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

Restartpoint

A restartpoint is an action similar to a *checkpoint*. Restartpoints are only performed when the instance is in *archive recovery mode* or in *standby mode*.

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, ...

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*.

Master Server

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

Standby Server

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

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.

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.

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*.

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.

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*.

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.

Architecture

The daily work as a PostgreSQL DBA is based on the knowledge of PostgreSQL's architecture: strategy, processes, buffers, files, configration, backup and recovery, replication, and a lot more. The page on hand describes the most basic concepts.

Introduction

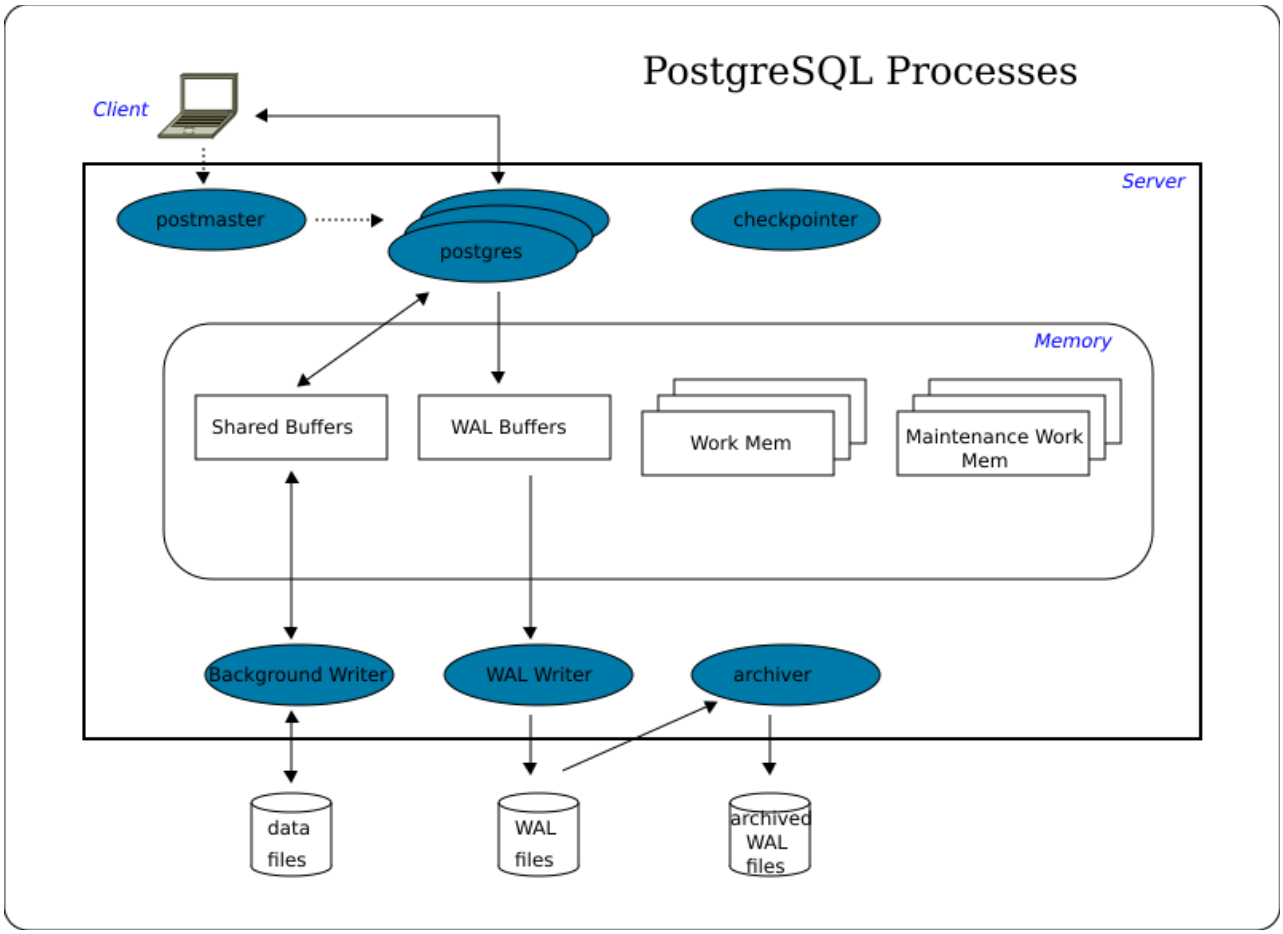
PostgreSQL is a relational database management system with a client-server architecture. At the server side the PostgreSQL's processes and shared memory work together and build an *instance*, which handles the accesses to the data. Client programs connect oneself to the *instance* and request read and write operations.

The Instance

The instance always consists of multiple processes (https://wiki.postgresql.org/wiki/FAQ#How_does_PostgreSQL_use_CPU_resources.3F). PostgreSQL does not use a multi-threaded model:

- *postmaster* process

- Multiple *postgres* processes, one for each connection
- *WAL writer* process
- *background writer* process
- *checkpointer* process
- *autovacuum launcher* process (optional)
- *logger* process (optional)
- *archiver* process (optional)
- *stats collector* process (optional)
- *WAL sender* process (if Streaming Replication is active)
- *WAL receiver* process (if Streaming Replication is active)
- *background worker* processes (if a query gets parallelised, which is available since 9.6)



How Data is processed

Connecting to the Instance

Client applications, which run on a different server than the instance, use the IP protocoll to connect to it. If client application and instance run on the same server, the same connection method is possible. But it is also possible to use a conncetion via a local socket.

In a first step the application connects to the *postmaster* process. The *postmaster* checks the application's rights and - if successful - starts a new *postgres* process and connects it with the client application.

