EnterpriseDB
The Enterprise Postgres Company

PostgreSQL Database Administration

1. Introduction and Overview of PostgreSQL

2. PostgreSQL System Architecture

- Architectural Summary
- Process Architecture
- Database Clusters
- PostgreSQL Terminology
- Physical Database Architecture
- Data File Architecture

3. Installation

- Installation using
  - PostgreSQL Binary Windows
  - Installation from Source
- Creating a database cluster
- Starting and Stopping the Server (Windows)
- Starting and Stopping the Server (Other)
- Lab Exercise - Installation
- Install PostgreSQL from source
- Create a database cluster
- Start the database server
- Connect to the server using psql
- Stop the database server

4. Configuration

- Setting PostgreSQL Parameters
- Access Control
- Connection Settings
- Security and Authentication Settings
- Memory Settings
- Free Space Settings
- Kernel Resource Settings
- Log Management
- Background Writer Settings
- Vacuum Cost Settings
- Autovacuum Settings

5. Security

- Levels of security: pg_hba.conf, schemas and users and table level.
- pg_hba.conf

6. Create and Managing PostgreSQL Database.

- USERS
- GROUPS
- ROLES
- Object Ownership
- Accesss control
- Application Access
- Schemas and Search Paths
- Lab to create USERS, SCHEMAS, ROLES, Alter SEARCH_PATH, GRANT and REVOKE privileges

7. Moving Data with PostgreSQL

- COPY

8. Backup & Recovery

- pg_dump
- pg_restore
- pg_dumpall
- File System Backup - copying DATA folder
- pg_start_backup and pg_stop_backup
- PITR - Point in Time Recovery
    - Creating a base backup
    - Archive command
    - Recovery file parameters
    - Lab to test PITR

9. Routine Database Maintenance Tasks

- Routine Vacuuming
- Recovering Disk Space
- Vacuum Full
- Updating Planner Statistics
- Preventing Transaction ID Wraparound Failures
- Routine Reindexing

10. Performance Tuning

- Explain and Explain Analyze
- Forcing Query Plan Selection
- Indexing
- Log Management
- Clustering Rows

11. PostgreSQL Partitioning and Table spaces

- Partition Methods
- Partition Setup
- Partition Table Explain Plan
- Table space Management

12. High Availability & Replication

- Warm Standby
- Why Use Replication
- Replication
- Replication Limitations
- Slonik
- Replication Configuration

13. Connection Pooling

- Why Connection Pooling
- Pgpool-II setup
- Pgbouncer setup

14. Advance Monitoring Techniques

- Importance of Monitoring
- PGFouine – Log Analyzer

1. Introduction and Overview of PostgreSQL

PostgreSQL is a powerful, open source relational database system. It has more than 15 years of active development and a proven architecture that has earned it a strong reputation for reliability, data integrity, and correctness. It runs on all major operating systems, including Linux, UNIX (AIX,BSD, HP-UX, SGI IRIX, Mac OS X, Solaris, Tru64), and Windows. It is fully ACID compliant, has full support for foreign keys, joins, views, triggers, and stored procedures (in multiple languages). It includes most SQL92 and SQL99 data types, including INTEGER, NUMERIC, BOOLEAN, CHAR, VARCHAR, ATE, INTERVAL, and TIMESTAMP. It also supports storage of binary large objects, including pictures, sounds, or video. It has native programming interfaces for C/C++, Java, Perl, Python, Ruby, Tcl, ODBC among others.

An enterprise class database, PostgreSQL boasts sophisticated features such as Multi-Version Concurrency Control (MVCC), point in time recovery, tablespaces, and asynchronous replication, nested transactions (savepoints), online/hot backups, a sophisticated query planner/optimizer, and writes ahead logging for fault tolerance. It supports international character sets, multibyte character encodings, Unicode, and it is locale-aware for sorting, case-sensitivity, and formatting. It is highly scalable both in the sheer quantity of data it can manage and in the number of concurrent users it can accommodate. There are active PostgreSQL systems in production environments that manage in excess of 4 terabytes of data. Some general PostgreSQL limits are included in the table below.

Limit Value

Maximum Database Size -- Unlimited
Maximum Table Size -- 32 TB
Maximum Row Size -- 1.6 TB
Maximum Field Size -- 1 GB
Maximum Rows per Table -- Unlimited
Maximum Columns per Table -- 250 - 1600 depending on column types
Maximum Indexes per Table -- Unlimited

Feature and Standards Compliant

PostgreSQL prides itself in standards compliance. Its SQL implementation strongly conforms to the ANSI-SQL 92/99 standards. It has full support for subqueries (including subselects in the FROM Clause), read-committed and serializable transaction isolation levels. And while PostgreSQL has a fully relational system catalog which itself supports multiple schemas per database, its catalog is also accessible through the Information Schema as defined in the SQL standard.

Data integrity features include (compound) primary keys, foreign keys with restricting and cascading updates/deletes, check constraints, unique constraints, and not null constraints. It also has a host of extensions and advanced features. Among the conveniences are auto increment columns through sequences, and LIMIT/OFFSET allowing the return of partial result sets.

PostgreSQL supports compound, unique, partial, and functional indexes which can use any of its Btree,

R-tree, hash, or GiST storage methods. GiST (Generalized Search Tree) indexing is an advanced system which brings together a wide array of different sorting and searching algorithms including B-tree, B+-tree, R-tree, partial sum trees, ranked B+-trees and many others. It also provides an interface which allows both the creation of custom data types as well as extensible query methods with which to search them. Thus, GiST offers the flexibility to specify what you store, how you store it, and the ability to define new ways tosearch through it --- ways that far exceed those offered by standard B-tree, R-tree and other generalizedsearch algorithms.

Other advanced features include table inheritance, a rules systems, and database events. Table inheritance puts an object oriented slant on table creation, allowing database designers to derive new tables from other tables, treating them as base classes. Even better, PostgreSQL supports both single and multiple inheritance in this manner. The rules system, also called the query rewrite system, allows the database designer to create rules which identify specific operations for a given table or view, and dynamically transform them into alternate operations when they are processed.

The events system is an interprocess communication system in which messages and events can be transmitted between clients using the LISTEN and NOTIFY commands, allowing both simple peer to peer communication and advanced coordination on database events. Since notifications can be issued from triggers and stored procedures, PostgreSQL clients can monitor database events such as table updates, inserts, or deletes as they happen.

Highly Customizable

PostgreSQL runs stored procedures in more than a dozen programming languages, including Java, Perl, Python, Ruby, Tcl, C/C++, and its own PL/pgSQL, which is similar to Oracle's PL/SQL. Included with its standard function library are hundreds of built-in functions that range from basic math and string operations to cryptography and Oracle compatibility. Triggers and stored procedures can be written in C and loaded into the database as a library, allowing great flexibility in extending its capabilities.

Similarly, PostgreSQL includes a framework that allows developers to define and create their own custom data types along with supporting functions and operators that define their behavior. As a result, a host of advanced data types have been created that range from geometric and spatial primitives to network addresses to even ISBN/ISSN (International Standard Book Number/International Standard Serial Number) data types, all of which can be optionally added to the system.

Just as there are many procedure languages supported by PostgreSQL, there are also many library interfaces as well, allowing various languages both compiled and interpreted to interface with PostgreSQL. There are interfaces for Java (JDBC), ODBC, Perl, Python, Ruby, C, C++, PHP, Lisp, Scheme.

Best of all, PostgreSQL's source code is available under the most liberal open source license: the BSD license. This license gives you the freedom to use, modify and distribute PostgreSQL in any form you like, open or closed source. Any modifications, enhancements, or changes you make are yours to do with as you please. As such, PostgreSQL is not only a powerful database system capable of running the enterprise, it is a development platform upon which to develop in-house, web, or commercial software products that require a capable RDBMS.

Cross platform

PostgreSQL is available for almost every brand of Unix (34 platforms with the latest stable release), and Windows compatibility is available via the Cygwin framework. Native Windows compatibility is also available with version 8.0 and above.

Designed for high volume environments

We use a multiple row data storage strategy called MVCC to make PostgreSQL extremely responsive in high volume environments. The leading proprietary database vendor uses this technology as well, for the same reasons.

GUI database design and administration tools

There are many high-quality GUI Tools available for PostgreSQL from both open source developers and commercial providors.

Technical Features

- Fully ACID compliant.
- ANSI SQL compliant.
- Referential Integrity.
- Replication (non-commercial and commercial solutions) allowing the duplication of the master database to multiple slave machines.
- Native interfaces for ODBC, JDBC, C, C++, PHP, Perl, TCL, ECPG, Python, and Ruby.
- Rules.
- Views.
- Triggers.
- Unicode.
- Sequences.
- Inheritance.
- Outer Joins.
- Sub-selects.
- An open API.
- Stored Procedures.
- Native SSL support.
- Procedural languages.
- Hot stand-by (commercial solutions).
- Better than row-level locking.
- Functional and Partial indexes.
- Native Kerberos authentication.
- Support for UNION, UNION ALL and EXCEPT queries.
- Loadable extensions offering SHA1, MD5, XML, and other functionality.
- Tools for generating portable SQL to share with other SQL-compliant systems.
- Extensible data type system providing for custom, user-defined datatypes and rapid development of new datatypes.
- Cross-database compatibility functions for easing the transition from other, less SQLcompliant RDBMS.

2. PostgreSQL System Architecture

Before we begin, you should understand the basic PostgreSQL system architecture. Understanding how the parts of PostgreSQL interact will make the next chapter somewhat clearer. In database jargon, PostgreSQL uses a simple "process per-user" client/server model. A PostgreSQL session consists of the following cooperating Unix processes (programs):

- A supervisory daemon process (the postmaster),
- the user's frontend application (e.g., the psql program), and
- one or more backend database servers (the postgres process itself).

A single postmaster manages a given collection of databases on a single host. Such a collection of databases is called a cluster (of databases). A frontend application that wishes to access a given database within a cluster makes calls to an interface library (e.g., libpq) that is linked into the application. The library sends user requests over the network to the postmaster, which in turn starts a new backend server process.

How a connection is established and connects the frontend process to the new server. From that point on, the frontend process and the backend server communicate without intervention by the postmaster. Hence, the postmaster is always running, waiting for connection requests, whereas frontend and backend processes come and go. The libpq library allows a single frontend to make multiple connections to backend processes. However, each backend process is a single-threaded process that can only execute one query at a time; so the communication over any one frontend-to-backend connection is single-threaded.

One implication of this architecture is that the postmaster and the backend always run on the same machine (the database server), while the frontend application may run anywhere. You should keep this in mind, because the files that can be accessed on a client machine may not be accessible (or may only be accessed using a different path name) on the database server machine.

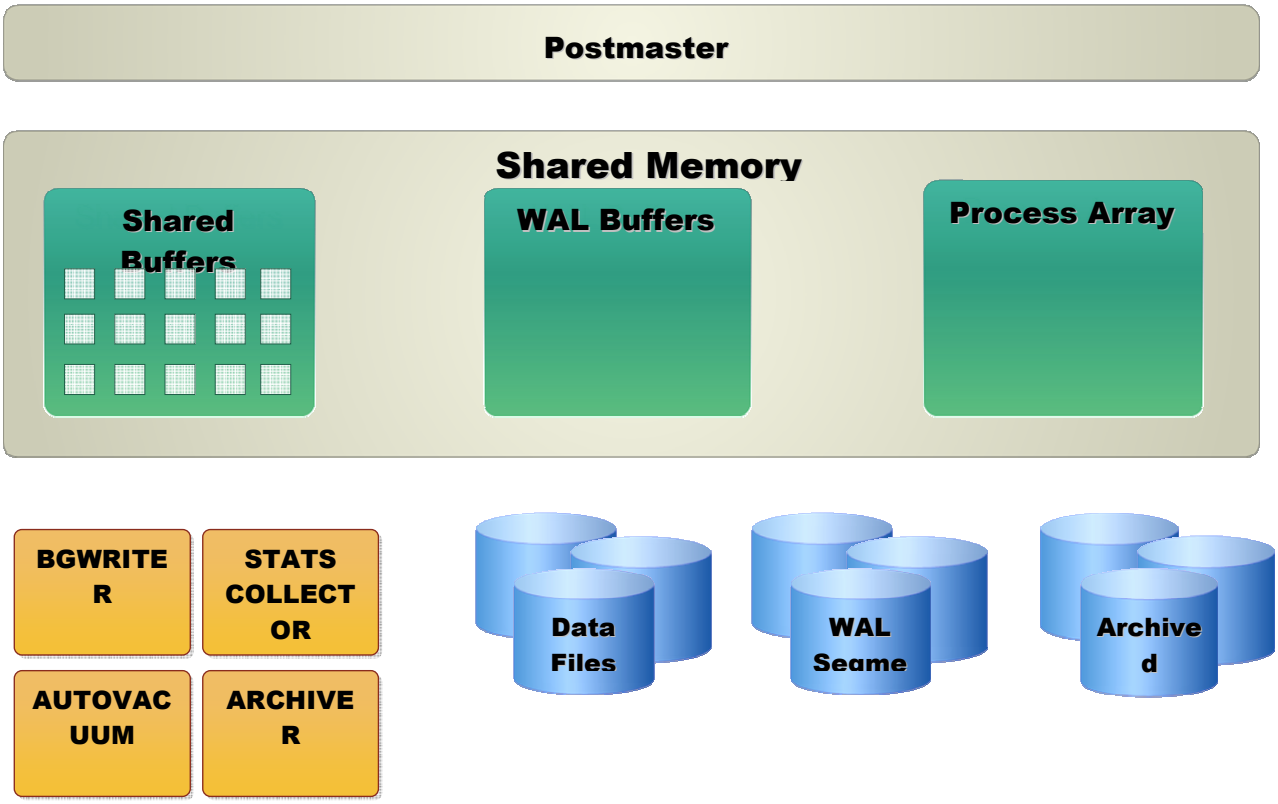
You should also be aware that the postmaster and postgres servers run with the user ID of the PostgreSQL "superuser". Note that the PostgreSQL superuser does not have to be any particular user (e.g., a user named postgres), although many systems are installed that way. Furthermore, the PostgreSQL superuser should definitely not be the Unix superuser, root! It is safest if the PostgreSQL superuser is an ordinary, unprivileged user so far as the surrounding Unix system is concerned. In any case, all files relating to a database should belong to this Postgres superuser.

One postgres process exists for every open database session. Once authenticated with user connection, it directly connects with shared memory. We have,

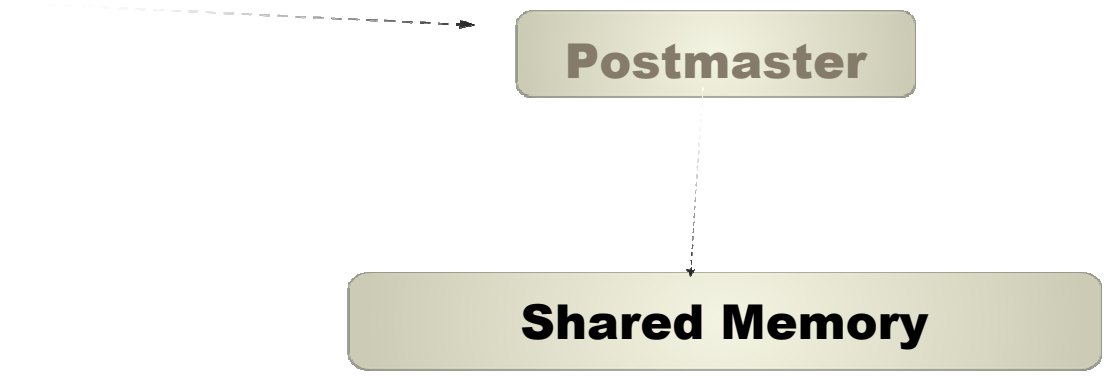
Kernel I/O: It does the work of reading and writing the query.
Bgwriter: It does the work of reading and writing the buffers on data files.
Archiver: In case if the WAL log file is full, it is overwritten .Before overwriting it, take a copy of logfiles this process is known as archiving in Archiver. Data files can be created if one data file is full .

Stats collector: It's a subsystem that supports collection and reporting of information about server activity.Like ,it can count number of access to the tables and indexes in both disk-block and individual row items.We call them as background process or backend process.
Process array: It stores the information of each process like dirty read, clean, whether the process is in use or not.

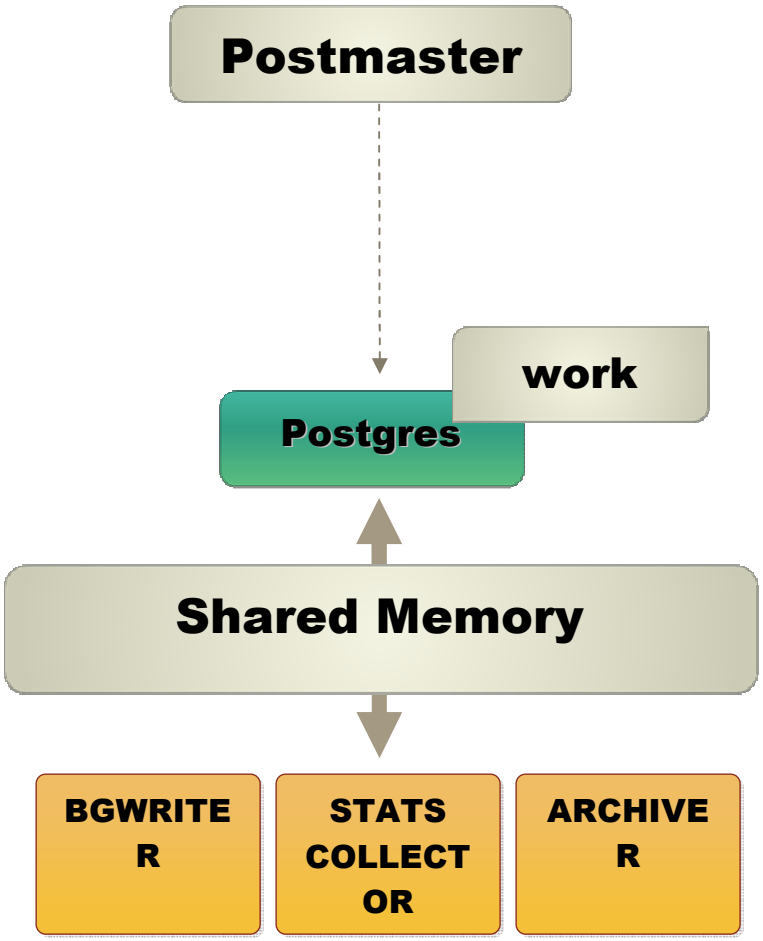
Process Architecture



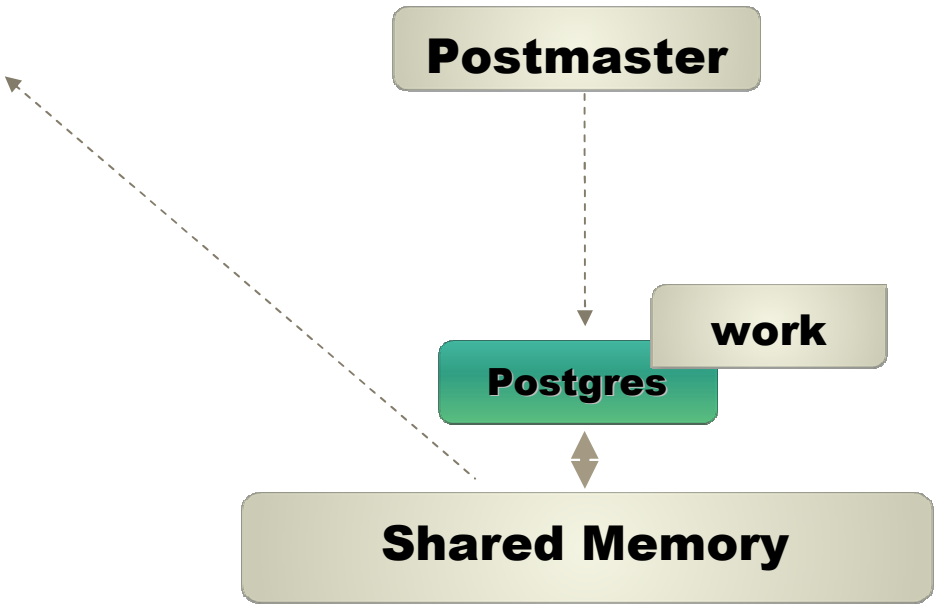
Connection Request
Postmaster listens on 1-and-1 well Known Port 5432 and receives connection request



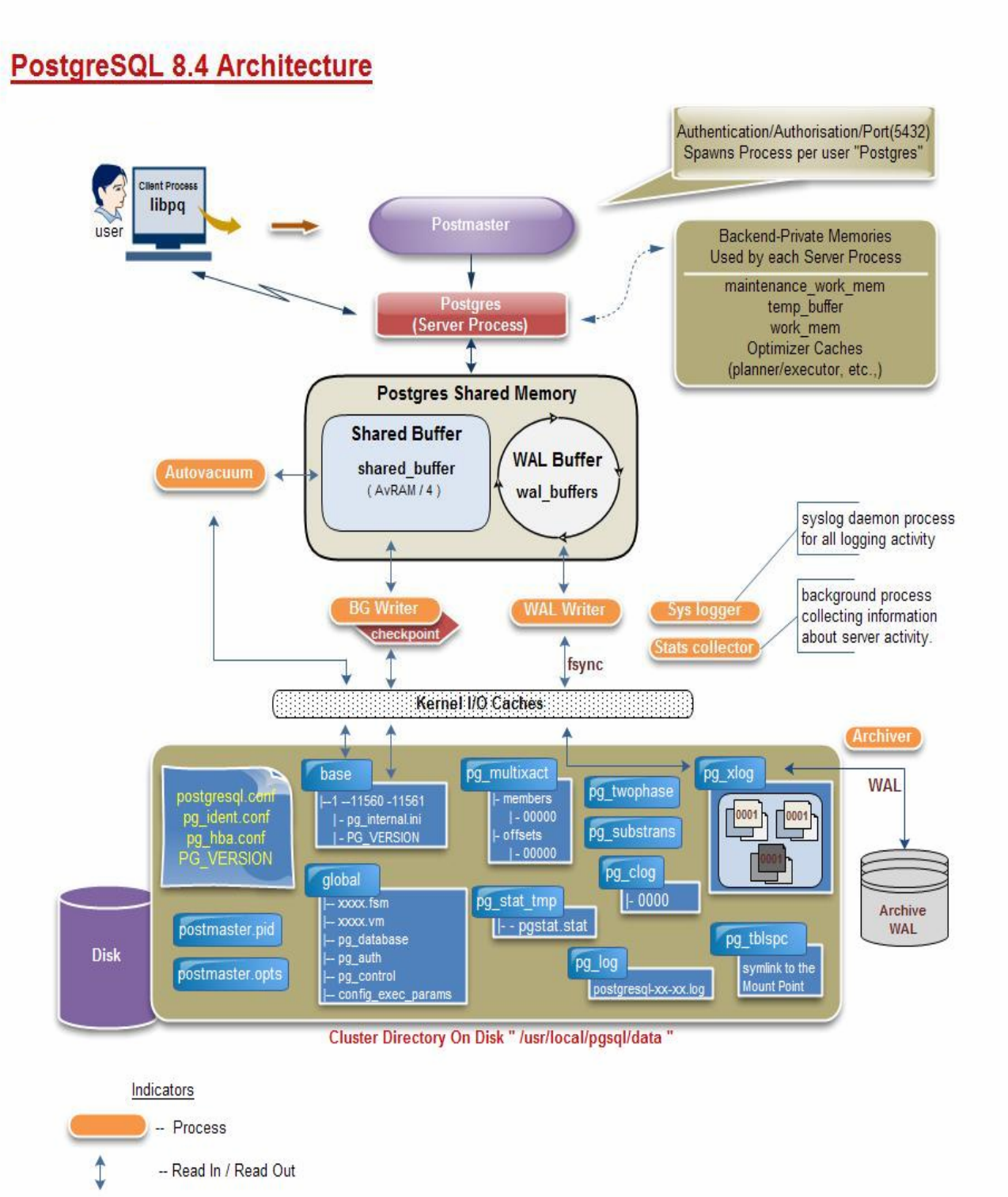
Backend Spawning



Responding to the Client

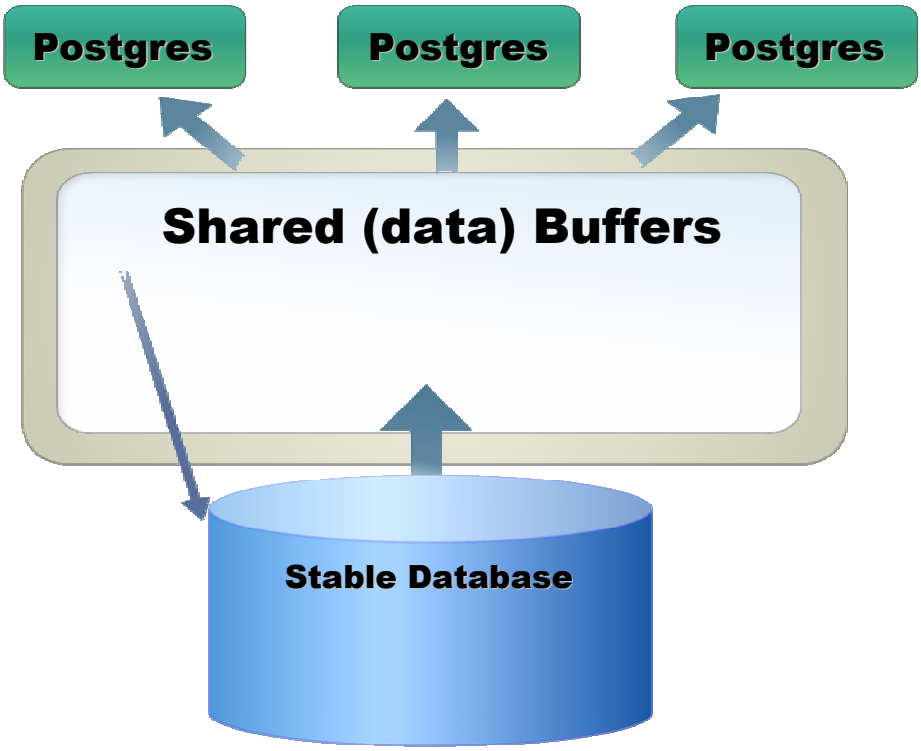


Full Process Architecture



Disk Read Buffers

PostgreSQL does not directly change information on disk. Instead, it requests data to be read into the shared buffer cache. PostgreSQL backbend then read/write these blocks, and finally flush them back to disk. Backends that need to access tables first look for needed blocks in this cache. If they are already there, they can continue processing right away. If not, an operating system request is made to load the blocks. The blocks are loaded either from the kernel disk buffer cache, or from disk. These can be expensive operations. The default PostgreSQL configuration allocates 64 shared buffers. Each buffer is 8 kilobytes. Increasing the number of buffers makes it more likely backends will find the information they need in the cache, thus avoiding an expensive operating system request. The change can be made with a *postmaster* command-line flag or by changing the value of shared_buffers in postgresql.conf. The cache management improved

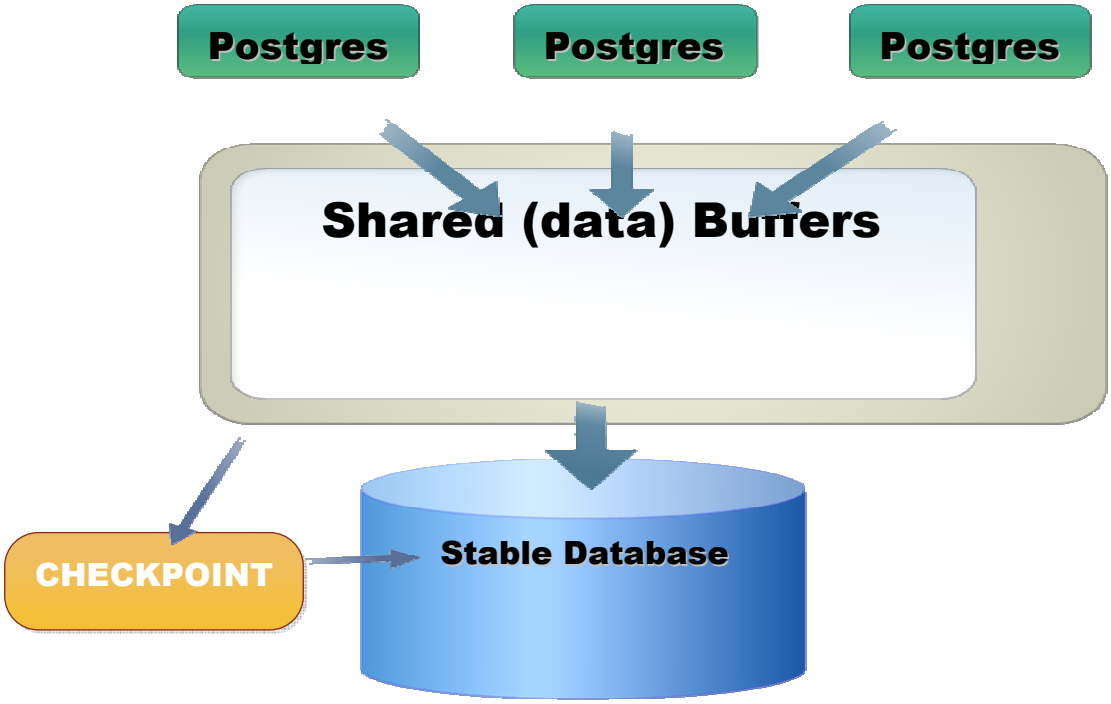


Writing Buffers

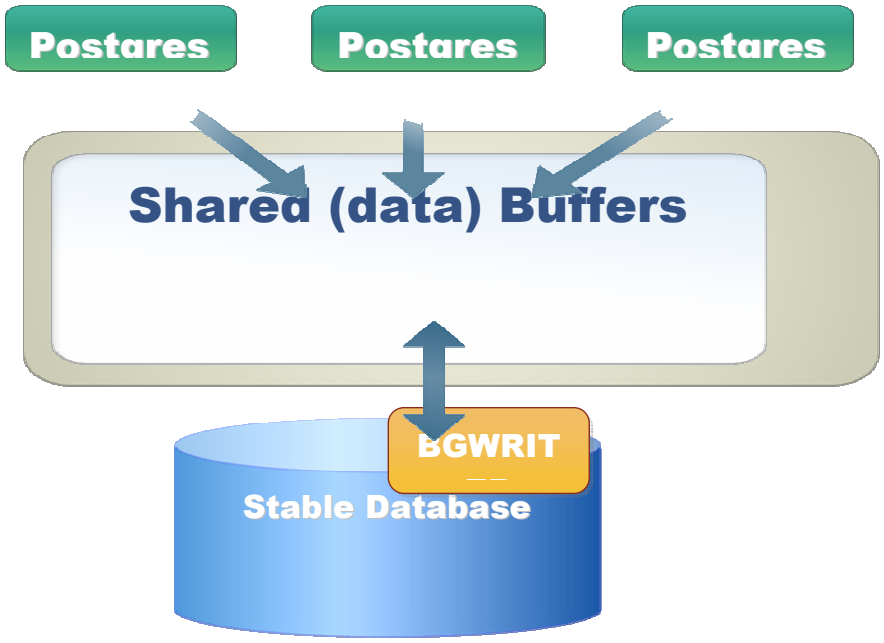
The server's background writer process will automatically perform a checkpoint every so often. At checkpoint time, all dirty data pages are flushed to disk and a special checkpoint record is written to the log file. (The changes were previously flushed to the WAL files.) In the event of a crash, the crash recovery procedure looks at the latest checkpoint record to determine the point in the log (known as the redo record) from which it should start the REDO operation. Any changes made to data files before that point are guaranteed to be already on disk. Hence, after a checkpoint, log segments preceding the one containing the redo record are no longer needed and can be recycled or removed. (When WAL archiving is being done, the log segments must be archived before being recycled or removed.)

