PostgreSQL

Database Administration
Volume 1

Federico Campoli

PostgreSQL Database Administration
Volume 1
Basic concepts

Federico Campoli

June 17, 2014

Contents

Preface

When I first came with the idea to write a PostgreSQL DBA book, I decided to publish via a commercial publisher. Shortly I changed my mind, as there's a massive lack of knowledge on the PostgreSQL database and few dba oriented books around.

Then I decided this book series shall stay free in order to reach as many people as possible, and hopefully become a reference to keep under the pillow.

Be warned before you start reading this book, I'm asking to be patient with me and my bad English.

As I did not study English and I'm not native, my writing will be probably wrong with many typos (I'm affected by some dyslexia as well) and bad grammar.

In exchange I'll promise this book, and all the subsequent, will start free and will remain free. Free as freedom and as beer.

This first book covers the basics on PostgreSQL.

A brief introduction, the installation and a deep enough to wet your taste analysis on the RDBMS, but not so deep to scare you. Not yet at least.

Intended audience

Database administrators, System administrators, Developers

Book structure

In this book assumes the reader knows how to perform the user operations like connecting to the database or creating tables.

The book covers the basic aspects of the database administration from the product installation to the cluster management.

A couple of chapters are dedicated to the logical and physical structure in order to show the two coin's faces. The maintenance and the important task of backup/restore completes the "other side of the monitor's" picture. The final chapter is dedicated to the developers who want to avoid the common mistakes and wrong assumptions on PostgreSQL.

Version and platform

This book cover the database version 9.3 on Debian GNU Linux 7.0. References to older version or different platform are explicitly specified.

Chapter 1

PostgreSQL at glance

PostgreSQL is a first class product with high end enterprise class level features. This first chapter is a general review on the product with a brief talk on the database's history.

1.1 Long time ago in a galaxy far far away...

Following the works of the Berkeley's Professor Michael Stonebraker, in the 1996 Marc G. Fournier asked for any volunteer interested in revamping the Postgres 95 project.

Date: Mon, 08 Jul 1996 22:12:19-0400 (EDT) From: "Marc G. Fournier" ¡scrappy@ki.net¿
Subject: [PG95]: Developers interested in improving PG95?
To: Postgres 95 Users ¡postgres95@oozoo.vnet.net¿
Hi... A while back, there was talk of a TODO list and development moving forward on Postgres95 ...
at which point in time I volunteered to put up a cvs archive and sup server so that making updates (and getting at the "newest source code") was easier to do...
... Just got the sup server up and running, and for those that are familiar with sup, the following should work (ie. I can access the sup server from my machines using this):
....

The answer came from Bruce Momjian,Thomas Lockhart e Vadim Mikheev, the very first PostgreSQL Global Development Team.

1.2 Features

Every time a new major release is released, new powerful features join the rich set of the product's functionalities. There's a small excerpt of what the latest version offer in terms of flexibility and reliability.

1.2.1 Write ahead logging

Like any RDBMS worth of this name PostgreSQL have the write ahead logging feature. In short, when a data block is updated the change is saved in a reliable location, the so called write ahead log. The effective write on the datafile is performed later. Should the database crash the WAL is scanned and the saved blocks are replayed during the crash recovery. PostgreSQL stores the redo records in fixed size segments, usually 16 MB. When the wal segment is full PostgreSQL switches to a newly created or recycled wal segment in the process called log switch.

1.2.2 Point in time recovery

When the log switch happens is possible to archive the previous segment in a safe location. Taking an inconsistent copy of the data directory is possible to restore a fully functional cluster because the archived wal segments have all the informations to replay the physical data blocks on the inconsistent data files. The restore can be, optionally stopped at a given point in time. For example is possible to recover a PostgreSQL cluster to one second before the a catastrophic happening (e.g. a table drop).

1.2.3 Standby server and high availability

The inconsistent snapshot can be configured to stay up in continuous archive recovery. PostgreSQL 8.4 supports the warm standby configuration where the standby server does not accept connections. From the version 9.0 is possible to enable the hot standby configuration to access the standby server in read only mode.

1.2.4 Streaming replication

The wal archiving doesn't work in real time. The wal shipping happens only after the log switch and in a low activity server this can leave the standby behind the master for a while. Using the streaming replication a standby server can get the wal blocks over a database connection in almost real time.

1.2.5 Transactional

PostgreSQL fully supports the transactions and is ACID compliant. From the version 8.0 the save points were introduced.

1.2.6 Procedural languages

Amongst the rich of feature procedural language pl/pgsql, many procedural languages such as perl or python are available for writing database functions. The DO keyword was introduced in the 9.1 to have anonymous function's code blocks.

1.2.7 Partitioning

The partitioning, implemented in PostgreSQL is still very basic. The partitions are tables connected with one empty parent table using the table's inheritance. Defining check constraints on the partitioned criteria the database can exclude, querying the parent table, the partitions not affected by the where condition. As the physical storage is distinct for each partition and there's no global primary key enforcement nor foreign keys can be defined on the partitioned structure.

1.2.8 Cost based optimizer

The cost based optimizer, or CBO, is the one of PostgreSQL's point of strenght. The query execution is dynamically determined and self adapting to the underlying data structure or the estimated amount of data affected. PostgreSQL supports also the genetic query optimizer GEQO.

1.2.9 Multi platform support

PostgreSQL nowadays supports almost any unix flavour and from the the version 8.0 is native to Windows.

1.2.10 Tablespaces

The tablespace support permits the data files fine grain distribution on the OS filesystems.

1.2.11 MVCC

The way PostgreSQL keeps things consistent is the MVCC which stands for Multi Version Concurrency Control. The mechanism is neat and efficient, offering great advantages and one single disadvantage. We'll see in detail further but keep in mind this important sentence.
There's no such thing like an update in PostgreSQL.

1.2.12 Triggers

Triggers to execute automated tasks on when DML is performed on tables and also views are supported at any level. The events triggers are also supported.

1.2.13 Views

The read only views are well consodlidated in PostgreSQL. In the version 9.3 was added the support for the materialized and updatable. Also the implementation is still very basic as no incremental refresh for the mat views nor update is possible on complex views. Anyway is still possible to replicate this behaviour using the triggers and procedures.

1.2.14 Constraint enforcement

PostgreSQL supports primary keys and unique keys to enforce local data meanwhile the referential integrity is guaranteed with the foreign keys. The check constraint to validate custom data sets is also supported.

1.2.15 Extension system

PostgreSQL implements the extension system. Almost all the previously known contrib modules are now implemented in this efficient way to add feature to the server using a simple SQL command.

Chapter 2

Database installation

This chapter will cover the install procedure, on Debian Gnu linux compiling from source and using the packaged install from the pgdg archive.

2.1 Install from source

Installing from source, using the default configuration settings requires the root access as the default install location is in /usr/local/. To simplify things I've created a procedure with minimal need for root access. This of course is still required but only for the os user creation and to install the dependencies.

Before starting with the postgresql part, ask your sysadmin, or do it by yourself, to do the following

- Create a postgres group and a postgres user
- Add the postgres user to the postgres group
- Install the packages
- build-essential
- libreadline6-dev
- zlib1g-dev

When everything is in place login as postgres user and download the source's tarball.

```
mkdir download
cd download
wget http://ftp.postgresql.org/pub/source/v9.3.4/postgresql-9.3.4.tar.bz2
```

Then extract the tarball with

```
tar xfj postgresql-9.3.4.tar.bz2
cd postgresql-9.3.4
```

Using the configuration's script option –prefix is possible to change the install directory to a custom location. Assuming the postgres user have his home directory in /home/postgres, we'll put the install target into the bin directory organised per mayor version. In this way is possible to have multiple versions on the same box without hassle.

```
mkdir -p /home/postgres/bin/9.3
./configure --prefix=/home/postgres/bin/9.3
```

The script will check all the dependencies and will generate the makefiles. Any error at configure time is usually displayed clearly.

When everything looks fine you can start the build process with the *make* command.

The time required to compile depends from the box speed. On a laptop usually this doesn't require more than 30 minutes. After the build is complete it's a good idea to run the regression tests before completing the installation.

```
make check
```

All the output is written in the directory src/test/regress/results.

If everything looks fine the installation can be completed with

```
make install
```

When the install is complete, into the /home/postgres/bin/9.3 directory will appear with 4 new subfolders *bin include lib* and *share.*

- **bin** is where the database binaries are stored
- **include** contains the server's header files
- **lib** is the shared objects location
- **share** is where the example files and the extension config are stored

2.2 Packaged install

The PostgreSQL Global Group mantains an apt repository to simplify the install on the GNU/Linux based on debian.

The supported Linux versions are listed on the wiki page http://wiki.postgresql.org/wiki/Apt and at moment are

- Debian 6.0 (squeeze)
- Debian 7.0 (wheezy)
- Debian unstable (sid)
- Ubuntu 10.04 (lucid)
- Ubuntu 12.04 (precise)
- Ubuntu 13.10 (saucy)
- Ubuntu 14.04 (trusty)

The packages are available for amd64 and i386.

The available database versions are

- PostgreSQL 8.4
- PostgreSQL 9.0
- PostgreSQL 9.1
- PostgreSQL 9.2
- PostgreSQL 9.3

Before starting you shall import the GPG key to validate the packages.

In a root shell simply run

```
wget --quiet -O - https://www.postgresql.org/media/keys/ACCC4CF8.asc | sudo apt-key add -
```

Then add the file pgdg.list into the directory /etc/apt/sources.d/ with the following contents

```
deb http://apt.postgresql.org/pub/repos/apt/ {codename}-pgdg main
```

Change the codename value accordingly with your distribuition. (e.g. wheezy) then you can run the installation in two simple steps.

```
apt-get update
apt-get install postgreql-9.3 postgreql-contrib-9.3 postgreql-client-9.3
```

The debian's packaged installation as post installation task, if not yet present, will create a new running database cluster in the home directory /var/lib/postgresql.

Chapter 3

Install structure

In this chapter we'll look at the PostgreSQL installation. Whatever installation method you did, the PostgreSQL binaries are the same. The packaged version comes with few extra utilities we'll take a look later as those present some caveats to be aware.

The source installation puts all the binaries into the bin sub directory in the target location specified by the –prefix parameter.

The packaged install puts the binaries into a folder organised per major PostgreSQL version.

e.g. /usr/lib/postgresql/9.3/bin/

3.1 The core binaries

We'll look first at the core binaries like postgres or psql.

A separate section is dedicated to wrappers and the contributed modules.

3.1.1 postgres

Is the database itself. It's possible to start it directly or using the pg_ctl utility.

The latter is the best way to control the instance except in case of the XID wraparound failure. In this case the only way to run the instance is executing postgres in single user mode.

For historical reason there's usually also the postmaster symbolic link to postgres.

3.1.2 pg_ctl

As mentioned before this is the utility for managing the PostgreSQL instance.

It can start,stop, reload the postgres process. It's also capable to send kill signals to the running instance.

pg_ctl accepts on the command line various options. The most important are the -D or –pgdata= to specify the database storage area and the -m to specify the shutdown mode. Check the section 4.3 for details.

pg_ctl also requires the action to perform on the instance.

The supported actions are

- **init[db]** initialises a directory as PostgreSQL data area
- **start** starts a PostgreSQL instance
- **stop** shutdowns a PostgreSQL instance
- **reload** reloads the configuration's files
- **status** checks the PostgreSQL instance running status
- **promote** promotes a standby server
- **kill** sends a custom signal to the running instance

3.1.3 initdb

Is the binary which initialises the PostgreSQL data area. initdb requires an empty directory to initialise. Various options can be specified on the command line, like the character enconding or the collation order.