Accessing Data

Client processes send and request data to and from the instance. For performance reasons, the instance doesn't write or read the requested data

directly to or from disk files. Instead, it buffers them in a shared memory area which is called the *shared buffers*. The flushing to disc is done at a later stage.

To perform a client request, the corresponding *postgres* process acts on the *shared buffers* and *WAL buffers* and manipulates their contents. When the client requests a `COMMIT`, the *WAL writer* process writes and flushes all *WAL records* resulting from this transaction to the WAL file. As the WAL file - in contrast to the data files - is written strictly sequentially, this operation is relatively fast. After that, the client gets its `COMMIT` confirmation. At this point, the database is inconsistent, which means that there are differences between *shared buffers* and the corresponding data files.

Periodically the *background writer* process checks the *shared buffers* for 'dirty' pages and writes them to the appropriate data files. 'Dirty' pages are those whose content was modified by one of the *postgres* processes after their transfer from disk to memory.

The *checkpointer* process also runs periodically, but less frequently than the *background writer*. When it starts, it prevents further buffer modifications, forces the *background writer* process to write and flush all 'dirty' pages, and forces the *WAL writer* to write and flush a `CHECKPOINT` record to the WAL file after which the database is consistent, which means: a) the content of the *shared buffers* is the same as the data in the files, b) all modifications of *WAL buffers* are written to WAL files, and c) table data correlates with index data. This consistency is the purpose of checkpoints.

In essence the instance contains at least the three processes *WAL writer*, *background writer*, and *checkpointer* - and one *postgres* process per connection. In most cases there are some more processes running.

Optional Processes

The *autovacuum launcher* process starts a number of worker processes. They remove superflous row versions according to the MVCC architecture of PostgreSQL. This work is done in shared memory and the 'dirty' pages are written to disc in the same way as any other 'dirty' pages, such as the ones resulting from data modification by clients.

The *logger* process writes log, warning, and error messages to a log file (not to the WAL file!).

The *archiver* process copies WAL files, which are completely filled by the *WAL writer*, to a configurable location for mid-term storing.

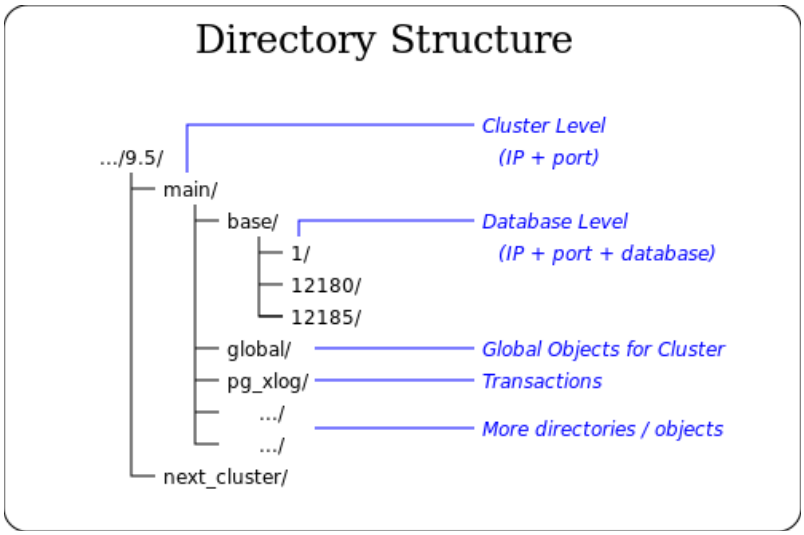
The *stats collector* process continuously collects information about the number of accesses to tables and indices, total number of rows in tables, and works in coordination with VACUUM/ANALYZE and ANALYZE.

The *WAL sender* and *WAL receiver* processes are part of the Streaming Replication feature. They exchange data about changes in the master server bypassing the WAL files on disc.

Since version 9.6 it is possible to execute queries in parallel on several CPUs. In this case those parts of the execution plan, which shall run in parallel, are executed by additional *background worker* processes. They have access to the *shared buffers* in the same way as the original *postgres* processes and handle different buffer pages at the same time.

The Directory Structure

Within a cluster there is a fix structure of subdirectories and files. At last all information is stored within these files. Some information contains to the cluster at all, and some belongs to single databases - especially tables and indexes.



Managing the Instance

The PostgreSQL instance consists of several processes running on a server platform. They work together in a coordinated manner using common configuration files and a common start/stop procedure. Thus all are running or none of them.

The program `pg_ctl` controls and observes them as a whole. When you are logged in as user *postgres* you can start it from a shell. The simplified syntax is:

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

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: 864)
/usr/lib/postgresql/9.4/bin/postgres "-D" "/var/lib/postgresql/9.4/main" "-c" "config_file=/etc/postgresql/9.4/main
/postgresql.conf"
$
```

You can observe, whether the instance is running or not, the process id (PID) of the postmaster, the directory of the cluster and the name of the configuration file.

start

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

```
$ pg_ctl start
...
database system is ready to accept connections
$
```

When you see the above message everythink works fine.

stop

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