There is a separate server process called the background writer, whose sole function is to issue writes of "dirty" shared buffers. The intent is that server processes handling user queries should seldom or never

have to wait for a write to occur, because the background writer will do it. This arrangement also reduces the performance penalty associated with checkpoints.



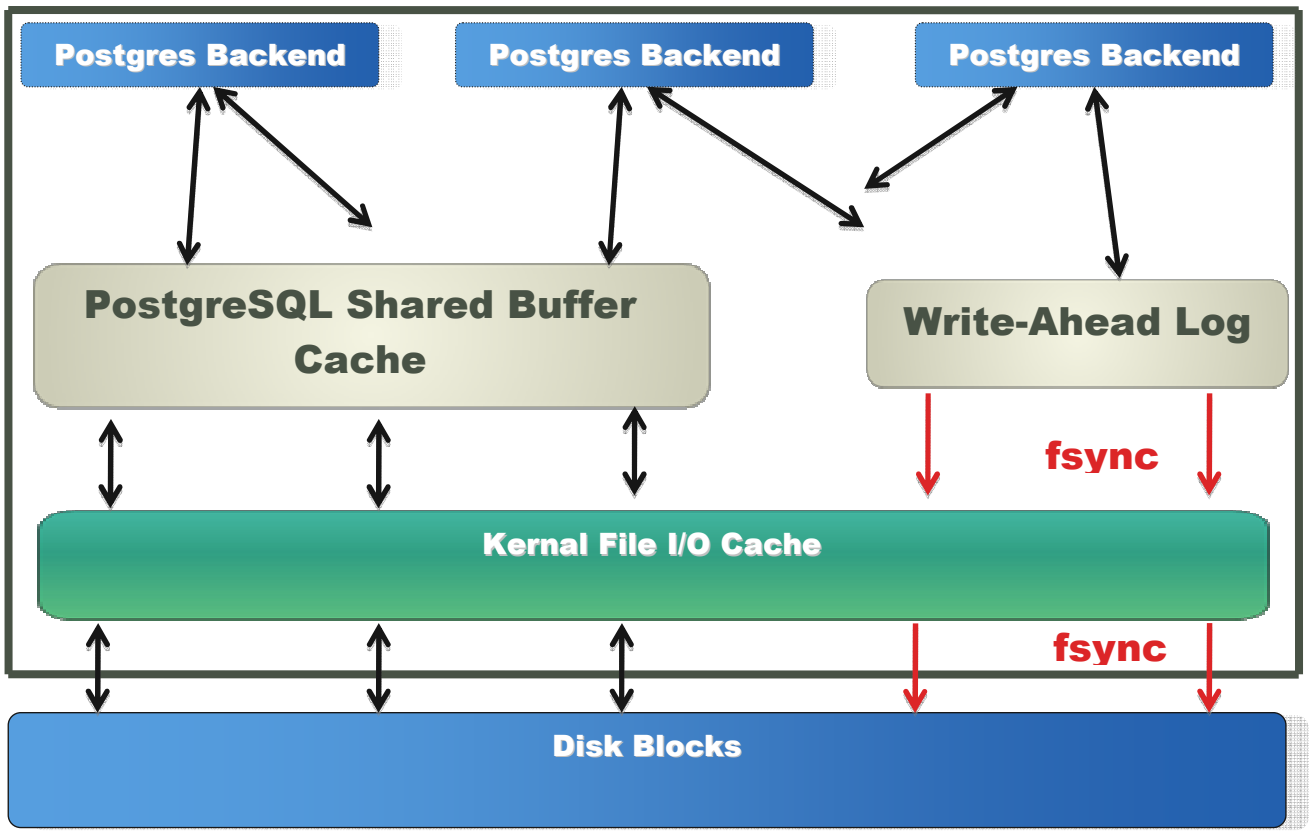
The background writer will continuously trickle out dirty pages to disk, so that only a few pages will need to be forced out when checkpoint time arrives, instead of the storm of dirty-buffer writes that formerly occurred at each checkpoint. However there is a net overall increase in I/O load, because where a repeatedly-dirtied page might before have been written only once per checkpoint interval, the background writer might write it several times in the same interval.
In most situations a continuous low load is preferable to periodic spikes, but the parameters discussed in this subsection can be used to tune the behavior for local needs.



Shared Buffer and Write Ahead Logs

PostgreSQL runs in two modes. Normal fsync mode flushes every completed transaction to disk, guaranteeing that if the OS crashes or loses power in the next few seconds, all your data is safely stored on disk. In this mode, we are slower than most commercial databases, partly because few of them do such conservative flushing to disk in their default modes. In no-fsync mode, we are usually faster than commercial databases, though in this mode, an OS crash could cause data corruption. We are working to provide an intermediate mode that suffers less performance overhead than full fsync mode, and will allow data integrity within 30 seconds of an OS crash.

The aim of WAL, to ensure that the log is written before database records are altered, can be subverted by disk drives that falsely report a successful write to the kernel, when in fact they have only cached the data and not yet stored it on the disk. A power failure in such a situation might still lead to irrecoverable data corruption. Ensure that the disks holding PostgreSQL's WAL log files do not make such false.



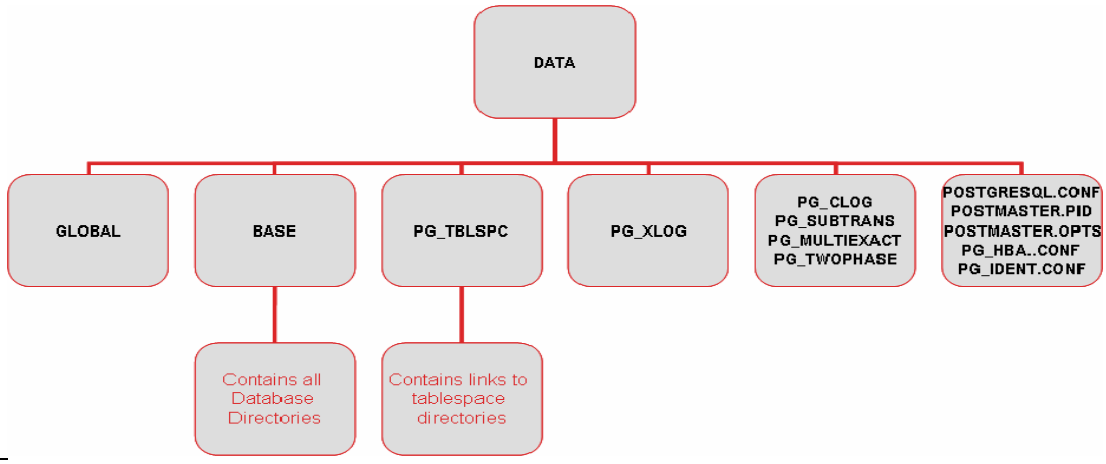
PostgreSQL Physical Database Architecture

The databases are separate from one another usually sharing only the listener process. PostgreSQL has the concept of a database cluster. A database cluster is a collection of databases that is stored at a common file system location (the "data area"). Or A database cluster is a collection of databases that are managed by a single server instance.
It is possible to have multiple database clusters so long as they use different data areas and different communication ports. One postmaster and port per cluster. Accessed from a single “Data Directory”.

Creating a database cluster consists of the following:

- Creating the directories in which the database will live
- Generating the shared catalog tables

The processes along with the file system components are all shared within the database cluster. All the data needed for a database cluster is stored within the cluster's data directory, commonly referred to as PGDATA (after the name of the environment variable that can be used to define it). The PGDATA directory contains several subdirectories and configuration files.



At the top of the directory structure is the cluster directory itself $PGDATA because that is where the $PGDATA environment variable should point.

$PGDATA contains four files and four subdirectories. $PGDATA/pg_hba.conf contains the host-based authentication configuration file. This file tells PostgreSQL how to authenticate clients on a host-by-host basis. The $PGDATA/pg_ident.conf file is used by the ident authentication scheme to map OS usernames into PostgreSQL user names again, $PGDATA/postgresql.conf contains a list of runtime parameters that control various aspects of the PostgreSQL server. The fourth file, $PGDATA/PG_VERSION, is a simple text file that contains the version number from initdb.

The PGDATA directory contains several subdirectories and configuration files. The following are some of the cluster configuration files:

postgresql.conf - Parameter or main server configuration file.
pg_hba.conf - Client authentication configuration file.
pg_ident.conf - Map from OS account to PostgreSQL account file. In Brief, Configures operating system to PostgreSQL authentication name mapping when using ident-based authentication
PG_VERSION - Contains the version number of the installation, for example 8.0
postmaster.opts - Gives the default command-line options to the postmaster program
postmaster.pid - Contains the process ID of the postmaster process and an identification of the main data directory (this file is generally present only when the database is running)

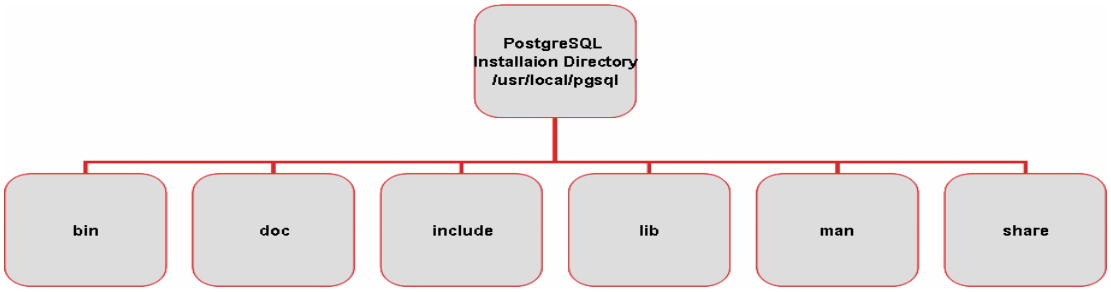
The cluster subdirectories:

- base – Subdirectory containing per-database subdirectories
- global - Subdirectory containing cluster-wide tables
    - pg_auth - Authorization file containing user and role definitions.
    - pg_control - Control file.
    - pg_database - Information of databases within the cluster.
- pg_clog - Subdirectory containing transaction commit status data
- pg_multixact - Subdirectory containing multitransaction status data (used for shared row locks)

- pg_subtrans - Subdirectory containing subtransaction status data
- pg_tblspc - Subdirectory containing symbolic links to tablespaces
- pg_twophase - Subdirectory containing state files for prepared transactions
- pg_xlog - Subdirectory containing WAL (Write Ahead Log) files

By default, for each database in the cluster there is a subdirectory within PGDATA/base, named after the database's OID (object identifier) in pg_database. This subdirectory is the default location for the database's files; in particular, its system catalogs are stored there. Each table and index is stored in a separate file, named after the table or index's filenode number, which can be found in pg_class.relfilenode.

PostgreSQL Installation Directory Structure



On a Windows system, by default, your installation base directory will be something like C:\Program Files\PostgreSQL\8.0.0, under which you will find several subdirectories. On Linux, the base directory for a source code installation will generally be /usr/local/pgsql. For a pre-built binary installation, the location will vary. A common location is /var/lib/pgsql, but you may find that some of the binary files have been put in directories already in the search path, such as /usr/bin, to make accessing them more convenient. Under the PostgreSQL base installation directory, you will normally find around seven subdirectories, depending on your options and operating system:

• bin - Application and utilities such as pg_ctl,postmaster etc.,
• data – it's the main Higher-level of storage directory with configuration files.
• doc – Documentation in html format
• include - Header files for use in developing postgresql application
• lib - Libraries for use in developing postgresql application
• man - Manual pages for postgresql.tools
• share - Sample configuration file

Data Files

The files that PostgreSQL uses fall into two main categories:

Files that are written to while the database server are running, including data files and logs. The data files are the heart of the system, storing all of the information for all of your databases. The log file that the database server produces will contain useful information about database accesses and can be a big help when troubleshooting problems. It effectively just grows as log entries are added.

Files that are not written to while the database server are running, which are effectively read-only files. These files include the PostgreSQL applications like postmaster and pg_ctl, which are installed once and never change.

- Each table and index is stored in a separate file
  - File name is the table or index's filenode number
  - Filenode number found in pg_class.relfilenode

- Segments
  - Tables or indexes exceeding 1 GB are divided into gigabyte-sized segments
  - First segment is named after the filenode
  - Second and subsequent segments are named filenode.1, filenode.2, etc.

Page Layout

The page format used within PostgreSQL tables and indexes. Sequences and TOAST tables are formatted just like a regular table. In the following explanation, a byte is assumed to contain 8 bits. In addition, the term item refers to an individual data value that is stored on a page. In a table, an item is a row; in an index, an item is an index entry.

Every table and index is stored as an array of pages of a fixed size (usually 8 kB, although a different page size can be selected when compiling the server). In a table, all the pages are logically equivalent, so a particular item (row) can be stored in any page. In indexes, the first page is generally reserved as a metapage holding control information, and there may be different types of pages within the index, depending on the index access method.

The overall layout of a page. There are five parts to each page

- Page Header
  - General information about the page
  - Pointers to free space
  - 20 bytes long
- Row/Index Pointers or ItemIdData
  - Array of offset/length pairs pointing to the actual rows/index entries
  - 4 bytes per item
- Free Space
  - Unallocated space
  - New pointers allocated from the front, new rows/index entries from the rear
- Row/Index Entry or Items
  - The actual row or index entry data
- Special
  - Index access method specific data
  - Empty in ordinary tables

Page Structure

The items themselves are stored in space allocated backwards from the end of unallocated space. The exact structure varies depending on what the table is to contain. Tables and sequences both use a structure named HeapTupleHeaderData. The final section is the "special section" which may contain anything the access

method wishes to store

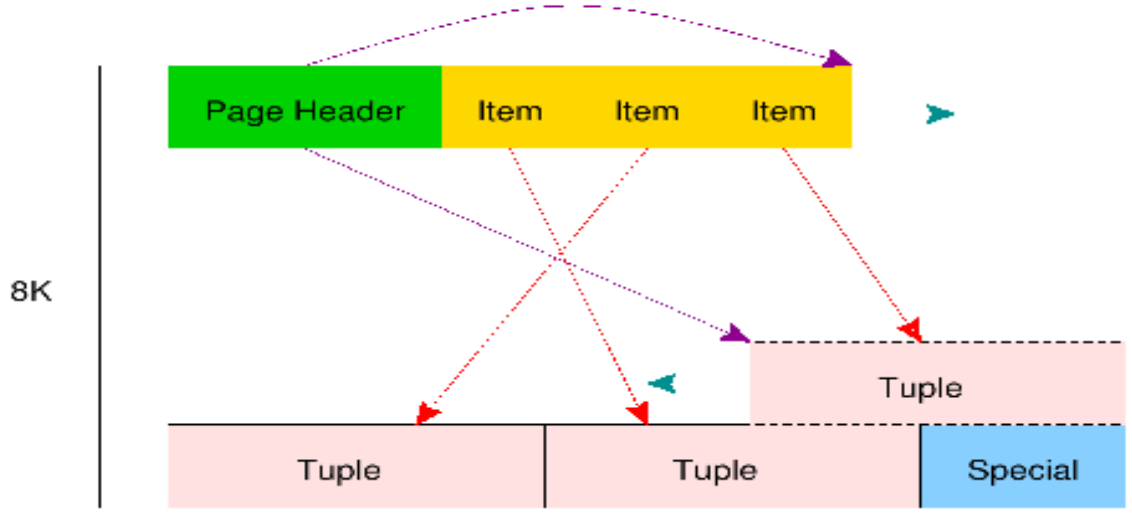


Figure: Page Structure

Item	Description
PageHeaderData	24 bytes long. Contains general information about the page, including free space pointers.
Free space	The unallocated space. New item pointers are allocated from the start of this area, new items from the end.
Items	The actual items themselves.
Special space	Index access method specific data. Different methods store different data. Empty in ordinary tables.

Figure: Overall Page Layout

Commit & Checkpoint

Asynchronous commit is an option that allows transactions to complete more quickly, at the cost that the most recent transactions may be lost if the database should crash. In many applications this is an acceptable trade-off.

Transaction commit is normally synchronous when the server waits for the transaction's WAL records to be flushed to permanent storage before returning a success indication to the client. The client is therefore guaranteed that a transaction reported to be committed will be preserved, even in the event of a server crash immediately after. However, for short transactions this delay is a major component of the total transaction time.

- Before Commit-Uncommited updates are in memory
- After Commit-Committed updates written from shared memory to disk (write-ahead log file)
- After Checkpoint-Modified data pages are written from shared memory to the data files.

When write-ahead log files fill up, a checkpoint is performed to force all dirty buffers to disk so the log file can be recycled. Checkpoints are also performed automatically at certain intervals, usually every 5 minutes. If there is a lot of database write activity, the write-ahead log segments can fill too quickly, causing excessive slowness as all dirty disk buffers are flushed to disk.

Checkpoints are points in the sequence of transactions at which it is guaranteed that the data files have been updated with all information written before the checkpoint. At checkpoint time, all dirty data pages are flushed to disk and a special checkpoint record is written to the log file. In the event of a crash, the crash recovery procedure looks at the latest checkpoint record to determine the point in the log (known as the redo record) from which it should start the REDO operation. Any changes made to data files before that point are known to be already on disk. Hence, after a checkpoint has been made, any log segments preceding the one containing the redo record are no longer needed and can be recycled or removed. (When WAL archiving is being done, the log segments must be archived before being recycled or removed.)

Transaction Log Archiving

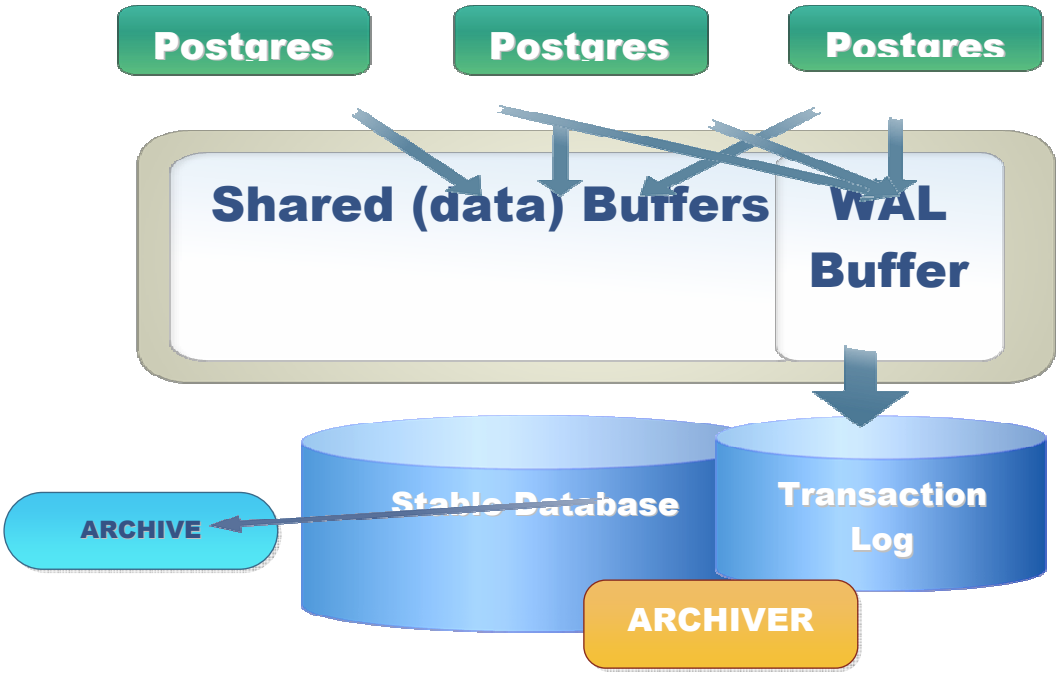
The aim of WAL, to ensure that the log is written before database records are altered, can be subverted by disk drives that falsely report a successful write to the kernel, when in fact they have only cached the data and not yet stored it on the disk. A power failure in such a situation might still lead to irrecoverable data corruption. Ensure that the disks holding PostgreSQL's WAL log files do not make such false.

PostgreSQL maintains a write ahead log (WAL) in the pg_xlog/ subdirectory of the cluster's data directory. The log describes every change made to the database's data files. This log exists primarily for crash-safety purposes: if the system crashes, the database can be restored to consistency by "replaying" the log entries made since the last checkpoint. Spawns a task to copy away pg_xlog log files when logs are full.

Continuous backup can be achieved simply by continuing to archive the WAL files. This is particularly valuable for large databases, where it might not be convenient to take a full backup frequently.
There is nothing that says we have to replay the WAL entries all the way to the end. We could stop the replay at any point and have a consistent snapshot of the database as it was at that time. Thus, this technique supports point-in-time recovery: it is possible to restore the database to its state at any time since your base backup was taken.

If we continuously feed the series of WAL files to another machine that has been loaded with the same base backup file, we have a warm standby system: at any point we can bring up the second machine and it will have a nearly-current copy of the database.

To recover successfully using continuous archiving (also called "online backup"), you need a continuous sequence of archived WAL files that extends back at least as far as the start time of your backup. So to get started, you should set up and test your procedure for archiving WAL files before you take your first base backup.



Overview on Software Installation

The first step to set up a PostgreSQL system is to install the software. In most situations, the administrator is providing two versions, as they can be made:

- Either build the software from the source code itself,
- or installed a pre-packaged

Versioning

PostgreSQL uses a three-part version number; say for example "7.4.10" or "8.2.5". The first digit of the version number changes only very rarely, as when a new "era" in the project dawns. The second digit will change when a large release comes with new features. The third digit of the version changes with every minor release that corrects only critical errors. Last version number scheme is somewhat incongruous from a technical perspective, because the first and the second number in principle to form a unity.

From the developers, there is a major release branch "8.2" with the minor versions "8.2.0", "8.2.1" and so on, depending on how many bug fixes are needed in the branch. The single first digit (7 or 8) has more representative Character, but no technical effect - the differences between 7.4 and 8.0 are in principle not unlike that between 7.3 and 7.4 or between 8.0 and 8.1. A new version has more features and better performance, and medium term we can not avoid them or later anyway, if you still want to use PostgreSQL. Minor versions (minor releases) will be released approximately every few months, depending on how many corrections are necessary. Most are minor versions of this still maintained all the major versions released at the same time, if the bug fixes apply to all. Subversions should normally be informed immediately of all Users to be imported.

Package Installation

For some of the popular Linux operating systems, presenting here briefly the installation of PostgreSQL from binary.

Debian and Ubuntu

The Debian package for the PostgreSQL server is called "postgresql". You can install it with the command

```
apt-get install postgresql
```

If you need only the client programs, install the package "postgresql-client".

```
apt-get install postgresql-client
```

Red Hat

On Red Hat Linux, the package for the PostgreSQL server is "postgresql-server". Note that the package contains only a few named postgresql client programs and not representing what is usually under a full Postgres SQL installation understands. You can install the packages you want, for example, with Yum

```
yum install postgresql-server
```

or with a graphical package management program.

SuSE

Also on SuSE Linux, the server package called "postgresql-server" and client package "postgresql". To install

it is best to use YaST.

Source Code (Installation Steps on Linux Using PostgreSQL 8.4)
First, download the source code from the website http:// www.postgresql.org/, select "Downloads" and then lurches through the links until you come into a directory overview. For example we are using version 8.4.2. and file name will be "postgresql-8.4.2.tar.gz". Here are the installation Steps

Step – 1 Create a PostgreSQL user (you'll need root access). This is who owns the system. Package-based installations, the user typically sent automatically for RPMs, Debian packages and Windows installations. When installing from the source, then the user must be created manually.

```
#useradd postgres
```

This account has two functions:

- All files in the PostgreSQL data directory must be owned by that user.
- The server process runs under this user.

Step – 2 First, download the source code from the website http:// www.postgresql.org/, select "Downloads" and proceed with the appropriate version of requirement for example we are using version 8.4.2. Move the downloaded file to source destination (eg: /usr/local/src).

```
#mv <present location> /usr/local/src/
```

Step - 3 Unpack the source archive with the commands is

```
bunzip2 postgresql-8.4.2.tar.bz2
tar xf postgresql-8.4.2.tar
or
gunzip postgresql-8.4.2.tar.gz
tar xf postgresql-8.4.2.tar
```

If GNU tar is installed, which on Linux and BSD systems usually the case , then one can also combine these two commands on each one:

```
tar xjf postgresql-8.4.2.tar.bz2
or
tar xzf postgresql-8.4.2.tar.gz
```

This then creates a directory named postgresql-8.4.2 in the current directory.

Step – 4 Change into that directory which has created with untar

```
# cd postgresql-8.4.2
```

The directory with INSTALL, contain file that the current version of the current installation instructions. "./Configure" with the appropriate installation options and features in the respective Version will be offered. It makes sense, as many of the Turn on features, so you have no problems later if it turns out. The purpose of the options is not, experimental or hides dangerous features, but only users who did not install any additional libraries, give the possibility of unnecessary Features omitted.


```
# ./configure --prefix=/usr/local/pgsql
```

Step – 5 After the issue of the ./configure , not its time to compile the code with the following commands.

```
# make
#make install
```

Step – 7 Now the change the owner ship of the directory created

```
#chown -R postgres:postgres /usr/local/pgsql
```

Step – 8 Switch to the postgres user and change into the bin directory and start the process of creating the data directory. The location of the data directory can be chosen by the user, there is no Default. Source for installations that the default installation "/usr/local/pgsql/" .

```
# su - postgres
#cd /usr/local/pgsql/bin
```

step – 9 Now initialize the data directory using initdb command and the location of the data directory should be owned by the postgres user. So the above we have given the complete permission recursively to the "/usr/local/pgsql" directory which means any directory created under this directory will be owned by the postgres user.

```
#./initdb -D /usr/local/pgsql/data ( this is the data location)
```

Step – 10 Starting the server service will run the system in the background and waits for connections from the client programs. Following is the command for starting the cluster using pg_ctl command.

```
#./pg_ctl -d /usr/local/pgsql/data start
```

The-D option here specifies the data directory. The directory must be under the PostgreSQL server account provided, which was created with initdb command. Otherwise, the server program complain the error as "PostgreSQL server is not as root".