3.1.4 psql

Is the PostgreSQL command line client. Despite his look very essential is one of the most flexible tools available to interact with the server. As is part of the core distribution is always present.

3.1.5 pg_dump

Is the binary dedicated to the backup. Generates consistent backups in various formats. The default is plain text. Supports the parallel dump implemented using the snapshot exports.

The switch to set is -F followed by a letter to indicate the wanted output format.

- **p** saves the sql statements to reconstruct the schema and/or data in plain text with no compression.

- **c** is the custom PostgreSQL format. Supports parallel restore, compression and object search.
- **d** the dump is saved in a directory. With this format is possible to dump in parallel.
- **t** saves the dump in the standard tar format.

Please on't be confused by the pg_dumpall . This does look more like a wrapper for pg_dump rather a dedicated program.

As pg_dumpall doesn't support all the pg_dump features is still very useful to save the cluster wide objects like the users with the switch –globals-only.

3.1.6 pg_restore

As the name suggests this is used to restore the database's dump. It can read the backups generated in all formats by pg_dump. The restore target can be a PostgreSQL connection or a file. If the backup's format is directory or custom then pg_restore can run the data load and index/key creations in multiple jobs.

3.1.7 pg_controldata

The program query the pg_control file where instance's vital informations are stored. The pg_control is one of the most important cluster's files. With a corrupted pg_control the instance cannot start.

3.1.8 pg_resetxlog

If the WAL files or the get corrupted the instance cannot perform a crash recovery. pg_resetxlog can solve the problem and make the instance startable but keep in mind this must the last chance, after trying any other possible solution.

The reset removes the WAL files and creates a new pg_control. The XID are also restarted.

The instance becomes startable at the cost of losing any reference between the transactions and the data files. All the physical data integrity is lost and any attempt to run DML queries results in data corruption.

The on line manual is absolutely clear on this point.

After running pg_resetxlog the database must start without user access, the entire content must be dumped, the data directory must be dropped and recreated from scratch using initdb and then the dump file can be restored using psql or pg_restore

3.2 Wrappers and contributed modules

In this section we'll take a brief look to the contributed binaries and the sql wrappers.

3.2.1 create/drop binaries

These binaries, createdb createlang createuser and dropdb droplang dropuser, are wrappers for the corresponding SQL functions. Each binary can create/drop a database an user or a procedural language. The command line parameters are quite the same as psql for the connection part.

3.2.2 clusterdb

Performs a database wide cluster on previously clustered tables. The binary can run on single tables specified on the command line. The word cluster can be confusing as the PostgreSQL implementation is very peculiar. Check the chapter 7 for further readings.

3.2.3 reindexdb

Performs a database wide reindex. It's possible to specify on the command line the target table or the target index. In chapter 7 we'll look deeply to the index maintenance, how this can affect the performances.

3.2.4 vacuumdb

This binary is a wrapper for the VACUUM SQL command. The VACUUM is the most important maintenance task and is required every 2 billion transactions to avoid the XID wraparound failure Similar to reindexdb and clusterdb, vacuumdb performs a database wide normal VACUUM. It's possible to specify a target table on the command line, plus various options to have a VACUUM FULL or an ANALYZE .

3.2.5 vacuumlo

Despite the name this binary doesn't perform any vacuum, its purpose is to remove orphaned large object from the pg_largeobject. The pg_largeobject is a system table where the database stores the binary objects. Usually is used when the size of the large object is bigger than 1GB. The theoretical limit for the large object is now 4 TB. Before the version 9.3 the limit were 2 GB.

3.3 Debian's specific utilities

The debian packaged install ships with some other, not official, utilities, mostly written in PERL.

3.3.1 pg_createcluster

Creates a new PostgreSQL cluster naming the configuration's directory in /etc/postgresql after the major version and the cluster's name. It's possible to specify the data directory and the initd options.

3.3.2 pg_dropcluster

Removes a PostgreSQL cluster created previously with pg_createcluster. The cluster must be stopped before the drop.

3.3.3 pg_lscluster

Lists the clusters created with pg_createcluster.

3.3.4 pg_ctlcluster

Controls the cluster in an almost similar way pg_ctl does. Using this wrapper for the shutdown is not a good idea as there's no way to tell the script which shutdown mode to use. For further readings on the shutdown sequence check the section 4.3 By default pg_ctlcluster performs a *smart* shutdown mode. Using the –force option the script first try a *fast* shutdown mode. If the database doesn't shutdown in a *reasonable time* the script then try an *immediate* shutdown. If the the instance is still up the script then sends a **kill -9** on the postgres process.

Chapter 4

Managing the instance

In this chapter we'll look how to initialise a PostgreSQL cluster and the management tasks. Later we'll take a look to the data area structure, the processes and the memory.

4.1 Initialising the data directory

A PostgreSQL instance is composed by a shared memory process and a data area managed by the postgres processes. The data area is initialised by initdb which requires an empty and writable directory to succeed. The binary location depends on the installation method, take a look to the chapters 3 and 2 for further informations.

Initdb accepts various parameters. If not supplied then the probam will try to get the informations from the environment variables.

Its basic usage is
initdb DATADIRECTORY
If the variable PGDATA is set then the DATADIRECTORY can be omitted.

```
 e.g

postgres@tardis:~/tempdata$ export PGDATA=`pwd`
postgres@tardis:~/tempdata$ /usr/lib/postgresql/9.3/bin/initdb
The files belonging to this database system will be owned by user "postgres".
This user must also own the server process.

The database cluster will be initialized with locale "en_GB.UTF-8".
The default database encoding has accordingly been set to "UTF8".
The default text search configuration will be set to "english".
```

```
Data page checksums are disabled.

fixing permissions on existing directory /var/lib/postgresql/tempdata ... ok
creating subdirectories ... ok
selecting default max_connections ... 100
selecting default shared_buffers ... 128MB
creating configuration files ... ok
creating template1 database in /var/lib/postgresql/tempdata/base/1 ... ok
initializing pg_authid ... ok
initializing dependencies ... ok
creating system views ... ok
loading system objects' descriptions ... ok
creating collations ... ok
creating conversions ... ok
creating dictionaries ... ok
setting privileges on built-in objects ... ok
creating information schema ... ok
loading PL/pgSQL server-side language ... ok
vacuuming database template1 ... ok
copying template1 to template0 ... ok
copying template1 to postgres ... ok
syncing data to disk ... ok

WARNING: enabling "trust" authentication for local connections
You can change this by editing pg_hba.conf or using the option -A, or
--auth-local and --auth-host, the next time you run initdb.

Success. You can now start the database server using:

    /usr/lib/postgresql/9.3/bin/postgres -D /var/lib/postgresql/tempdata
or
    /usr/lib/postgresql/9.3/bin/pg_ctl -D /var/lib/postgresql/tempdata -l
logfile start
```

After initialising the data area initdb shows the two methods to start the database instance. The first one is useful for debugging and development purposes as starts the database directly from the command line without daemonising the process.

```
postgres@tardis:~/tempdata$ /usr/lib/postgresql/9.3/bin/postgres -D
/var/lib/postgresql/tempdata
LOG:  database system was shut down at 2014-03-23 18:52:07 UTC
LOG:  database system is ready to accept connections
LOG:  autovacuum launcher started
```

To stop it simply press ctrl+c

The second method is more interesting as the PostgreSQL process is managed as daemon via pg_ctl.

The option *-l logfile* is to enable the logfile redirection. Without this option the server's output will appear on the standard output.

```
without -l

postgres@tardis:~/tempdata$ /usr/lib/postgresql/9.3/bin/pg_ctl -D
/var/lib/postgresql/tempdata  start
server starting
postgres@tardis:~/tempdata$ LOG:  database system was shut down at 2014-03-23
19:00:36 UTC
LOG:  database system is ready to accept connections
LOG:  autovacuum launcher started

with -l

postgres@tardis:~/tempdata$ /usr/lib/postgresql/9.3/bin/pg_ctl -D
/var/lib/postgresql/tempdata -l logfile start
server starting

postgres@tardis:~/tempdata$ tail logfile
LOG:  database system was shut down at 2014-03-23 19:01:19 UTC
LOG:  database system is ready to accept connections
LOG:  autovacuum launcher started
```

As seen in the subsection 3.1.2 pg_ctl accepts the command to perform on the cluster.

In order to stop the running instance simply run

```
postgres@tardis:~$ /usr/lib/postgresql/9.3/bin/pg_ctl -D
/var/lib/postgresql/tempdata -l logfile stop
```

```
waiting for server to shut down.... done
server stopped
```

4.2 The startup sequence

When the server process is started allocates the shared segment in memory. In the versions before the 9.3 this was the a potential point of failure because, if the shared_buffers was bigger than the kernel's max allowed shared memory segment the startup will fail with this sort of error

```
FATAL: could not create shared memory segment: Cannot allocate memory

DETAIL: Failed system call was shmget(key=X, size=XXXXXX, XXXXX).

HINT: This error usually means that PostgreSQL's request for a shared memory
segment exceeded available memory or swap space, or exceeded your kernel's
SHMALL parameter. You can either reduce the request size or reconfigure the
kernel with larger SHMALL. To reduce the request size (currently XXXXX bytes),
reduce PostgreSQL's shared memory usage, perhaps by reducing shared_buffers or
max_connections.
```

In this case the kernel parameters needs adjustment. As from my Oracle experience I take no chance and I do use the values suggested for an Oracle installation which incidentally works perfectly for PostgreSQL.

```
kernel.shmall = 2621440
kernel.shmmax = 18253611008
kernel.shmmni = 4096
kernel.sem = 250 32000 100 128
fs.file-max = 658576
```

Put those values into the file /etc/sysctl.conf and, as root, run *sysctl -p.*

The latest major version PostgreSQL switched from the SysV to Posix shared memory and mmap. This solved completely the shared memory tuning on Linux.

When the memory is allocated the postmaster reads the pg_control file to check if the instance requires recovery.

The pg_control contains references to the last checkpoint and the last known status for the instance.

If the instance is in dirty state, because a crash or an unclean shutdown, the startup replays the blocks from the WAL segments in the pg_xlog directory starting from the last checkpoint position read from the pg_control file.

Any corruption in the wal files or, even worse, the pg_control file results in a not startable instance.

After the recovery is complete or the cluster is in clean state, then the startup completes opening the database in production state.

4.3 The shutdown sequence

For those coming from Oracle, the PostgreSQL's shutdown sequence will sound familiar with few, but very important, exceptions.

A PostgreSQL process enters the shutdown status when receive a specific OS signal.

This can happen using the os kill or via pg_ctl.

As seen in the section 3.1.2 the latter accepts the -m switch to specify the shutdown mode. If not specified defaults to the mode smart which sends the signal SIGTERM to the running PostgreSQL process. This way the database will not accept new connections and will wait for the existing connections to exit before the shutdown.

The most useful PostgreSQL shutdown mode is the fast mode. This way the SIGQUIT is sent to the postgresql process and the database will disconnect all the sessions rolling back the open transactions and then will enter the shutdown sequence.

Either modes, smart and fast, make the database shutdown clean, leaving the cluster in consistent state when the postgres process exits.