```
$ pg_ctl stop
...
database system is shut down
$
```

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 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] ]`

restart

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

reload

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

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.

Automated start at boot time

In most cases it is necessary that PostgreSQL starts immediately after the server boots. Whether this happens - or not - may be configured in the file `start.conf`, which is located in the special directory $PGDATA (Debian/Ubuntu) or in the main directory of the cluster (RedHat). There is only one entry and its allowed values are:

- auto: automatically start/stop
- 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)

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).

Numerous definitions have a dynamic nature, which means that they take effect with a simple `pg_ctl reload`. Others require a restart of the instance `pg_ctl restart`. The comments within the delivered default configuration files describe which one of the two actions has to be taken.

postgresql.conf

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.

Connections

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

Resources

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

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

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

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*, *achive*, *hot_standby*, and *logical*. Depending on the decision, which kind of archiving or replication you want to use, the WAL mechanism must write only basic information to the WAL files or some more information. *minimal* is the basic methode 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.

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 pay attention only on the master-definitions and the standby only on 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.

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 seach for an optimal query execution plan. The values are no real values (in sense of milliseconds or number of CPU cycles), they are only a) a rough guideline for the query planer and b) relative to each other. The real values during later 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.

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.

Statistics

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

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. But even in such situations the client may enforce the *host* connection type by using the TCP/IP syntax (e.g.: 'localhost:5432') referring 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.

MVCC

Multiversion Concurrency Control (MVCC) is a broadly used database technic to accomplish two goals: first, it allows to show every concurrently running transaction a consistent state of the data as of the time of its start - which is called a *snapshot* -, and second, it minimizes lock contemption and guaranties high performance in a multiuser environment. MVCC implementation is very complex, thus this short introduction is only the tip of an iceberg.

Hint: Please consider the wording on this page: *row* is used in the conventional sense of relational database systems, whereas *record* denotes one or more versions of this *row*.

The Concept

The basic idea is, that the modification of a column value does not lead to a modification of the value in the original record. Instead PostgreSQL creates an additional record which contains the changed value. Thus an UPDATE command does not really update values in a record. It creates a new record and leaves the values in the old record unchanged.

This behaviour raises the question, which one of the multiple records of the row shall be delivered to concurrent or later running processes. To resolve the situation PostgreSQL adds two additional hidden system columns to every record. The columns `xmin` and `xmax` contains transaction IDs (*xid*). Those transaction IDs act like a serial: they are increased by 1 for every new transaction within the system. Therefore a lower *xid* indicates an older transaction. `xmin` contains the *xid* of the transaction, which has created the record. Transactions with an *xid* greater that `xmin` are younger and may see this record. `xmax` is initially 0. During write operations it changes to the *xid* of the transaction, which has created the next version of this record - or has deleted the record. Transactions with an *xid* greater that `xmax` are younger and cannot see this record, if `xmax` differs from 0.

Examples

Example 1: INSERT

Time	xmin	xmax	column value	Transaction #20	Transaction #21
				INSERT ...	
t_1	20	0	1		
				COMMIT	
					SELECT ...

The initial INSERT creates one record. `xmin` contains the *xid* of this transaction and `xmax` the value 0 to indicate, that the record is not deleted. The SELECT of transaction #21 sees this record because its *xid* is greater than `xmin`.

Example 2: UPDATE

Time	xmin	xmax	column value	Transaction #25	Transaction #33
t_1	20	0	1		
				UPDATE ...	
t_2	20	25	1		
	25	0	2		
				COMMIT	
					SELECT ...

UPDATE stores its own *xid* in `xmax` of the already existing record and creates a new record with the new column value in the same way as an INSERT. The SELECT of transaction #33 sees only the record with the column value 2. The first record is invisible as its `xmax` is smaller (= older) than the transactions *xid*.

Example 3: DELETE

Time	xmin	xmax	column value	Transaction #101	Transaction #120
t_2	20	25	1		
	25	0	2		
				DELETE ...	
t_3	20	25	1		
	25	101	2		
				COMMIT	
					SELECT ...

DELETE changes `xmax` in the last record to its own *xid* - nothing else is done! Transaction #120 cannot see any of the two records as its *xid* is heigher than `xmax` (and `xmax` is not 0).

The three examples show only the basic operations. The situation can become much more complicated, if the SELECT runs before the COMMIT of the writing transaction, if the writing transaction use multiple write-operations within its transaction context, if the writing transaction aborts, if the number of *xid*s gets exhaused and wraps arround, ...

Vacuum

If no additional action is taken, the MVCC technic blows up every database, which performs a great number of writing activities. The number of records grows and grows even if the overall number of rows keeps constant or decreases. It is absolutely necessary to physically remove from time to time such records which are no longer valid. The utility `vacuumdb` respectively the SQL command `vacuum` takes this task.

Vacuum retrieves the smallest (= oldest) *xid* from all actually active transactions. All records with an `xmax` smaller than this *xid* cannot been seen by any of the running or future transactions. Therefore it is save, to physically remove them.

This task is very I/O intensive. Therefore, it's not running constantly, but on a periodical basis.

WAL

Usage

WAL files are files, where changed data values are stored in a binary format. This is additional information and in this respect it is redundant to the information in the database files. WAL files are a specific kind of 'diff' files.

Writing to WAL files is very fast as they are written always sequentially. In contrast to WAL files database files are organized in special structures like trees, which possibly must be reorganized during write operations or which points to blocks at far positions. Thus writes to database files are much slower.

Whenever a client requests a write operation like `UPDATE` or `DELETE`, the modifications to the data are not instantly written to database files. For the mentioned performance reasons this is done in a special sequence and - in some parts - asynchroniously to the client requests. First, data is written and flushed to WAL files. Second, it is stored in shared buffers in RAM. And in the end it is written from shared buffers to database files. The client must not 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 asynchroniously at an later (or prior) point in time.

Remove

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. The basic criterion for such a deletion is, that the changes in the shared buffers (which correlate to the content of the WAL files) are flushed to the database files. As it is guarantied that this criterion is met after a `CHECKPOINT`, there are some dependencies between WAL file delete operations and `CHECKPOINT`s:

- You can define a limit for the total size of all files in the directory: `max_wal_size`. If it is reached, PostgreSQL performs automatically a `CHECKPOINT`.
- You can define a `checkpoint_timeout`. Not later than this number of seconds, PostgreSQL performs automatically a `CHECKPOINT`.

In both cases the shared buffers gets 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 criterions, especially by failling archive commands. `max_wal_size` is a soft limit and may be exeeded in such situations.

Tools

There are various tools which supports the DBA in its daily work. Some parts of this work can be done in standard SQL syntax, eg: `CREATE USER ...`, whereas a lot of important tasks like backups or cleanups are out of scope of SQL and are supported only by ventor-specific SQL extentions or

utilities, eg: VACUUM. Thus in most cases the DBA tools support standard-SQL syntax as well as PostgreSQL-specific SQL syntax and the spawning of PostgreSQL's utilities.

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 at 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 actual cluster
- `\echo :DBNAME` lists the actual database (consider upper case letters). In most cases the name is part of the psql prompt.
- `\dn` lists all schemas in the actual database
- `\d` lists all tables, views, sequences, materialized views, and foreign tables in the actual schema
- `\dt` lists all tables in the actual schema
- `\d+ TABLENAME` lists all columns and indexes in table TABLENAME
- `\du` lists all users in the actual 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*

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 (http://www.pgadmin.org/download/).

pgAdmin extends the functionalities of *psql* by a lot of 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 (http://www.pgadmin.org/screenshots/) are online.

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

phpPgAdmin

phpPgAdmin is a graphical tool which 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 sourgeforge (http://sourceforge.net/projects/phppgadmin/) and seems to be outdated as the actual release 5.1 dates from April 14, 2013.

Other Tools

There are a lot of other general tools (https://wiki.postgresql.org/wiki/Community_Guide_to_PostgreSQL_GUI_Tools) and monitoring tools (https://wiki.postgresql.org/wiki/Monitoring#Postgres-centric_monitoring_solutions) with a GUI interface. Their functionality variies greatly from pure SQL support up to entity-relationship and UML support. Some of the tools are open/free source, others proprietary.

ClientServerComm Client

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 (or cluster). 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 name of the database within the cluster (= IP/port combination).
- The name of the user (= role) with which the client program wants to work
- The password of this user.

You can specify this values in three different ways: as explicit parameters of the client program, as environment variables or as a fix line of text in the special file *pgpass*.

Parameters

You can specify the parameters in the usual short (-) or long (--) format.