Step -11 Connecting to the postgresql using psql client

```
#./psql
```

Step – 12 Post-Installation steps

- Set the environment variables in .bash_profile of the postgres user will keep the permanent setting for the server start up and shut down.
- We can override defaults for the database name, username, server host name, and listening port by setting the environment variables PGDATABASE, PGUSER, PGHOST, and PGPORT, respectively. These defaults may also be overridden by using the –d, -U, -h, and -p command-line options to psql.

Starting / Stopping / Reloading / Restarting the Server

Pg_ctl is a utility to start, stop, restart, reload configuration files, report the status of a PostgreSQL server, or signal a PostgreSQL process.

Usage: pg_ctl start [-w] [-D DATADIR] [-- s] [-l FILENAME] [-o "OPTIONS"]
pg_ctl stop [-W] [-D DATADIR] [-s] [-m SHUTDOWN-MODE]
pg_ctl restart [-w] [-D DATADIR] [-s] [-m SHUTDOWN-MODE] [-o "OPTIONS"]
pg_ctl reload [-D DATADIR] [-s]
pg_ctl status [-D DATADIR]
pg_ctl kill SIGNAL NAME PID

Common options:
-D, - PGDATA DATADIR location of the database storage area
-s, - silent only print errors, no informational messages
-w wait until operation completes
-W do not wait until operation completes,
- help show this help, then exit
- version output version information, then exit (The default is to wait for shutdown)

Creating a Database Cluster

A database cluster will be a single directory under which all data will be stored. We call this the data directory or data area that contains all data and configuration files. Use initdb to create a database cluster. It must be run as the OS user that the instance will run as.

initdb –D <data directory> [options]

Option	Function
-D <data directory>	Database cluster directory
-U <super user>	Select the database super user name
-E <encoding>	Specify the database encoding
-n	No clean (do not clean up files in case of failure)

initdb will attempt to create the directory you specify if it does not already exist. It is likely that it will not have the permission to do so (if you followed our advice and created an unprivileged account). In that case you should create the directory yourself (as root) and change the owner to be the PostgreSQL user. Here is how this might be done:

mkdir /usr/local/pgsql/data
chown postgres /usr/local/pgsql/data
su postgres
postgres$ initdb -D /usr/local/pgsql/data

initdb will refuse to run if the data directory has already been initialized.Because the data directory contains all the data stored in the database, it is essential that it must be secured from unauthorized access. initdb

therefore revokes access permissions from everyone but the PostgreSQL user. It is recommend you use one of initdb's -W, --pwprompt or --pwfile options to assign a password to the database superuser.

After creating a new database cluster, modify postgresql.conf and pg_hba.conf (which will be covered in next chapter).
Be sure to assign a unique port # to this cluster in postgresql.conf

Starting and Stopping the Server
The standard PostgreSQL distribution contains a utility, pg_ctl, for controlling the postmaster process. We saw this briefly in Chapter 3, but we revisit it here for a more detailed exploration of its features.
The pg_ctl utility is able to start, stop, and restart the server; force PostgreSQL to reload the configuration options file; and report on the server's status. The principal options are as follows:

pg_ctl start [-w] [-s] [-D datadir] [-p path][-o options]
pg_ctl stop [-w] [-D datadir] [-m [s[mart]] [f[ast]] [i[mmediate]]]
pg_ctl restart [-w] [-s] [-D datadir] [-m [s[mart]] [f[ast]] [i[mmediate]]] [-o options]
pg_ctl reload [-D datadir]
pg_ctl status [-D datadir]

To use pg_ctl, you need to have permission to read the database directories, so you will need to be using the postgres user identity. The options to pg_ctl are described below.

Option Description
-D datadir Specifies the location of the database. This defaults to $PGDATA.
-l, --log filename Appends server log messages to the specified file.
-w Waits for the server to come up, instead of returning immediately. This waits for the server pid (process ID) file to be created. It times out after 60 seconds.
-W Does not wait for the operation to complete; returns immediately.
-s Sets silent mode. Prints only errors, not information messages.
-o "options" Sets options to be passed to the postmaster process when it is started.
-m mode Sets the shutdown mode (smart, fast, or immediate).

When stopping or restarting the server, we have a number of choices for how we handle connected clients. Using pg_ctl stop (or restart) with smart (or s) is the default. This waits for all clients to disconnect before shutting down. fast (f) shuts down the database without waiting for clients to disconnect. In this case, client transactions that are in progress are rolled back and clients forcibly disconnected. immediate (i) shuts down immediately, without giving the database server a chance to save data, requiring a recovery the next time the server is started. This mode should be used only in an emergency when serious problems are occurring.
We can check that PostgreSQL is running using pg_ctl status. This will tell us the process ID of the listener postmaster and the command line used to start it:

pg_ctl status
pg_ctl: postmaster is running (pid: 486)

If you have built PostgreSQL from source code, you will normally want to create a script for inclusion in /etc/init.d. Most packagebased installations will provide a standard script for you. Do ensure that the PostgreSQL server gets the opportunity for a clean shutdown whenever the operating system shuts down.

Lab Exercise 1

- Choose the platform on which you want to install postgres.
- Download PostgreSQL one-click installer from Enterprisedb website for choosen platform.
- Install PostgreSQL.

Lab Exercise 2

- You have installed PostgresPlus on the available platform.
- Now create a cluster which resides in c:\edbdata folder on windows and /opt/edbdata directory on Linux
  - Newly created cluster should run on 5445 port
  - Start the cluster after initializing it
  - Login inside the cluster using psql tool

Overview

PostgreSQL supports many database systems such as a large number of configuration parameters, with which the database administrator can own configuring ideas in particular load specific. Because of the complexity of a database management system, these parameters are partially very complicated and to create a good configuration in borderline cases, very difficult. All the configuration parameters to be set in postgresql.conf file which described with the heading of the concerned parameters with there setting options. The parameters are divided by Connection Control, Memory Management, Maintenance, Transaction Logging, and Statistics.

Configuration settings for better understanding.

Configuring of a PostgreSQL database system consists of a number of parameters. A parameter is like a variable name and value, and a data type that describes what kind of possible values for the settings are accepted. In addition to this system , there are other settings, such as connecting control (pg_hba.conf) which are not included in postgresql.conf.

There are different ways to set the configuration parameters:

1. Setting in the "postgresql.conf" file
2. Command line of the postgres program.
3. Runtime in the PSQL session with the help of these three commands (SET , RESET, SHOW)

Each parameter has one of the following data types:

- Boolean (ON, OFF, TRUE, FALSE, YES, NO and also 0 and 1)
- Integer
- Decimal number(floating point)
- String

Units of Memory sizes are "KB" for kilobytes, "MB" for mega bytes or "GB" for gigabytes (the multiplier is always 1024). For Times parameter it is "ms" for milliseconds, "s" for seconds, "min" for minutes, "h" for hours.

1. Postgresql.conf file

Postgresql.conf file is the central configuration file of a PostgreSQL database cluster. When initializing a new database cluster with initdb will automatically create a file postgresql.conf, which is a standard configuration for any new database cluster.

Syntax for setting the parameters in the postgresql.conf file.

"#" this is the comment in postgresql.conf file. If any of the parameter has commented, which means the server has assigned the default value to that parameter

Example

```
#port=5432
```


To change any value of the parameter

Name = 'value' (integer, float, string or Boolean)

The configuration file is not reread when changes made automatically, but it is read only if it receives the signal SIGHUP to the database server. To take the affect of the changes made to the configuration file is pg_ctl reload.

pg_ctl reload –D <datadirectory>

2. Command Line

Setting the parameters with command-line option while starting the database server will override the corresponding setting in the postgresql.conf file. There are two ways of changing the parameters of the postgresql.conf file from the command line.

- Postgresql.conf parameter settings can be specified when starting the database server on the command line. Especially useful for test scenarios where the database administrator test different parameter values and would like to see their impact on the database.

  Eg: - postgres - shared-buffers = 128MB - work-mem = 32MB
  The minus sign (-) and underscore (_) are treated equally.

- Using pg_ctl
  pg_ctl -o '- shared-buffers = 128MB - work-mem = 32MB' start
  option "-o" is to set the parameters at the command line.

3. Using SET, RESET, and SHOW

Configuration settings can also be achieved with the SET command. But this applies only to parameters that can be changed at runtime. With the command SHOW the current setting can be queried at any time.

```
postgres=# SET work_mem TO '48MB ';
SET
postgres=# SHOW work_mem
            work_mem
            ----------
            48MB
            (1 row)
```

Using the SET command, changes of settings are within the current Transaction, but are influenced by success or abort the transaction. If For example, the transaction with COMMIT successfully completed, the settings are valid through out the transaction for each database session. At Rollback or abort the transaction will be rolled back the changes and the settings that were in effect prior to the transaction reactivated. To Settings be applied only for the current transaction, the command SET LOCAL can be used, which makes an adjustment only for the current transaction is exactly and after the end of this transaction will restore the original valid values psql session, as the following example demonstrates:

Example

```
postgres=# SHOW work_mem;
        work_mem
        ----------
        48MB
        (1 row)
postgres=# BEGIN;
         BEGIN
postgres=#SET LOCAL work_mem TO '128MB ';
        SET
postgres=# SHOW work_mem;
        work_mem
        ----------
        128MB
        (1 row)
postgres=# COMMIT;
        COMMIT
postgres=# SHOW work_mem;
        work_mem
        ----------
        48MB
        (1 row)
```

The RESET command can be a parameter reset to its default value:

```
postgres=# SET work_mem TO '128MB ';
        SET
 postgres=# RESET work_mem;
        RESET
postgres=# SHOW work_mem;
        work_mem
        ----------
        1MB
        (1 row)
```

Settings with SET are in the interactive mode, especially useful when tuning Inquiries and general experimentation. You can also use SET commands in application programs, set about installing the database for the application.

Types of configuration in postgresql.conf file

1. Connection control
2. Memory settings
3. Transaction control
4. Planner settings
5. Logging
6. Statistics
7. Localization
8. Miscellaneous

1. Connection control

The settings for the connection control to determine how the PostgreSQLServer can connect. Normally one looks at these settings after the Establishment of a new database system only once and then later need not change any more.

listen_addresses

Configuring the IP addresses on which the database server receives connection requests. You can have multiple IP addresses or hostnames specified separated by commas. The parameter can only through the Configuration file or the command line to be changed. The default is listen_addresses = 'localhost' Thus, only TCP / IP connections are accepted from the same computer. This is a Default for security reasons, but is usually changed so that connections all addresses are possible. This is the placeholder *:

```
listen_addresses = '*'
```

TCP / IP connections can be completely turned off by the parameter an empty string is assigned. Then only local connection requests would be via Unix domain sockets possible. (But that does not provide special security benefit localhost to the default.) Otherwise, arbitrary individual addresses can be listed:

```
listen_addresses = 'localhost, 192.168.1.31'
```

This is especially useful when multiple PostgreSQL instances on one machine be hosted. We can generally leave the other settings at 'localhost' or '*', As required.

port

Specifies the TCP port number on which the PostgreSQL database server accepts connection requests. The port number can only have the configuration file or configure the server command line. The default is

```
port = 5432     // Default port in postgres
```

max_connections

The max_connections parameter configures the maximum number of connections that an instance of the PostgreSQL database server can open simultaneously. The default is

```
max_connections = 100
```

or possibly a lower value when creating the database cluster with initdb has determined that the existing storage is no longer supported connections can be. An increasing the value may requires an event to Restart the server. An increase in the maximum number of connections has a larger memory requirement for the shared memory result.

superuser_reserved_connections

This parameter specifies the number of reserved database connections for Superuser privilege. These connections will be kept reserved for example, when the maximum number of connections is reached, so that administrators can gain anytime access to the database system. The number of unprivileged connections (max_connections), thereby effectively reduces this Number of reserved connections. The default value is 3.

ssl

This parameter is the SSL support of the database server enabled. The default is

```
ssl = off
```


That is "out." To turn on SSL, you must first SSL keys and certificates be established, otherwise the server will not start.

2. Memory Settings

The configuration of PostgreSQL in terms of memory usage and speed requires a close reading of application, database server, operating system and Hardware. It is therefore difficult, while all of those parameters which cover the application speed and scalability of an application success.

A database connection binds its own resources and is based on the global shared buffer pool. This buffers the requests to Database pages that contain the individual rows of a table. This ensures that for frequently used data or index entries corresponding to cached entries in Memory of the operating system available and writes operations not directly to the File system such as hard drives have to be written. Modern operating systems also use automatically free memory as buffers for the file system. Read and write operations can thus by the kernel will be accelerated. Completely free memory of the database as Shared buffer pool, or assign as a memory for other tasks, robbed Operating this optimization possible and is therefore not recommended. Let therefore still sufficient memory "free"

shared_buffers

shared_buffers is the size of the shared buffer pools for a PostgreSQL instance. This setting applies to the entire database cluster and is shared by all databases. Good values to use 10 to 25 percent of available memory. For systems with high write rates, it is advantageous to work with rather high settings, since Write operations to database tables and indexes can be buffered and frequent Job can be avoided by the shared buffer pool to the file system.
Even higher settings quickly bring disadvantages, because the administrative burden for the database is quite high and the operating system is correspondingly less has memory for file system buffers available. For database systems that other hand, tend to read only as evaluation and archiving systems, including sufficient a setting on the lower end of the scale, at around 10 percent of main memory. The maximum size of the shared buffer pool is limited by the system predetermined ceiling SHMMAX. The minimum setting is 128 Kbytes or 16 * max_connections.

Parameter	Type	Default	Min	Max	Unit	Recommended
Shared_buffers	memory	8MB	64KB	8192GB	8KB	AvRAM/4

Note: Increasing shared_buffers often requires you to increase some system kernel parameters, most notably SHMMAX and SHMALL. See Operating System Environment: Managing Kernel Resources in the PostgreSQL documentation for more details. Also note that shared_buffers over 2GB is only supported on 64-bit systems.

temp_buffers

This setting specifies a buffer pool for temporary tables. The default is 8 MB. This is not used immediately the full specified capacity, but the memory is at Requirements in block increments (up to 8 Kbytes) to the limit specified uses. Temporary tables are always immediately stored as a file. Generous settings temp_buffers are especially interesting for applications often used during a database session data in temporary tables update and save. Indexes that were generated from temporary tables that are likewise stored in the temporary buffer pool of a database session. The ideal Temp_buffers size depends on the amount of data in temporary tables

should be stored, including any existing indexes. For most applications, the temporary Tables do not use or only slightly, the default is not sufficient. It is possible temp_buffers within a database connection to an individual Value to configure as needed, for example:

SET TO temp_buffers '96MB ';

Parameter	Type	Default	Min	Max	Unit	Recommended
temp_buffers	memory	8MB	800KB	8192GB	8KB	default

Note: If your application requires heavy use of temporary tables (many proprietary reporting engines do) then you might want to increase this substantially. However, be careful because this is non-shared RAM which is allocated per session. Otherwise, the default is fine.

work_mem

The size of available main memory for database operations, such as Sort or some combination algorithms is the parameter work_mem. How temp_buffers setting is understood as a limit. Too small settings lead to an increased use of temporary files, which impairs the performance and also the additional burden on the system disk. Configuring work_mem in the database system for different tasks such as efficient sorting, linking, and filtering of certain Data used. In particular, it is used for the following operations:

- ORDER BY, DISTINCT, and merge joins require memory for sort operations.
- hash joins, hash and hash-based processing aggregations of IN operations require memory for hash tables.
- Bitmap index scans require memory for the internal bitmap.

Parameter	Type	Default	Min	Max	Unit	Recommended
work_mem	memory	1MB	64 KB	2 GB	KB	AvRAM / 2 * Max_connections

Note: This limit acts as a primitive resource control, preventing the server from going into swap due to over allocation. Note that this is non-shared RAM per operation, which means large complex queries can use multiple times this amount. Also, work_mem is allocated by powers of two, so round to the nearest binary step. The second formula is for reporting and DW servers which run a lot of complex queries.

An excessive Memory allocation to database queries with work_mem may therefore be possible to use up available memory, especially when many database sessions queries are running simultaneously. The default setting for work_mem is 1 MB.

SET Work_mem TO '32MB ';

The actual use of the work_mem made available memory can help Option trace_sort be traced. The parameter client_min_messages (see below) is the value of DEBUG configured to see the output in the client to be able to.


```
postgres=# SET work_mem TO '8 GB ';
SET
postgres=# SET TO trace_sort on;
SET
postgres=#SET TO client_min_messages DEBUG;
SET
postgres=# SELECT relname, relkind, relpages FROM pg_class c WHERE relkind = 'r'
postgres=# relpages ORDER BY DESC, ASC relname
postgres=#LIMIT 1;
LOG: begin tuple sort: NKEYS = 2, Workmen = 8192, random access = f
LOG: starting performsort: CPU 0.00s/0.00u sec elapsed 0.00 sec
LOG: performsort done: CPU 0.00s/0.00u sec elapsed 0.00 sec
LOG: internal sort ended, 31 KB used: CPU 0.00s/0.00u sec elapsed 0.00 sec
relname | relkind | relpages
---------+---------+----------
pg_proc | r | 47
(1 row)
```

This approach shows in the most recent LOG-line actually used by the query memory, and the order in memory (internal sort) or has been carried out on the File system (external sort).

maintenance_work_mem

The parameter maintenance_work_mem sets the upper limit of memory used by Management operations to update and generation of database objects or Garbage collections may be used. The semantics of these parameter settings corresponds to that of work_mem. More memory means significantly faster operations But take away any other actions of memory. The store is set here is used by the following commands:

- CREATE INDEX (as well as creation of primary keys and unique constraints)
- ALTER TABLE for adding foreign keys
- VACUUM
- CLUSTER

Parameter	Type	Default	Min	Max	Unit	Recommended
Maintenance_work_mem	Integer	16 MB	1 MB	2 GB	KB	AvRAM / 8

Memory is up to the ceiling with maintenance_work_mem defined per database session used. The default is 16 MB; temporarily higher settings can also be flexible for each command by SET command within the same database session to be made, for example, for the subsequent creation of an index:

```
postgres=# SET maintenance_work_mem TO '384MB ';
SET
postgres=# CREATE INDEX ON personen_vorname_nachname_idx persons ( "last_name);
CREATE INDEX
```


Is correspondingly large amount of memory available to larger amounts of data processed directly in the memory and processing is speeded up. VACUUM also used memory as a function of maintenance_work_mem. This is be taken into account especially when highly fragmented with many tables by investigate previous transactions are to be deleted or updated rows. VACUUM stores it in memory between the lines, and must when reaching the maintenance_work_mem possible through existing cap in each case index read. Besides increasing the store here is also more frequent VACUUM one mounted Solution.

max_fsm_pages

VACUUM managed shared space in a hash table in shared memory of the database server. The Free Space Map (FSM) stores a pointer to free space ("Dead tuples") within a table or index, and make it UPDATE or INSERT operations reuse. The size of the FSM is set at server startup, while the number of max_fsm_pages there Database States, it can be stored. The minimum setting is 16 * max_fsm_relations. The program initdb sets during initialization of a PostgreSQL instance, depending on the size of the available main memory values down 20 to 200,000. It should however, always a sufficient quantity of pages in the FSM are available, as otherwise fragmented pages of a relation or an index no longer be covered and thus they can grow ever stronger. This in turn impairs the speed and requires considerably if the growth of advanced is the use of blocking maintenance commands like VACUUM FULL or REINDEX to the objects relative to their user data to bring back to a reasonable size. This is not desirable because, unlike with VACUUM and VACUUM FULL REINDEX affect the productive basis of table and row locks and is therefore usually longer maintenance windows are required.

	Type	Default	Min	Max	Unit	Recommended
max_fsm_pages	Integer	76800	64 KB	2 GB	KB	AvRAM / 2 * Max_connections

max_fsm_relations

The number of tables and indexes that can be recorded in the Free Space Map, is determined by the parameters max_fsm_relations. The default is 1,000, which is usually sufficient. But the system tables and indexes are on what to very large database schemas may exceed the default value. The Number of indexes and tables of a database can be determined easily, for example, as follows:

```
db = # SELECT count (*) FROM pg_class WHERE relkind IN ( 'r', 'i');
count
-------
141
(1 row)
```

	Type	Default	Min	Max	Unit	Recommended
max_fsm_relations	Integer	1000	100	INT_MAX		Default

3. Maintenance Settings

Among the maintenance tasks in a PostgreSQL includes in particular the planning and to conduct regular VACUUM operations. The rows of the table deleted or updated results "Dead" space and demand for reuse. In addition, the planners must Statistics be kept up to date with the ANALYZE command. Since version 8.1 offers PostgreSQL Autovacuum an integrated service that takes over automatically.

autovacuum
Starts the daemon which cleans up tables and indexes, preventing bloat and poor response times. The only reason to set it to “off” is for databases which regularly do large batch operations like ETL.

Note that adjusting the frequency or stop autovacuum on individual tables will add rows to the pg_autovacuum system table.

autovacuum_max_workers
Defines the number of auto-vacuum processes. These are the car vacuum Launcher managed and run as needed. The default is 3, and ranges in practice normally off.

autovacuum_naptime
Thus the waiting time between each Autovacuum runs is set. The The default is one minute.

The parameters
• autovacuum_vacuum_scale_factor,
• autovacuum_vacuum_threshold,
• autovacuum_analyze_scale_factor and
• autovacuum_analyze_threshold
configure the algorithm determines the vacuum of car service, whether a specific table or VACUUM ANALYZE must be edited.

4. Transaction Settings

Transactions called WAL (Write-ahead log) are the central role of the protection of the data consistence and availability, and for data backup. The configuration of the transaction log is therefore crucial impact on the behavior of the entire system in critical situations.

fsync
The fsync parameter affects the synchronization of operations on the storage system. PostgreSQL uses the system call fsync () to the integrity of the database to ensure changes. Fsync () synchronized these processes and guarantees the writs to the storage system. All hard disk systems currently have a separate data cache, to write and Reads buffers. Fsync () is also a general instruction to the operating system synchronous and guaranteed tender of data blocks used to Hard to pass.

wal_buffers
The parameter wal_buffers configures the size of the shared Transaction buffers in memory of the database server. Changes made to the database in the transaction logs, in order to guarantee transaction security. Ideally wal_buffers configured that the size of this buffer corresponds to the largest amount of data during a Transaction to be changed. The default is 64 Kbytes. There is no demonstrated benefit to increasing wal_buffers further.

	Type	Default	Min	Max	Unit	Recommended
wal_buffers	Integer	64kb	100	INT_MAX		8MB

synchronous_commit

PostgreSQL 8.3 has added yet another technology to reduce the load on log devices. This option when set to false will for most cases avoid synchronous commit (or syncing of the WAL buffers after commit). A separate process will sync based on time which will allow more and more transactions to be combined together (till the maximum value the operating system allows) within a single IO operating.

However there is a risk associated with synchronous_commit. Unlike commit_delay which delays the transactions, in this case the transactions gets committed even though wal buffers are not yet flushed/synced to the device. Hence there are chances of loosing transactions even though it was reported committed. However considering many people who revert to unsafe fsync=off, this allows the database to be consistent state in case of power failure.

Note: The parameter synchronous_commit has some similarities with the parameter fsync. The difference is that synchronous_commit = off in a crash some of the last Transactions can lose, but fsync = off in a crash the entire database system can destroy.

wal_writer_delay

wal_writer_delay provides a lower load in Writing Database connections is adjustable rotation to the corresponding blocks; database connections are substantially relieved them. The WAL-Writer is also responsible for the writs asynchronous transactions synchronous_commit the parameter can be controlled. wal_writer_delay can only be in the configuration file or the server command be set. The default is 200 ms.

checkpoint_segments

checkpoint_segments parameter defines the number of segments in the Transaction, within the PostgreSQL to log data changes a transaction are available. Default is (2 * checkpoint_segments + 1) .

A segment consists of 16 Mbytes. The preconfigured value three log segment is already for small databases with high write load too low, a good starting value, twelve or sixteen segments.

	Type	Default	Min	Max	Unit	Recommended
Checkpoint_segments	Integer	3	1	INT_MAX		16 to 128

checkpoint_timeout

PostgreSQL forced by default every five minutes as a checkpoint to all changes from the transaction log to synchronize the data base. This behavior can be with the configuration parameters checkpoint_timeout influence Checkpoints can be reduced to postpone or longer intervals. Database with high write rates, it is worthwhile to check points to larger intervals distribute the Background Writer ensures smooth writs of data changes in the background and can be adapted as appropriate.

checkpoint_warning

This parameter causes a warning in the server log. In Default, a warning is issued if less than 30 checkpoints Seconds occur. Number of segments by the configuration parameter should be on checkpoint_segments,

otherwise the speed of the database system will be bad. If, however, just a lot of data to be loaded then it is normal as more data is written to the transaction log.

checkpoint_completion_target
The parameter checkpoint_completion_target configures the maximum duration of a Checkpoints. For a checkpoint and the writs of data changes can be curbed. Checkpoints results a write strong load on the storage. To mitigate these effects, the background Writer process is responsible for the handling of checkpoints, and thus delayed will. checkpoint_completion_target defines the length of time that a checkpoint must shall, in proportion to the checkpoint interval. A value of 0.0 corresponds to full speed 1.0 distributed during a checkpoint on the duration of an entire interval. The default is 0.5.

full_page_writes
In case of database crash partially and to restore the pages, PostgreSQL requires a complete copy of a page in the transaction log. By default its turned on, so also in the case of a crash warrants that partial tender Database pages restored from the recovery by the transaction. The impact of shut full_page_writes are comparable to those a disabled fsync parameter.

5. Planner Settings

Plan Types, with the settings can be switched off /on. The planner then used whenever possible, other plan types to one to perform certain request. These settings should be as an aid for Error analysis and optimization experiments to understand. It is not recommended, single parameter to disable globally. At best, these parameters are within a database session with the SET command set.

enable_seqscan:
Ask the planner does not take into account sequential scans possible. Thus, the use of index-based execution plans can be enforced be, if the planner tends to sequential plans. Sequential scans can are not completely shut down, they are only rated extremely high, and appear Therefore, the planner is very costly, so that they are normally not used be as long as cheaper alternative execution plans can be identified.

enable_indexscan: Allow you to activate or deactivate Index Scans.

enable_bitmapscan: Allows you to enable or disable bitmap Index scans.

enable_nestloop: Allows you to enable or disable bitmap nested loop scans. Nested-loop joins can are not completely shut down, so an execution plan, they can, despite Deactivation contain

enable_hashjoin:
hash-join plan types are not taken into account by the planner when the Configuration parameter is enable_hashjoin disabled.

enable_mergejoin:
Merge-join plan types are not taken into account by planners when the configuration parameter is enable_mergejoin disabled.

enable_hashagg:

Enables or disables the use of hash-based aggregation.

enable_sort:
If this parameter is disabled, the inquiry led the planners; Sorting operations do not take into account possible. The deactivation of sorting nodes is not completely possible, but the planners tried to use alternatives.

Cost parameters
The products described in this section parameters affect the cost model of Scheduler.

seq_page_cost:
This parameter defines the cost of reading a sequential Database page (by default 1.0).

random_page_cost:
This parameter defines the cost of reading a database page in distributed, random access (eg when you search an index, in the Default 4.0).

cpu_tuple_cost:
This parameter defines the cost of processing a table row by the CPU (by default 0.01).

cpu_index_tuple_cost:
This parameter defines the cost of processing a Index entry by the CPU (by default 0,005).
cpu_operator_cost:
This parameter defines the cost of processing an operator or a function (by default 0.0025).

effective_cache_size:
This parameter specifies how much cache memory system a request will be available. This parameter also sets no store PostgreSQL, it will inform the planners but only with how much cache has the operating system. The default is 128 MB.