As soon as the postgres process enters the shutdown sequence will issue a last checkpoint consolidating the updated physical blocks in the shared buffers to the data files.

As soon as the checkpoint is complete and the last consistent position is logged on the pg_control file, the database main process exits the accessory processes like the walwriter or the checkpointer and terminates removing the pid file.

The last checkpoint can slow down the entire sequence because is commonly spread through time to avoid any disk IO activity spike. If the shared_buffer is big and contains many dirty blocks, the checkpoint can run for a very long time. In addition, if at the shutdown time, another checkpoint is running the database will wait the completion before starting the final checkpoint. This meanwhile the new connections are forbidden, whatever shutdown mode you decided to use.

In order to have an idea of what's happening on the running cluster, is a good practice to change the GUC log_checkpoints = off to log_checkpoints = on.

For more informations about the database processes take a look to the section 4.4.

If the cluster is taking too long to stop, as last resort is possible to use the immediate mode. This will send a SIGQUIT to the running process causing the immediate exit of the main process and all the sessions.

The shutdown in this case is not clean and the subsequent start will perform a crash recovery starting from the last consistent state read from the pg_control file. The process is usually harmless with one important exception.

If the cluster have unlogged tables those relations are recreated from scratch when the recovery happens and all the data in those table **is lost**.

This is the main reason I suggest to avoid the pg_ctlcluster shipped with debian to stop the cluster. The program doesn't offer any control on the shutdown mode and can result in disastrous data loss. For more details take a look to the subsection 3.3.4.

A last word about the SIGKILL signal. It can happen the cluster refuse to stop even using the immediate mode. In this case, as last resort the SIGKILL or kill -9 can be used. The online manual is very clear on this point. As the SIGKILL cannot be trapped and evicts from the ram the process with the shared memory, its use will results in not freeing the resources used by the killed process.

This will very likely affect the start of a fresh instance. Please refer to your sysadmin to find out the best way to perform a memory and semaphore cleanup before starting PostgreSQL after a SIGKILL.

4.4 The processes

Since the early version 7.4, when the only process running the cluster were the old loved postmaster, PostgreSQL has enriched with new dedicated processes, becoming more complex but even more efficient.

With a running cluster there are at least six postgres processes, the one without colon in the process name is is the main database's process, started as seen in the section 4.2.

4.4.1 postgres: checkpointer process

As the name suggest this process take care of the cluster's checkpoint. The checkpoint is an important event in the database activity. When a checkpoint starts all the dirty pages in memory are written to the data files. The checkpoint by the time and the number of cluster's WAL switches. To adjust the checkpoin's frequency the GUC parameters checkpoint_timeout and checkpoint_segments are used. A third parameter, checkpoint_completion_target is used to spread the checkpoint over a percentage of the checkpoint_timeout, in order to avoid a massive disk IO spike.

4.4.2 postgres: writer process

To ease down the checkpoint activity the background writer scans the shared buffer for dirty pages to write down to the disk. The process is designed to have a minimal impact on the database activity. It's possible to tune the rounds length and delay using the

GUC parameters bgwriter_delay, time between two rounds, and bgwriter_lru_maxpages, the number of buffers after the writer's sleep.

4.4.3 postgres: wal writer process

This background process has been introduced recently to have a more efficient wal writing. The process works in rounds were write down the wal buffers to the wal files. The GUC parameter wal_writer_delay sets the milliseconds to sleep between the rounds.

4.4.4 postgres: autovacuum launcher process

This process is present if the GUC parameter autovacuum is set to on. It's scope is to launch the autovacuum backends at need. Anyway autovacuum can run even if autovacuum is turned of, when there's risk of the XID wraparound failure.

4.4.5 postgres: stats collector process

The process gathers the database's usage statistics for human usage and stores the informations into the location indicated by the GUC stats_temp_directory, by default pg_stat_temp. Those statistics are useful to understand how the database is performing, from pyshical and logical point of view.

4.4.6 postgres: postgres postgres [local] idle

This kind of process is the database backend, one for each established connection. The values after the colon square brackets show useful informations like the connected database, the username, the host and the executing query. The same informations are stored into the pg_stat_activity table.

4.5 The memory

The PostgreSQL's memory structure is not complex like other databases. In this section we'll take a to the various parts.

4.5.1 The shared buffer

The shared buffer, as the name suggests is the segment of shared memory used by PostgreSQL to manage the data pages.

Its size is set using the GUC[1] parameter shared_buffers and is allocated during the startup process.Any change requires the instance restart.

The segment is formatted in blocks with the same size of the data file's blocks, usually 8192 bytes. Each backend connected to the cluster is attached to this segment. Because usually its size is a fraction of the cluster's size, a simple but very efficient mechanism keeps in memory the blocks using a combination of LRU and MRU.

Since the the version 8.3 is present a protection mechanism to avoid the massive block eviction when intensive IO operations, like vacuum or big sequential reads, happens.

Each database operation, read or write, is performed moving the blocks via the shared buffer. This ensure an effective caching process and the memory routines guarantee the consistent read and write at any time.

PostgreSQL, in order to protect the shared buffer from potential corruption, if any unclean disconnection happens, resets by default all the connections.

This behaviour can be disabled in the configuration file but exposes the shared buffer to data corruption if the unclean disconnections are not correctly managed.

4.5.2 The work memory

This memory segment is allocated per user and its default value is set using the GUC parameter work_mem. The value can be altered for the session on the fly. When changed in the global configuration file becomes effective to the next transaction after the instance reload.

This segment is used mainly for expensive operations like the sort or the hash.

If the operation's memory usage exceeds the work_mem value then the PostgreSQL switches to a disk sort/hash.

Increasing the work_mem value results generally in better performances for sort/hash operations.

Because is a per user memory segment, the potential amount of memory required in a running instance is max_connections * work_mem. It's very important to set this value to a reasonable size in order to avoid any risk of out of memory error or unwanted swap.

In complex queries is likely to have many sort or hash operations in parallel and each one consumes the amount of work_mem for the session.

4.5.3 The maintenance work memory

The maintenance work memory is set using the GUC parameter maintenance_work_mem and follow the same rules of work_mem. This memory segment is allocated per user

[1]GUC, Grand Unified Configuration, this acronym refers to the parameters used to configure the instance

and is used for the maintenance operations like VACUUM or REINDEX. As usually this kind of operations happens on one relation at time, this parameter can be safely set to a bigger value than work_mem.

4.5.4 The temporary memory

The temporary memory is set using the GUC parameter temp_buffers. This is the amount of memory per user for the temporary table creation before the disk is used. Same as for the work memory and the maintenance work memory it's possible to change the value for the current session but only before any temporary table creation. After this the parameter cannot be changed anymore.

4.6 The data area

As seen before the data storage area is initialized by initdb . Its structure didn't change too much from the old fashioned 7.4. In this section we'll take a look to the various subdirectories and how their usage can affect the performances.

4.6.1 base

As the name suggests, the base directory contains the database files. Each database have a dedicated subdirectory, named after the internal database's object id. A freshly initialised data directory shows only three subdirectories in the base folder.

Those corresponds to the two template databases,template0 and template1, plus the postgres database. Take a look to chapter 5 for more informations.

The numerical directories contains various files, also with the numerical name which are actualy the database's relations, tables and indices.

The relation's name is set initially from the relation's object id. Any file altering operation like VACUUM FULL or REINDEX, will generate a new file with a different name. To find out the real relation's file name the relfilenode inside the pg_class system table must be queried.

4.6.2 global

The global directory contains all the cluster wide relations. In addition there's the very critical control file mentioned in 3.1.7 .

This small file is big exactly one database block, usually 8192 bytes, and contains critical informations for the cluster. With a corrupted control file the instance cannot start. The control file is written usually when a checkpoint occurs.

4.6.3 pg_xlog

This is the most important and critical directory, for the performances and for the reliability.

The directory contains the transaction's logs, named wal file. Each file is usually 16 Mb and contains all the data blocks changed during the database activity. The blocks are written first on this not volatile area to ensure the cluster's recovery in case of cras. The data blocks are then written later to the corresponding data files. If the cluster's shutdown is not clean then the wal files are replayed during the startup process from the last known consistent location read from control file.

In order to ensure good performance this location should stay on a dedicated device.

4.6.4 pg_clog

This directory contains the committed transactions in small 8k files, except for the serializable transactions. The the files are managed by the cluster and the amount is related with the two GUC parameters autovacuum_freeze_max_age and vacuum_freeze_table_age. Increasing the values for the two parameters the pg_clog must store the commit status to the "event horizon" of the oldest frozen transaction id. More informations about vacuum and the maintenance are in the chapter 7.

4.6.5 pg_serial

Same as pg_clog this directory stores the informations about the commited transactions in serializable transaction isolation level.

4.6.6 pg_multixact

Stores the informations about the multi transaction status, used generally for the row share locks.

4.6.7 pg_notify

Stores informations about the LISTEN/NOTIFY operations.

4.6.8 pg_snapshots

This directory is used to store the exported snapshots. From the version 9.2 PostgreSQL offers the transaction's snapshot export where one session can open a transaction and export a consistent snapshot. This way different session can access the snapshot and read all togheter the same consistent data snapshot. This feature is used, for example, by pg_dump for the parallel export.

4.6.9 pg_stat

This directory contains the permanent files for the statistic subsystem.

4.6.10 pg_stat_tmp

This directory contains the temporary files for the statistic subsystem. As this directory is constantly written, is very likely to become an IO bottleneck. Setting the GUC parameter stats_temp_directory to a ramdisk speeds can improve the database performances.

4.6.11 pg_subtrans

Stores the subtransactions status data.

4.6.12 pg_twophase

Stores the two phase commit data. The two phase commit allows the transaction opening independently from the session. This way even a different session can commit or rollback the transaction later.

4.6.13 pg_tblspc

The directory contains the symbolic links to the tablespace locations. A tablespace is a logical name pointing a physical location. As from PostgreSQL 9.2 the location is read directly from the symbolic link. This make possible to change the tablespace's position simply stopping the cluster, moving the data files in the new location, creating the new symlink and starting the cluster. More informations about the tablespace management in the chapter 5.

Chapter 5

The logical layout

This chapter is dedicated to the PostgreSQL logical layout. After the connection process we'll look to the logical objects, the relations like tables, indices and views. The chapter will end with the tablespaces logical aspect and how PostgreSQL handle the transactions.

5.1 The connection

When a client tries to connect to a PostgreSQL cluster the process follow few stages which can result in rejection or connection.

The first stage is the host based authentication where the cluster scans the pg_hba.conf file searching a match for the connection's parameters, like the host, the username etc. This file is usually stored into the data area amongst the postgresql.conf file and is read from the top to the bottom. If there's match the corresponding method is used, otherwise if there's no match then the connection is refused.

The pg_hba.conf is structured as shown in table 5.1

Type	Database	User	Address	Method
local	name	name	ipaddress/network mask	trust
host	*	*	host name	reject
hostssl				md5
hostnossl				password
				gss
				sspi
				krb5
				ident
				peer
				pam
				ldap
				radius
				cert

Table 5.1: pg_hba.conf

The column type specifies if the connection is local and happens via unix socket or host,hostssl,hostnossl, in this case the tcp/ip is used.The host type matches either an SSL or plain connection, the hostssl only a SSL connection and hostnossl only a plain connection.

The Database and User columns are used to match specific databases or users from the incoming connection. It's possible to use a wildcard to specify everything.