```
$ # 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.

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)

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 sucessfull connection and stores the actual connection values.

ClientServerComm

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 consistence.

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.

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 implemenations at the 'products' site [2].

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 authentification against 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. The lines are evaluated from start to end and the first matching line applies. The main layout of this lines is as follows

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

Words in upper case must be replaced by specific values whereas *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 noteable 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 respectivelly all database-users.

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

METHODE is one of the following. The thereby defined rule (=line) applies, if database/user/address combination matches.

- 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 METHODE.

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 comeing 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
```

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.

References

1. Driver Wiki [4] (https://wiki.postgresql.org/wiki/List_of_drivers)
2. Commercial and open source driver [5] (http://www.postgresql.org/download/products/2/)

Security

Roles

PostgreSQL supports the concept of `roles` [1] to realize security issues within the database. Roles are independent from operating system user accounts (with the exception of the special case peer authentication 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.

References

1. Concept of roles [6] (http://www.postgresql.org/docs/current/static/user-manag.html)

BackupAndRecovery

Creating backups is an essential task for every database administrator. He must ensure that in the case of a hardware or software crash the database can be restored with minimal data loss. PostgreSQL offers different strategies to support the DBA in his effort to achieve this goal.

First off all, backup technology can generally be divided into two classes: A **cold** backup is a backup which is taken during a period of time, where no database file is open. In the case of PostgreSQL this means, that the instance is not running. The second class of backups is called **hot** backup. Hot backups are taken during normal working times, which means, that applications can perform read and write actions in parallel to the backup creation. There are different types of hot backups.

```
Backup
   |
   +-- Cold Backup
   |
   +-- Hot Backup
         |
         +-- Logical Backup
```