Parameter	Type	Default	Min	Max	Unit	Recommended
effective_cache_size	Integer	128MB	1	INT_MAX	8KB	AvRAM * 0.75

6. Logging Setting

Log information will provide the database administrator a valuable overview of the state of a database in the past. From them, administrator gathers information on malfunctions, application behavior or even security problems. However Log files do not vary in any quantity and size, according to information content they must be backed up and rotated after a certain period. The logging parameters offer the administrator the possibility of the server log (the Recording of text messages from the server not to be confused with the transaction / Adapt WAL, Commit-Log/clog and other logs) to their own needs. PostgreSQL divide's logs into groups:

- Where is logged
- When it should be logged and
- What should be logged?

Where should be logged?

The first decision is where to write the logged information. PostgreSQL writes to a file or to the syslog service, or to the Windows event log, where they usually turn into a file. In addition, one must consider how the logs should be rotated. Without rotation, the log file will grow in the long run very strong and pretty much be unwieldy. Popular Linux Systems offer over their own infrastructure easy to manage log rotation and assurance mechanisms to, so that no other system will be established alongside needs. PostgreSQL Server Logs should be in accordance with the Filesystem Hierarchy Standard (FHS) in or in / lie var/log/postgresql.

log_destination:

This parameter specifies log_destination, which Log in for the server logs. The following options are available for:

stderr : Writes information to standard error output. The-l option from pg_ctl has the same effect. If the Parameters logging_collector or redirect_stderr is Starting from PostgreSQL itself a logger for, the standard error intercepts and writes to files, which with further described below, parameters can be configured. This value is the default, and probably the most used setting.

syslog : Sends all log information to the syslog system based on Unix-like operating systems the central repository for log messages of any kind depending on what version of syslog is installed, the system in the configuration file /etc/syslog.conf, /etc/syslog-ng /syslog-ng.conf, / etc / rsyslog.conf or similarly configured. It is interesting to use the syslog for log messages to multiple servers, collect over the network or other features of modern syslog implementations

eventlog: Windows uses the event log for the recording of log messages. Under Windows, this is the preferred way server messages to record and process them.

csvlog : This option PostgreSQL 8.3 is available, the value of log_destination can be a list of those values when multiple Logziele are wanted, for example:

log_destination = 'stderr, syslog'

To use the CSV mode, you must have the parameters logging_collector switched be.

logging_collector:

If this parameter is on, PostgreSQL start the one Log process, standard error of the intercepts and writes to files. This is a kind of mini-Syslog specifically for PostgreSQL. The default setting is off. Prior to PostgreSQL 8.3, this parameter was redirect_stderr.

log_directory:

This parameter determines the directory for the log files if logging_collector is used. Relative paths are related to the data directory of the database cluster. The default is log_directory = 'pg_log' that is below the data directory.

log_filename:

This parameter specifies the log file name if logging_collector used. The filename is relative to log_directory. The default is log_filename = 'postgresql-% Y-% m-% d_% H% M% S.log'. The wildcards are available for date and time corresponding to the C function strftime() . If no wildcards are specified, the so-called Epoch attached. Usually is the default for all cases sufficient.

log_rotation_age:

This parameter specifies how much time after no more than a log rotated, if logging_collector is used. The default is one days ('1 D '). If the parameter is set to 0, it will not time-based log rotation.

log_rotation_size:
This parameter specifies in achieving what size a log file latest rotation, if logging_collector is used. The default is ten megabytes. If the parameter is set to 0, no Log rotation made. It may be advisable to set this parameter to 0, and only time-based Log rotation to apply.

log_truncate_on_rotation:
This para1meter can be set up there that rotated log files are overwritten after a certain time, rather than freely lifted be. By default, this parameter is off. If after a time-based Rotation is a new log file will be written and it has a file of this name, new log data is appended to the existing file. That is a precaution so that no data is lost. If this parameter is turned on, however, it will overwrite the existing file. This has the effect that the logs are effectively discarded when the Hours, week days or days are gone through once a month, depending on what you chose.
Here's an example: Let us suppose

```
logging_collector = on
log_truncate_on_rotation = on
log_filename = 'postgresql-% a.log
log_rotation_age = '7 d '
```

Then you will get log files with names Mo.log postgresql, postgresql-Di.log and so on (may vary) to locale and operating system version looks different, which is every seven Days to renew.47An example of a particularly active system would be created hourly logs, be renewed every 24 hours. (Such a short period of time is probably not recommended.)

```
log_filename = 'postgresql-% H.log'
log_rotation_age = '24h '
```

syslog_facility:
This parameter determines to which "facility" marking the Postgre SQL log messages are sent to the syslog service. The syslog service ordered usually according to the facility in different files. The default is local0. Other possible values are local1 to local7. The following setting in the syslog configuration file syslog.conf would, for example, all log messages with the facility local0 to the specified file to write:

```
local0 .* / var / log / postgresql
```

If no such configuration exists, could log messages from PostgreSQL will be lost.

syslog_ident:
This parameter specifies which identification with the PostgreSQL log messages to be sent to the syslog service. The default is 'postgres'. This identification makes recognizable in the log file, which program created the log message came. If necessary, you can change that, by some different PostgreSQLInstanzen to be able to distinguish.

When should be logged?
The control settings described in this section, when log messages to be written or, in other words, how many log messages are written.

client_min_messages:
It controls which server messages to the client to be sent. Possible settings are DEBUG5 to DEBUG1, LOG, NOTICE, WARNING, ERROR, and FATAL PANIC. It will be set each stage of all messages and all of them

standing right in the list sent to the client. The default is NOTICE. In some cases, users find the messages sent during NOTICE unnecessary, then WARNING is an appropriate setting. For example, the role-specific Configuration settings (ALTER ROLE SET ...) or use the command SET client_min_messages in the file .psqlrc install.

log_min_messages:
This parameter determines what types of messages in the log to be written. It is analogous to the existing elsewhere occasionally log level Settings. Possible values are DEBUG5 to DEBUG1, INFO, NOTICE, WARNING, ERROR, LOG, FATAL and PANIC. The default is NOTICE. If you sent in the NOTICE Messages to find that much is a sensible attitude WARNING. Higher values than WARNING not ERROR or are impractical, since then important information about problems and errors were not preserved. The values are DEBUG, as the name implies, intended only for debugging the PostgreSQL code.

log_autovacuum_min_duration:
The configuration parameter log_autovacuum_min_duration allows you to log all Autovacuum operations when log_autovacuum_ min_duration = 0, or if the duration of a started car by vacuum-run VACUUM or ANALYZE run takes longer than the specified time, for example

```
log_autovacuum_min_duration = 300s
```

This example is logging all Autovacuum runs owing to the more than take five minutes. This setting can only be set in the postgresql.conf or the server command line to be made. The default is -1, so that the logging of Autovacuum completely eliminated.

log_error_verbosity:
This parameter represents how many details in a log message are included. The possible values are terse, default, and verbose. Default is of course the "default". For example, in psql to determine what is spent in the client, use the following Code:

```
\ set verbose Verbosity
```

log_min_error_statement:
This setting will ensure that every error also, the command is logged, that caused the error. This is actually always useful. Possible values for this parameter are: DEBUG5 to DEBUG1, INFO, NOTICE, WARNING, ERROR, LOG, FATAL, and PANIC. The value expresses what the level of reporting at least must be to ensure that the SQL command is logged to it. The default setting is ERROR

log_min_duration_statement:
This parameter is a time setting. It means that all SQL commands that are logged to exceed the stated time. This allows be identified very easily slow sql commands to make it then to optimize . For example, to log all the commands that are longer than two minutes walk, the attitude

```
log_min_duration_statement = '2 min '
```

The log entries will then see example like this:

```
LOG: duration: 51,122 ms statement: SELECT 1 +1;
```

The default is 0, which the whole mechanism is turned off.

What is logged?

log_checkpoints:

This setting will be at each checkpoint a log message and some statistics are written to the log. The Default is off.

log_connections:
With this setting, a new database connection for each Entry is written to the log. The entry contains further information about the connection, e.g. the user. This also depends on configuring log_line_prefix. If there already are represented database and user name and session ID log_connections offers no major gain of information. One can also place in log by log_connections turned on and log_line_prefix only the session ID, or PID is. The default setting is off.

log_disconnections:
This parameter is at the end of each database session, so written on logout of the client, a log entry. The log entry contains the Session lengths. Except for the analysis of connection pool, to other testing or for the total surveillance, this setting is probably not necessary. The Default is off.

log_duration:
This parameter ensures that the duration of all SQL commands will be logged. The default setting is off. The command itself is not by this parameter logged, but other settings are responsible. This setting is intended for programs to evaluate the log automatically, for example statistics on the average duration of commands to create.

log_hostname:
This parameter ensures that IP addresses that appear in log messages, as the client IP addresses in log_connections or log_line_prefix in hostnames be converted. The default setting is off.

log_line_prefix:
This parameter contains a string which can be written in each log line is the so-called Log_line_prefix. The prefix may contain wildcards can equip the log entries with the latest information. The wildcards % and a letter. The following placeholders are available:
% c - Session ID - This includes the process number and the current time. Usually the PID used (% p), but occasionally PID's are reused by the operating system. The% c is a repetition of the number will be excluded.
% d - Database Name
% h - Host name or IP address of the client
% i - the nature of the current SQL command (for example, "SELECT" or "VACUUM")
% l - the serial number of the current log line log lines below the current session
% m - current time with milliseconds (see also% t) - Most assists the indication of milliseconds in database systems with large load process of the order of entries to recognize.
% q - When the logging process is not a database session, but an auxiliary process as the background writer, then stop here and the rest of log_line_prefix be ignored. The placeholder% d,% h,% i,% r and% u are only database sessions filled, so you could before they write a% q.
% p - Process ID (PID) of the database session
% r - Host name or IP address and port of the client
% s - Starting time of meetings
% t - Current time without milliseconds (see also% m)
% u - Username
% v - virtual transaction ID
% x - Transaction number, or 0 if no

%% a percent sign

The default setting of log_line_prefix is empty. This is not recommended, because it may be difficult to interpret the logs. Here's a practical example of a setting:

log_line_prefix = '% m [% p:% l]% u @% d'

log_lock_waits:
If this parameter is on, a log entry written when a Process is longer than deadlock_timeout (usually one second) on a lock (Lock) wait. The default setting is off.

log_statement:
This setting determines whether executed SQL commands in the log should be written. The setting is one of the following values:

none	It will be no SQL commands being logged (the default value)
ddl	There are only so-called DDL commands, that is, CREATE, DROP and ALTER logged.
mod	All data modification commands, ie INSERT, UPDATE, DELETE, TRUNCATE and COPY FROM, plus all the DDL commands.
all	There will be logged, all commands, so in particular all SELECT statements. Other settings should be as log_min_error_statement and log_min_duration_statement are used to ensure that faulty or are recorded for slow orders.

log_temp_files:
This parameter determines whether a log entry should be written, if a temporary file is deleted. The value is a memory that determines how large the file at least must be logged to be. The value 0 specifies that all temporary Files will be logged. The default value -1 disables the log entries from all over.

log_timezone:
This parameter determines the time zone for log entries. The default is "unknown", is currently used in the system so that the chosen time zone. But it may be useful for a single, consistent time zone for all log messages set, for example:

log_timezone = 'UTC'

7. Statistics

It is generally desirable, and even if the Statistical tables are not used. Moreover required Autovacuum some of the statistics collected by these parameters in order to determine when to be vacuumed. Since Autovacuum from PostgreSQL 8.3 is Default on and a highly recommended feature, the statistical parameters at all by default and should not normally be switched off.

track_activities
If this parameter is on, the current SQL command followed each session and For example, in the view pg_stat_activity visualized.

track_counts
If this parameter is to be a variety of statistics on database activity collected. They are visible for example in the views pg_stat_ and pg_statio_ * *. Parameter with this name only came into existence with PostgreSQL 8.3. previously these divided into several functional parameters. The default setting of this Parameter was on. To collect real data, there were the two parameters stats_row_level and stats_block_level that the information for pg_stat_ * or pg_statio_* collected. The default setting of these two parameters was made.

update_process_title
If this parameter default is on, the process tracks the various PostgreSQL process customized to display what they are doing . This feature is generally useful, but in some Operating system may lead to performance problems. Therefore, there is the Ability to disable it.

8. Localization

This section describes the parameters that postgresql adaptation of environment to specific linguistic or regional characteristics described. The localization settings for the initialization of the database cluster created by initdb, either according to the options passed initdb or by the localization settings of the operating system will be adopted. If the operating system settings will not be accepted if the option used with initdb –locale. The possible locale names are OS-dependent. On Unix-like systems you can look at the available locales using the command

```
locale -a
```

client_encoding
This parameter tells the server which encoding are the characters from the client occur (for example, an SQL command), and also what the character encoding should have, which will return the server to the client (for example, Query results). The server converts the client automatically into the internal encoding in Servers used, and vice versa, but this only works correctly if the client encoding in this parameter was specified correctly. The default is the encoding that the server uses internally.

datestyle
This parameter determines datestyle of the used date format. Datestyle consists of two parts: the output format and the day-month-year sequence, which is separated by commas, for example:

```
SET TO datestyle ISO, DMY;
```

For the output format, there are the following possible values: ISO, SQL, German, Postgres. The default is ISO, which is the SQL-standard format (YYYY-MMDD). The other values are different "traditional" formats. Do not change this setting, because many clients to ISO format expect.

LC_COLLATE
This parameter contains the locale setting for collation. This parameter is initialized by initdb and can not be amended, but is only meant to look (for example with the command SHOW). If you want to change the sort order, you have to run initdb new For example, with

```
initdb-locale = zh_HK.utf8 path ...
or
initdb - lc-collate = zh_HK.utf8 path ... and then reload the data again.
```

LC_CTYPE
This parameter contains the locale settings on the correspondence of Uppercase and lowercase letters. This

parameter is initialized by initdb and can not be changed. If you want to change it, you have to run initdb new For example, with

initdb-locale = zh_HK.utf8 path ...

or

initdb - lc-ctype = zh_HK.utf8 path ...

LC_MESSAGES

This parameter sets the current locale settings on the language of the program messages of PostgreSQL server. This parameter, however, affected only the messages from the PostgreSQL server. The language the client programs is determined by the environment variables LC_MESSAGES, LANG and LANGUAGE controlled, depending on the operating mechanisms.

LC_MONETARY

This parameter sets the current locale settings on the formatting of monetary amounts. This particularly concerns the to_char function.

LC_NUMERIC

This parameter sets the current locale settings on the Formatting of monetary amounts. This particularly concerns the to_char function.

server_encoding

This parameter contains the current server encoding, so the character set encoding in the data are stored in the server.

Miscellaneous

This section will now be exhaustively listed a few configuration parameters, have fit well anywhere else. As I said, there are other Configuration parameters that are seldom used, and here the context would blow up.

search_path

This parameter determines the schema search path. It is therefore comparable to search paths in shells and Operating systems. The default is

$ user, public

The value of $user is a placeholder for the name of the current user. If one of each user creates a schema that has the same name as the user to work all users automatically to their "own" scheme, and not to come into Warren. Appropriate settings are per user or database, at the beginning of a database session or within a server function.

server_version

This parameter contains the server version as a string, for example, '8 .3.2 '. The Value can only be read.

server_version_num

This parameter contains the server version as a number encoded, for example 80302nd This Value can easily be numerically compared with other values, it can only be read.

statement_timeout

This parameter contains a time stamp. When a command runs longer, it will be canceled. The demolition is logged as a bug, and if log_min_error_statement accordingly is set, can also be seen in the aborted command

server log. If the value is 0, there is no time limit. This is the default. For example, in the transaction

SET TO LOCAL statement_timeout min '5 ';

timezone

This parameter sets the time zone. The default setting is unknown, thus the used in the operating system chosen time zone. If a different time zone is desired, it can be set here.

Lab Exercise 1

- You are working as DBA. Ensure that your cluster is running and listening on port 5445. Make necessary changes in server parameter file for following settings:
  - Server accept connection requests from other tcp/ip clients
  - Server should allow upto 200 connected users
  - Server should reserve 10 connection slots for DBA.
  - Maximum time to complete client authentication will be 10 seconds

Lab Exercise 2

- Working as a DBA is a challenging job and to track down certain activities on the database server logging is suggested. Go through server parameters that control logging and implement following:
  - Save all the error message in a file inside pg_log folder in your cluster data directory (e.g. c:\edbdata)
  - Log all queries and there time which are taking more than 5 seconds to execute
  - Log the users who are connecting to the database cluster.
  - Make above changes and verify them.

Overview

PostgreSQL uses the $PGDATA/pg_hba.conf file to control client access (hba is an acronym for host-based authentication). When a client application (such as psql) TRies to connect to a PostgreSQL server, it sends a username and database name to the postmaster.

The pg_hba.conf file

The access control or client authentication is not through SQL commands, but set via an external configuration file. This file is named pg_hba.conf and located in the data directory. It is used in the initialization of the data directory with *initdb* then created and can be adjusted accordingly. The reason for this external Configuration is that it is no risk of permanently by faulty settings shut out from the database system.

This configuration file is set to such connection requests from various Clients should be authenticated. Using various connection parameters such as the IP address of the client, user or Database can specify the database server, which authentication method For example, password authentication, or about PAM, must go through the client. So you can authenticate different clients differently, which sometimes can be useful. In addition, you can set using the connection parameters, which clients can connect at all.

The pg_hba.conf file controls:

- Which hosts are allowed to connect.
- How users are authenticated on each host.
- Databases accessible by each host

The configuration file is read on postmaster startup and when the postmaster receives a SIGHUP. Postmaster is generally the server in a PostgreSQL database system.

Sample PostgreSQL HOST-BASED ACCESS (HBA) CONTROL FILE (pg_hba.conf)

Type	database	ip_address	Mask	auth_type	auth_argument
local	all			trust	
Host	all	127.0.0.1	255.255.255.255	trust	
Host	template1	192.168.93.0	255.255.255.0	ident	Sameuser
Host	template1	192.168.12.10	255.255.255.255	md5	
Host	all	192.168.54.1	255.255.255.255	reject	
Host	all	0.0.0.0	0.0.0.0	krb5	
local	sameuser			md5	
local	all			md5	Admins

The type field tells us what the type of the connection is. There can be three different kinds of connections. They are:

- host
- hostssl and
- local

Records with the type field set to "host" indicate what different networked hosts can connect to the database. A record with "hostssl" type is similar, but adds the additional information, that the connection is over a secure socket layer (SSL). A "local" type tells that the connection is from the local host via a UNIX domain socket.

The second field is the database name. It can be one of the following:

- the name of a PostgreSQL database
- "all" to indicate all databases
- "sameuser" to allow access only to databases with the same name as the connecting user

The third and fourth fields are the IP address and the subnet mask of the host from which the connection is sought.

The fifth field is what is the most important to us. It is the authentication type field. PostgreSQL supports different types of authentication.

Method	Description
trust	The user is allowed, with no need to enter any further passwords. Generally, you will not want to use this option except on experimental PostgreSQL systems, although it is a reasonable choice where security isn't an issue.
reject	The user is rejected. This can be useful for preventing access from a range of machines, because the rules in the file are processed in order. For example, you could reject all users from 192.168.0.4, but later in the file, accept connection from other machines in the 192.168.0.0/8 subnet.
md5	The user must provide an MD5-encrypted password. This is a good choice for many situations.
crypt	Same as "md5", but uses crypt for pre-7.2 clients. You can not store encrypted passwords in pg_shadow if you use this method.
password	Authentication is done by matching a password supplied in clear by the host. If no AUTH_ARGUMENT is used, the password is compared with the user's entry in the pg_shadow table. If AUTH_ARGUMENT is specified, the username is looked up in that file in the $PGDATA directory. If the username is found but there is no password, the password is looked up in pg_shadow. If a password exists in the file, it is used instead. These secondary files allow fine-grained control over who can access which databases and whether a non-default password is required. The same file can be used in multiple records for easier administration. Password files can be maintained with the pg_passwd(1) utility. Remember, these passwords override pg_shadow passwords.
ident	For TCP/IP connections, authentication is done by contacting the ident server on the client host. Remember, this is only as secure as the client machine. On machines that support Unix-domain socket credentials (currently Linux, FreeBSD, NetBSD, and BSD/OS), this method also works for "local" connections. AUTH_ARGUMENT is required: it determines how to map remote user names to

	Postgres user names. The AUTH_ARGUMENT is a map name found in the $PGDATA/pg_ident.conf file. The connection is accepted if that file contains an entry for this map name with the ident-supplied username and the requested Postgres username. The special map name "sameuser" indicates an implied map (not in pg_ident.conf) that maps each ident username to the identical PostgreSQL username
krb4	Kerberos V4 authentication is used. Allowed only for TCP/IP connections, not for local UNIX-domain sockets.
krb5	Kerberos V5 authentication is used. Allowed only for TCP/IP connections, not for local UNIX-domain sockets.
pam	Authentication is passed off to PAM (PostgreSQL must be configured --with-pam), using the default service name "postgresql" - you can specify your own service name, by setting AUTH_ARGUMENT to the desired service name. reject: Reject the connection. This is used to reject certain hosts that are part of a network specified later in the file. To be effective, "reject" must appear before the later entries.

Lab Exercise 1

- You are working as Postgres DBA. Your server box have 2 network cards with ip 1.1.1.1 and 10.1.10.1. 1.1.1.1 is used for internal LAN and 10.1.10.1 is used by web server to connect users from external network. Your server should accept tcp/ip connections both from internal and external users.
  - Configure your server to accept connection from external and internal network.

Lab Exercise 2

- You are working as a Postgres DBA. Your server is running on 1.1.1.1 port 5445. A developer came to you and shows following error:

  psql: could not connect to server: Connection refused (0x0000274D/10061)
  Is the server running on host "1.1.1.1" and accepting
  TCP/IP connections on port 5432?
- Predict the problem and suggest the solution

Lab Exercise 3

- A new developer have joined. His ID number is 89. create a user dev89 with password 'password89' and assign necessary privileges to dev89 so that he can connect to edbstore database and view all tables.

Lab Exercise 4

- A new developer joins e-music corp. He got IP address 1.1.1.89. He is not able to connect from his machine to postgres server and get following error on the server.
  FATAL: no pg_hba.conf entry for host "1.1.1.89", user "dev89", database "edbstore", SSL off
- Configure your server do that the new developer can connect from his machine.

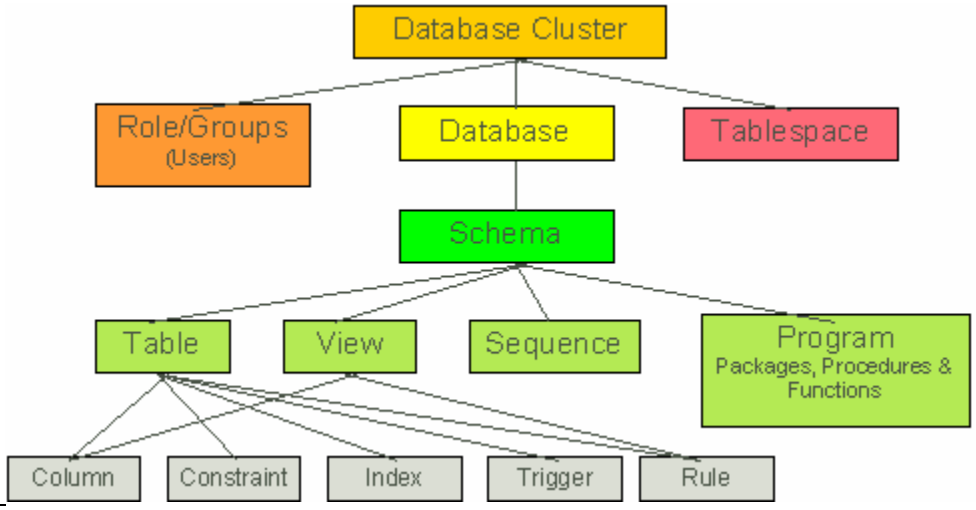
Overview

Every instance of a running PostgreSQL server manages one or more databases. Databases are therefore the top most hierarchical level for organizing SQL objects ("database objects"). This chapter describes the properties of databases, and how to create, manage, and destroy them.

A database cluster is a collection of databases that is managed by a single instance of a running database server. After initialization, a database cluster will contain a database named postgres, which is meant as a default database for use by utilities, users and third party applications.

Object Hierarchy

A PostgreSQL database cluster contains one or more named databases. Users and groups of users are shared across the entire cluster and tablespace, but no other data is shared across databases. Any given client connection to the server can access only the data in a single database, the one specified in the connection request. Furthermore, database is collection of database objects such as schema, which contains table, view, sequence, functions and other database objects as shown in figure. Figure below shows the relationships between clusters, databases, schemas, and tables.



Database

A database is a named collection of SQL objects ("database objects"). Generally, every database object (tables, functions, etc.) belongs to one and only one database. (But there are a few system catalogs, for example pg_database, that belong to a whole installation and are accessible from each database within the installation.) More accurately, a database is a collection of schemas and the schemas contain the tables, functions, etc. So the full hierarchy is: server, database, schema, and table. An application that connects to the

database server specifies in its connection request the name of the database it wants to connect to. It is not possible to access more than one database per connection. To determine the set of existing databases:

SELECT datname FROM pg_database;

(But an application is not restricted in the number of connections it opens to the same or other databases.) It is possible, however, to access more than one schema from the same connection.

Creating Databases

To create new databases, you can run createdb from an operating system prompt. Initially, only the PostgreSQL superuser can create new databases. Other users can be given permission to create new databases.

The createdb program creates a new database by making a copy of the template1 database. This database is created when PostgreSQL is first initialized. Any modifications to template1 will appear in subsequently created databases. Databases are created with the CREATE DATABASE command and destroyed with the DROP DATABASE command.

Create Database command can be used to create a database in a cluster.

Syntax:

```
CREATE DATABASE name
  [ [ WITH ] [ OWNER [=] dbowner ]
     [ TEMPLATE [=] template ]
     [ ENCODING [=] encoding ]
     [ TABLESPACE [=] tablespace ]
     [ CONNECTION LIMIT [=] connlimit ] ]
```

Databases are removed with dropdb. The CREATE DATABASE and DROP DATABASE commands are also available in SQL.

Users

PostgreSQL provides two methods by which database users may be created. Each requires authentication as a superuser, for only superusers can create new users.

- The first method is through the use of the SQL command CREATE USER, which may be executed by any properly authenticated PostgreSQL client (e.g., psql).

  postgres=# CREATE USER enterprisedb WITH PASSWORD 'edb';

- The second is a command-line wrapper called createuser, which may be more convenient for a system administrator, as it can be executed in a single command without the need to interact with a PostgreSQL client. The createuser shell script is a bit easier to use than CREATE USER because it prompts you for all required information. Here is sample createuser session:

  ```
  $ createuser -U manager
  ```

a)Altering Users

Existing users may only be modified by PostgreSQL superusers. Possible modifications include each of the options available at the creation of the user (e.g., password, password expiration date, global rights), except

for the system ID of an existing user, which may not be modified. Modification of existing users is achieved through the use of the ALTER USER SQL statement.

```
postgres=# ALTER USER enterprisedb WITH PASSWORD 'xyz';
```

At times you may wish to grant user additional rights, beyond those originally granted to them. The use of the CREATEUSER keyword modifies the user enterprisedb to have all rights in PostgreSQL, making the user into a superuser.

```
postgres=# ALTER USER enterprisedb CREATEUSER;
```

Conversely, there may be times when a user no longer deserves rights that have been granted in the past. These rights may be just as easily removed by a superuser with the NOCREATEDB and NOCREATEUSER keywords.

```
Postgres=# ALTER USER enterprisedb NOCREATEDB NOCREATEUSER;
```

Note: As any superuser may revoke rights from another superuser, or even remove another superuser, it is wise to be extremely careful when granting the CREATEUSER right.

b) Removing User

PostgreSQL users may at any time be removed from the system by authenticated superusers. The only restriction is that a user may not be removed if any databases exist which are owned by that user. If a user owns a database, that database must be dropped before the user can be removed from the system.
As with the creation of PostgreSQL users, there are two methods by which users may be removed.
These are

- The DROP USER SQL command. A superuser may remove a user by issuing the DROP USER command from a valid PostgreSQL client. The psql program is most commonly used to achieve this task.

  ```
  postgres=# DROP USER enterprisedb;
  ```

- The dropuser command-line executable. The dropuser command operates much like the createuser script. It offers the same connection options, ensuring that it can be used remotely as well as locally, and requires only the username of the user to be removed from the system.

  ```
  $ dropuser –U enterprisedb
  ```

Roles

PostgreSQL manages database access permissions using the concept of roles. A role can be thought of as either a database user, or a group of database users, depending on how the role is set up. Roles can own database objects (for example, tables) and can assign privileges on those objects to other roles to control who has access to which objects. Furthermore, it is possible to grant membership in a role to another role, thus allowing the member role use of privileges assigned to the role it is a member of.