The column address is used only if the type uses the tcp/ip and can be an ip address with network mask or a hostname. Both ipv4 and ipv6 are supported.

The last column is the authentication method. PostgreSQL supports many methods, from the password challenge to the sophisticated radius or kerberos.

For now we'll take a look to the most common.

- **trust** allow the connection without any request. Is quite useful if the password is lost but represent a threat on production.

- **peer** allow the connection if the OS user matches the database user. Useful to authenticate to the database on the local boxes. Initdb sets this as default method for the local connections.

- **password** allow the connection matching the user and password with pg_shadow system table. Beware asthis method sends the password in clear over the network.

- **md5** same as for password this method offer a md5 encryption for the passwords. As the md5 is deterministic a pseudo random subroutine is used during the password challenge to avoid the same string to be sent over the network.

When the connection request matches the pg_hba.conf and the authentication method is cleared, the connection becomes established. The postgres main process forks a new backend process which attaches to the shared buffer.

As the fork process is expensive the connection is a potential bottleneck. A great amount of opening connections can degenerate in zombies resulting in the worst case in a denial of service.

A solution could be to keep all the needed connections constantly established. Even if it seems reasonable, this approach have a couple of unpleasant side effects.

Each connection's slot in the GUC max_connections, consumes about 400 bytes of shared memory; each established connection requires the allocation of the work_mem and the os management of the extra backend process.

For example a 512 MB shared_buffer and 100MB work_mem, with with 500 established connections consumes about 49 GB. Even reducing the work_mem to 10MB the required memory is still 5 GB; in addiction, as seen in 4.5.2, this parameter affects the sorts and subsequently the performance, his value requires then extra care.

In this case a connection pooler like pgpool http://www.pgpool.net/ or the lightweight pgbouncer http://pgfoundry.org/projects/pgbouncer is a good solution. In particular the latter offers a very simple to use configuration, with different pooling levels.

The GUC parameter listen_addresses in a freshly initialised data area is set to localhost. This way the cluster accepts only tcp connections from the local machine. In order to have the cluster listening on the network this parameter must be changed to the correct address to listen or to * for 'all'. The parameter accepts multiple values separated by commas.

Changing the parameters max_connections and listen_addresses require the cluster shutdown and startup as described in 4.2 and 4.3

5.2 Databases

In order to establish the connection, PostgreSQL requires a target database.

When the connection happens via the client psql, the database can be omitted. In this case the environment variable $PGDATABASE is used. If the variable is not set then the target database defaults to the login user.

This can be confusing because even if the pg_hba.conf authorises the connection, this aborts with the message

```
postgres@tardis:~$ psql -U test -h localhost
Password for user test:
psql: FATAL:  database "test" does not exist
```

In this case if the database to connect to is unknown, the connection should contain the *template1* as last parameter. This way the connection will establish to the always present template1 database . After the connection is established a query to the pg_database system table will return the database list.

```
postgres@tardis:~$ psql -U test -h localhost template1
Password for user test:
psql (9.3.4)
SSL connection (cipher: DHE-RSA-AES256-SHA, bits: 256)
Type "help" for help.
```

```
template1=> SELECT datname FROM pg_database;
    datname
---------------
 template1
 template0
 postgres
(3 rows)
```

A brief note for the database administrators coming from MS SQL or MySql. The database postgres can be confused for the system database, like the master db or the mysql database.

That's not correct, the postgres database have nothing special and it's created by default only in the recent major PostgreSQL versions because is required by specific tools like pg_bench.

The template0 and template1 are the template databases. A template database is used to build new database copies via the physical file copy.

During the initdb the template1 is initialised with the correct references to the WAL records. The system views and the procedural language PL/PgSQL are then loaded into the template1. Finally the template0 and postgres databases are created from template1.

The database template0 doesn't allow the connections and is used to rebuild template1 if gets corrupted or to build a new database with character encoding or ctype an different from the cluster wide values.

```
postgres=# CREATE DATABASE db_test WITH ENCODING 'UTF8' LC_CTYPE '
    en_US.UTF-8';
ERROR:  new LC_CTYPE (en_US.UTF-8) is incompatible with the LC_CTYPE
    of the
template database (en_GB.UTF-8)
HINT:  Use the same LC_CTYPE as in the template database, or use
    template0 as
template.

postgres=# CREATE DATABASE db_test WITH ENCODING 'UTF8' LC_CTYPE '
    en_US.UTF-8'
```

```
TEMPLATE template0;
CREATE DATABASE
postgres=#
```

If not specified, the CREATE DATABASE statement will use template1. A database can be renamed or dropped with the ALTER DATABASE and DROP DATABASE statements. Those operations require the exclusive access to the database. If any connection except the one performing the operation is present on the database the operation will abort.

```
postgres=# ALTER DATABASE db_test RENAME TO db_to_drop;
ALTER DATABASE

postgres=# DROP DATABASE db_to_drop;
DROP DATABASE
```

5.3 Tables

A database is the logical container of the relations. A relation is the relational object making the data accessible.

The first kind of relation we'll take a look is the table.

This is the fundamental storage unit for the data. PostgreSQL implements various kind of tables having the full support, a partial implementation or no support at all for the durability.

The table creation is performed using the standard SQL command CREATE TABLE.

The PostgreSQL implementation does not guarantee the data is stored in a particular order. This is a straight MVCC consequence Take a look to more information in 6.6.

5.3.1 Logged tables

If executed without options, CREATE TABLE creates a logged table. This kind of table implements fully the durability being WAL logged at any time. The data is managed in the shared buffer, logged to the WAL and finally consolidated to the data file.

5.3.2 Unlogged tables

This kind of table were introduced in the 9.1. The data is still consolidated to the data file but the blocks aren't WAL logged. This make the write operations considerably faster at the cost of the data consistency. This kind of table is not crash safe and the database truncate any existing data during the crash recovery. Also, because there's no WAL record the unlogged tables aren't replicated to the physical standby.

5.3.3 Temporary tables

A temporary table's lifespan lasts the time of the connection. This kind of tables are useful for any in memory operation. The temporary table stays in memory as long as the amount of data is no bigger than temp_buffers seen in 4.5.4.

5.3.4 Table inheritance

As PostgreSQL is an Object Relational Database Management System, some of the object oriented programming concepts are implemented. The relations are referred generally as classes and the columns as attributes.

The inheritance binds a parent table to one or more child tables which have the same parent's attribute structure. The inheritance can be defined at creation time or later. If a manually defined table shall inherit another the attribute structure shall be the same as the parent's.

The PostgreSQL implementation is rather confusing as the unique constraints aren't globally enforced on the inheritance tree and this prevents the foreign key to refer inherited tables. This limitation makes the table partitioning tricky.

5.3.5 Foreign tables

The foreign tables were first introduced with PostgreSQL 9.1, improving considerably the way the remote data can be accessed. A foreign table requires a called foreign data wrapper to define a foreign server. This can be literally anything. Between the contrib modules PostgreSQL has the file_fdw to create foreign tables referring CSV or COPY formatted flat files. In the the version 9.3 finally appeared the postgres_fdw with the read write for the foreign tables. The postgres_fdw implementation is similar to dblink with a more efficient performance management and the connection caching.

5.4 Indices

An index is a relation capable to map the values in an structured way pointing the table's position where the rows are stored. The presence of an index doesn't mean this will be used for read. By default and index block read have an estimated cost four times than the table block read. However the optimiser can be controlled using the two GUC parameters seq_page_cost for the table sequential read read and random_page_cost for the index. Lowering the latter will make more probable the optimiser will choose the indices when building the execution plan.

Creating indices is a complex task. At first sight adding an index could seem a harmless action. Unfortunately their presence adds overhead to the write operations unpredictably. The rule of thumb is *add an index only if really needed.* Monitoring the index usage is crucial. Querying the statistics view pg_stat_all_indexes is possible to find out if the indices are used or not.

For example, the following query finds all the indices in che public schema, with zero usage from the last database statistics reset.

```
SELECT
        schemaname,
        relname,
        indexrelname,
        idx_scan
FROM
         pg_stat_all_indexes
WHERE
                schemaname='public'
        AND     idx_scan=0
;
```

PostgreSQL supports many index types.

The general purpose B-tree, implementing the Lehman and Yao's high-concurrency B-tree management algorithm. The B-tree can handle equality and range queries and returns ordered data. As the data is actually stored in the page and because the index is not TOASTable, the max length for an index entry is 1/3 of the page size. This is the limitation for the variable length indexed data (e.g. text).

The hash indices can handle only equality and aren't WAL logged. That means their changes are not replayed if the crash recovery occurs, requiring a reindex in case of unclean shutdown.

The GiST indices are the Generalised Search Tree. The GiST is a collection of indexing strategies organized under an infrastructure. They can implement arbitrary indexing schemes like B-trees, R-trees or other. The operator classes shipped with PostgreSQL are for the two elements geometrical data and for the nearest-neighbor search. As the GiST indices are not exact , when scanned the returned set doesn't requires a to remove the false positives.

The GIN indices are the Generalised Inverted Indices. This kind of index is optimised for indexing the composite data types or vectors like the full text search elements. The GIN are exact indices, when scanned the returned set doesn't require recheck.

There's no bitmap index implementation in PostgreSQL. At runtime the executor can emulate partially the bitmap indices reading the B-tree sequentially and matching the occurrences in the on the fly generated bitmap.

The index type shall be specified in the create statement. If the type is omitted then the index will default to the B-tree.

```
CREATE INDEX idx_test ON t_test USING hash (t_contents);
```

As the index maintenance is a delicate matter, the argument is described in depth in 7.

5.5 Views

A view is the representation of a query, stored in the system catalogue for quick access. All the objects involved in the view are translated to the internal identifiers at the creation time; the same happens for any wild card which is expanded to the column list.

An example will explain better the concept. Let's create a simple table. Using the generate_series() function let's put some data into it.

```


CREATE TABLE t_data
        (
                i_id serial,
                t_content       text
        );

ALTER TABLE t_data
ADD CONSTRAINT pk_t_data PRIMARY KEY (i_id);


INSERT INTO t_data
        (
                t_content
        )
SELECT
        md5(i_counter::text)
FROM
        (
                SELECT
                        i_counter
                FROM
                        generate_series(1,200) as i_counter
        ) t_series;

CREATE OR REPLACE VIEW v_data
AS
  SELECT
```

```
        *
  FROM
        t_data;
```

The SELECT * from t_data or v_data looks exactly the same, the view simply runs the stored SQL used at creation time. If we look to the stored definition in pg_views we'll find the wildcard is expanded into the table's columns.

```
 db_test=# SELECT * FROM pg_views where viewname='v_data';
-[ RECORD 1 ]--------------------
schemaname | public
viewname   | v_data
viewowner  | postgres
definition |  SELECT t_data.i_id,
           |     t_data.t_content
           |    FROM t_data;
```

Now let's add a new column to the t_data table and run again the select on the table and the view.

```
 ALTER TABLE t_data ADD COLUMN d_date date NOT NULL default now()::
     date;

 db_test=# SELECT * FROM t_data LIMIT 1;
 i_id |            t_content             |   d_date
------+----------------------------------+------------
    1 | c4ca4238a0b923820dcc509a6f75849b | 2014-05-21
(1 row)


db_test=# SELECT * FROM v_data LIMIT 1;
 i_id |            t_content
------+----------------------------------
    1 | c4ca4238a0b923820dcc509a6f75849b
(1 row)
```

The view doesn't show the new column. To update the view definition a new CREATE OR REPLACE VIEW statement must be issued.

```
 CREATE OR REPLACE VIEW v_data
AS
  SELECT
        *
  FROM
        t_data;

db_test=# SELECT * FROM v_data LIMIT 1;
 i_id |            t_content             |   d_date
------+----------------------------------+------------
    1 | c4ca4238a0b923820dcc509a6f75849b | 2014-05-21
(1 row)
```

Because the views are referring the objects identifiers they will never invalidate when the referred objects are altered. The CREATE OR REPLACE statement updates the view definition only if the column list adds new attributes in the end. Otherwise, any change to the existing columns requires the view's drop and recreate.