```
       |
       +-- Physical Backup
       |
       +-- Physical Backup plus PITR
```

Cold Backup (= Offline Backup)

A cold backup is a **consistent** copy of all files which constitutes the cluster with all of its databases. To be 'consistent' this copy cannot be taken at an arbitrary point in time. There is only one way to create a consistent, usefull cold backup: You must shut down the PostgreSQL instance `pg_ctl stop`, which disconnects all applications from all databases.

After the instance is shut down, you can use any operating system utility (cp, tar, dd, rsync, ...) to create a copy of all cluster files to a secure location: on the disc of a different server, on any backup system at a SAN or the intranet, a tape system, ...

Which files constitutes the cluster and are therefor necessary to copy?

- All files under the directory node, where the cluster is installed. The logical $PGDATA points to this directory. Its name is somethink like '.../postgresql/9.4/main'.
- If the cluster uses the directory layout as it is used on the Linux derivate Ubuntu, the configuration files are located outside of the above directory structure in a separate directory. In this case you must additionally copy the directory with the configuration files.
- All files, which are used as a tablespace.

One may try to backup only special parts of a cluster, eg. a huge file which represents a table on a separate partition or tablespace - or the opposite: everything without this huge file. Even if the instance is shut down during the generation of such a partial copy, copies of this kind are useless. The restore of a cold backup needs all data files and files with metainformation of the cluster.

Cold backups are sometimes called *offline backups*.

Advantages

- The methode is easy to install.

Disadvantages

- A continuous 7x24 operation mode of any of the databases of this cluster is not possible.
- It is not possible to backup smaller parts of a cluster like a single database or table.
- You cannot restore parts of the backup. Either you restore everything or nothing.
- After a crash you cannot restore the data to any point in time after the last backup generation. All changes to the data after this moment gets lost.

How to Recover

In the case of a crash you can restore the data from a cold backup by performing the following steps:

- Stop the instance.
- Delete all original files of the crashed cluster: $PGDATA plus, for the Ubuntu way, the configuration files.
- Copy the files of the backup to their original places.
- Start the instance. It shall start in the normal way, without any special message.

Hot Backup (= Online Backup)

In opposite to *cold backups* a *hot backup* is taken during the instance is running and applications may change data during the backup is taken.

Hot backups sometimes are called *online backups*.

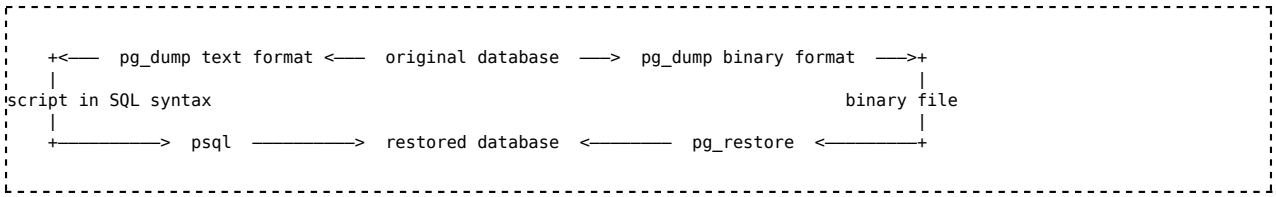
Logical Backup

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. If any data changes some values during the creation of the backup, the backup takes the old values whereas the application sees the new ones. (Logical backups runs in

serializable transactions.) This is possible because of PostgreSQL's MVCC implementation.

`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 backup-ed data. The second format is a binary format and is called the *archive format*. To restore files with this format you must use the tool `pg_restore`.

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



The format type is choosen by `pg_dump`s parameter `--format`.

`pg_dump` (in both formats) is able to dump data, schema definitions or both. The parameters `--data-only` and `--schema-only` restrict the output to data resp. to the schema.

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. `pg_dumpall` without detailed parameters will dump everything of the cluster: all data and structures of all databases plus all user definitions plus definitions of their rights. With the parameter `--globals-only` you can restrict its behaviour to dump global objects only.

```
Some Examples:

$ # dump complete database 'finance' in text format to a file
$ pg_dump --dbname=finance --username=boss --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=c --file=finance_person.archive
$
$ restore table 'person' from binary archive
$ pg_restore --dbname=finance_x --username=boss --format=c <finance_person.archive
$
```

Advantages

- Continuous 7x24 operation mode is possible.
- Small parts of 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 mentioned in the above diagram the recovery process depends on the format of the dump. Text files are in standard SQL syntax. To recreate objects from this format you can use SQL utilities like `psql`. Binary files are in the *archive format*. They can only be used by the utility pg_restore.

Physical Backup

A physical backup is an **inconsistent** copy of the files **of a cluster**, created with operating system utilities like cp or tar taken at a time whereas applications modify data. At first glance such a backup seems to be useless. To understand its purpose, you must 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 reflects all changes made to the data and the schema. Prior to their transfer to the data files of the database the *log records* are stored in the (sequentially written) WAL file. In the case of a system crash those *log records* are used to recover the cluster to a consistent state during restart. The recover process searches the timestamp of the last *checkpoint* and replays all subsequent *log records* in chronological order against this version of the cluster. Through that action the cluster gets recovered to a consistent state and contains all changes up to the last COMMIT.

The existence of a physical backup, which is inconsistent by definition but contains its *WAL files*, in combination with this recovery-from-crash technique can be used for backup/recovery purposes. To implement this, you have to restore the previous taken physical backup (including its *WAL files*). When the instance starts again, it uses the described recovery-from-crash technique and replays all *log records* in the *WAL files* against the database files. In the exact same way as before, the cluster comes to a consistent state and contains all changes up to the point in time when the backup-taken has started.

Please keep in mind, that physical backups work only on cluster level, not on any finer granularity like database or table.

Physical backups without PITR sometimes are called *standalone hot physical backup*.

Advantages

- Continuous 7x24 operation mode is possible.

Disadvantages

- Physical backup works only on cluster level, not on any finer granularity like database or table.
- Without PITR (see below) you will lose all data changes between the time, when the physical backup is taken, and the crash.

How to Take the Backup and Recover

To use this technique it is necessary to configure some parameters in the *postgres.conf* file for WAL and archive actions. As the usual technique is *Physical Backup plus PITR* we describe it in the next chapter.

Physical Backup plus PITR

The term *PITR* stands for *Point In Time Recovery* and denotes a technique, where you can restore the cluster to any point in time between the creation of the backup and the crash.

The *Physical Backup plus PITR* strategy takes a physical backup plus all WAL files, which are created since the time of taking this backup. To implement it, three actions must be taken:

- Define all necessary parameters in *postgres.conf*
- Generate a physical backup
- Archive all arising WAL files

If a recovery becomes necessary, you have to delete all files in the cluster, recreate the cluster by copying the physical backup to its original location, create the file *recovery.conf* 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.
- Generating WAL files is the basis for additional features like *replication*.

Disadvantages

- Physical backup works only on cluster level, not on any finer granularity like database or table.
- If your database is very busy and changes a lot of data, many WAL files may arise.

How to Take the Backup

Step 1

You have to define some parameters in *postgres.conf* so that: WAL files are on the level 'archive' or higher, archiving of WAL files is activated and a copy command is defined to transfers WAL files to a failsafe location.