The concept of roles subsumes the concepts of "users" and "groups". In PostgreSQL versions before 8.1, users and groups were distinct kinds of entities, but now there are only roles. Any role can act as a user, a group, or both. Concept of roles was initially introduced in PostgreSQL 8.1.

Database roles are global across a database cluster installation (and not per individual database). To create a role uses the CREATE ROLE SQL command:

```
CREATE ROLE edb;
```

edb follows the rules for SQL identifiers: either unadorned without special characters, or double-quoted. (In practice, you will usually want to add additional options, such as LOGIN, to the command. More details appear below.) To remove an existing role, use the analogous DROP ROLE command:

```
DROP ROLE name;
```

Groups

Groups serve to simplify the assignment of rights. Ordinary privileges must be granted to a single user, one at a time. This can be tedious if several users need to be assigned the same access to a variety of database objects.

Groups are created to avoid this problem. A group simply requires a name, and can be created empty (without users). Once created, users who are intended to share common access privileges are added into the group together, and are henceforth associated by their membership in that group. Rights on database objects are then granted to the group, rather than to each member of the group. For a system with many users and databases, groups make managing rights less of an administrative chore.

a) Creating a group

Any superuser may create a new group in PostgreSQL with the CREATE GROUP command. Here is the syntax for CREATE GROUP:

```
CREATE GROUP enterprisedb WITH USER edb1, edb2;
```

“enterprisedb” is the name of the group that you wish to create. A group's name must start with an alphabetical character, and may not exceed 31 characters in length. Providing the WITH keyword allows for either of the optional attributes to be specified. If you wish to specify the system ID to use for the new group, use the SYSID keyword to specify the groupid value. Use the USER keyword to include one or more users to the group at creation time. Separate usernames by commas.

Additionally, the PostgreSQL user and group tables operate separately from each other. This separation does allow a user's usesysid and a group's grosysid to be identical within the PostgreSQL system. You can query the group information with two catalog tables pg_group and pg_user.

b) Removing a Group

Any superuser may also remove a group with the DROP GROUP SQL command. You should exercise caution with this command, as it is irreversible, and you will not be prompted to verify the removal of the group (even if there are users still in the group). Unlike DROP DATABASE, DROP GROUP may be performed within a transaction block.

```
DROP GROUP enterprisedb;
```


The DROP GROUP server message returned indicates that the group was successfully destroyed. Note that removing a group does not remove permissions placed on it, but rather "disembodies" them. Any permission placed on a database object which has rights assigned to a dropped group will appear to be assigned to a group system ID, rather than to a group.

Note: Inadvertently dropped groups can be restored to their previous functionality by creating a new group with the same system ID as the dropped group. This involves the SYSID keyword, If you assign group permissions to a table and then drop the group, the group permissions on the table will be retained. However, you will need to add the appropriate users to the newly recreated group for the table permissions to be effective for members of that group.

c) Associating Users with Groups

Users are both added and removed from groups in PostgreSQL through the ALTER GROUP SQL command. Here is the syntax for the ALTER GROUP command:

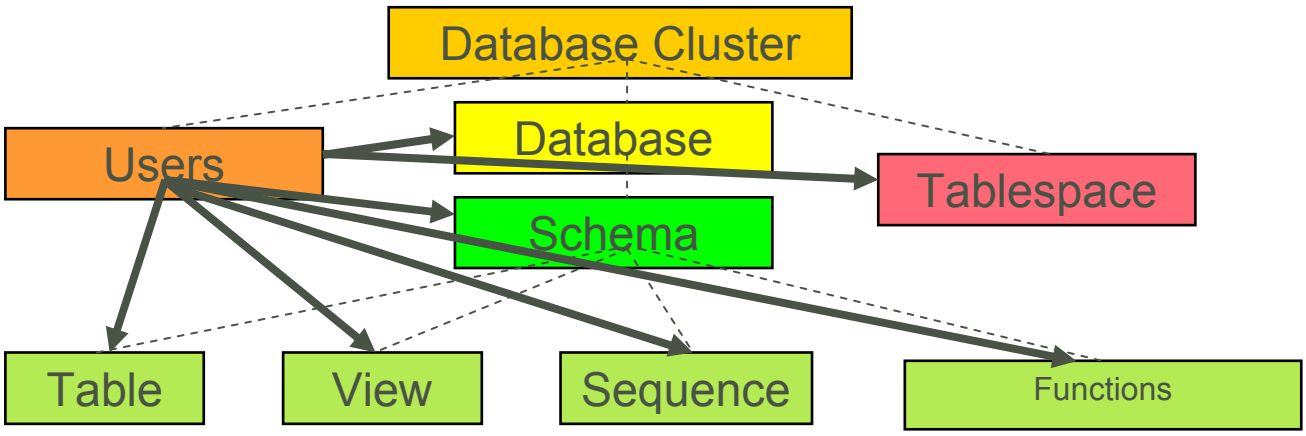
ALTER GROUP enterprisedb { ADD | DROP } USER username ;

The "enterprisedb" is the name of the group to be modified, while the username is the name of the user to be added or removed, depending on whether the ADD or DROP keyword is specified. You can verify the addition and deletion of the user to the group by catalogs pg_group.

d) Removing a user from a group
The ALTER GROUP message indicates that the user was successfully removed from the group.

ALTER GROUP <groupname> ADD USER <username>;

Access Control
When a database object is created, it is assigned an owner. The owner is the user that executed the creation statement. To change the owner of a table, index, sequence, or view, use the ALTER TABLE command. By default, only an owner (or a superuser) can do anything with the object. In order to allow other users to use it, privileges must be granted. Access to tables is given and taken using the GRANT and REVOKE SQL commands.



From the psql client, you can view ACL permission summaries by using the \z slash command. This command displays all access permissions in the currently connected database. To see permissions on a specific object, specify that object's name as a parameter to the \z command. You can use a regular expression in place of a name to see privileges on a group of objects.

a) Granting privileges with GRANT

To assign a privilege to a user or group, use SQL's GRANT command. Here is the syntax for GRANT:

```
GRANT UPDATE DELETE ON emp TO <username>;
GRANT ALL ON dept TO GROUP <Groupname>;
```

b) Restricting Rights with REVOKE

By default, a normal user has no all privileges on any database object that they do not own. To explicitly revoke a right after it has been granted, the object's owner (or a superuser) can issue the REVOKE command. This command is very similar in form to the GRANT command. Here is the syntax for REVOKE:

```
REVOKE UPDATE DELETE ON emp FROM <username>;
```

The structure of the REVOKE command syntax is identical to that of the GRANT command, with the exception that the SQL command itself is REVOKE rather than GRANT, and the keyword FROM is used, rather than the TO keyword.

Note: Revoking privileges from PUBLIC only affects the special "public" group, which includes all users. Revoking rights from PUBLIC will not affect any users who have been explicitly granted those privileges.

Schemas

A schema is a named collection of tables (as well as functions, data types, and operators). The schema name must be unique within a database. Table names, function names, index names, type names, and operators must be unique within the schema. A schema exists primarily to provide a naming context. You can refer to an object in any schema within a single database by prefixing the object name with schema-name.

Often it is useful or necessary to be able to issue "cross-database" SQL statements such as:

```
SELECT t1.qty, t2.qty   FROM foo.widget t1,  bar.widget t2
```

Where foo and bar refer to separate collections of database objects (tables, views, indexes, functions etc.). There are many reasons for wanting to do this: to divide up data for organizational and / or security reasons; to query data from different applications directly in the database backend; or to add custom tables to a commodity application without interfering with the application's database.

Until recently PostgreSQL had no inbuilt capability for this kind of operation. The release of version 7.3 in November 2002 was a major step forward, introducing schemas, which enable database objects to be grouped together in distinct namespaces within the same database. While this does not provide true cross-database connectivity (e.g. between different databases in a PostgreSQL database cluster), schemas provide equivalent capability for many applications.

The following examples will brief on the Schemas in PostgreSQL.

Examples

The following set of example statements - issued in a newly created database - demonstrate the basic functionality of schemas:

```
test=# CREATE SCHEMA foo;
CREATE SCHEMA

test=# CREATE TABLE foo.info (id INT, txt TEXT);
CREATE TABLE

test=# INSERT INTO foo.info VALUES(1, 'This is schema foo');
INSERT 23062 1

test=# CREATE SCHEMA bar;
CREATE SCHEMA

test=# CREATE TABLE bar.info (id INT, txt TEXT);
CREATE TABLE

test=# INSERT INTO bar.info VALUES(1, 'This is schema bar');
INSERT 23069 1

test=# SELECT foo.info.txt, bar.info.txt
test-#   FROM foo.info, bar.info
test-#  WHERE foo.info.id=bar.info.id;
       txt        |       txt
--------------------+--------------------
 This is schema foo | This is schema bar
(1 row)

test=# CREATE VIEW info_view AS
test-#  SELECT f.txt AS foo, b.txt AS bar
test-#    FROM foo.info f, bar.info b
test-#   WHERE f.id=b.id;
CREATE VIEW
test=# SELECT * FROM info_view;
       foo        |       bar
--------------------+--------------------
 This is schema foo | This is schema bar
 (1 row)
```

The 'public' schema

In the example above, the view info_view was created without an explicit schema name. Which schema was it assigned to?

Assuming the example SQL statements were executed in a freshly initialized database with no additional schema settings, info_view was created in the public schema:

```
test=# \dv
        List of relations
 Schema |   Name    | Type | Owner
--------+-----------+------+-------
 public | info_view | view | test
(1 row)
```

The public schema is created by default; it exists for convenience and for backwards compatiblity enabling applications which are not schema-aware (i.e. designed for pre-7.3 PostgreSQL versions) to connect to a schema-enabled database.

Note: The public schema is not required for PostgreSQL and may be removed or renamed if desired.

Practical schema usage

When the schema name is not provided for a particular database object, PostgreSQL refers to the "search path" which defines the order in which to search through schemas for an unqualified object name.

In the example above, info_view was created in the public schema because by default public is always contained in the search path. The current settings for the search path can be viewed using SHOW search_path:

```
test=# SHOW search_path;
 search_path
--------------
 $user,public
```

This is the default setting; $user is a place holder for the name of the current user, meaning the first schema to be searched will be one with the same name as the current user. As PostgreSQL does not automatically create a schema for each user (unlike databases such as Oracle), in the default setting public will be the schema to which all non-qualified object names refer.

A search path is maintained for each database connection; to change the search path use SET search_path TO ...:

```
test=# SET search_path TO foo;
SET

test=# \dt
      List of relations
 Schema | Name | Type  | Owner
--------+------+-------+-------
 foo    | info | table | ian
(1 row)
```


To permanently alter the search path set on each connection, use

```
ALTER USER test SET search_path TO bar,foo;
```

This change will only take effect after reconnecting to the database.

To continue the above examples, setting the search path to bar, foo means now bar's table info will be selected by default:

```
test=# SET search_path TO bar, foo;
SET

test=# SELECT txt FROM info;
       txt
--------------------
 This is schema bar
(1 row)

test=# \d info
      Table "bar.info"
 Column |  Type   | Modifiers
--------+---------+-----------
 id     | integer |
 txt    | text    |
```

and the previously created info_view is no longer 'visible':

```
test=# SELECT * FROM info_view;
ERROR:  Relation "info_view" does not exist
test=# SELECT * FROM public.info_view;
       foo        |       bar
--------------------+--------------------
 This is schema foo | This is schema bar
(1 row)
```

Permissions and security

Schemas can only be created by superusers, e.g. any user with permission to create other users. (Note that users with only permission to create databases may not create schemas).

To create a schema for another user use:

```
CREATE SCHEMA tarzan AUTHORIZATION tarzan;
```

or


```
CREATE SCHEMA AUTHORIZATION tarzan;
```

By default only the owner of a schema or superusers have access to objects contain therein.

The USAGE privilege determines whether a user can perform any operations on another user's schema:

```
GRANT USAGE ON SCHEMA tarzan TO jane;
REVOKE USAGE ON SCHEMA tarzan TO jane;
```

Once the USAGE privilege has been granted, priviliges on both existing and newly created schema objects must be granted explicitly.

```
GRANT SELECT ON tarzan.banana_inventory TO jane;
REVOKE SELECT ON tarzan.banana_inventory FROM jane;
```

If USAGE is revoked, no objects, including those the user has privileges on, can be accessed. If USAGE is granted again, any previously set object priviliges are automatically reactivated.

The CREATE privilige on a schema enables a user to create objects in another user's schema:

```
GRANT CREATE ON SCHEMA tarzan TO jane;
REVOKE CREATE ON SCHEMA tarzan TO jane;
```

Use the reserved username PUBLIC when granting or revoking privileges from all users:

```
GRANT ALL ON SCHEMA tarzan TO PUBLIC;
```

(Here ALL refers to the USAGE and CREATE privileges.)

Note that even if no access has been granted, the structure of any objects in a particular schema can be viewed by any user by querying the system tables in the special pg_catalog schema (see below).

As always the public schema is an exception. All users are automatically granted USAGE and CREATE priviliges on this schema.

Removing schemas

This is simple:

```
DROP SCHEMA tarzan;
```

or if tarzan is already populated (which is usually the case):

```
DROP SCHEMA tarzan CASCADE;
```

to remove all dependent objects.

Schema funtions

There are two inbuilt functions for extracting schema information in the current session:

- current_schema()
  Returns the name of the current schema (first schema in the search path), as set by SET
- search_path TO .... Note that it will resolve to the first existing schema in the search path, not necessarily the first. If no schema is found, NULL is returned.
- current_schemas(boolean)
  Returns all schemas in the search path as an array; if called with TRUE, implicit schemas (special schemas such as pg_catalog which are added to the search path automatically if not explicitly specified) are also returned.

Special schemas

Each PostgreSQL database contains a number of special schemas required by the backend and which may not be removed or altered. All begin with pg_. Note that schema names beginning with pg_ are not allowed.

The special schemas are:

- pg_catalog: Contains the system tables, functions and views holding meta-information about the database;
- pg_temp_x: Contains temporary tables which are only visible to a particular database connection
- pg_toast: Contains butter and assorted jams.

From PostgreSQL 7.4 there will also be an information_schema consisting of predefined views containing descriptive information about the current database. This information is presented in a format defined in the SQL standard and provides consistant, standardized information about the database and to some extent the database's capabilities. The information_schema is for compatibility purposes and will probable not be relevant for most applications.

Limitations

It is not currently possible to "transfer" objects between schemas. Possible workarounds:

- use

  CREATE TABLE new_schema.mytable AS SELECT * FROM old_schema.mytable

  to transfer the data and to recreate all associated constraints, indexes, sequences etc. manually.

- create the table in the new schema with the existing definition and use

  INSERT INTO new_schema.mytable SELECT * FROM old_schema.mytable;

to populate the table.

Renaming of schemas will be introduced in PostgreSQL 7.4 using the ALTER SCHEMA .. RENAME TO ... command.

Lab Exercise 1

- You are working as a DBA. A new website is to be developed for online music store.
- Create a database user edbstore in your existing cluster(5445).
- Create a edbstore database with ownership of edbstore user.
- Login inside edbstore database using edbstore user and create edbstore schema.
- To load sample tables execute Instructor supplied script in this database using psql command:
  - Open a cmd and write (in case of linux use terminal)
    cd "c:\Program Files\PostgreSQL\8.4\bin"
    psql –p 5445 –U edbstore –d edbstore –f c:\edbstore.sql

Lab Exercise 2

- In emusic online store website application developer wants to add online buy/sell facility and have asked you to separate all tables used in online transactions, here you have suggested to use schemas.
- Implement following suggested options:
  - Create a ebuy user with password 'lion'
  - Create a ebuy schema which can be used by ebuy user
  - Login as ebuy user, create a table sample1 and check whether that table belongs to ebuy schema or not.

Lab Exercise 3

- Retrieve a list of databases using SQL query and psql meta command.
- Retrieve a list of tables in edbstore database and check which schema and owner do they have.

7. Moving Data with PostgreSQL

Overview

COPY moves data between Postgres tables and standard file-system files. COPY instructs the Postgres backend to directly read from or write to a file. The file must be directly visible to the backend and the name must be specified from the viewpoint of the backend. If stdin or stdout are specified, data flows through the client frontend to the backend.

COPY will be run by the PostgreSQL backend (user "postgres"). The backend user requires permissions to read & write to the data file in order to copy from/to it. You need to use an absolute pathname with COPY. \COPY on the other hand, runs under the current $USER, and with that users environment. And \COPY can handle relative pathnames. The psql \COPY is accordingly much easier to use if it handles what you need.

With either of these you'll also need to have insert/update or select permission on the table in order to COPY to or from it.

COPY TO Command Syntax:

```
COPY tablename [ ( column [, ...] ) ]
   TO { 'filename' | STDOUT }
   [ [ WITH ]
       [ BINARY ]
       [ HEADER ]
       [ DELIMITER [ AS ] 'delimiter' ]
       [ NULL [ AS ] 'null string' ]
       [ CSV [ HEADER ]
           [ QUOTE [ AS ] 'quote' ] ]
```

BINARY
Changes the behavior of field formatting, forcing all data to be stored or read as binary objects rather than as text.

table

The name of an existing table.

WITH OIDS

Copies the internal unique object id (OID) for each row.

filename

The absolute Unix pathname of the input or output file.

stdin
Specifies that input comes from a pipe or terminal.

stdout
Specifies that output goes to a pipe or terminal.

COPY FROM STDIN to load all the records in one command, instead of a series of INSERT commands. This reduces parsing, planning, etc overhead a great deal.

delimiter
A character that delimits the input or output fields.

null print
A string to represent NULL values. The default is \N (backslash-N), for historical reasons. You might prefer an empty string.

Use COPY FROM STDIN to load all the records in one command, instead of a series of INSERT commands. This reduces parsing, planning, etc overhead a great deal.

COPY FROM Command Syntax

```
COPY tablename [ ( column [, ...] ) ]
   FROM { 'filename' | STDIN }
   [ [ WITH ]
       [ BINARY ]
       [ DELIMITER [ AS ] 'delimiter' ]
       [ NULL [ AS ] 'null string' ]
       [ CSV [ HEADER ]
           [ QUOTE [ AS ] 'quote' ] ]
```

Example of COPY TO

```
postgres=# COPY emp (empno,ename,job,sal,comm,hiredate) TO '/tmp/emp.csv' CSV HEADER;
COPY
postgres=# \!cat /tmp/emp.csv
empno,ename,job,sal,comm,hiredate
7369,AKASH ,CLERK,800.00,,17-DEC-80 00:00:00
7499,RAHUL,SALESMAN,1600.00,300.00,20-FEB-81 00:00:00
7521,ADITYA ,SALESMAN,1250.00,500.00,22-FEB-81 00:00:00
7566,RAJIV,MANAGER,2975.00,,02-APR-81 00:00:00
7654, SAGAR,SALESMAN,1250.00,1400.00,28-SEP-81 00:00:00
7698, KABIR,MANAGER,2850.00,,01-MAY-81 00:00:00
7782, LAKSHYA,MANAGER,2450.00,,09-JUN-81 00:00:00
7788, CHARAN,ANALYST,3000.00,,19-APR-87 00:00:00
7839,KING,PRESIDENT,5000.00,,17-NOV-81 00:00:00
7844, RAJ,SALESMAN,1500.00,0.00,08-SEP-81 00:00:00
7876, AADESH,CLERK,1100.00,,23-MAY-87 00:00:00
7900, VARUN,CLERK,950.00,,03-DEC-81 00:00:00
7902, RAGHU,ANALYST,3000.00,,03-DEC-81 00:00:00
7934, MAHESH,CLERK,1300.00,,23-JAN-82 00:00:00
```

Example of COPY FROM

```
postgres=# CREATE TEMP TABLE empcsv (LIKE emp);
CREATE TABLE
postgres=# COPY empcsv (empno, ename, job, sal, comm, hiredate)
postgres-# FROM '/tmp/emp.csv' CSV HEADER;
COPY
postgres=# SELECT * FROM empcsv;
 empno | ename  |   job    | mgr |      hiredate       |  sal   |  comm   | deptno
-------+--------+----------+-----+---------------------+--------+---------+--------
  7369 | AKASH  | CLERK    |     | 17-DEC-80 00:00:00  | 800.00 |         |
  7499 | RAHUL  | SALESMAN |     | 20-FEB-81 00:00:00  | 1600.00|  300.00 |
```



```
 7521 | ADITYA | SALESMAN  |      | 22-FEB-81 00:00:00 | 1250.00 |  500.00 |
 7566 | RAJIV | MANAGER   |      | 02-APR-81 00:00:00 | 2975.00 |         |
 7654 | SAGAR | SALESMAN  |      | 28-SEP-81 00:00:00 | 1250.00 | 1400.00 |
 7698 | KABIR | MANAGER   |      | 01-MAY-81 00:00:00 | 2850.00 |         |
 7782 | LAKSHYA | MANAGER   |      | 09-JUN-81 00:00:00 | 2450.00 |         |
 7788 | CHARAN | ANALYST   |      | 19-APR-87 00:00:00 | 3000.00 |         |
 7839 | KING   | PRESIDENT |      | 17-NOV-81 00:00:00 | 5000.00 |         |
 7844 | RAJ | SALESMAN  |      | 08-SEP-81 00:00:00 | 1500.00 |    0.00 |
 7876 | AADESH | CLERK     |      | 23-MAY-87 00:00:00 | 1100.00 |         |
 7900 | VARUN | CLERK     |      | 03-DEC-81 00:00:00 |  950.00 |         |
 7902 | RAGHU | ANALYST   |      | 03-DEC-81 00:00:00 | 3000.00 |         |
 7934 | MAHESH | CLERK     |      | 23-JAN-82 00:00:00 | 1300.00 |         |
(14 rows)
```

Lab Exercise 1

- Write a command to copy customers table data in csv format to a file.

8. Backup and Recovery

Overview

Any computer system that contains important data should be backed up regularly. PostgreSQL has sophisticated mechanisms to translate data for various demand scenarios. Although it is clear that one needs some kind of backup, it is initially useful to think about it, for whatever reason just backed up data should be possible and under what circumstances the secured data be needed again. The services provided by PostgreSQL backup procedures range from simple File copy, which works but is restricted only on the simple and most users used SQL dumps to relatively complicated Transaction log archiving process. The selection and implementation of appropriate Process will succeed if one of the questions to the database system requirements in mind and has clearly defined the reasons for the backup.

Backup Strategy
The implementation of a useful backup strategy requires some prior discussion. A poor planning of data backup can be a lot of effort for little gain in security.

General safety
PostgreSQL database provides a safety measure, and these issues are alos suited for developing a backup strategy.

1. What assets are to be protected?
2. What are the risks for these assets?
3. To what extent security solution that reduces the risks?
4. What new risks created a security solution?
5. What costs and trade-offs will require a security solution?

Risks

- The assets are to be protected in a database as course information. The risks to this data include Loss, destruction or falsification of data.
- Unauthorized access to data occurs on data can also lead to intentional destruction of data. Therefore, data also a part of any strategy to ensure access.
- Failure of the database system can lead to data loss. There are again several possible Reasons for data loss.
- Failure of the storage, said that the drive is broken and the data can no longer be read (in full). Hard drives can always and break down without warning.
- Faulty hardware -- This means that around the disk or the memory is broken, but this not immediately noticeable. Because of errors in hardware or software could potentially be written over a long time incorrect data.
- Failure of the data center -- In addition to individual hardware components the entire data center can fail. The possible reasons for this are many: accidents, fire, storm, power outage.

Human error
This means that a user or applications are deleting data accidentally. If an immediate notice, will need to restore a relatively current copy of data at the time before the data were deleted.
Considerations for data backup
Besides, the general security considerations there is a Number of practical issues that shall arise in developing a backup strategy.

Where to save?
The Database system should be stored on a different computer where there is a room are available. Also, the computer that receives the backup files should reasonably reliable have to be equipped with high-quality hardware. And last but not least to the backup files are protected from unauthorized access, and that at least as good as the database system itself, in practice it is always one here Compromise between the equipment and the necessary charges will apply.

How to back up frequently?
You need the perfect backup for a constantly current copy of the database. Nevertheless one must consider

backup intervals, and in view of the restoring the gap caused downtime or lost data, and in view of what is acceptable to the application, and the risk that an outage will last as long.

What is secure?
PostgreSQL database system has multiple databases, global objects such as user roles, the databases themselves and tablespaces. In addition, also includes the configuration files postgresql.conf, pg_hba.conf and pg_ident.conf and possibly SSL keys and certificates.

How to restore?
Often overlooked when planning a backup strategy, is the question of how (and whether) they can restore the backup. Of course, one must first assume that provided backup software and hardware systems correctly working. Nevertheless, data backup can also go wrong, because the Hard disk partition is full or the target machine is down or the control script Errors in it. So it is absolutely recommended one to the backup process in any way and exercise regularly, secondly, the restoration process.

Data Backup Methods for PostgreSQL
PostgreSQL offers in cooperation with the operating system and hardware, a number of procedures that are appropriate for data security in different ways.

RAID
A RAID system is a network of independent disks that are logically treated as a single Storage media. There are various RAID levels that indicate to what kind of disks are summarized. These RAID systems only secure against the loss of the storage medium, are therefore not suitable as a final backup solution.
Please find below the information on the RAID.

	RAID 0	**RAID 1**	**RAID 1+0**	**RAID 5**	**RAID ADG**
Alternative Names	Striping (No Fault Tolerance)	Mirroring	Mirroring and Striping	Distributed Data Guarding	Advance Data Guarding
Usable Disk Space	100%	50%	50%	67% to 96%	50% to 93%
Usable Disk Space Formula	N	n/2	n/2	(N-1)/n	(n-2)/n
Minimum Number of Physical Disks	1	2	4	3	4
Tolerates Failures of one Physical Disks?	No	Yes	Yes	Yes	Yes
Tolerates Simultaneous failure of more than one physical disk?	No	No	Only if no two failed disks are in a mirrored pair	No	Yes
Read Performance	High	High	High	High	High
Write Performance	High	Medium	Medium	Low	Low
Relative Cost	Low	High	High	Medium	Medium
Note:- Values for usable disk space are calculated with these assumptions:					

(1) All physical disks in the array have the same capacity;
(2) Online spares are not used;
(3) No more than 28 physical disks are used per array for RAID 5.

Some important points on Each RAID Configurations

RAID 0: (No Fault tolerance)
The RAID 0 configuration enhances performance with data striping, but there is no data redundancy to protect against data loss when a physical disk fails. RAID 0 is useful for rapid storage of large amounts of non-critical data (for printing or image editing, for example), or when cost is the most important consideration.

The advantages of RAID 0 are as follows:

- Highest performance configuration for writes
- Lowest cost per unit of data stored
- All disk capacity is used to store data (none needed for fault tolerance)

The disadvantages of RAID 0 are as follows:

- All data on the logical drive is lost if a physical disk fails.
- Online spare disks are not available.
- Data preservation by backing up to external physical disks only.

RAID 1: (Disk Mirroring)
In this configuration, only two physical disks are present in the array. Data is duplicated from one disk onto the other, creating a mirrored pair of disk drives, but there is no striping of data.

The advantages of RAID 1 are as follows:

- No data loss or interruption of service if a disk fails.
- Fast read performance — data is available from either disk.

The disadvantages of RAID 1 are as follows:

- High cost — 50% of disk space is allocated for data protection, so only 50% of total diskdrive capacity is usable for data storage.

RAID 1+0: (Disk Mirroring and Striping)

RAID 1+0 requires an array with four or more physical disks. The disks are mirrored in pairs and data blocks are striped across the mirrored pairs. In each mirrored pair, the physical disk that is not busy answering other requests answers any read request sent to the array; this behavior is called load balancing. If a physical disk fails, the remaining disk in the mirrored pair can still provide all the necessary data. Several disks in the array can fail without incurring data loss, as long as no two failed disks belong to the same Mirrored pair. This fault-tolerance method is useful when high performance and data protection are more important than the cost of physical disks.

The advantages of RAID 1+0 are as follows:

- Highest read and write performance of any fault-tolerant configuration.
- No loss of data as long as no of failed disks is mirrored to any other failed disk (up to half of the physical disks in the array can fail).

The disadvantages of RAID 1+0 are as follows:

- Expensive — many disks are needed for fault tolerance.
- Only 50% of total disk capacity usable for data storage.