When one or more view are pointing a relation this cannot be dropped. The option CASCADE in the drop statement will drop the dependant objects before the final drop. This is a dangerous approach though. Dropping objects regardless can result in data or functionality loss.

When a drop is blocked by dependant objects the database emits a message with the informations about the dependencies. If the amount of objects is too much big it's better to query the pg_depend table to find out the correct dependencies. This table lists all the dependencies for each object using a peculiar logic.

As seen before a view is a logical short cut to a pre saved query. This means the database will follow all the steps to execute exactly the same way if the entire query has been sent via client, except for the network overhead.

Nothing forbids a view to point another view inside the definition or join the one or more views in a different query. This can cause massive regression on the overall performance because each view require an execution plan and mixing the views will cause not efficient planning.

To mark a relation is a view it's a good idea to use a naming prefix like v_. This will distinguish them from the tables marked with the prefix t_. In 9 we'll take a to the naming conventions to let the database schema self explanatory.

PostgreSQL from the version 9.3 supports the updatable simple views. A view is simple if

- Have exactly one entry in its FROM list, which must be a table or another updatable view
- Does not contain WITH, DISTINCT, GROUP BY, HAVING,LIMIT, or OFFSET clauses at the top level
- Does not contain set operations (UNION, INTERSECT or EXCEPT) at the top level
- All columns in the view's select list must be simple references to columns of the underlying relation. They cannot be expressions, literals or functions. System columns cannot be referenced, either
- columns of the underlying relation do not appear more than once in the view's select list
- does not have the security_barrier property

If the view doesn't fit those rules it's still possible to make it updatable using the triggers with the INSTEAD OF clause.

The major version 9.3 introduces also the materialised view concept. This is a physical snapshot of the saved SQL and can be refreshed with the statement REFRESH MATERIALIZED VIEW.

5.6 Tablespaces

A tablespace is a logical shortcut for a physical location. This feature was first introduced with the major release 8.0, recently with the 9.2 had a small adjustment to make the dba life easier.

When a new relation is created without tablespace specification, the relation's tablespace is set to the GUC prameter default_tablespace or, if this is missing, is set to the database's default tablespace. Anyway, without any specification the default tablespace is the pg_default, corresponding to the $PGDATA/base directory.

In order to create a new tablespace the chosen directory must be owned by the os user which started the postgres process and must be specified as absolute path.

For example, having a folder named /var/lib/postgresql/pg_tbs/ts_test a tablespace we can create a new tablespace ts_test.

```
CREATE TABLESPACE ts_test
OWNER postgres
LOCATION '/var/lib/postgresql/pg_tbs/ts_test' ;
```

Only superusers can create tablespaces. The OWNER clause is optional, if omitted the tablespace is owned by the user issuing the command.

The tablespaces are cluster wide, each database sees the same list in the pg_tablespace system table.

To create a relation into the tablespace ts_test just add the TABLESPACE clause followed by the tablespace name at creation time.

```
CREATE TABLE t_ts_test
        (
                i_id serial,
                v_value text
        )
TABLESPACE ts_test ;
```

It's possible to move a relation from a tablespace to another using the ALTER command.

For example, this is the command to move the previously created table to the pg_default tablespace.

```
ALTER TABLE t_ts_test SET TABLESPACE pg_default;
```

The move is transaction safe but requires an exclusive lock on the affected relation. If the relation have a significant size this means no access to the data for the time required by the move.

In addition, changing the tablespaces is not permitted when the backup is in progress, the exclusive lock is not compatible with the locks issued by the schema and data export.

The tablespace feature adds flexibility to the space management. Even if is still primitive a careful design can improve sensibly the performances, for example, putting tables and indices on different devices to maximise the disks bandwidth.

To remove a tablespace there is the DROP TABLESPACE command. The tablespace must be empty before the drop. There's no CASCADE clause to have the tablespace's contents dropped with the tablespace.

```
postgres=# DROP TABLESPACE ts_test;
ERROR:  tablespace "ts_test" is not empty

postgres=# ALTER TABLE t_ts_test SET TABLESPACE pg_default;
ALTER TABLE
postgres=# DROP TABLESPACE ts_test;
DROP TABLESPACE
```

In 6.5 we'll take a look to the how PostgreSQL implements the tablespaces on the physical side.

5.7 Transactions

PostgreSQL implements the MVCC which stands for Multi Version Concurrency Control. This offers high efficiency in multi user access for read and write queries. When a new query starts a transaction identifier is assigned, the XID a 32 bit quantity. To determine the transaction's snapshot visibility, all the committed transactions with XID lesser than the current XID are in the past and then visible. Otherwise, all the transactions with XID greater than the current XID are in the future and not visible.

This comparison happens at tuple level using two system fields xmin and xmax having the xid data type. When a transaction creates a new tuple then the transaction's xid is put into the tuple's xmin value. When a transaction deletes a tuple then the xmax value is set to the transaction's xid leaving the tuple in place for read consistency. When a tuple is visible to any transaction is called a live tuple, a tuple which is no longer visible is a dead tuple.

PostgreSQL have no dedicated field for the update's xid. That's because when an UPDATE is issued PostgreSQL creates a new tuple's version with the updated data and sets the xmax value in the old version making it disappear.

A dead tuple can be reclaimed by VACUUM if no longer required by running transactions, anyway tables updated often can result in data bloat for the dead tuples and for the eventual indices. Look to 6.3 for more information on the tuples.

When designing a new data model, the PostgreSQL's peculiar behaviour on the update should be the first thing to consider, in order to limit the table and index bloat.

Among the xmin,xmax two other system fields the cmin and cmax which data type is CID, command id. Those are similar to the xmin/xmax quantities and usage and their usage is to track the internal transaction's commands, in order to avoid the command execution on the same tuple more than one time. The pratical issue is explained in the well known Halloween Problem. For more informations take a look here http://en.wikipedia.org/wiki/Halloween_Problem.

The SQL standard defines four level of transaction's isolation levels where some phenomena are permitted or forbidden.
Those phenomena are the following.

- **dirty read** A transaction reads data written by a concurrent uncommitted transaction
- **nonrepeatable read** A transaction re reads the data previously read and finds the data changed by another transaction which has committed since the initial read
- **phantom read** A transaction re executes a query returning a set of rows satisfying a search condition and finds that the set of rows satisfying the condition has changed because another recently-committed transaction

Table 5.2 shows the isolation levels with the allowed phenomena. In PostgreSQL it's possible to set all the four isolation levels but only the three more strict are supported. Setting the isolation level to read uncommited fallback to the read committed in any case.

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read
Read uncommitted	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible
Repeatable read	Not possible	Not possible	Possible
Serializable	Not possible	Not possible	Not possible

Table 5.2: Standard SQL Transaction Isolation Levels

By default the global isolation level is set to read committed, it's possible to change the session's transaction isolation level using the command:

```
SET TRANSACTION ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ |
    READ
COMMITTED | READ UNCOMMITTED };
```

To change the default transaction isolation level cluster wide there is the GUC parameter transaction_isolation.

Chapter 6

The physical layout

This chapter explores the physical storage. We'll start from the data files moving deep into the data blocks and finally to the tuples. After looking to the tablespaces we'll complete the outline started in 5.7 with the MVCC.

6.1 Data files

Wherever the database stores the files, the $PGDATA/base or a different tablespace, those files are named, initially, after the relation's object identifier, a 4 byte unsigned integer. The word *initially* means there's no guarantee the file will stay the same in the future. Some file altering operations like the REINDEX or the VACUUM FULL change the file name leaving the relation's object identifier unchanged.

The maximum size allowed for a datafile is 1 GB, then a new segment with the same name and a sequential suffix is created. For example if the relation 34554001 reaches the upper limit a new file named 34554001.1 is added, when this one reaches 1 GB then a 34554001.2 is added and so on.

Alongside the main data files there're some additional forks needed for the database activity.

6.1.1 Free space map

The free space map segment is present for the index and table's data files . It's named after the relation's filenode with the suffix _fsm and is used to track the free space in the relation.

6.1.2 Visibility map

The table's file are also called heap files. Alongside those files there is a second fork called visibility map. Like before this file is named after the relation's filenode with the suffix _vm. Its usage is for tracking the data pages having all the tuples visible to all the active transactions. This fork is used also for the index only scans where the data is retrieved from the index page only.

6.1.3 Initialisation fork

The initialisation fork is an empty table or index page, stored alongside the unlogged relation's data file. As seen in 5.3.2 when the database performs a crash recovery the unlogged relations are zeroed. The initialisation fork is used to reset them and all the relation's accessory forks are deleted.

6.1.4 pg_class

All the current database's relations are listed in the pg_class system table. The field relfilenode shows the relation's filename.

The oid field, hidden if wildcard is used in the select list, is the internal object identifier. PostgreSQL is shipped with plenty of useful functions to get informations from the relation's OID. For example the function pg_total_relation_size(regclass) returns the space used by the table plus the indices, the additional forks and the eventual TOAST table. The function returns the size bytes. Another function, the pg_size_pretty(bigint), returns a human readable format for better reading.

The field relkind is used to track the relation's kind

Value	Relation's kind Read
r	ordinary table
i	index
S	sequence
v	view
m	materialised view
c	composite type
t	TOAST table
f	foreign table

Table 6.1: Relkind possible values

6.2 Pages

The datafiles are organized as array of fixed length elements called pages. The default size is 8k. Pages of a table's datafile are called heap pages to distinguish from the index pages which differs from the former only for the special space present in the page's end. The figure 6.1 shows an index page structure. The special space is small area in the page's end used to track down the index structure. For example a B-tree index stores in the special space the pointers to the leaf pages.

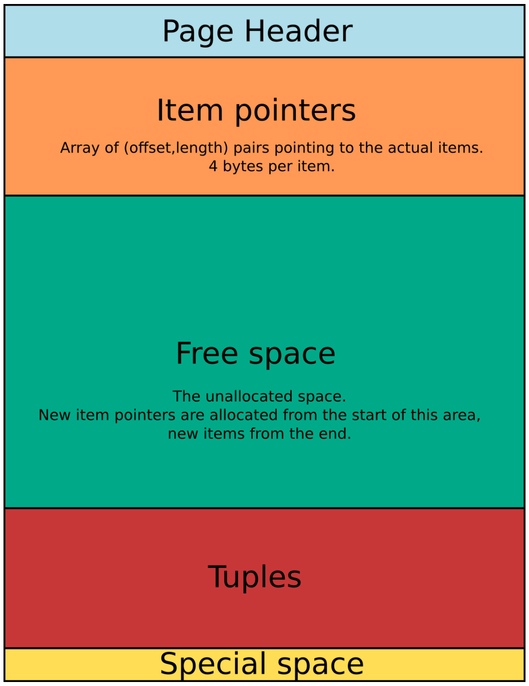
Page Header
Item pointers
Array of (offset,length) pairs pointing to the actual items.
4 bytes per item.
Free space
The unallocated space.
New item pointers are allocated from the start of this area,
new items from the end.
Tuples
Special space

Figure 6.1: Index page

The page starts with a header 24 bytes long followed by the item pointers, usually 4 bytes long. In the page's bottom are stored the actual items, the tuples.