```
# collect enough information in WAL files
wal_level = 'archive'
# activate ARCHIVE mode
archive_mode = on
# supply a command to transfer WAL files to a failsafe location (cp, scp, rsync, ...)
# %p is the pathname including filename. %f is 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 a *physical* or *base backup* with an operating system utility during the instance is in a special 'backup' mode. In this mode the instance will perform a checkpoint and create some additional files.

```
$ # start psql and set the instance to 'backup' mode, where it creates a checkpoint
$ psql -c "SELECT pg_start_backup('AnyBackupLabel');"
$
$ # copy the cluster's files
$ scp -r $PGDATA dba@archive_server:/postgres/whole_cluster/
$
$ # start psql again and finish 'backup' mode
$ psql -c "SELECT pg_stop_backup();"
$
```

If you like to do so, you can replace the three steps by a single call to the utility *pg_basebackup*.

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.

How to Recover

To perform a recovery the original *physical* or *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 in a later stage.
- Delete all files of the crashed cluster.
- Recreate the cluster files from the base backup.
- 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.
- Start the instance. During startup the instance will copy and process all WAL files found in the archive location.

The fact, that *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 the *recovery.conf* file. 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 *recovery.conf* file may look like this:

```
restore_command      = 'scp dba@archive_server:/postgres/wal_archive/%f %p'
```

```
recovery_target_time = '2016-01-31 06:00:00 CET'
```

Additional Tools

There is an open source project *Barman* [1], which simplifies the steps of backup and recovery. The system helps you, if you have to manage a lot of servers and instances and it becomes complicate to configure and remember all the details about your server landscape.

References

1. Barman [7] (http://www.pgbarman.org/)

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.

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 opposite to the above categories, the two different architectures are not strictly distinct to each other, e.g.: if you focus to atomic replication channels of a *Multi-Master* architecture, you will 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.

Configuration

This configuration is done in the 'postgres.conf' file (some on the master site, others on the standby site), whereas security configuration is stored in 'pg_hba.conf' (master site), and some important decisions are derived from the existence of 'recovery.conf' (standby site) and its values. The great number of possible combinations of concepts and their correlation to values within the config files may be confusing at the beginning. Therefore, we reduce our explanations to the minimal set of 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 dabases 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 (http://www.postgresql.org/docs/current/static/logicaldecoding.html). And there is an important project Bi-Directional Replication (http://bdr-project.org/docs/stable/index.html), which is based on this technique.

Synchronism: synchron vs. asynchron

The default behaviour is asynchronuous 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 synchronuous replication waits for a successful processing of COMMIT statements at the standby before he confirms it to its client.

The synchronuous 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
```

Typical Use Cases

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

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:

- Take a physical backup exactly as described in Backup with PITR 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 arrising WAL files to a lot of servers and activate warm standby on each of them. Doing so, you get a lot of standbys.

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

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 ...;

# postgres.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_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.

An Additional Tool

If you have to manage a complex replication use case, you may want to check the open source project 'repmgr' (http://www.repmgr.org/). It supports you to monitor the cluser of nodes or perform a failover.

Partitioning

If you have a table with a very huge amount of data, it may be helpfull to scatter the data to different physical tables which shares 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.

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 disapeares, but in comparision to the previous solution the functionality keeps 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');
...
```

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();
```

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.

Tablespace

The default behaviour of PostgreSQL is, that all data, indexes, and management information is stored in subdirectories of a single directory. But this approach does not fit always. 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;
```

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 incompatiblities 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.

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.

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.

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.

Extensions

PostgreSQL offers an extensibility architecture and implements its internal datatypes, 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 (http://www.postgresql.org/docs/current/static/extend.html).

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

- Data types, operators, and function for the handling of **spatial data** like points, polylines, overlaps(), ... as defined by OSGeo and SQL Multimedia and Application Packages Part 3: Spatial.
- Functionality for **full text** search as defined by SQL Multimedia and Application Packages Part 2: Full-Text.
- 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.

The lifecycle of such an extension starts with the implementaion 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 (http://www.postgresql.org/docs/current/static/contrib.html) 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`.

PostGIS

PostGIS 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 and SQL Multimedia and Application Packages Part 3: Spatial. 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.

The PostGIS project has its own representation on the WEB (http://postgis.net/) where all its aspects are described, especially the download process and the activation of the extension itself.

Foreign Data Wrappers

Foreign Data Wrappers (FDW) are PostgreSQL extensions which offers 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 (https://wiki.postgresql.org/wiki/Foreign_data_wrappers) gives an overview.

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

Here is an example 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 ...
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
```

Bidirectional Replication (BDR)

BDR is an extension which allows replication in both directions between involved (master-) nodes in parallel to their regular read and write activities of their client applications. So it realizes a multi-master replication. Actually the project is a standalone project (http://bdr-project.org/docs/next/index.html). But multiple technologies emerging from BDR development have already become an integral part of core PostgreSQL, such as Event Triggers (http://www.postgresql.org/docs/current/static/event-triggers.html), Logical Decoding (http://www.postgresql.org/docs/current/static/logicaldecoding.html), Replication Slots (http://www.postgresql.org/docs/current/static/logicaldecoding-explanation.html#AEN68501), Background Workers (http://www.postgresql.org/docs/current/static/bgworker.html), and more.

Data Types

PostgreSQL supports the basic set of data types which are defined by the SQL standard and described in the wikibook SQL (but: CLOB is called

TEXT and BLOB is called BYTEA).

Character Types
Character (CHAR)
Character Varying (VARCHAR)
Character Large Object (TEXT/CLOB)
Binary Types
Binary (BINARY)
Binary Varying (VARBINARY)
Binary Large Object (BYTEA/BLOB)
Numeric Types
Exact Numeric Types (NUMERIC, DECIMAL, SMALLINT, INTEGER, BIGINT)
Approximate Numeric Types (FLOAT, REAL, DOUBLE PRECISION)
Datetime Types
(DATE, TIME, TIMESTAMP. With and without timezone.)
Interval Type
(INTERVAL)
Boolean
XML

In addition to this basic types there are some more predefined types as well as a mechanism to define own, composite data types.

Extensions to the Standard

SERIAL
SERIAL generates a sequence of integer values, usually used as a primary key. However SERIAL is not a real data type. Under the hood it uses the type INTEGER and accomplishes it with a sequence.

ENUM
It defines a static, ordered set of values, eg: colors, days of the week, ...

MONETARY
It represents currency values with a fixed fractional precision.

GEOMETRIC
POINT, LINE, LSEG, BOX, PATH, POLYGON and CIRCLE are supported 'out of the box' (without the need to install the extension PostGIS).

Network Address Types
MACADDR, MACADDR8: They hold MAC addresses.
INET: It holds an IPv4 or IPv6 host address and optionally a netmask, eg: `123.45.67.89` or `192.168.0.1/24`. INET accepts nonzero bits to the right of the netmask (on input).
CIDR: It holds an IPv4 or IPv6 network specification and optionally a netmask, eg: `123.45.67.89` or `192.168.0.0/24`. CIDR does not accept nonzero bits to the right of the netmask (on input).

Bit Types
BIT(n) and BIT VARYING(n) stores '0's and '1's in the same way as CHAR stores character.

UUID
It stores a sequence of 128 bits according to RFC 4122. It's main usage is to hold unique values.

JSON Types
JSON and JSONB stores data according to RFC 7159. The difference between the to two data types is, that JSON internally uses a pure text representation whereas JSONB uses a decomposed binary format.

Special Types

Arrays

According to SQL:2011 PostgreSQL supports arrays. Columns of a table can be defined as variable-length multidimensional arrays of the above presented data types.