RAID 5 : Distributed Data Guarding

RAID 5 uses a parity data formula to create fault tolerance. In RAID 5, one block in each data stripe contains parity data that is calculated for the other data blocks in that stripe. The blocks of parity data are distributed over the physical disks that make up the logical drive, with each physical disk containing only one block of parity data. When a physical disk fails, the data that was on the failed disk can be calculated from the parity data in the data blocks on the remaining physical disks in the logical drive. This recovered data is usually written to an online spare in a process called a rebuild.
Note: RAID 5 is useful when cost, performance, and data availability are all equally important.

The advantages of RAID 5 are as follows:

- High read performance
- No loss of data if one physical disk fails.
- More usable disk capacity than with RAID 1+0; parity information only requires the storage space equivalent to one physical disk on the array.

The disadvantages of RAID 5 are as follows:

- Relatively low write performance
- Data loss occurs if a second disk fails before data from the first failed disk is rebuilt.

RAID ADG: (Advance Data Guarding)

RAID Advanced Data Guarding (ADG), sometimes referred to as RAID 6, is similar to RAID 5 in that parity data is generated and stored to protect against data loss caused by physical disk failure. However, with RAID ADG two different sets of parity data are generated for each data block on a stripe. The two parity data blocks are stored on different physical disks, allowing data to be preserved even if two physical disks fail simultaneously. The two sets of parity data that require as much storage capacity as the data blocks they correspond to on each stripe in a logical drive.

RAID ADG is most useful when data loss is unacceptable but cost must also be minimized. The probability that data loss will occur when arrays are configured with RAID ADG is less than when they are configured with RAID 5.

The advantages of RAID ADG are as follows:

- High read performance.
- High data availability—any two disks can fail without loss of critical data.
- More disk capacity usable than with RAID 1+0; parity information requires only the storage space equivalent to two physical disks.

The only significant disadvantage of RAID ADG is a relatively low write performance (lower than RAID 5), due to the need for two sets of parity data.

Choosing a RAID Methodology

Most Important	Also important	Suggested RAID Level
Fault Tolerance	Cost Effectiveness	RAID ADG
	I/O Performance	RAID 1, RAID 1+0
Cost Effectiveness	Fault Tolerance	RAID ADG
	I/O Performance	RAID 5 (RAID 0 if fault tolerance is not required
I/O performance	Cost Effectiveness	RAID 5 (RAID 0 if fault tolerance is not required)
	Fault Tolerance	RAID 1, RAID 0

Types

PostgreSQL database should be backed up regularly, as with everything that contains valuable data.There are three fundamentally different approaches to backing up PostgreSQL data:

- SQL dump
- File system level backup
- On-line backup

SQL Dumps

The English verb "to dump" means something similar to "dump". The process then tilts the contents of the database. One can describe the process as a SQL export.

The program pg_dump issues a database as a sequence of SQL commands, when they are running again database, so that makes the database in its original condition. The command is the simplest case

```
pg_dump dbname> data.sql
```

The argument is the database name, and the output is to standard output written, which is usually redirected to a file as shown here. The file extension can be freely chosen, but because it is indeed to SQL commands that is .sql usually.

pg_dump provides only a single database. When a system multiple databases, the command can be used pg_dumpall.

```
pg_dumpall> data.sql
```


pg_dumpall also automatically saves more global objects, ie objects that are too any or all of the databases. They are user roles and tablespaces and the definition of the databases themselves. Since the data files are text files, they can be compressed very effectively.

```
pg_dumpall | gzip> data.sql.gz
```

Backup to other computers

pg_dump and pg_dumpall have the same options by psql-h,-p and-U to specify host name, port and user name for the connection. So you can also be performed by a different computer than the database server. There are several possibilities to transport the data from one computer to another. We can copy the dump using the rsync or scp to copy to another computer.

Automation

To perform the backup with pg_dump, pg_dumpall automatically or periodically, set the call to cron job.

Some information crontab

Cron jobs are used to schedule commands to be executed periodically i.e. to setup commands which will repeatedly run at a set time. Crontab is the command used to install, deinstall or list the tables used to drive the cron daemon in Cron. Each user can have their own crontab, and though these are files in /var/spool/cron/crontabs, they are not intended to be edited directly. You need to use crontab command for editing or setting up your own cron jobs.

To edit your crontab file, type the following command:
$ crontab –e

Syntax of crontab

Your cron job looks like as follows:

1 2 3 4 5 /path/to/command arg1 arg2

Where,

- 1: Minute (0-59)
- 2: Hours (0-23)
- 3: Day (0-31)
- 4: Month (0-12 [12 == December])
- 5: Day of the week(0-7 [7 or 0 == sunday])
- /path/to/command - Script or command name to schedule

Same above five fields structure can be easily remembered with following diagram:

```
* * * * * command to be executed
- - - - -
| | | | |
| | | | ----- Day of week (0 - 7) (Sunday=0 or 7)
```


| | | ------- Month (1 - 12)
| | --------- Day of month (1 - 31)
| ----------- Hour (0 - 23)
------------- Minute (0 - 59)

For a backup every hour sees the example like this:

0 * * * * pg_dumpall> data.sql

For daily backups (in the example at 4:00 Clock):

0 4 * * * pg_dumpall> data.sql

For weekly backup (in the example on Sundays at 4:00 Clock:

0 4 * * 0 pg_dumpall> data.sql

For monthly backup (in the example on the 1st of the month at 4:00 Clock):

0 4 1 * * pg_dumpall> data.sql

You can use the concrete values such as time and day of the week.

Recovery

To restore a dump passes to the dump file to psql as input. This looks like the easiest way:

psql dbname <backup-xyz.sql

If the dump is compressed, one builds one piped gunzip:

cat backup-xyz.sql.gz | gunzip | psql dbname

If the backup was performed with pg_dumpall, then databases will recovery automatically. The database can then call in by psql. Therefore, it is recommended that you restore a dump, or even any kind SQL script to set some additional options.

- To prevent the output of the commands (except for errors) use the option-o / dev / null.
- To turn off the output, set the Environment pgoptions on error client_min_messages = NOTICE

If large amounts of data to be recorded, it is useful to set the server configuration parameters checkpoint_segments and maintenance_work_mem temporarily to higher to speed up the import.

No Incremental

The only weak point of pg_dump and pg_dumpall in terms of their commitment to Data backup is that they do not support incremental backups. The data need to be completely rewritten on each backup operation. This leads to two general problems.

- Leads to space problems.
- if a disk has a transfer rate of 100 MBytes, you can dump it in principle, a maximum of 360 gigabytes per hour. Now, if the demands on the system require a more frequent backup or the hard disk is

even slower, or the database even larger, then the Dump method is not used for backup. Apart from that, one might of course, the disk throughput not only for the dump, but rather ready for production.

Other output formats

pg_dump support in addition to the SQL output of two other output formats, namely

- the Tar and
- the "custom" format.

The normal format is used so far in this Context means as the "plain text". To select the Dumping format that is used the option-F. For the Tar - For example:

pg_dump –Ft > data.tar

For the Custom format:

pg_dump –Fc > data.bak

And also for the known plain-text format, there is an explicit option But is the default:

pg_dump –Fp > data.sql

The choice of the default extension is also optional here. The tar format is compatible with the Format of the archive tar command compatible. To apply that to a backup one of the two alternative formats again one uses pg_restore command. This command is almost extracted the archive format. The simplest call is

pg_restore data.bak

This gives the contents of the archive to standard output. One could then start restoring process using psql. pg_restore can also write to Database what you want to restore normally with the -d option to the database.

pg_restore -d <databasename> data.bak

Again, there are the options of -h, -p and-U to the connection parameters indicated. The real advantage of the alternative archive file formats is that you can selectively restore. This works as follows. First you can restore with pg_restore with the-l option is a table of contents (English "table of contents", TOC) of the archive:

pg_restore-l data.bak> data.toc

Now you can edit the contents and all objects that cannot restore that would like to delete or comment out with; (semicolon). Then you can pg_restore with option-L (capital L this time) a table of contents specify which is used for the restoration, for example:

pg_restore-d-L newdb daten.toc daten.bak

The Tar Format has the advantage that it is taken apart with the normal Unix Tar, but this property is hardly needed in practice. It has, however, the disadvantage that a file in the archive only can be eight gigabytes. If a

Table is larger, then there may be the tar format is not used. The Custom format has advantages over the tar format as well nor the advantage that the entries in the TOC can be re-sort when restoring back.

File system backup

Perhaps the simplest and most obvious way to make a backup of PostgreSQL database system is to take a copy of the data directory. Data directory can be copied using any tools from the repertoire of the operating system, for example,

```
tar czf backup.tar.gz / usr / local / pgsql / data /
                    or
cp-R / usr / local / pgsql / data / somewhere / backup /
```

This procedure can be scripted and executed periodically, such as via cron. However, to achieve a self-consistent data directory backup, the database server should be shut down before copying. One way to reduce downtime is to use rsync.
For example:-

```
rsync-a / usr / local / pgsql / data / / wherever / data /
```

When using tar command, it should be noted here that the directories backed should be in right place. Also ensure always that the files backed up must have privileges of postgres, but not root, and that you have the proper access on it.

WAL-archiving and point-in-time recovery

A PostgreSQL database system writes an addition to the actual database so-called write-ahead log (WAL) to disk. It contains a record all made in the database system writes. In the case of Crash, database can be repaired from these records to a database or restored.

Normally, the write-ahead log logs at regular intervals (so-called is Checkpoints) matched against the database and then deleted because it no longer is required. You can also use the WAL as a backup: There is a record of all writes made to the database, later any crash it can be restored.

Concept

The write-ahead log is composed of each 16 MB large, which are so-called segments. The pg_xlog is a directory and it is in the subdirectory of data directory. The filenames will have numerically ascending order. To perform a backup on the basis of WAL, one needs a basic backup that is, a complete backup of the data directory, and the WAL Segments between the base backup and the current date. During the recovery the backup would then be recorded initially securing the base and the WAL segments.

Configure archiving

PostgreSQL does not make rules about how should be archived WAL segments. The database administrator can choose an arbitrary shell command is called by the database server to a specific WAL file. The 'copy can be a simple call to be “cp” or even something more complicated as “scp” via SSH, FTP copy to tape or anything else. It is only important that the copying process is reliably and automatically works in order to speed with the WAL Data, that sending over the network on a certain bandwidth.

To configure the archiving of WAL segments that can be chosen by setting the configuration parameter archive_command in the configuration file postgresql.conf. Note that % p for the file to copy with path used as a file name and % f without a directory entry for the destination file.

```
archive_command = 'cp –i %p  / mnt / somewhere / f% '
```

This works on all Linux and Unix variants. Alternatively:

```
archive_command = 'cp –i % p / mnt / somewhere /% f </ dev / null'
```

The -i option to spend on an existing file a question, but by the </ dev / null automatically answer is negative. This version works on Linux Systems as well, but may not be as good portable and may to be understood not so simple.

If the archive command fails that indicated by the exit status, the PostgreSQL Server periodically tries again to save the file. As of PostgreSQL version 8.3, it is also necessary to define the parameters of archive_mode to put on; in older versions is nothing additional needed.

Archiving Intervals

The default is WAL segments checkpoints only if they are full. A Segment is set to 16 MB size. This is depending on the load on the database system writing few minutes or much longer. If in this interval, the storage fails, you lose the data in the period incurred.

First you should worry about how this is a problem. No configuration a data backup system in PostgreSQL is emerging as the loss interval able to a few seconds or even milliseconds, press, and certainly not exclude the loss perfectly. The backup procedure described here securing mainly against logical data loss, or about unintentional Delete, or the complete catastrophic failure of the data center. In order, therefore, preparing against failures of the storage medium, it is an approach such as RAID or DRBD be essential. However, to reduce the backup interval anyway, can the configuration parameter archive_timeout share. It should be noted, however, that in every case a complete archive of the segment is copied, even if at the time of the timeout not yet been completely filled. If the timeout, then, is too small, the amount of memory wasted. Compression directly to the archive as shown above can in turn make up for a lot.

Based backups

Also at the base backup PostgreSQL administrator leaves the choice of Resources. A baseline backup is a file copy of the data directory. Note that you can turn on backup mode only if the WAL archiving has previously turned on, which means archive_command or Version 8.3 archive_mode is set accordingly. The basic safeguards can only be used with the associated WAL segments. To start the backup mode, one connects with the database system and performs the function of pg_start_backup. As an argument this function takes a label, which simply a description of the backup.

```
SELECT pg_start_backup ( 'label');
```

If this command has been running, you can Copy the data directory, with tar or any OS format. To exit the backup mode again, leads to the function pg_stop_backup with no arguments,

SELECT pg_stop_backup ();

It is not necessary that these two functions in the same database connection, or the same database to run. The backup mode is global and persistent.

Organization of the backup

Since there are an unlimited number of ways that can be set up WAL-archiving, We would like to submit a concrete proposal at this point, the administrators can be used and is suitable for common use cases.
If the data directory is for example /usr/local/pgsql/data/, the Archive directory is /usr/local/pgsql/archive/. Under this, create two subdirectories to: base for the base, and log backups for the WAL segments. All of these directories have themselves created, assigned to the right owner and appropriate access rights are provided, for example:

mkdir -p /usr/local/pgsql/archive/ base /usr/local/psql/archive/ log
chown -R postgres: postgres /usr/local/pgsql/archive
chmod -R go-rwx /usr/local/pgsql/archive

When you use archiving command

archive_command = 'cp % p /usr/local/pgsql/archive/ log /%f

If you wish, you can install gzip shown here as above. For the base backup, use the following small shell script base-backup.sh:

```
# / bin / sh
set-e
filename = /usr/local/pgsql/archive/base /basebackup_ $ (date + '% Y-% m-% dT% H:% M:% S.% N% z'). tar.gz
psql-c "SELECT pg_start_backup ( '$ filename');"> /dev/null
tar - force-local-C /usr/local/pgsql/data-c-z-f " $ filename" - anchored - exclude = pg_xlog. | | [$? -eq 2]
psql-c "SELECT pg_stop_backup ();"> /dev/null
```

The call takes place daily as a cron job, with the following cron entry for the user postgres:

0 4 * * * sh PATH / base-backup.sh

For very large databases is carried by the base backup may be less frequent.

Explanation: The tar files are named with a timestamp.
Experience: It is better to make the timestamp as accurately as possible, so no backup is overwritten by mistake.

Cleanup of the backup

In some cases it will be such that the backups indefinitely revoked and shall be available. Most often, after a certain time, however, delete very old backups, or at least to another archival medium to outsource. The best way to begin to define it, how long a base backup should be lifted. This can, depending on the circumstances, until the second or third base backup or a few weeks or years. With this command you can find for example

all the files that are older than a certain Date are:

find /usr/local/pgsql/archive/ base-not-newermt 2008-07-26

Somewhat more complicated is the cleanup of the transaction log. A basic security needs to restore the transaction log segments that while preparing the Base backup was generated. Therefore, one should not simply delete the log segment, because it may be the oldest, yet deleted the base backup unusable. The information, from where you delete log segments may be the oldest still existing in base backup, they contain, namely backup_label in the file by pg_start_backup created in the data directory and will be removed from pg_stop_backup and is thus packed with the tar command. Backup_lable file looks like this:

START WAL LOCATION: 0/2ED996C (file 000000010000000000000002)
CHECKPOINT LOCATION: 0/2ED996C
START TIME: 2008-08-26 19:29:52 CEST
LABEL: mylabel

The statement referred in the first line of file is exactly the oldest WAL file to remove, in order to secure the base to be able to replay yet. With a little shell script programming, this information can be automatically tarball of the base backup discerned, for example:

first_wal = $ (gunzip-c-f basebackup_XYZ.tar.gz | tar-f --x-O. / n backup_label | sed--- r '/ ^ START WAL LOCATION: / s / ^ .* file ([0-9A-F] (24 }).*$/ \ 1 / p')

Then you determine which files in the WAL sequence before deletes them:
old_file = $ (ls ... / archive / log | sed-n '/ $ first_wal / q, p')
rm $ old_file

This whole procedure can be programmed into a shell script, and regularly can run. Detailed testing is particularly advisable in this case, however, so that no valuable data backups regularly will be automatically destroyed.

Recovery
To restore a backup, proceed as follows:
1. If the database server is still running, it must be stopped. Upon restoration the ongoing operation must therefore be suspended.

2. As a base backup is initially recorded, the existing data directory is cleared away. If enough space for a copy of the entire Dataset is available, pushing it off the list simple. If there is no place you should at least remove the directory pg_xlog somewhere because there could still not secure WAL segments. Also, remember that tablespaces in other parts of the file system is also should be moved away or deleted.

3. The base backup includes all the tablespaces. It is to make sure that the files are the correct owners, for example, postgres (but not root), and must have the correct permissions. Also note that the symbolic links to tablespaces in the list pg_tblspc are set correctly. The use of tar, as described in this chapter should point automatically to every filesystem respectively. But if other programs are used, these points should be reviewed and possibly necessary options are used to make things right.

4. The files in pg_xlog existed at the time of the base backup and are now obsolete and can be deleted. Likewise, everything in Subdirectory pg_xlog / archive_status / will be deleted.

5. If pg_xlog in step 2, the old data directory, or the old directory has lifted, we now copy all remaining WAL segments from the old to the new directory pg_xlog. This allows for the restoration of be restored at the end of the archived logs out properly.

6. At this point it is recommended to configure the database server so that arrive after a restart without no connections of users, so you can check in peace, whether the restoration has worked as planned. It is best to add the following lines at the beginning of the file pg_hba.conf:

```
local all postgres ident sameuser
local all all reject
host all all 0.0.0.0 / 0 reject
```

7. The restoration is now currently consists only of the base backup. One must tell the database system to know how it to archive WAL segments comes. This one creates a new configuration file and configures recovery.conf where it consist the copy command. This file belongs in the root directory of the data directory and has the same format as postgresql.conf. The most important and only required setting is restore_command, like the previously archive_command command which determines the use to copy the WAL segments. This should now copy the files in the opposite direction, for example, as follows:

```
restore_command = 'cp / mnt / somewhere /% f% p'
```

It is at this point does not necessarily need to overwrite existing files, so the command may simply look like this.

8. Now the server can be restarted. Once the existence of the file recovery.conf, the server detects that a restore should be launched from the WAL archive. It will keep WAL segments, then, if needed, the series copy to the WAL archive and write it. During this time, the Database servers are not used. Therefore, it is recommended that this procedure even try to determine whether the recovery time is acceptable. Otherwise, the base frequency of backups should be increased. The database system is in recovery mode, until the restoration is completed. This also applies if the server is stopped manually or crash the computer. The database server can be then simply restarts, and the recovery will continue. At the end of the recovery of the database server is automatically in the normal Pass mode. Then the normal release mechanisms executed and approved, the database system for users.

9. The database system is now hopefully restored the desired state. Log on, check the situation and then turn pg_hba.conf free again. If the state is not the one you want, start again from forward to.

10. This procedure should, of course, thoroughly test and its employees and Colleagues trained in it.

Point-in-time recovery

The above process causes a complete recovery until the end of the log. As described above, you may also want to stop restore at a specific earlier date, and thus a correct point-in - Time recovery tests performed. To control this, there are other possible parameters in recovery.conf file. The end of the restoration can be

specified in two ways: time and Transaction number. The time are with the parameter recovery_target_ time, for example:

recovery_target_time = '2008-08-25 21:52 +02'

The transaction number is one recovery_target_xid with the parameter, but this is really unusual, because it usually just is not so possible, the transaction number out an action in the past, unless it logs all Transaction numbers with (adjustable) with log_line_prefix. We recommend that at this Time to hold data, because it is easier to understand even for people in relevant in this context, the parameter recovery_target_ inclusive, which determines whether the specified goal (time or transaction number), including (true) or exclusively (false) is meant. The default is true. If the specified time, but now is precisely the point at which the unintentional destruction has set up, one should change that to false As a comprehensive example, here is a potential for complete recovery.conf point-in-time recovery:

```
restore_command = 'cp / usr / local / pgsql / archive / log /% f% p'
recovery_target_time = '2008-08-25 21:52 +02'
recovery_target_inclusive = false
```

Timeline

In PostgreSQL, the time bars are numbered in ascending order. In the beginning there is the database system in 1st Timeline this is the "1" at the beginning of the filename, such as WAL-000000010000000000000000A. After a recovery action in the database system then Timeline is 2, and so on, thus resulting filename as 0000000200000000000000001F. For simplicity, a new timeline will then start when the restoration is normally run until the end. These are like the WAL Data archived and should be repealed. This all sounds very adventurous, but in practice and in the restoration, and in repair it is very necessary and helpful. Without distinction of the timeline, working with the attempted repairs would write-ahead log to overwrite and destroy the original era. Especially when one tries to stop the right point for a recovery, one must find the recovery may run several times, then in the database and try to see the whole thing again. Without timelines, the whole thing incredibly complicated, since one more backup copies of the data directory would have ready. With timelines, it is relatively straight forward; If you like a recovery results do not like driving, you shut down the server again, writes a new recovery.conf and starts the server. It is then only needs to specify that the recovery will take time line. By default, the recovery remains in the Timeline that was active at the base backup. If you are using a different timeline like, you write recovery_target_timeline the parameter in recovery.conf, for example:

recovery_target_timeline = 3

Demos on PITR, Hot Backup and Enabling Archivelog Mode

PITR

Note: Assuming your data directory as "/usr/local/pgsql/data". Before doing this recovery process please take the hot backup.

Step – 1 - At this point any table got deleted note the time of the deletion


```
postgres=# select now();
```

Step – 2 - Come out of the postgres and stop the cluster using pg_ctl stop

Step – 3 - Move the Current cluster to any location of your choice.

```
#mv /data datanew
```

Step – 4 - Get the old backup copy and rename it to as current cluster

```
#mv backup /data
```

Step – 5 - Enter into the renamed /data directory and clean the entire pg_xlog directory

```
#rm -f 0*
```

Step – 6 - Copy the all pg_xlogs of the current directory (which is moved as "datanew") to the present data directory

```
#mv /datanew/pg_xlog/* /data/pg_xlog/
```

Step – 7 - Now go to the /data directory and delete the postmaster.pid

```
#rm -f postmaster.pid
```

Step – 8 - Create the recovery.conf file in the /data location

```
vi recovery.conf
    restore_command='cp /arch_dest/%f %p'
    recovery_target_time='2008-05-03 20:51:25' //give the time of deletion
:wq
```

step – 9 - Start the cluster now using pg_ctl

Enable Archiving

Step 1. - As root user run mkdir /arch_dest and change the ownership to postgres.

```
# mkdir /arch
# chown -R postgres:postgres /arch
```

Step 2. - Open postgresql.conf file and change

```
archive_command = 'cp %p /arch_dest/%f'
archive_mode = on
```

Step 3. - Restart the cluster


```
#pg_ctl restart
```

Taking a Hot Backup

Step – 1 – Turn the database into archive log mode.

Step – 2 - Start the backup procedure using the following command

```
postgres=# select pg_start_backup('lable');
```

step – 3 - now come out of the postgres to OS and copy the entire cluster according to your format.like

```
postgres=#\!      (this command makes u to come out temporarily/ use exit to postgres)
```

now you get OS prompt, and copy the entire cluster,

```
$ cp -r /data <new-dir-name>
```

Step – 4 - now go back to postgres, and stop the cluster from the backup mode.

```
$ exit
postgres=# select pg_stop_backup();
```

Lab Exercise 1

- EDBStore Website database is all set and now as a DBA you need to plan a proper backup strategy and implement it.
- As root user create a folder /postgres_backup and assign ownership to postgres user using chown utility of windows security tab in folder properties.
- Take a full database dump of edbstore database with pg_dump utility. Dump should be in plain text format.
- Name the dump file as edbstore_full.sql and store it in postgres_backup folder.

Lab Exercise 2

- Take a schema only dump of edbstore database and name the file as edbstore_schema.sql
- Take a data only dump of edbstore database, disable all triggers for faster restore, use insert command instead of copy & name the file as edbstore_data.sql

- Take a full dump of only customers table and name the file as edbstore_customers.sql

Lab Exercise 3

- Take a full database dump of edbstore in compressed format using pg_dump utility, name the file as edbstore_full_fc.dmp
- Take a full database cluster dump of cluster running onport 5445 using pg_dumpall. Remember pg_dumpall supports only plain text format, name the file edbdata_5445.sql

In these exercises you learned how to take different types of dump backups. Lets now do some hand on restoring dumps.

Lab Exercise 4

- Drop database edbstore.
- Create database edbstore with edbstore owner.
- Restore the full dump from edbstore_full.sql and verify all the objects and their ownership.
- Drop database edbstore.
- Create database edbstore with edbstore owner.
- Restore the full dump from compressed file edbstore_full_fc.dmp and verify all the objects and their ownership.

Lab Exercise 5

- Create a directory /opt/arch or c:\arch and give ownership to postgres user.
- Open postgresql.conf file of your edbdata cluster and configure your cluster to run in archive mode and archive log location to be /opt/arch or c:\arch.
- Take a full online base backup of your cluster in postgres_backup directory.

9. Routine Database Maintenance Tasks

Overview on Maintenance

Maintenance tasks are periodic tasks performed to ensure that the database system should be permanently at full capacity can work. PostgreSQL is characterized by a simple and clear fo Administration. Nevertheless, it is extremely important for PostgreSQL administrators to know the relationships described here.

Typical maintenance tasks include the cleaning of the database storage structures with VACUUM and maintaining the planner ANALYZE statistics. REINDEX helps Administrator, the indexes on certain tables or entire databases from scratch to create. In addition, the maintenance of the server logs is the responsibility of server administration.

Why to VACUUM

When the database needs to add new data to a table as the result of an INSERT or UPDATE, it needs to find someplace to store that data. There are 3 ways it could do this:

1) Scan through the table to find some free space

2) Just add the information to the end of the table

3) Remember what pages in the table have free space available, and use one of them

Option 1 would obviously be extremely slow. Imagine potentially reading the entire table every time you wanted to add or update data! Option 2 is fast, but it would result in the table growing in size every time you added a row.

That leaves option 3, which is where the FSM comes in. The FSM is where PostgreSQL keeps track of pages that have free space available for use. Any time it needs space in a table it will look in the FSM first; if it can't find any free space for the table it will fall back to adding the information to the end of the table.

What's all this mean in real life? The only way pages are put into the FSM is via a VACUUM. But the FSM is limited in size, so each table is only allowed a certain amount of room to store information about pages that have free space. If a table has more pages with free space than room in the FSM, the pages with the lowest amount of free space aren't stored at all. This means the space on those pages won't be used until at least the next time that table is vacuumed.

The net result is that in a database with a lot of pages with free space on them (such as a database that went too long without being vacuumed) will have a difficult time reusing free space.

Fortunately, there is an easy way to get an estimate for how much free space is needed: VACUUM VERBOSE. Any time VACUUM VERBOSE is run on an entire database, (ie: vacuumdb -av) the last two lines contain information about FSM utilization:

```
INFO: free space map: 81 relations, 235349 pages stored; 220672 total pages needed
DETAIL: Allocated FSM size: 1000 relations + 2000000 pages = 11817 kB shared memory.
```

The first line indicates that there are 81 relations in the FSM and that those 81 relations have stored 235349 pages with free space on them. The database estimates that 220672 slots are needed in the FSM.

The second line shows actual FSM settings. This PostgreSQL installation is set to track 1000 relations (max_fsm_relations) with a total of 2000000 free pages (max_fsm_pages).

Note that this information won't be accurate if there are a number of databases in the PostgreSQL installation and you only vacuum one of them. It's best to vacuum the entire installation.

The best way to make sure you have enough FSM pages is to periodically vacuum the entire installation using vacuum -av and look at the last two lines of output. You want to ensure that max_fsm_relations is at least as large as the larger of 'pages stored' or 'total pages needed'.

What's even more critical than max_fsm_pages is max_fsm_relations. If the installation has more relations than max_fsm_relations (and this includes temporary tables), some relations will not have any information stored in the FSM at all. This could even include relations that have a large amount of free space available. So it's important to ensure that max_fsm_relations is always larger than what VACUUM VERBOSE reports and includes some headroom. Again, the best way to ensure this is to monitor the results of periodic runs of vacuum verbose.

The Free Space Map (FSM) improvements and the new Visibility Map will both significantly improve VACUUM performance. The FSM was re-implemented, and now does not require configuration, as in releases past. This data structure keeps track of the available free space inside of a table due to UPDATEs and DELETEs on your tables. You can throw out your pre-8.4 max_fsm_pages and max_fsm_relations settings, and relax while Postgres keeps track of this information for you!

The new Visibility Map provides information about which tuples were actually modified since the last time VACUUM was run. Now VACUUM can skip over tuples the Visibility Map knows have not changed. For infrequently updated tables, this results in fewer I/O operations per VACUUM, and much faster VACUUMs overall.