The item pointers is an array of pairs, offset and length, pointing the actual tuples in the page's bottom. The tuples are put in the page's bottom and going backwards to fill up all the available free space.

The header contains the page's generic space management informations as shown in figure 6.2.

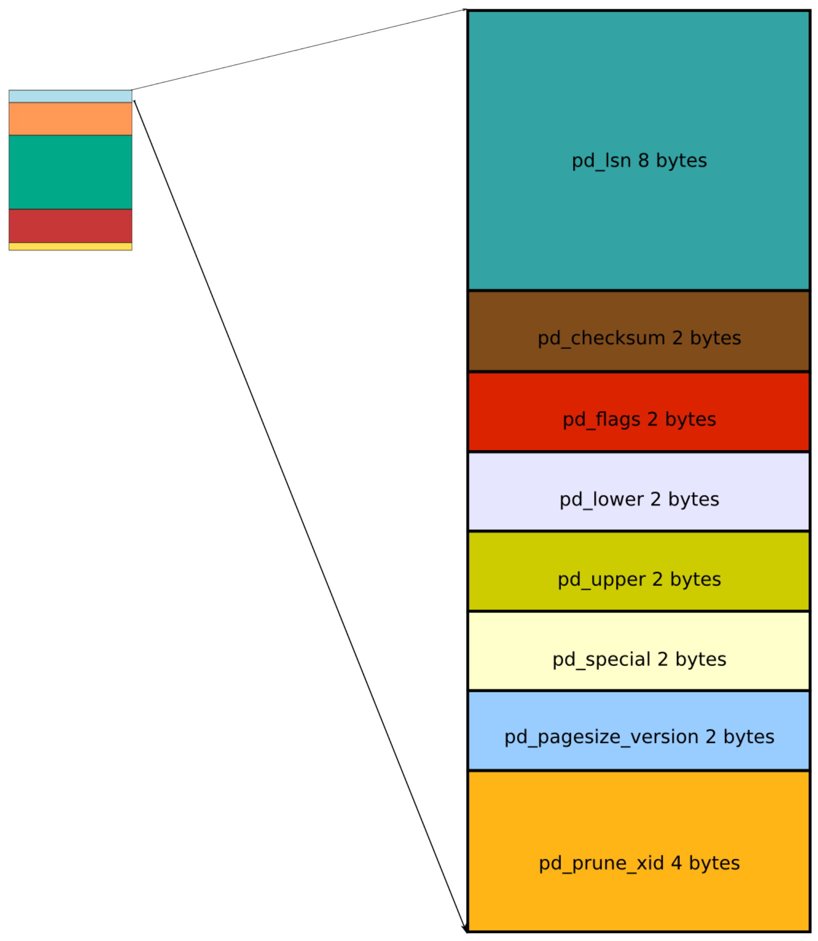

Figure 6.2: Page header

- **pd_lsn** identifies the xlog record for last page's change. The buffer manager uses the LSN for WAL enforcement. A dirty buffer is not dumped to the disk until the xlog has been flushed at least as far as the page's LSN.

- **pd_checksum** stores the page's checksum if enabled, otherwise this field remain unused
- **pd_flags** used to store the page's flags
- **pg_lower** offset to the start of the free space
- **pg_upper** offset to the end of the free space
- **pg_special** offset to the start of special space
- **pd_pagesize_version** page size and page version packed together in a single field.
- **pg_prune_xid** is a hint field to helpt to determine if pruning is useful. It's used only on the heap pages.

The pd_checksum field substitute the pd_tli field present in the page header up to PostgreSQL 9.2 and used to track the xlog record across the timeline id. The page's checksum can be enabled only when the data area is initialised with initdb and cannot be disabled later.

The offset fields, pg_lower, pd_upper and the optional pd_special, are 2 bytes long, this means PostgreSQL can only support pages up to 32KB.

The page version was introduced with PostgreSQL 7.3. For the prior releases the page version is arbitrarily considered 0; PostgreSQL 7.3 and 7.4 had the page version to 1; PostgreSQL 8.0 used the version 2; PostgreSQL 8.1 and 8.2 used version number 3; From PostgreSQL 8.3 the page's version number is 4.

6.3 Tuples

As seen in 5.7 PostgreSQL when updating the rows, generates new row versions stamping the old as dead.

Each tuple comes with a fixed header of system columns usually 23 bytes as shown in the figure 6.3.

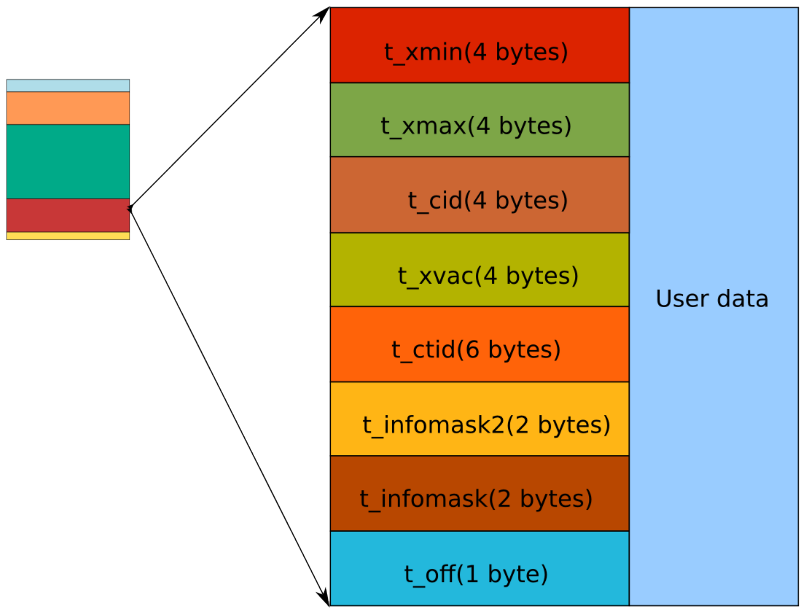

Figure 6.3: Tuple structure

The two fields t_xmin and t_xmax are stamped respectively at tuple's insert and delete with the operation's transaction id.

The field t_cid is a "virtual" field used either for cmin and cmax. This is possible because the command id have meaning only inside a transaction.

The field t_xvac is used by VACUUM when moving the rows, according with the source code's comments in src/include/access/htup_details.h this is used only by old style VACUUM FULL.

The t_cid is the tuple's location id, a couple of integers pointing the page number and the tuple's index. When a new tuple is created t_cid is set to the actual row's value; When the tuple is updated the this value changes to the location of the newer tuple's version.

A tuple is then recognised as last version when xmax is invalid or t_cid points to itself. If xmax is also valid then the tuple is the last locked or deleted version of the tuple's version chain.

The two infomask fields are used to store various flags like the presence of the tuple OID or if the tuple has NULL values.
The last field t_off is used to mark the offset to the composite data, the actual tuple's data. This field's value is usually zero if the table doesn't have NULLable fields or is created WITH OIDS. If present, tThe OID and the a NULL bitmap are placed just after the tuple's header. The bitmap begins just after the fixed header and occupies enough bytes to have one bit per data column. The OID comes after the bitmap and is 4 bytes long.

6.4 TOAST

The oversize attribute storage technique is the PostgreSQL implementation for the data overflowing the page size.

The user data shown in figure 6.3 is a stream of composite data. Actually the data itself is logically described by the composite model stored in the system catalogue. The attributes in the model can be grouped in two categories, the fixed length and the variable length data type (varlena).

For example, a four bytes integer is a fixed length type and a text is a variable length. For the PostgreSQL's internal routines the data at physical level appears all the same, as a generic datum. When the datum is loaded into the shared buffer becomes meaningful and is managed accordingly with its nature.

The attribute's kind is stored in the first two bits[1] of the varlena length word. When both bits are zero then the attribute is a fixed length data type and the remaining bits give the datum size in bytes including the length word.

If the first bit is set then the value have only a single-byte header and the remaining bits describe the total datum size in bytes including the length byte. If the remaining bits are all zero, then the value is a pointer to an out of line data stored in a separate TOAST table which structure is shown in figure 6.4.

If the first bit is zero but the second bit is set then the datum is compressed and must be decompressed before the use. The compression uses the LZ family algorithm.

[1]On the big-endian architecture those are the high-order bits; on the little-endian those are the low-order bits

the OID of the toasted data, the chunk_seq and integer for ordering the chunks within the value and the chunk_data, a bytea field containing the the overflown data.

The chunk size is normally 2k and is controlled at compile time by the symbol TOAST_MAX_CHUNK_SIZE. The TOAST code is triggered by the value TOAST_TUPLE_THRESHOLD also 2k by default. When the tuple's size is bigger then the TOAST routines are triggered.

The TOAST_TUPLE_TARGET normally 2 kB as well governs the compression's behaviour. PostgreSQL will compress the datum to achieve a final size lesser than TOAST_TUPLE_TARGET. If cannot then the out of line storage is used.

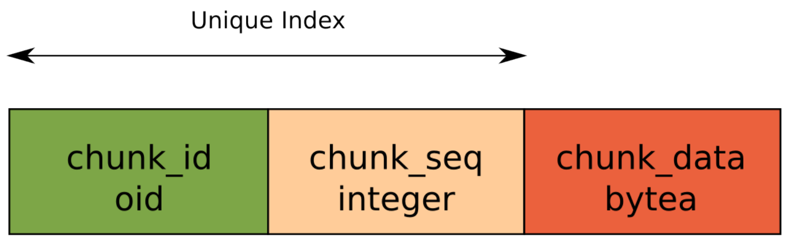

Figure 6.4: Toast table structure

TOAST offers four different storage strategies. Each strategy can be changed per column using the ALTER TABLE SET STORAGE statement.

- PLAIN prevents either compression or out-of-line storage; It's the only storage available for fixed length data types.
- EXTENDED allows both compression and out-of-line storage. It is the default for most TOAST-able data types. Compression will be attempted first, then out-of-line storage if the row is still too big.
- EXTERNAL allows out-of-line storage but not compression.
- MAIN allows compression but not out-of-line storage. Actually the out-of-line storage is still performed as last resort.

The out of line storage have the advantage of leaving out the stored data from the row versioning; if the TOAST data is not affected by the update there will be no dead row for the TOAST data. That's possible because the varlena is a mere pointer to the chunks and a new row version will affect only the pointer leaving the TOAST data unchanged.
The TOAST table are stored like all the other relation's in the pg_class table, the associated table can be found using a self join on the field reltoastrelid.

Because the TOAST usurps two bits in the varlena length word, this limits the maximum allocated size for the datum to 1GB ($2^{30} - 1 bytes$) .

6.5 Tablespaces

PostgreSQL implements the tablespaces creating the symbolic links, pointing the tablespace's location, into the pg_tblspc. The links are named like the tablespace's OID. The tablespaces are available only on systems supporting the symbolic links.

Since PostgreSQL 9.1 the tablespace location was stored into the field spclocation of the pg_tablespace system table. The information was used only to dump the tablespace's definition during a pg_dump and was removed in the version 9.2 and it was was introduced the function pg_tablespace_location(tablespace_oid) to get the tablespace's absolute path from the tablespace oid.