```
CREATE TABLE tbl_1 (
  id          SERIAL,
  -- A one dimensional array of integers. It's possible to use multidimensional arrays, eg: INTEGER[][]
  column_1    INTEGER[]
);

-- construct arrays with curly braces or with the ARRAY[] syntax
INSERT INTO tbl_1 VALUES (1, '{1, 2, 3, 4}');
INSERT INTO tbl_1 VALUES (2, ARRAY[5, 6, 7]);

-- specify certain elements of an array with the [] syntax
SELECT * FROM tbl_1 WHERE column_1[2] = 2;
 id | column_1
----+-----------
  1 | {1,2,3,4}
```

Composite Types

You can create new types by arranging an ordered set of data types - like a *struct* or *record* in other programming languages. This new type can be used at all places where the predefined data types are applicable (columns, function parameters, ...).

```
-- create a composite type
CREATE TYPE person AS (
  first_name    CHAR(15),
  last_name     CHAR(15)
);

-- Use the composite type to define columns with the desired structure
CREATE TABLE tbl_2 (
  id          SERIAL,
  -- the composite type
  pers        person,
  -- an array of up to 5 hobbies
  hobbies     CHAR(10) [5]
);

-- construct values of the composite type with () syntax
INSERT INTO tbl_2 VALUES (1, '(John D., Walker)', '{Sports, Chess}');

SELECT * FROM tbl_2;
 id |                 pers                  |           hobbies
----+---------------------------------------+----------------------------
  1 | ("John D.        "," Walker        ") | {"Sports    ","Chess     "}
```

Range Types

Range types are data types representing a range of values, eg: an integer range or a date or timestamp range. This is similar to a BETWEEN constraint. But it offers the additional possiblity to ensure, that values of different rows do not overlap; see the description of an EXCLUSION CONSTRAINT with an GIST index (https://www.postgresql.org/docs/current/static/rangetypes.html#RANGETYPES-CONSTRAINT)

Index Types

PostgreSQL supports the following index types: B-tree, GIN, GiST, SP-GiST, BRIN, and Hash (which is discouraged).

B-tree

The default index type of PostgreSQL (and may other database systems) is B-tree. Those indexes are suitable in situations where there is a well

defined sort order for the affected values. In particular this applies to all numeric values, e.g.: `... WHERE numeric_column < 5;`. The syntax is simple and conforms to those of other implementations - see wikibook SQL.

```
postgres=# CREATE INDEX indexname ON tablename (columnname);
```

There are two situations where the use of this basic index type is not possible:

- The comparision operator may have no notion of a sort order, eg: does a text CONTAIN a given word?
- The type of the column may be to complex to define a sort order, eg: two- or n-dimensional grafical objects like polygons.

GIN - for muliplicities

GIN (https://www.postgresql.org/docs/current/static/gin-intro.html) (Generalized Inverted Index) indexes are appropriate for data which is constructed out of multiple components (of equal type). Examples are texts (= many words) or arrays (= many values of any basic type). Typically, queries (where-clauses) to such columns does not address the entire column value (complete text or complete array) but the values of its elements (words or array elements). A typical operator in such situation is 'contains' which may be abbreviated as `@@` or `@>`, depending on the data type.

The syntax of the `CREATE INDEX` statement is extended by the key words `USING GIN` plus some more parameters in dependence of the data type.