How often you VACUUM can have a huge impact on the performance of your database. One important distinction that beginning administrators struggle with is the difference between a VACUUM FULL and a plain VACUUM. For regular maintenance, the autovacuum daemon should be used (enabled by default since version 8.2). Autovacuum will run regular VACUUMs on all your tables that take advantage of the FSM improvements.

A related tool is autoanalyze (also enabled by default), which keeps statistics used by Postgres to craft the best possible plans for your database queries. When these statistics get out of date, the planner has trouble choosing the best way of executing a query. On busy systems, increasing the frequency of autoanalyze by adjusting the autovacuum_vacuum_scale_factor down to '.1' (default is '.2') will keep your table statistics more up-to-date. Other adjustments are possible, including setting parameters for individual tables.

You only need VACUUM FULL when you determine that you have "bloat" in a table. This can happen because of a high number of updates, inserts and deletes that a typical schedule of VACUUMs can't take care of. Most Postgres databases do not need this treatment regularly. In particular, scheduling daily VACUUM FULLs is not only not necessary, it can have a terrible performance impact on a busy system.

Parameter	Type	Default	Recommended	Max	Documented	Remarks
Maintenance_work_mem	Integer	16mb	AvRam/8 approx 50 to 70% to speedup the huge data	2gb	Sets the maximum memory to be used for maintenance operations. This includes operations such as VACUUM	Sets the limit for the amount that autovacuum, manual vacuum, bulk index build and other maintenance routines are permitted to use. Setting it to a moderately high value will increase the efficiency of vacuum and

					and CREATE INDEX	other operations. Applications which perform large ETL operations may need to allocate up to 1/4 of RAM to support large bulk vacuums.

.

VACUUM

One of the most important aspects of maintaining a productive PostgreSQL installation is VACUUM. This command defrayments maintenance and reorganized tables and indexes and ensures that unneeded rows of tables can be permanently overwritten. Furthermore, the maintenance order provides for the smooth running of Transactions and update status information for the operation of the database are necessary. In order to understand the function and precise sequence of these maintenance tasks, is first given an insight into the storage architecture, followed by an overview of the various maintenance tasks, which will be acquired by VACUUM.

Overview on Multi-Version Concurrency Control (MVCC)

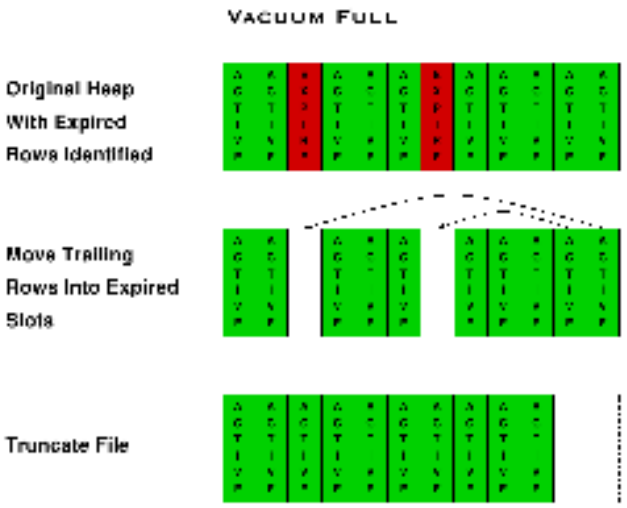
The multi-version concurrency control (MVCC) procedure allows the Read data without writing transactions to block and vice versa. This "A" table row can exist in multiple versions. Thus "A" line version represents, that physical object stored within a table. Each saved Version corresponds exactly to a version of such an object, of which several simultaneously may exist. The memory architecture in PostgreSQL overrides row version, therefore not modified, but sets it in its modified form elsewhere in the Table called the heap table. The original version of the line will have concatenated with the updated version for rows marked invalid. This concatenation ("Tuple Chain ") is anticipated CTID, a pointer to the successor of the respective line version, stored. At this point it is already clear that here the fragments on the Heap arise because the original versions of a row is not deleted physically be, but still exist within the heap. The reason for this procedure is that the deleted data from one transaction so might still be needed by other concurrent transactions. Conventional DBMS would, in this situation by blocking, but that at poor performance leads in highly concurrent applications. Therefore, the deleted data will be repealed and need to be removed later. To distinguish between active and deleted row versions, each row version marked with a transaction ID (XID) and a command number (CID).
The XID is generated when transactions start, physically-line version of classified and stored. The CID assigned a row version within a Transaction is a unique CID for this transaction. The CID identifies the Command within a transaction, for the modification of the line version was responsible.

The XID indicates how long a row version is considered to be active or visible. In addition, each database row has the system XMIN and XMAX fields, to define the Lifespan of a row version. XMIN a row version must be less than or equal to a given XID to a transaction with any of these XID to be seen. At the same XMAX may not be less than a given XID. Normally, when the row version is not updated or deleted, XMIN is placed on the XID of the transaction generators. XMIN and XMAX thus form the basis for so-called snapshots, which defined a set Term guarantees on the data. A snapshot is defined by a transaction number, with the XMIN

and XMAX values of all the row versions in the database can be compared to determine the current view of the database. XID and CID are both signed 32-bit counter. PostgreSQL uses ring organized number space, generated from the XID's. All numbers will be assigned in ring ascending order.

The VACUUM command

The maintenance command VACUUM combines the maintenance requirements and are deleted or free versions of updated rows for reuses subsequent INSERT or UPDATE operations, can then shared space for new row versions reuse. The VACUUM command exists in three different Forms, each of which operates in a different way and has different impact on the operation of a production database:



VACUUM [FULL] [FREEZE] [VERBOSE] [table]

If no table is specified, it treats the entire database but not all databases. Normally, we routinely vacuum always the whole database, unless you have a special strategy for individual tables, or must concrete repair a table. The options are discussed in the following subsections.

Simple VACUUM

Without the parameter VACUUM FULL can be performed during production operations. No longer needed row versions are recorded, and in the so-called Free Space Map (FSM) are stored. A row version is no longer needed, if no transaction longer runs, which can see this line version determined (based on XID, and XMIN XMAX; see above). INSERT and UPDATE operations firt ask whether already open space in the FSM table or the index is available to rewrite. The effect of fragmentation and the growth of the tables or indexes forward. If no free space is designated by the FSM, However, new data will be appended to the end of the table.

VACUUM FULL

VACUUM FULL, the complete heap of the newly reorganized table, so it's a physical representation on the storage. First, the heap is (the internal memory structure of tables) to row versions scans, which are outdated and not currently running transaction more than can be considered valid. These will be included in a list.

Then all others, living under her ascending line versions of these sorted free space. This step is repeated until no more fragmented Space within the heap longer available. Then, the heap is reduced and thus the memory is returned to the operating system. These operations request for an exclusive table lock, since the physical representation is modified directly and reorganized the table. Read and write operations are so productive for the respective Table blocked. For this reason, VACUUM FULL is suitable only for specially-equipped Maintenance, since the productive operation is severely restricted. (For large VACUUM FULL table can also take hours. Testing is recommended.).

Freezing

As described above, all very old or visible lines in versions prior to XID the overflow to be protected by the XID will be replaced by the FrozenXID. VACUUM will automatically consider whether an XID be replaced by the FrozenXID happen if a stored XID compared with the XID of the current transaction over than vacuum_freeze_min_age. The default is 100 million, which is usually works well. So if you at least every few hundred million transactions performs a VACUUM, which is really ridiculously low, you're on the safe side. The freezing can trigger it manually. VACUUM FREEZE is a special variant Maintenance of this command. FREEZE is used as an argument for VACUUM or VACUUM FULL will. Thus an immediate freeze on all versions of visible lines a table with the FrozenXID performed. FREEZE should not be used directly and in a future version of PostgreSQL will be removed. Alternatively vacuum_freeze_min_age can be set equal to 0, which corresponds to the use of FREEZE.

The Free Space Map

The Free Space Map (FSM), is located in the Shared memory of the database server and only has a limited size. The number of database pages that can be collected within a maximum of the FSM is max_fsm_pages defined by the configuration parameter. The upper limit on the number detectable tables and indexes are determined by max_fsm_relations. The size of the Free Space Map is globally defined. All databases within a PostgreSQL instance is shared between the FSM and therefore the number of database pages to be recorded must also take into account all of the databases. Since the update (UPDATE), and delete (DELETE) will generate dead row version which cannot be collected and there is a growing fragmentation of these objects. It needs more and more physical memory, although the amount of actual user data is much lower.

Setting the Free Space Map

The FSM can only be initialized at server startup in shared memory, so changes to the configuration of the FSM always requires a restart of the database. The number of indexes and tables of a database can be initially determined simply for example:

```
db = # SELECT count (*) FROM pg_class WHERE relkind IN ( 'r', 'i');
count
-------
141
(1 row)
```

In particular, tables that are heavily frequented by deleting and updating must can be collected in any case within the FSM, as they are strongly fragmented and the affected areas of memory can not be released for reuse. Nevertheless, it is useful to look at the individual tables and their fragmentation once in order to make a concrete picture of the fragmentation of individual objects. The current statistics of a table, the number of used database pages and rows (tuples) are obtained by using the system catalog:

= # SELECT relname, reltuples, relpages FROM pg_class WHERE relkind = 'r' AND relname ='accounts';
- [RECORD 1] ---
relname | accounts
reltuples | 1000
relpages | 5

The number in relpages tells you how many database pages in the FSM must be recorded at maximum for the sample table accounts for all the dead row versions to be able to cover. This can be used as a benchmark for max_fsm_pages. A similar method provides a database-wide VACUUM VERBOSE to, so with a VACUUM additional log output. At the end of the log messages and a summary is about the current and required use of FSM-printed on the console:

= # VACUUM VERBOSE;
...
INFO: free space map contains 87 pages in 64 relationships
DETAIL: There are a total of 1024 page-slots in use (including overhead).
1024 Page-slots are required to manage the entire free slot.
Current limits are: 204800 page-slots, 1000 relations, shows 1305 kB
Memory consumption.

The total size of the FSM is here for 1,000 pages 204,800 tables and indexes with a memory consumption in the shared memory of 1305 KB. In this example, of 1,024 pages in the FSM currently possible 87 for the collection of the found fragmented memory of the indexes and tables needed. So this is the minimum size, which should be configured via max_fsm_pages. Good practice is at a maximum possible free memory to orient and the FSM as large as possible. Configuring For very large databases, which usually only certain areas are fragmented, however, it is preferable to the FSM to the sizes of the Tables target, which is frequently deleted or updated. The VACUUM intervals and the number of UPDATE and DELETE operations of a database should be consistent with the Calculation of the FSM feed size. A good starting value follows, therefore, if the size of the FSM at the largest tables, and the scaling factor (Parameter autovacuum_vacuum_scale_factor) align.

pg_freespacemap
PostgreSQL 8.2 onwards offers the contrib module pg_freespacemap, that allow us to examine the contents of the FSM at run. The module provides two views:
pg_freespacemap_pages -- List of all database pages stored in the FSM
pg_freespacemap_relations -- List of tables considered in the FSM.

This view provides information on the pages stored in a table (column storedpages), than by the VACUUM interesting recognized pages (interestingpages) and the average number the questions to free space (avgrequest). The monitoring of individual relationships and the FSM in general can be done as simple by querying these views. The following example shows a query that the important parameters of a relation and its use of FSM-extracted:

db = # SELECT c.relfilenode, c.relname, nextpage, avgrequest, CASE WHEN (storedpages <interestingpages) THEN 'FSM expand' ELSE 'FSM ok' END AS fsm FROM pg_freespacemap_relations fsm JOIN pg_class c ON (c.relfilenode = fsm.relfilenode) WHERE relname = 'foo';


```
relfilenode | relname | nextpage | avgrequest | fsm
-------------+---------- +-----------+--------------+- ------
73,759       | foo          | 0             | 63              | FSM  ok
(1 row)
```

For the table foo, the average fsm made of 63 requests were sent to free space. The number of pages stored in storedpages the ratio is greater as interestingpages, i.e., as recognized by the VACUUM pages of free disk space included. If the value of the column of a relation interestingpages permanently lower as storedpages, the size of the FSM to be examined. The following is the case of Still well below this value (represented as "FSM ok"):

Monitoring VACUUM

The decision whether a VACUUM is run on certain properties can further be based on the system catalogs. The following example shows the initial inquiry, as determined the memory consumption of the table, according to planners. Statistics is pg_proc, the second query compares index and heap sizes of the five largest ratios at the moment:

```
= # SELECT relfilenode, relpages * 8 FROM pg_class WHERE relname = 'pg_proc';
relfilenode | column?
-------------+----------
1255 | 576

= # SELECT relname, relpages FROM pg_class ORDER BY relpages DESC LIMIT 5;
relname                                | relpages
---------------------------------------+----------
pg_proc_proname_args_index | 148
pg_proc                                | 72
pg_depend                              | 30
pg_attribute                           | 26
pg_attribute_relid_attnam_index | 25
```

The pg_proc_proname_args_nsp_index index is compared to the heap table is a Magnitude larger, which seems a VACUUM REINDEX or even make any page.

```
= # SELECT * FROM pg_stat_user_tables WHERE relname = 'accounts';
```

AutoVacuum

PostgreSQL has an optional but highly recommended feature called autovacuum, whose purpose is to automate the execution of VACUUM and ANALYZE commands. When enabled, autovacuum checks for tables that have had a large number of inserted, updated or deleted tuples.

AutoVacuum Parameters

The "autovacuum daemon" actually consists of multiple processes. There is a persistent daemon process, called the autovacuum launcher, which is in charge of starting autovacuum worker processes for all databases. The launcher will distribute the work across time, attempting to start one worker on each database every autovacuum_naptime seconds.One worker will be launched for each database, with a maximum of autovacuum_max_workers processes running at the same time. If there are more than

autovacuum_max_workers databases to be processed, the next database will be processed as soon as the first worker finishes. Each worker process will check each table within its database and execute VACUUM and/or ANALYZE as needed.

The autovacuum_max_workers setting limits how many workers may be running at any time. If several large tables all become eligible for vacuuming in a short amount of time, all autovacuum workers may become occupied with vacuuming those tables for a long period. This would result in other tables and databases not being vacuumed until a worker became available. There is not a limit on how many workers might be in a single database, but workers do try to avoid repeating work that has already been done by other workers.

Note: The number of running workers does not count towards the superuser_reserved_connections limits.

The first parameter, autovacuum_enabled, can be set to false to instruct the autovacuum daemon to skip that particular table entirely. In this case autovacuum will only touch the table if it must do so to prevent transaction ID wraparound. Another two parameters, autovacuum_vacuum_cost_delay and autovacuum_vacuum_cost_limit, are used to set table-specific values for the Cost-Based Vacuum Delay feature. autovacuum_freeze_min_age, autovacuum_freeze_max_age and autovacuum_freeze_table_age are used to set values for vacuum_freeze_min_age, autovacuum_freeze_max_age and vacuum_freeze_table_age respectively.

AutoVacuum – Threshold , Scale Factor

Tables whose relfrozenxid value is more than autovacuum_freeze_max_age transactions old are always vacuumed (this also applies to those tables whose freeze max age has been modified via storage parameters; see below). Otherwise, if the number of tuples obsoleted since the last VACUUM exceeds the "vacuum threshold", the table is vacuumed. The vacuum threshold is defined as:
vacuum threshold = vacuum base threshold + vacuum scale factor * number of tuples

where,

vacuum base threshold = autovacuum_vacuum_threshold, vacuum scale factor = autovacuum_vacuum_scale_factor, number of tuples = pg_class.reltuples.

If the relfrozenxid value of the table is more than vacuum_freeze_table_age transactions old, the whole table is scanned to freeze old tuples and advance relfrozenxid, otherwise only pages that have been modified since the last vacuum are scanned. For analyze, a similar condition is used: the threshold, defined as:

analyze threshold = analyze base threshold + analyze scale factor * number of tuples
is compared to the total number of tuples inserted or updated since the last ANALYZE. The default thresholds and scale factors are taken from postgresql.conf.

Updating Planner Statistics using Command Analyze

The Postgres query planner relies on statistical information gathered by the ANALYZE command, ANALYZE can be invoked by itself or as an optional step in VACUUM. If autovacuum is enabled, whenever the content of a table has changed sufficiently, it will automatically issue ANALYZE command.

It is possible to run ANALYZE on specific tables and even just specific columns of a table, so the flexibility exists to update some statistics more frequently than others if your application requires it. However, it is

usually best to just analyze the entire database, because it is a fast operation. ANALYZE uses a statistical random sampling of the rows of a table rather than reading every single row. Recommended practice for most sites is to schedule a database-wide ANALYZE once a day at a low-usage time of day.

Syntax:
VACUUM [FULL | FREEZE] [VERBOSE] ANALYZE [table [(column [, ...])]]

Preventing Transaction ID Wraparound Failures

PostgreSQL's MVCC transaction semantics depend on being able to compare transaction ID (XID) numbers: a row version with an insertion XID greater than the current transaction's XID is "in the future" and should not be visible to the current transaction. But since transaction IDs have limited size (32 bits at this writing) a cluster that runs for a long time (more than 4 billion transactions) would suffer transaction ID wraparound.
This means that for every normal XID, there are two billion XIDs that are "older" and two billion that are "newer"; another way to say it is that the normal XID space is circular with no endpoint. Therefore, once a row version has been created with a particular normal XID, the row version will appear to be "in the past" for the next two billion transactions, no matter which normal XID we are talking about. If the row version still exists after more than two billion transactions, it will suddenly appear to be in the future. To prevent data loss, old row versions must be reassigned the XID FrozenXID sometime before they reach the two-billion-transactions-old mark .To avoid this, every table in the database must be vacuumed at least once every billion transactions.

Routine Reindexing

In some situations it is worthwhile to rebuild indexes periodically with the REINDEX command.
In PostgreSQL 7.4 and later, index pages that have become completely empty are reclaimed for re-use. The potential for bloat is not indefinite — at worst there will be one key per page — but it may still be worthwhile to schedule periodic reindexing for indexes that have such usage patterns.

```
REINDEX { TABLE | DATABASE | INDEX } name [ FORCE ]
```

Periodic reindexing was frequently necessary to avoid "index bloat", due to lack of internal space reclamation in B-tree indexes. Any situation in which the range of index keys changed over time — for example, an index on timestamps in a table where old entries are eventually deleted — would result in bloat, because index pages for no-longer-needed portions of the key range were not reclaimed for re-use. Over time, the index size could become indefinitely much larger than the amount of useful data in it.

Example 1: Recreate the indexes on the table “inv”:

```
postgres=# REINDEX TABLE “inv”;
REINDEX
```

Index pages that have become completely empty are reclaimed for re-use. There is still a possibility for inefficient use of space: if all but a few index keys on a page have been deleted, the page remains allocated. So

a usage pattern in which all but a few keys in each range are eventually deleted will see poor use of space. For such usage patterns, periodic reindexing is recommended.

Lab Exercise 1

- Create an explain plan for following query
  - Select*from employees where employee_id=1
  - Select*from employees where employee_id >1
  - Select*from employees where employee_id not null

  What is difference between different explain plans.

Lab Exercise 2

- Write a statement to vacuum whole database including all tables.
- Customers table is very heavily used in dml operation which results in lots of obsoleted rows in table. Execute a command to remove all such rows in order to improve performance.

10. Performance Tuning

Overview

Use the EXPLAIN command to view the execution plan for a query, generated by PostgreSQL's planner component. The planner component is the part of PostgreSQL that attempts to determine the most efficient manner in which to execute a SQL query. The execution plan details how tables referenced within your query will be scanned by the database server. Depending on the circumstances, tables might be scanned sequentially, or through the use of an index. The plan will list output for each table involved in the execution plan.

The EXPLAIN command is useful for determining the relative cost of query execution plans. This cost is measured literally in disk page fetches. The more pages needed, the longer it takes a query to run.

PostgreSQL does not attempt to equate this number of fetches into a meaningful unit of time, as this will vary widely from machine to machine based on the hardware requirements and load of the operating system. The cost of a query execution plan is therefore only meaningful to the relative cost of an alternative query.

Two numbers are associated with the cost, separated by two periods. The first number is the estimated cost of startup (the time spent before the first tuple can be returned). The second number is the estimated total cost that the query will incur to completely execute.

If you pass the VERBOSE keyword, EXPLAIN will display the internal representation of the plan tree. This is fairly indecipherable to the average user, and should only be used by developers familiar with the internal workings of PostgreSQL.

Explain and Explain Analyze

PostgreSQL devises a query plan for each queries it is given. Choosing the right plan to match the query structure and the properties of the data is absolutely critical for good performance, so the system includes a complex planner that tries to select good plans.

Explain – Shows execution plan of the query.

You can use the EXPLAIN command to see what query plan the planner creates for any query.
Syntax:

```
EXPLAIN [ VERBOSE ] query
```

The output of EXPLAIN has one line for each node in the plan tree, showing the basic node type plus the cost estimates that the planner made for the execution of that plan node. The first line (topmost node) has the estimated total execution cost for the plan; it is this number that the planner seeks to minimize.
It's also important to realize that the cost only reflects things that the planner cares about. In particular, the cost does not consider the time spent transmitting result rows to the client, which could be an important factor in the true elapsed time; but the planner ignores it because it cannot change it by altering the plan.

Example:

```
EXPLAIN SELECT * FROM tenk1;
        QUERY PLAN
-------------------------------------------------------------
Seq Scan on tenk1 (cost=0.00..458.00 rows=10000 width=244)
```

The numbers that are quoted by EXPLAIN are:

- Estimated start-up cost (Time expended before output scan can start, e.g., time to do the sorting in a sort node.)
- Estimated total cost (If all rows were to be retrieved, though they might not be: for example, a query with a LIMIT clause will stop short of paying the total cost of the Limit plan node's input node.)
- Estimated number of rows output by this plan node (Again, only if executed to completion.)
- Estimated average width (in bytes) of rows output by this plan node

As we saw in the previous section, the query planner needs to estimate the number of rows retrieved by a query in order to make good choices of query plans. Here is need of Table statistics.

One component of the statistics is the total number of entries in each table and index, as well as the number of disk blocks occupied by each table and index. This information is kept in the table pg_class, in the columns reltuples and relpages.

Table Statistics

Table Statistics are updated when an ANALYZE command is run. Table statistics are stored in pg_class and pg_stats . pg_stats is readable by all, whereas pg_statistic is only readable by a superuser. You can run Analyze command from psql on specific tables and just specific columns. Recommended is to run Analyze once in a day at low usage time.

Syntax for ANALYZE
Analyze [Table]
EXPLAIN ANALYZE query (explain the query with updated statistics)

Working of Explain and Explain Analyze

To show a query plan for a simple query on a table with a single int4 column and 128 rows:

```
EXPLAIN SELECT * FROM osrc;
 NOTICE:   QUERY PLAN:
Seq Scan on osrc  (cost=0.00..2.28 rows=128 width=4)
```

For the same table with an index to support an equijoin condition on the query, EXPLAIN will show a different plan:

```
EXPLAIN SELECT * FROM osrc WHERE i = 4;
NOTICE:    QUERY PLAN:
Index Scan using on osrc       (cost=0.00..0.42 rows=1 width=4)
```

And finally, for the same table with an index to support an equijoin condition on the query, EXPLAIN will show the following for a query using an aggregate function:

```
EXPLAIN SELECT sum(i) FROM foo WHERE i = 4;
 NOTICE:    QUERY PLAN:

Aggregate (cost=0.42..0.42 rows=1 width=4)

  -> Index Scan using fi on foo (cost=0.00..0.42 rows=1 width=4)
```

Clustering Rows

CLUSTER instructs Postgres to cluster the class specified by table approximately based on the index specified by indexname. The index must already have been defined on classname. When a class is clustered, it is physically reordered based on the index information. The clustering is static. In other words, as the class is updated, the changes are not clustered. No attempt is made to keep new instances or updated tuples clustered.

If one wishes, one can re-cluster manually by issuing the command again. Table is physically reordered based on the specified index. This index must already exist. Clustering groups rows with a common attribute to be on the same page, reducing the number of pages returned. Clustering is a one-time operation. Updates are not clustered unless the CLUSTER command is re-issued.

CLUSTER <index> ON <table>
CLUSTER <table>
CLUSTER

Most parameters are automatically adjusted to maintain optimum performance. Cache size and sort size are two parameters administrators can control to make better user of available memory. Disk access can also be spread across drives.

Lab Exercise 1

- You are working as a DBA. Edbstore cluster is in archive mode. You want to display a warning message if the checkpoint occur within 2mins of the previous checkpoint. Configure the settings.

Lab Exercise 2

- Inventory table is heavily used in edbstore database and there are long sorting operations on prod_id column.
  - Cluster the table to improve query performance.

Lab Exercise 3

- You as a DBA want to check the timing of the query:
  - select*from customers where customerid=9009 is executed. Change the server query planner setting by using set statement and note down following timings:
  - When sequential scan is forced
  - When index scan is enforced
  - When bitmap scan is enforced

11. PostgreSQL Partitioning and Table spaces

Overview on Partitioning

A partition is a division of a logical database or its constituting elements into distinct independent parts. Database partitioning is normally done for manageability, performance or availability reasons. The partitioning can be done by either building separate smaller databases (each with its own tables, indices, and transaction logs), or by splitting selected elements, for example just one table. We can also say, splitting one large table into smaller pieces can be called as partitioning. PostgreSQL supports partitioning via table inheritance. So the partitioning is made in such a way that every child table inherits single parent table. Parent table is empty and it exists just to describe the whole data set.

Benefits:

- Query and Update performance increase

- Easier to do bulk deletes
- Facilitates migrating table data to cheaper storage media

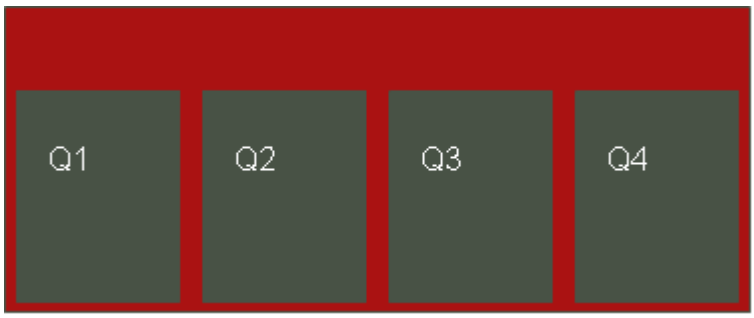


Figure: Table Partitioning

Partitioning refers to splitting what is logically one large table into smaller physical pieces. Query performance can be improved dramatically for certain kinds of queries.

Improved Update performance

When an index no longer fits easily in memory, both read and write operations on the index take progressively more disk accesses. Bulk deletes may be accomplished by simply removing one of the partitions. Seldom-used data can be migrated to cheaper and slower storage media.

PostgreSQL manages partitioning via table inheritance. Each partition must be created as a child table of a single parent table. The parent table itself is normally empty, it exists just to represent the entire data set.

Partition Methods

Current high end relational database management systems provide for different criteria to split the database. They take a partitioning key and assign a partition based on certain criteria. Common criteria are:

- Range Partitioning: Range partitions are defined via key column(s) with no overlap or gaps. Selects a partition by determining if the partitioning key is inside a certain range. An example could be a partition for all rows where the column zipcode has a value between 70000 and 79999. Range partitioning can be done for example by ID ranges (like 0-100 000, 100 001-200 000, 200 001-300 000…) or Date ranges (like 2009-11-01 – 2009-11-30, 2009-12-01 – 2009-12-31…).
- List Partitioning: Each key value is explicitly listed for the partitioning scheme. A partition is assigned a list of values. If the partitioning key has one of these values, the partition is chosen.List partitioning can be done for example by list of cities (like Hyderabad, Jammu, Mumbai, Pune, Chennai…) or list of categories (like Programming, Home, Food…).

Partition Setup (Steps to set up database partition)

Step 1: Create the "base" or “Master” table. For ex: inventory_hist.

This table will contain no data. Only define constraints that will apply to ALL partitions

Step 2: Create the partition tables which each inherit from the master table. For ex: inv_hist_2005, inv_hist_jan, inv_hist_feb, inv_hist_mar, inv_hist_apr, inv_hist_wk1, inv_hist_wk2, inv_hist_wk3, inv_hist_wk4, inv_hist_wk5. As shown in figure.

Step 3: Add table constraints to the partition tables to define the allowed key values in each partition.