Querying the system catalogue to acquire any sort of informations is quite simple. In this example the query returns the tablespace's location seen in 5.6

```
 postgres=#
 SELECT
        pg_tablespace_location(oid),
        spcname
FROM
        pg_tablespace;
       pg_tablespace_location        |  spcname
-------------------------------------+------------
                                     | pg_default
                                     | pg_global
 /var/lib/postgresql/pg_tbs/ts_test  | ts_test
(3 rows)
```

The function returns the empty string for the system tablespaces, pg_default and pg_global, because those locations have an immutable location, relative to the data directory. We can get the data area's absolute path using the function current_settings.

```
 postgres=# SELECT current_setting('data_directory');
       current_setting
```

```
-------------------------------
 /var/lib/postgresql/9.3/main
(1 row)
```

Using the CASE construct is then possible to build up an more complete query to lookout at the tablespaces locations.

```
 postgres=#
SELECT
        CASE
                WHEN
                                pg_tablespace_location(oid)=''
                        AND     spcname='pg_default'
                THEN
                        current_setting('data_directory')||'/base/'
                WHEN
                                pg_tablespace_location(oid)=''
                        AND     spcname='pg_global'
                THEN
                        current_setting('data_directory')||'/global/'
        ELSE
                pg_tablespace_location(oid)
        END
        AS      spclocation,

        spcname
FROM
        pg_tablespace;
             spclocation              |  spcname
--------------------------------------+------------
 /var/lib/postgresql/9.3/main/base/   | pg_default
 /var/lib/postgresql/9.3/main/global/ | pg_global
 /var/lib/postgresql/pg_tbs/ts_test   | ts_test
(3 rows)
```

Before the version 8.4 the tablespace location pointed directly to the referenced directory, causing a potential location's clash. The newer versions introduced the usage of a container directory into the tablespace location named after the major version and the system catalogue version number.

```
postgres@tardis:~$ ls -l /var/lib/postgresql/pg_tbs/ts_test
total 0
drwx------ 2 postgres postgres 6 Jun  9 13:01 PG_9.3_201306121
```

The tablespace's directory container is structured this way
PG_{MAJOR_VERSION}_{CATALOGUE_VERSION_NUMBER}

The major version is the PostgreSQL version truncated to the second cypher and the catalogue's version number is the same shown in the pg_controldata output, a formatted date.

```
postgres@tardis:~$ export PGDATA=/var/lib/postgresql/9.3/main
```

```
postgres@tardis:~$ /usr/lib/postgresql/9.3/bin/pg_controldata
pg_control version number:            937
Catalog version number:               201306121
Database system identifier:           5992975355079285751
Database cluster state:               in production
pg_control last modified:             Mon 09 Jun 2014 13:05:14 UTC
.
.
.
WAL block size:                       8192
Bytes per WAL segment:                16777216
Maximum length of identifiers:        64
Maximum columns in an index:          32
Maximum size of a TOAST chunk:        1996
Date/time type storage:               64-bit integers
Float4 argument passing:              by value
Float8 argument passing:              by value
Data page checksum version:           0
```

Inside the container directory the structure is same as the directory base seen in 4.6, with one difference. There're only the subdirectories for the databases having relations on the tablespace.

To get all the databases with objects on the current tablespace it's present the function pg_tablespace_databases(tablespace_oid) which returns the set of the database OID with objects on the specified tablespace. In order to have a better picture we can join the query with the pg_database system table.

Here's an example query using the CASE construct with the pg_tablespace_databases function, to get all the databases with objects on the ts_test tablespace.

```
 db_test=#
 SELECT
        datname,
        spcname,
        CASE
                WHEN
                                pg_tablespace_location(tbsoid)=''
                        AND     spcname='pg_default'
                THEN
                        current_setting('data_directory')||'/base/'
                WHEN
                                pg_tablespace_location(tbsoid)=''
                        AND     spcname='pg_global'
                THEN
                        current_setting('data_directory')||'/global/'
        ELSE
                pg_tablespace_location(tbsoid)
        END
        AS      spclocation
FROM
        pg_database dat,
        (
                SELECT
                        oid as tbsoid,
                        pg_tablespace_databases(oid) as datoid,
                        spcname
                FROM
                        pg_tablespace where spcname='ts_test'
        ) tbs
WHERE
        dat.oid=tbs.datoid
;
 datname | spcname |              spclocation
---------+---------+-------------------------------------
 db_test | ts_test | /var/lib/postgresql/pg_tbs/ts_test
(1 row)
```

Moving a tablespace to another physical location it's not complicated; the cluster of course has to be shut down; see 4.3 for more informations about the shutdown sequence.

When the cluster is stopped the container directory can be copied to the new location; the directory's access permissions must be the same as the origin; read write for the os user running the postgresql process only, otherwise the cluster will refuse to start.

When the copy is complete and the symbolic link in $PGDATA/pg_tblspc is fixed to point the new location the cluster can be started as shown in 4.2.

6.6 MVCC

The multiversion concurrency control is the access method used by PostgreSQL to provide the transactional model as seen in 5.7.

At logical level this is completely transparent to the user and the new row versions become visible after the commit, accordingly with the transaction isolation level.

At physical level we have for each new row version, the insert's XID stamped into the t_xmin field. The PostgreSQL's internal semantic makes visible only the committed rows stamped with the XID lesser than the current transaction's XID because considered *in the past*. The rows with a XID greater than the current transaction's XID are considered *in the future* and then invisible.

Because the XID is a 32 bit quantity, it wraps at 4 billions. When this happens theoretically all the tuples should suddenly disappear because they switch from in the XID's past to its future. This is the XID wraparound failure, a serious problem for the older PostgreSQL versions, which only fix was to re init a new data area each 4 billion transactions and dump reload the databases.

PostgreSQL 7.2 introduced the $modulo - 2^{32}$ arithmetic for evaulating the XID age where a special XID, the FrozenXID[2] was assumed as always in the past and having, for any given XID 2 billion transactions in the future and 2 billion transactions in the past.

When the age of the stamped t_xmin becomes old then the VACUUM can freeze the tuple stamping the FrozenXID and preserving it from the disappearance. The pg_class and the pg_database table have a dedicated field to track the oldest tuple inside the relation and the database, respectively the relfrozenxid and the datfrozenxid where the oldest not frozen XID's value is stored. The builtin function age() shows how many transactions are between the current XID and the value stored in the system catalogue.

For example this is a query to get all the databases with the datfrozenxid and the age.

[2]The FrozenXID's value is 2. The docs of PostgreSQL 7.2 also mention the BootstrapXID with value 1

```
postgres=#
        SELECT
                datname,
                age(datfrozenxid),
                datfrozenxid
        FROM
                pg_database;
   datname     | age  | datfrozenxid
---------------+------+--------------
 template1     | 4211 |          679
 template0     | 4211 |          679
 postgres      | 4211 |          679
 db_test       | 4211 |          679
```

The datfroxenxid value is meaningful only through the age function which shows the "distance" between the current XID and the datfroxenxid. PostgreSQL assigns the new XID only for the write transactions and only if the tuples are updated in the so called "lazy XID assignment".

When a tuple's XID becomes older than 2 billion transactions, the tuple simply disappears jumping from the the current XID's past to its future. Before the version 8.0 there was no prevention for this problem, except the periodic cluster wide VACUUM. The latest versions introduced a passive protection mechanism emitting messages in the activity log when the age of datfrozenxid is ten million transactions from the wraparound point.

```
WARNING:  database "test_db" must be vacuumed within 152405486 transactions
HINT:  To avoid a database shutdown, execute a database-wide VACUUM in
"test_db".
```

Another active protection is the autovacuum daemon which take care of the affected tables and starts a VACUUM to freeze the tuples even if autovacuum is turned off. However if something goes wrong and the datfrozenxid reaches one million transactions from the wraparound point, the cluster shutdown and keeps shutting down for each transaction. When this happens the cluster can be only started in single-user backend to execute the VACUUM.

To limit the effect of data bloat, unavoidable with this implementation, PostgreSQL have the feature called HOT which stands for Heap Only Tuples. The RDBMS tries to keep the updated tuples inside the same page avoiding also any index reference update, if present. This is possible only if there's available free space. By default PostgreSQL when inserting the tuples, fills up the pages completely; however is possible to reserve a page portion for the updates with the fillfactor storage parameter. This is the percentage of the page to reserve for the inserts. The default value for the heap pages is 100, complete packing. For the indices is 70 for the not leaf pages and 90 for the leaf pages leaving some space available for the unavoidable updates. A smaller fill factor will result, at insert time, with a bigger table but with lesser grow rate when updated.

Finally if the MVCC is not carefully considered at design time, this can result in data bloat and generally poor performances. In the 7 we'll see how and how to keep the cluster in efficient conditions or at least how to try.

Chapter 7

Maintenance

The database maintenance is something crucial for the efficiency of the data access, the integrity and the reliability. Any database sooner or later will need a proper maintenance plan.

When a new tuple's version is generated by an update it can be put everywhere there's free space. Frequent updates can result in tuples moving across the data pages many and many times leaving a trail of dead tuples behind them. Because the dead tuples are physically stored but no longer visible this creates an extra overhead causing the table to bloat. Indices makes things more complicated because when a tuple changes page the index entry is updated to point the new page and because of the index's ordered structure, the bloating is more probable than the table.

7.1 vacuum

VACUUM is a PostgreSQL specific command which reclaims back the dead tuple's space. When called without specifying a target table, the command processes all the tables in the database. Running regulary VACUUM have some beneficial effects.

- It reclaims back the dead tuple's disk space.
- It updates the visibility map making the index scans run faster.
- It freezes the tuples with old XID protecting from the XID wraparound data loss

The optional ANALYZE clause also gather the statistics on processed table, more details here 7.2.

A standard VACUUM's run, frees the space used by the dead rows inside the data files but doesn't returns the space to the operating system. VACUUM doesn't affects the common database activity but prevents any schema change on the processed table. Because the pages are rewritten, a VACUUM run increases substantially the I/O activity.

The presence of one or more empty pages in the table's end can be removed by VACUUM if an exclusive lock on the relation can be obtained immediately. When this happens the table is scanned backward to find all the empty pages and then it's truncated to the first not empty page. The index pages are scanned as well and the dead tuples are also cleared. The VACUUM's truncate scan works only on the heap data files. VACUUM'S performances are influenced by the maintenance_work_mem only if the table have indices, otherwise the VACUUM will run the cleanup sequentially without storing the tuple's references for the index cleanup.