```
postgres=# CREATE INDEX indexname ON tablename USING GIN (columnname);
```

Here is a first example: text retrieval

```
postgres=# -- create a table with a text column
postgres=# CREATE TABLE t1 (id serial, t text);
CREATE TABLE
postgres=# CREATE INDEX t1_idx ON t1 USING gin (to_tsvector('english', t));
CREATE INDEX
postgres=# INSERT INTO t1 VALUES (1, 'a fat cat sat on a mat and ate a fat rat');
INSERT 0 1
postgres=# INSERT INTO t1 VALUES (2, 'a fat dog sat on a mat and ate a fat chop');
INSERT 0 1
postgres=# -- is there a row where column t contains the two words? (syntax contains some magic to hit index)
postgres=# SELECT * FROM t1 WHERE to_tsvector('english', t) @@ to_tsquery('fat & rat');
 id |                    t
----+-------------------------------------------
  1 | a fat cat sat on a mat and ate a fat rat
(1 row)
```

Hint: If you havn't created an index, the SELECTs will lead to the same result. But the performance is quite different. Because of the missing index, the SELECT executes a sequential scan over the complete table T1 and - even more costly - normalizes every term on the fly. This must be done with every SELECT whereas the INDEX terms are created and modified rarely: only during INSERT and UPDATE.

A second example: array elements

```
postgres=# -- create a table where one column exists of an integer array
postgres=# --
postgres=# CREATE TABLE t2 (id serial, temperatures INTEGER[]);
CREATE TABLE
postgres=# CREATE INDEX t2_idx ON t2 USING gin (temperatures);
CREATE INDEX
postgres=# INSERT INTO  t2 VALUES (1, '{11, 12, 13, 14}');
INSERT 0 1
postgres=# INSERT INTO  t2 VALUES (2, '{21, 22, 23, 24}');
INSERT 0 1
postgres=# -- Is there a row with the two array elements 12 and 11?
postgres=# SELECT * FROM t2 WHERE temperatures @> '{12, 11}';
 id | temperatures
----+--------------
  1 | {11,12,13,14}
(1 row)
```

GiST - mainly for ranges

GiST (https://www.postgresql.org/docs/current/static/gist.html) (Generalized Search Tree) is an index type which supports - among others - range searches. GIN and B-tree index nodes consists of single values like '5' or 'rat'. In contrast GiST index nodes consists of multiple values - in mimimum

(in an one-dimensional space) a 'from' value and a 'to' value. The nodes are organized within a tree in a way, that the from/to of every upper node covers the from/to of all its child nodes. This is the basis for an excellent performance of range searches - the number of consulted nodes during tree traversal gets minimal.

The syntax of the CREATE INDEX statement is extended by the key words `USING GIST`.

```
postgres=# CREATE INDEX indexname ON tablename USING GIST (columnname);
```

Range searches are essential for spatial data. The example checks whether given circles lie within another circle. Please note, that basic geometry data types like point or circle are part of PostgreSQL's core - without the extension PostGIS.

```
postgres=# -- create a table with a column of non-trivial type
postgres=# --
postgres=# CREATE TABLE t3 (id serial, c circle);
CREATE TABLE
postgres=# CREATE INDEX t3_idx ON t3 USING gist(c);
CREATE INDEX
postgres=# INSERT INTO t3 VALUES (1, circle '((0, 0), 0.5)');
INSERT 0 1
postgres=# INSERT INTO t3 VALUES (2, circle '((1, 0), 0.5)');
INSERT 0 1
postgres=# INSERT INTO t3 VALUES (3, circle '((0.3, 0.3), 0.3)');
INSERT 0 1
postgres=# -- which circles lie in the bounds of the unit circle?
postgres=# SELECT * FROM t3 WHERE circle '((0, 0), 1)' @> c;
 id |        c
----+-----------------
  1 | <(0,0),0.5>
  3 | <(0.3,0.3),0.3>
(2 rows)
```

Hint: The GiST index type offers much more advantages, especially he is extensible (https://www.postgresql.org/docs/current/static/gist-extensibility.html). You can create new index types and operators which are suitable for particular use cases to support domain-specific queries, eg: 'is picture X similar to picture Y?'.

Parallel Queries

Since version 9.6 PostgreSQL supports parallel processing of queries. Within most of today's servers there are a lot of CPUs. Their concurrent usage can shorten the elapsed time of queries significantly. Therefore the query optimizer tries to create a plan, which leads to more than one executing process per query. At runtime those processes work concurrently and in a coordinated way on diverse parts of the shared buffers.

Parallel execution is initiated by so called *gather nodes* of the execution plan. When they are reached at runtime, the actual running process requests the planned number of additional processes (*background worker processes*). The original process plus the additional processes execute the child node of the plan in parallel. The *gather node* has the additional duty to collect and accumulate the results of its child processes.

The feature is not used in all situations. This results from three different effects: type of query, parameterization of PostgreSQL, and the actual implementation.

- If a query leads to an execution plan, which is hightly I/O intensive, he doesn't benefit greatly from parallelization as parallelization is a RAM-access feature. In contrast to this, queries which needs hight CPU activities (eg: ... where text like '%xyz%'; without an adaequat index) will benefit much more. Therefore it is more likely that parallelization is choosen for the second type of queries.
- The default behaviour of PostgreSQL (in version 9.6) is to use the traditional behaviour by invoking one single process. If one wants to use the parallelization feature, he must set some parameters: `max_parallel_workers_per_gather` defines the maximum number of processes which are allowed to run in parallel with each *gather node*. As it defaults to `0` it leads to the traditional behaviour - unless the value is changed. As mentioned above, each process working in parallel to the *gather node* is realized in a *background worker process*. The total number of *background worker processes* per instance is limited by `max_worker_processes` and defaults to 8. So it may be necessary to increase the value. Additionally, the

parameter `dynamic_shared_memory_type` must be set to a value other than `none`.
- The actual implementation in version 9.6 contains a lot of limitations which result from the fact that it must be enshured that this basic implementation is stable in all environments. It is likely that in further releases some of them disapear.
    - It is restricted to purely read-only commands: no UPDATE, DELETE, nor the CTE part of any writing command.
    - If there is a lock for any involved row.
    - If the transactions isolation level is `serializable`.
    - If the query is running inside of another query which is already parallelized. For example, if a function called by a parallel query issues an SQL query itself.

See also

- PostgreSQL documentation about parallel queries (https://www.postgresql.org/docs/9.6/static/parallel-query.html)
- Additional PostgreSQL wiki (https://wiki.postgresql.org/wiki/Parallel_Query)