To show the effect of the maintenance_work_mem let's build build a simple table with 10 million rows.

```
postgres=# CREATE TABLE t_vacuum
        (
                i_id serial,
                t_ts_value timestamp with time zone DEFAULT
                    clock_timestamp(),
                t_value text,
                CONSTRAINT pk_t_vacuum PRIMARY KEY  (i_id)
        )
;
CREATE TABLE

postgres=# INSERT INTO t_vacuum
        (t_value)
SELECT
         md5(i_cnt::text)
FROM
(
        SELECT
                generate_series(1,10000000) as i_cnt
) t_cnt
;
INSERT 0 10000000
```

To have a statical environment we'll disable the table's autovacuum and the analy. More infos on autovacuum here 7.5. We'll also increase the session's verbosity to look out what's happening during the VACUUM's run.

```
postgres=# ALTER TABLE t_vacuum
```

```
        SET
                (
                        autovacuum_enabled = false,
                        toast.autovacuum_enabled = false
                )
;
ALTER TABLE


SET client_min_messages='debug';
```

We are now executing a complete table rewrite running an UPDATE without the WHERE condition. This will create 10 millions of dead rows.

```
postgres=# UPDATE t_vacuum
        SET
                t_value = md5(clock_timestamp()::text)
;
UPDATE 10000000
```

Before running the VACUUM we'll change the maintenance_work_mem to a small value enabling the the timing to check the query duration.

```
postgres=# SET maintenance_work_mem ='20MB';
SET
postgres=# \timing
Timing is on.

postgres=# VACUUM t_vacuum;
DEBUG:  vacuuming "public.t_vacuum"
DEBUG:  scanned index "pk_t_vacuum" to remove 3495007 row versions
DETAIL:  CPU 0.80s/4.56u sec elapsed 21.36 sec.
DEBUG:  "t_vacuum": removed 3495007 row versions in 36031 pages
DETAIL:  CPU 0.63s/0.56u sec elapsed 19.31 sec.
DEBUG:  scanned index "pk_t_vacuum" to remove 3495007 row versions
DETAIL:  CPU 0.67s/4.18u sec elapsed 15.28 sec.
DEBUG:  "t_vacuum": removed 3495007 row versions in 36031 pages
DETAIL:  CPU 0.67s/0.53u sec elapsed 18.07 sec.
DEBUG:  scanned index "pk_t_vacuum" to remove 3009986 row versions
DETAIL:  CPU 0.53s/2.86u sec elapsed 12.29 sec.
DEBUG:  "t_vacuum": removed 3009986 row versions in 31031 pages
DETAIL:  CPU 0.47s/0.52u sec elapsed 20.06 sec.
DEBUG:  index "pk_t_vacuum" now contains 10000000 row versions in
    82352 pages
DETAIL:  10000000 index row versions were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
DEBUG:  "t_vacuum": found 10000000 removable, 10000000 nonremovable
    row versions in 206186 out of
```

```
206186 pages
DETAIL:  0 dead row versions cannot be removed yet.
There were 0 unused item pointers.
0 pages are entirely empty.
CPU 5.92s/17.08u sec elapsed 154.10 sec.
DEBUG:  vacuuming "pg_toast.pg_toast_28499"
DEBUG:  index "pg_toast_28499_index" now contains 0 row versions in 1
    pages
DETAIL:  0 index row versions were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
DEBUG:  "pg_toast_28499": found 0 removable, 0 nonremovable row
    versions in 0 out of 0 pages
DETAIL:  0 dead row versions cannot be removed yet.
There were 0 unused item pointers.
0 pages are entirely empty.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
VACUUM
Time: 154143.383 ms
postgres=#
```

During the VACUUM the the maintenance_work_mem is used to store an array of TCID referencing the dead tuples for the index cleanup. If the maintenance_work_mem is small and the dead rows are many, the memory fills up often. When this happens the table scan pauses and the index is scanned searching for the tuples stored into the array. When the index scan is complete the array of TCID is emptied and the table's scan resumes. Increasing the maintenance_work_mem to 2 GB[1] the index scan is executed in one single run resulting in a VACUUM 32 seconds faster.

```
postgres=# SET maintenance_work_mem ='2GB';
SET

postgres=# VACUUM t_vacuum;
DEBUG:  vacuuming "public.t_vacuum"
DEBUG:  scanned index "pk_t_vacuum" to remove 10000000 row versions
DETAIL:  CPU 1.58s/8.45u sec elapsed 52.41 sec.
DEBUG:  "t_vacuum": removed 10000000 row versions in 103093 pages
DETAIL:  CPU 1.78s/1.41u sec elapsed 33.90 sec.
DEBUG:  index "pk_t_vacuum" now contains 10000000 row versions in
    82352 pages
DETAIL:  10000000 index row versions were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
DEBUG:  "t_vacuum": found 10000000 removable, 10000000 nonremovable
    row versions in 206186 out of
206186 pages
```

[1] In order to have the table in the same conditions, a VACUUM FULL and a new update has been runt before the conventional VACUUM.

```
DETAIL:  0 dead row versions cannot be removed yet.
There were 0 unused item pointers.
0 pages are entirely empty.
CPU 5.62s/13.64u sec elapsed 121.99 sec.
DEBUG:  vacuuming "pg_toast.pg_toast_28499"
DEBUG:  index "pg_toast_28499_index" now contains 0 row versions in 1
    pages
DETAIL:  0 index row versions were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
DEBUG:  "pg_toast_28499": found 0 removable, 0 nonremovable row
    versions in 0 out of 0 pages
DETAIL:  0 dead row versions cannot be removed yet.
There were 0 unused item pointers.
0 pages are entirely empty.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
VACUUM
Time: 122021.251 ms
```

A table without indices does not use the maintenance_work_mem. For example if we run the VACUUM after dropping the table's primary key the execution is faster even with the low maintenance_work_mem setting.

```

postgres=# SET maintenance_work_mem ='20MB';
SET
postgres=# \timing
Timing is on.

postgres=# ALTER TABLE t_vacuum DROP CONSTRAINT pk_t_vacuum;
DEBUG:  drop auto-cascades to index pk_t_vacuum
ALTER TABLE
Time: 182.737 ms

postgres=# VACUUM t_vacuum;
DEBUG:  vacuuming "public.t_vacuum"
DEBUG:  "t_vacuum": removed 10000000 row versions in 103093 pages
DEBUG:  "t_vacuum": found 10000000 removable, 10000000 nonremovable
    row versions in 206186 out of
206186 pages
DETAIL:  0 dead row versions cannot be removed yet.
There were 0 unused item pointers.
0 pages are entirely empty.
CPU 2.16s/4.53u sec elapsed 47.30 sec.
DEBUG:  vacuuming "pg_toast.pg_toast_28499"
DEBUG:  index "pg_toast_28499_index" now contains 0 row versions in 1
    pages
DETAIL:  0 index row versions were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
```

```
DEBUG:  "pg_toast_28499": found 0 removable, 0 nonremovable row
    versions in 0 out of 0 pages
DETAIL:  0 dead row versions cannot be removed yet.
There were 0 unused item pointers.
0 pages are entirely empty.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
VACUUM
Time: 48823.132 ms
```

The table seen in the example begins with a size of 806 MB . After the update the table double its size which remains the same during the VACUUM runs the updates. This happens because after the first insert the table had all the rows packed together; the update added in the table's bottom the new row versions leaving the previous 10 millions row on the table's top as dead tuples. The VACUUM's run cleared the space on the table's top but weren't able to truncate because all the rows packed in the table's bottom. Running a new UPDATE followed by VACUUM would free the space in the table's bottom and a truncate scan would succeed but only if there's no tuple in the table's end free space. To check if the vacuum is running effectively the tables should show an initial growt followed by a substantial size stability in time. This happens only if the new rows are versioned at the same rate of the old rows clear down.

The XID wraparound failure protection is performed automatically by VACUUM which when it finds a live tuple with a t_xmin's age bigger than the GUC parameter vacuum_freeze_min_age, then it replaces the tuple's creation XID with the FrozenXID preserving the tuple's visibility forever. Because VACUUM by default skips the pages without dead tuples it will miss some aging tuples. That's the reason why it's present a second GUC parameter, vacuum_freeze_table_age, which triggers a VACUUM's full table scan when the table's relfrozenxid age exceeds the value.

VACUUM accepts the FREEZE clause which forces a complete tuple freeze regardless to the age. That's equivalent to run the VACUUM setting the vacuum_freeze_min_age to zero.

7.2 analyze

The PostgreSQL's query optimiser is based on the costs estimates. When building the execution plans the planner consults the internal statistics and assigns to each node plan an estimated cost. The plan with the smallest total estimated cost is then executed. Having up to date and accurate statistics will help the database to keep up the performances.

The ANALYZE command is used to gather the usage statistics. When launched reads the data, builds up the statistics and stores them into the pg_statistics system table. The command accepts the optional clause VERBOSE to increase verbosity and the optional target table and the column list. If ANALYZE is launched with no parameters it processes all the tables in the database. Launching ANALYZE with the table name only, will process all the table's columns.

When working on large tables ANALYZE runs a sample random read on a table's portion. The GUC parameter default_statistics_target determines the amount of entries read by the sample. The default limit is 100. Increasing the value will cause the planner to get better estimates. in particular for columns having data distributed irregularly. This accuracy costs more time for the statistics gathering and space because requires a bigger storage in the pg_statistics table.

To show how the default_statistics_target can affects the estimates, let's run an ANALYZE VERBOSE with the default setting on the table created in 7.1.

```
postgres=# SET default_statistics_target =100;
SET
postgres=# ANALYZE VERBOSE t_vacuum;
INFO:  analyzing "public.t_vacuum"
INFO:  "t_vacuum": scanned 30000 of 103093 pages, containing 2909979
    live rows and 0 dead rows;
30000 rows in sample, 9999985 estimated total rows
ANALYZE
```

Even if the table have 10 million rows, the analyse estimates only 2,909,979 rows, the 30% of the total effective storage.

Changing the default_statistics_target to its maximum value of 10000 ANALYZE will get better estimates.

```
SET
postgres=# ANALYZE VERBOSE t_vacuum;
INFO:  analyzing "public.t_vacuum"
INFO:  "t_vacuum": scanned 103093 of 103093 pages, containing 10000000
     live rows and 0 dead rows;
3000000 rows in sample, 10000000 estimated total rows
ANALYZE
```

This time the table estimate is correctly determined in 10 millions live rows.
The pg_statistics stores the gathered data for the database usage only. To help users and administrators it's present the view pg_stats providing a human readable visualization of the gathered statistics.

As general rule, before starting any performance tuning, it's important to check if database statistics are recent and accurate. The information is stored into the view pg_stat_all_tables [2].

For example this query gets the last execution of the manual and the auto vacuum with the analyze and auto analyze, for a given table.

```
postgres=# \x
Expanded display is on.
postgres=# SELECT
        schemaname,
        relname,
        last_vacuum,
        last_autovacuum,
        last_analyze,
        last_autoanalyze
FROM
         pg_stat_all_tables
WHERE
        relname='t_vacuum'
;
-[ RECORD 1 ]----+------------------------------
schemaname       | public
relname          | t_vacuum
last_vacuum      |
last_autovacuum  |
last_analyze     | 2014-06-17 18:48:56.359709+00
last_autoanalyze |

postgres=#
```

The statistics target can be set per column to fine tune the ANALYZE with the ALTER TABLE SET STATISTICS statement.

```


--SET THE STATISTICS TO 1000 ON THE COLUMN i_id
ALTER TABLE t_vacuum
        ALTER COLUMN  i_id
                        SET STATISTICS 1000
;
```

The SET can be used, as seen before, to change the default statistics target for the current session. Otherwise is possible to change the value cluster wide changing the parameter in the postgresql.conf file.

[2]The subset views pg_stat_user_tables and pg_stat_sys_tables are useful to search respectively the current user and the system tables only.

7.3 reindex

7.4 VACUUM FULL and CLUSTER

7.5 The autovacuum daemon

Chapter 8

Backup and restore

8.1 Saving with pg_dump

8.2 Restoring with pg_restore

Chapter 9

A couple of things to know before start coding...

Appendix A

License

List of Figures

List of Tables

Index