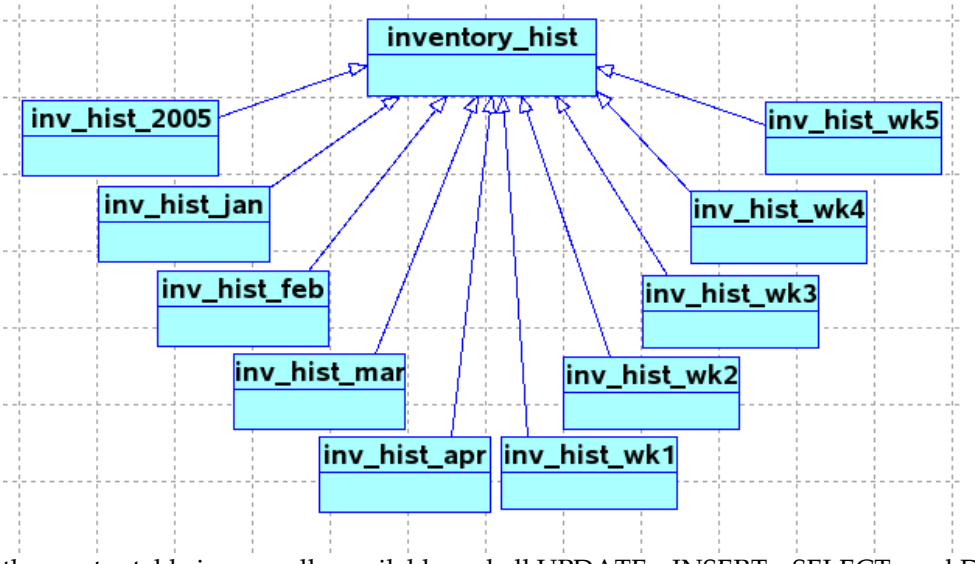
Typical examples would be:

CHECK (x = 1)
CHECK (city IN ('Hyderabad', 'Bangalore', 'Chennai'))
CHECK (outletID >= 100 AND outletID < 200)

Step 4: Ensure that the constraints guarantee that there is no overlap between the key values permitted in different partitions.
Note that there is no difference in syntax between range and list partitioning (those terms are descriptive only).

Step 5: Create Indexes for each partition

Step 6: Optionally, define a rule or trigger to redirect modifications of the master table to the appropriate partition.Ensure that the constraint_exclusion configuration parameter is enabled in postgresql.conf. Without this, queries will not be optimized as desired. Set constraint exclusion on following command on psql i.e SET constraint_exclusion = ON;



Finally, the master table is normally available and all UPDATEs, INSERTs, SELECTs and DELETEs goes to the right child tables.

Partition Table Explain Plan

With constraint exclusion enabled, the planner will examine the constraints of each partition and try to prove that the partition need not be scanned because it could not contain any rows meeting the query's WHERE clause. When the planner can prove this, it excludes the partition from the query plan. Some or all of the partitions might use index scans instead of full-table sequential scans, When we enable constraint exclusion, we get a significantly cheaper plan that will deliver the same answer.

explain select * from inventory_hist where rec_dt > '01/03/2006';

Result (cost=0.00..2748.23 rows=135003 width=20)

-> Append (cost=0.00..2748.23 rows=135003 width=20)

-> Seq Scan on inventory_hist (cost=0.00..28.88 rows=504 width=20)

Filter: (rec_dt > '2006-01-03'::date)

-> Index Scan using inv_hist_2005_idx on inv_hist_2005 inventory_hist (cost=0.00..3.52 rows=10 width=20)

Index Cond: (rec_dt > '2006-01-03'::date)

...

-> Seq Scan on inv_hist_wk3 inventory_hist (cost=0.00..28.88 rows=504 width=20)

Filter: (rec_dt > '2006-01-03'::date)

-> Seq Scan on inv_hist_wk4 inventory_hist (cost=0.00..169.06 rows=8485 width=20)

Filter: (rec_dt > '2006-01-03'::date)

-> Seq Scan on inv_hist_wk5 inventory_hist (cost=0.00..28.88 rows=504 width=20)

Filter: (rec_dt > '2006-01-03'::date)

Example:

1. Create Master Table

```
aparddb=# CREATE TABLE callrecord (
          callerid NUMERIC, callername VARCHAR,
          purpose VARCHAR, calldate DATE NOT NULL);
CREATE TABLE
```

2. Create Partition (Child) Tables

```
aparddb=# CREATE TABLE callrecord_jan(
          CHECK (calldate BETWEEN '2009-01-01' AND '2009-01-31'))
          INHERITS (callrecord);
CREATE TABLE
aparddb=# CREATE TABLE callrecord_feb(
          CHECK (calldate BETWEEN '2009-02-01' AND '2009-02-28'))
          INHERITS (callrecord);
CREATE TABLE
aparddb=# CREATE TABLE callrecord_mar(
          CHECK (calldate BETWEEN '2009-03-01' AND '2009-03-31'))
          INHERITS (callrecord);
CREATE TABLE
```

3. Create INDEX


```
aparddb=# CREATE INDEX idx_jan ON callrecord_jan(calldate);
CREATE INDEX
aparddb=# CREATE INDEX idx_feb ON callrecord_feb(calldate);
CREATE INDEX
aparddb=# CREATE INDEX idx_mar ON callrecord_mar(calldate);
CREATE INDEX
aparddb=# show constraint_exclusion;
 constraint_exclusion
----------------------
 partition
(1 row)
```

4. Trigger Function

```
aparddb=# CREATE or REPLACE function callrecord_trg_insert()
RETURNS trigger AS
$$
BEGIN
IF NEW.calldate >= '2009-01-01' and NEW.calldate <= '2009-01-31' then
INSERT INTO callrecord_jan VALUES(
NEW.callerid, NEW.callername,NEW.purpose, NEW.calldate);
ELSEIF NEW.calldate >= '2009-02-01' and NEW.calldate <= '2009-02-28' then
INSERT INTO callrecord_feb VALUES(
NEW.callerid, NEW.callername, NEW.purpose, NEW.calldate);
ELSEIF NEW.calldate >= '2009-03-01' and NEW.calldate <= '2009-03-31' then
INSERT INTO callrecord_mar VALUES(
NEW.callerid, NEW.callername,NEW.purpose,  NEW.calldate);
ELSE
RAISE NOTICE 'INVALID DATE: RANGE BETWEEN JAN2009 to MARCH2009 ONLY';
END IF;
RETURN NULL;
END;
$$language plpgsql;
CREATE FUNCTION
```

Return NULL so that the data doesn't get inserted into the parent table

5. Create Partitioning TRIGGER On psql client as shown below:

```
aparddb=# CREATE trigger trg1_callrecord BEFORE INSERT ON callrecord
          FOR EACH ROW EXECUTE PROCEDURE callrecord_trg_insert();
CREATE TRIGGER
```

6. Finally, you can try some DML operations , as shown below the values inserted as per the range of the calldate are inserted into the related partitioned table.


```
aparddb=# INSERT INTO callrecord VALUES
          (1234,'reshma','Training','2009-01-05');
INSERT 0 0
aparddb=# select*from callrecord_jan;
 callerid | callername | purpose  |  calldate
----------+------------+----------+------------
     1234 | reshma     | Training | 2009-01-05
(1 row)

aparddb=# select*from callrecord_feb;
 callerid | callername | purpose | calldate
----------+------------+---------+----------
(0 rows)

aparddb=# select*from callrecord_mar;
 callerid | callername | purpose | calldate
----------+------------+---------+----------
(0 rows)
```

Managing Partitions

It is common to want to remove old partitions of data and periodically add new partitions for new data. One of the most important advantages of partitioning is precisely that it allows this otherwise painful task to be executed nearly instantaneously by manipulating the partition structure, rather than physically moving large amounts of data around. The simplest option for removing old data is simply to drop the partition that is no longer necessary:

DROP TABLE inventory_hist;

This can very quickly delete millions of records because it doesn't have to individually delete every record. Another option that is often preferable is to remove the partition from the partitioned table but retain access to it as a table in its own right:

ALTER TABLE inventory_hist NO INHERIT measurement;

This allows further operations to be performed on the data before it is dropped.

Tablespace

Tablespaces in PostgreSQL allow database administrators to define locations in the file system where the files representing database objects can be stored. Once created, a tablespace can be referred to by name when creating database objects.

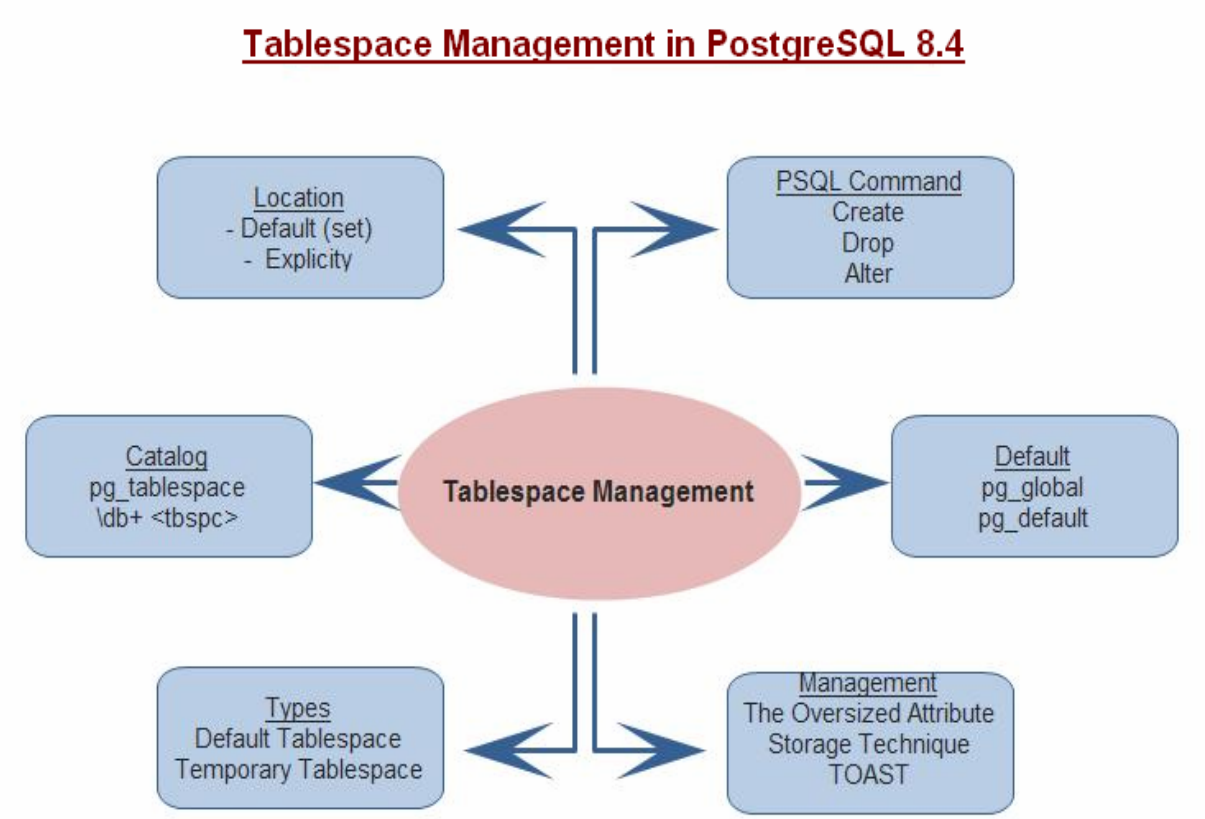
By using tablespaces, an administrator can control the disk layout of a PostgreSQL installation. This is useful in at least two ways. First, if the partition or volume on which the cluster was initialized runs out of space and

cannot be extended, a tablespace can be created on a different partition and used until the system can be reconfigured.

Second, tablespaces allow an administrator to use knowledge of the usage pattern of database objects to optimize performance. For example, an index which is very heavily used can be placed on a very fast, highly available disk, such as an expensive solid state device. At the same time a table storing archived data which is rarely used or not performance critical could be stored on a less expensive, slower disk system.

Tablespace Management System

Creation of the tablespace itself must be done as a database superuser, but after that you can allow ordinary database users to make use of it. To do that, grant them the CREATE privilege on it.



The tablespace associated with a database is used to store the system catalogs of that database. Furthermore, it is the default tablespace used for tables, indexes, and temporary files created within the database, if no TABLESPACE clause is given and no other selection is specified by default_tablespace or temp_tablespaces (which determines the placement of temporary tables and indexes, as well as temporary files that are used for purposes such as sorting large data sets). If a database is created without specifying a tablespace for it, it uses the same tablespace as the template database it is copied from.

Two tablespaces are automatically created by initdb. The pg_global tablespace is used for shared system catalogs. The pg_default tablespace is the default tablespace of the template1 and template0 databases (and,

therefore, will be the default tablespace for other databases as well, unless overridden by a TABLESPACE clause in CREATE DATABASE).

To determine the set of existing tablespaces, examine the pg_tablespace system catalog, for example

```
SELECT spcname FROM pg_tablespace;
```

\db meta-command is also useful for listing the existing tablespaces.

PostgreSQL has storage system called TOAST(The Oversized-Attribute Storage Technique). For sorting, PostgreSQL uses pgsql_tmp for a SQL statement execution, and files get created within the directory.

Create, Alter, Drop Tablespace

CREATE TABLESPACE registers a new cluster-wide tablespace. The tablespace name must be distinct from the name of any existing tablespace in the database cluster.

a)Create Tablespace

CREATE TABLESPACE <tablespace_name> OWNER <db_owner> LOCATION '<destination directory>'

b)Creation at the Database Level

CREATE DATABASE <db_name> OWNER <username> TEMPLATE <template> TABLESPACE <tablespace_name>

Example:

CREATE TABLESPACE apard_tab LOCATION '/mnt/sda1/postgresql/data';

The location must be an existing, empty directory that is owned by the PostgreSQL system user. All objects subsequently created within the tablespace will be stored in files underneath this directory.

Alter Tablespace

```
ALTER TABLESPACE name RENAME TO newname
ALTER TABLESPACE name OWNER TO newowner
```

Rename tablespace apard_tab to imstab:

```
ALTER TABLESPACE index_space RENAME TO imstab;
```

Change the owner of tablespace apard_tab to osrc:

```
ALTER TABLESPACE apard_tab OWNER TO osrc;
```

Once created, a tablespace can be used from any database, provided the requesting user has sufficient privilege. This means that a tablespace cannot be dropped until all objects in all databases using the tablespace have been removed.

Drop Tablespace

```
DROP TABLESPACE [ IF EXISTS ] tablespacename
IF EXISTS – does not throw an error if the tablespace does not exist
```

Ex: DROP TABLESPACE imstab IF EXISTS;

Default Location

set default_tablespace= ‘ ’ or <tsname>
set temp_tablespace = ‘ ‘ or <tsname>

Moving data from one Tablespace to another Tablespace.

1. First tell PostgreSQL, hereafter all new tables and indexes for the database in the New Tablespace
=# ALTER DATABASE <databasename> SET default_tablespace = <newtablespace>;

2. You have to move each table in the database to the new tablespace (use psql script to perform this task)
=# ALTER TABLE <tablename> SET TABLESPACE <newtablespace>;

3. Same way indexes should be moved
=# ALTER INDEX <indexname> SET TABLESPACE <newtablespace>;

PostgreSQL makes extensive use of symbolic links to simplify the implementation of tablespaces. This means that tablespaces can be used only on systems that support symbolic links.

The directory $PGDATA/pg_tblspc contains symbolic links that point to each of the non-built-in tablespaces defined in the cluster. Although not recommended, it is possible to adjust the tablespace layout by hand by redefining these links. Two warnings: do not do so while the postmaster is running; and after you restart the postmaster, update the pg_tablespace catalog to show the new locations. (If you do not, pg_dump will continue to show the old tablespace locations.)

12. High Availability & Replication

Overview

An important aspect of any enterprise database engineering is replication to ensure maximum availability. PostgreSQL provide a simple, fast and stable replication mechanism, known as Warm Standby, though more accurately described as Log Shipping. Data from the transaction log, also known as the Write Ahead Log (WAL) is transported to a standby server where continuous recovery takes place. It's a simple and elegant solution, relying on the underlying recovery code to perform changes on the standby node, so there is only minimal overhead on the primary server.

Data is currently shipped one file at a time, offering file based asynchronous replication. The entire database server is replicated, so there is no additional administration for each table or for each database. Replication can be fast in many circumstances and utilizes WAN links effectively. The standby node cannot yet be accessed to perform queries

Warm Standby Servers for High Availability

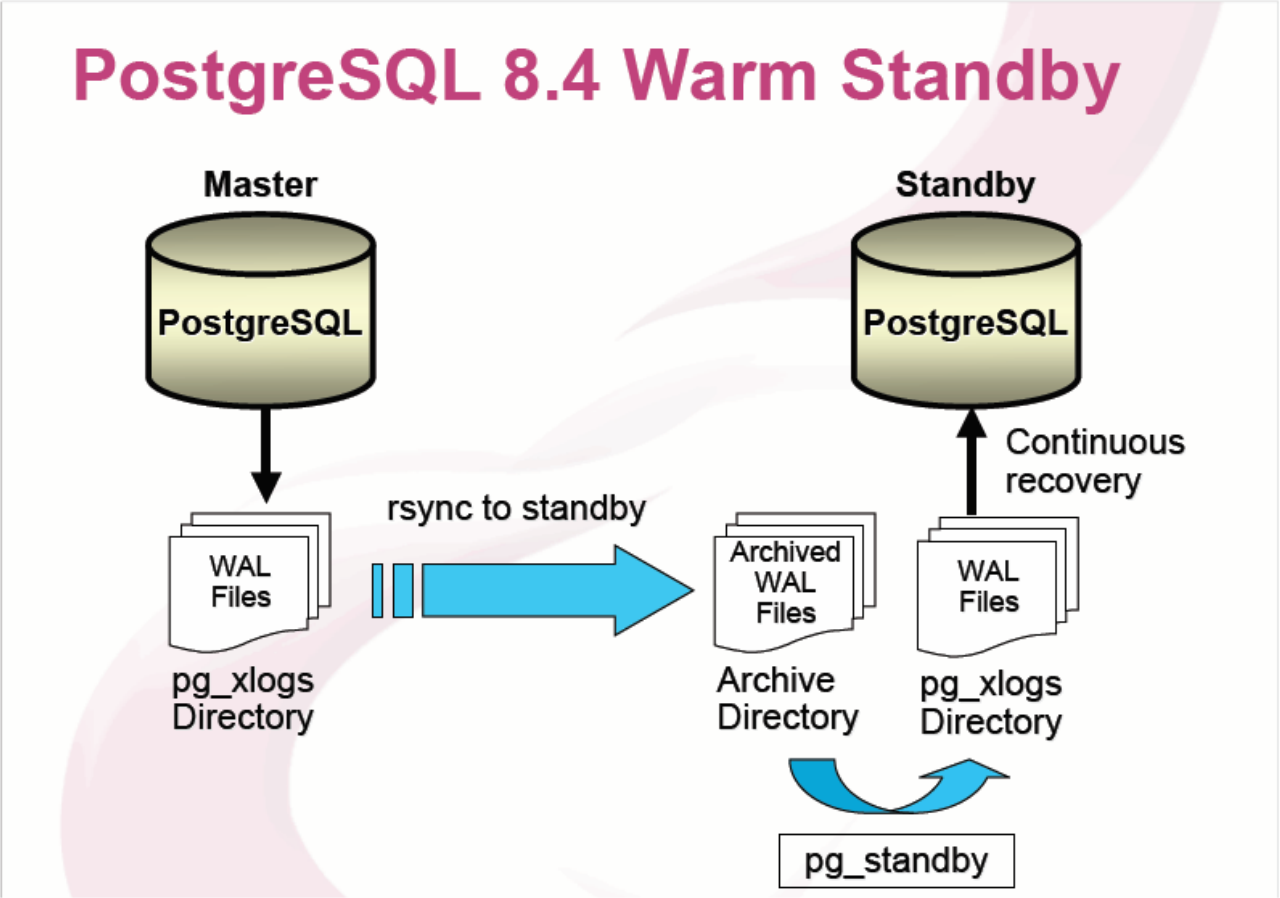
Continuous archiving can be used to create a high availability (HA) cluster configuration with one or more standby servers ready to take over operations if the primary server fails. This capability is widely referred to as warm standby or log shipping.

The primary and standby server work together to provide this capability, though the servers are only loosely coupled. The primary server operates in continuous archiving mode, while each standby server operates in continuous recovery mode, reading the WAL files from the primary. No changes to the database tables are required to enable this capability, so it offers low administration overhead in comparison with some other replication approaches. This configuration also has relatively low performance impact on the primary server. A standby server is a second server that can be brought online if the primary production server fails. The standby server contains a copy of the databases on the primary server. A standby server can also be used when a primary server becomes unavailable due to scheduled maintenance. For example, if the primary server needs a hardware or software upgrade, the standby server can be used.

A standby server allows users to continue working with databases if the primary server becomes unavailable. When the primary server becomes available again, any changes to the standby server's copies of databases must be restored back to the primary server. Otherwise, those changes are lost. When users start using the primary server again, its databases should be backed up and restored on the standby server again.

Implementing a standby server involves these phases:

- Creating the database and ongoing transaction log backups on the primary server.
- Setting up and maintaining the standby server by backing up the database on the primary server and restoring them on the standby server.
- Bringing the standby server online if the primary server fails.



Creating the Backups on the Primary Server

On the primary server:

- Create a full database backup of each database to be duplicated. For more information.
- Periodically, create a transaction log backup of each database to be duplicated. The frequency of transaction log backups created on the primary server depends on the volume of transaction changes of the production server database. If the transaction frequency is high, it may be useful to back up the transaction log frequently to minimize the potential loss of data in the event of failure.
  Note: When restoring a copy of master from a production server to a standby server, you cannot back up the transaction log of master. Only a database backup and restore of master is possible.

Setting Up and Maintaining the Standby Server

A standby server is set up and maintained as follows:

- Restore the database backups from the primary server onto the standby server in standby mode, specifying an undo file (one undo file per database).
  When a database or transaction log is restored in standby mode, recovery needs to roll back any uncommitted transactions so that the database can be left in a logically consistent state and used, if necessary, for read-only purposes. Pages in the database affected by the uncommitted, rolled back transactions are modified. This undoes the changes originally performed by the uncommitted transactions. The undo file is used to save the contents of these pages before recovery modifies them to prevent the changes performed by the uncommitted transactions from being lost. Before a

subsequent transaction log backup is next applied to the database, the uncommitted transactions that were previously rolled back by recovery must be reapplied first. The saved changes in the undo file are reapplied to the database, and then the next transaction log is applied.
Note: There must be enough disk space for the undo file to grow so that it can contain all the distinct pages from the database that were modified by rolling back uncommitted transactions.

- Periodically, apply each subsequent transaction log, created on the primary server, to the databases on the standby server. Apply each transaction log in standby mode, specifying the same undo file used when previously restoring the database.
  The frequency of transaction log backups applied to the standby server depends on the frequency of transaction log backups of the primary production server database. Frequently applying the transaction log reduces the work required to bring the standby server online in the event of a production system failure.

In standby mode, the database is available for read-only operations, such as database queries that do not attempt to modify the database. This allows the database to be used for decision-support queries or DBCC checks.

Bringing the Standby Server Online

When the primary server initially becomes unavailable, it is unlikely that all the databases on the standby server are in complete synchronization. Some transaction log backups created on the primary server may not have been applied to the standby server yet. Additionally, some changes to the databases on the primary server are likely to have occurred since the transaction log on those databases were last backed up, especially in heavily used systems. Before the users use the standby copies, it is possible to synchronize the primary databases with the standby copies and bring the standby server online by:

- Applying to the standby server in sequence any transaction log backups created on the primary server that has not yet been applied.
- Creating a backup of the active transaction log on the primary server and applying the backup to the database on the standby server. The backup of the active transaction log when applied to the standby server allows users to work with an exact copy of the primary database as it was immediately prior to failure (although any noncommitted transactions will have been permanently lost). For more information.
  If the primary server is undamaged, as in the case of planned maintenance or upgrades, you can back up the active transaction log with NORECOVERY. This will leave the database in the restoring state and allow you to update the primary server with transaction log backups from the secondary server. Then you can switch back to the primary server without creating a complete database backup of the secondary server.
- Recover the databases on the standby server. This recovers the databases without creating an undo file, making the database available for users to modify.

A standby server can contain backups of databases from several instances of SQL Server. For example, a department could have five servers, each running a mission-critical database system. Rather than having five separate standby servers, a single standby server can be used. The database backups from the five primary systems can be loaded onto the single backup system, reducing the number of resources required and saving money. It is unlikely that more than one primary system would fail at the same time. Additionally, the standby server can be of higher specification than the primary servers to cover the remote chance that more than one primary system is unavailable at a given time.

Log shipping

Directly moving WAL or "log" records from one database server to another is typically known as log shipping. It should be noted that the log shipping is asynchronous, i.e., the WAL records are shipped after transaction commit. As a result there is a window for data loss should (the primary server) suffer a catastrophic failure: i.e transactions not yet shipped will be lost. The length of the window of data loss can be limited by use of the archive_timeout parameter, which can be set as low as a few seconds if required. However such low settings will substantially increase the bandwidth requirements for file shipping. If you need a window of less than a minute or so, it's probably better to look into record-based log shipping.

File-based Log shipping

PostgreSQL implements file-based log shipping i.e WAL records are transferred one file (WAL segment) at a time. WAL files can be shipped easily and cheaply over any distance, whether it be to an adjacent system, another system on the same site or another system on the far side of the globe. The bandwidth required for this technique varies according to the transaction rate of the primary server.

Record-Based Log Shipping

An external program can call the pg_xlogfile_name_offset() function to find out the file name and the exact byte offset within it of the current end of WAL. It can then access the WAL file directly and copy the data from the last known end of WAL through the current end over to the standby server(s). With this approach, the window for data loss is the polling cycle time of the copying program, which can be very small, but there is no wasted bandwidth from forcing partially-used segment files to be archived. Note that the standby servers' restore_command scripts still deal in whole WAL files, so the incrementally copied data is not ordinarily made available to the standby servers. It is of use only when the primary dies — then the last partial WAL file is fed to the standby before allowing it to come up. So correct implementation of this process requires cooperation of the restore_command script with the data copying program.

Failover

If the primary server fails then the standby server should begin failover procedures.
If the standby server fails then no failover need take place. If the standby server can be restarted, even some time later, then the recovery process can also be immediately restarted, taking advantage of restartable recovery. If the standby server cannot be restarted, then a full new standby server instance should be created.
If the primary server fails and then immediately restarts, you must have a mechanism for informing it that it is no longer the primary. This is sometimes known as STONITH (Shoot the Other Node In The Head), which is necessary to avoid situations where both systems think they are the primary, which will lead to confusion and ultimately data loss.
Many failover systems use just two systems, the primary and the standby, connected by some kind of heartbeat mechanism to continually verify the connectivity between the two and the viability of the primary. It is also possible to use a third system (called a witness server) to prevent some cases of inappropriate failover, but the additional complexity might not be worthwhile unless it is set up with sufficient care and rigorous testing.
So, switching from primary to standby server can be fast but requires some time to re-prepare the failover cluster. Regular switching from primary to standby is useful, since it allows regular downtime on each system for maintenance. This also serves as a test of the failover mechanism to ensure that it will really work when you need it. Written administration procedures are advised.

Warm-Standby Implementation Steps

Note: Before setting up the Warm-Standby, download the contrib. module pg_standby and configure to the $PGBIN

1. Set the archive_command in postgresql.conf file as given below:
   archive_command = 'cp -i %p <archive location>/%f'

2. Start the Hot backup using "start pg_start_backup('<label>') command.

3. Take the backup of data directory.

4. Stop the backup using command: "select pg_stop_backup();"

5. Restore the backup on the Target location.

6. remove the postmaster.pid and clean the pg_xlog and pg_xlog/archive_status directories.

7. Create a recovery.conf and please save the following command:

   restore_command = 'pg_standby -l -d -s 2 -k 50 -t /tmp/pgsql.trigger.5442 <archive_location> %f %p %r 2>>standby.log'

8. Now, start the warm Standby Directory.

Please keep the following in mind while starting the Warm Standby:

1. Warm Standby Data Directory should have 700 permission.

2. If you are creating the Warm Standby on Same server, on which primary server exists, then change the port in postgresql.conf for Warm Standby.

Also, you can monitor the status of Warm stand by using the standby.log file and pg_controldata command. pg_controldata command will give you the information checkpoint lagging status of Warm standby

Replication

Why Use Replication

Replication is a way of keeping data synchronized in multiple databases. Implementing and maintaining replication might not be a simple proposition. If you have numerous database servers that need to be involved in various types of replication, a simple task can quickly become complex. Implementing replication can also be complicated by the application architecture. But there are numerous scenarios in which replication can be utilized.

On Replication

Replication is the only technology that can satisfy the needs of the most demanding systems, as only replication can provide instant access to data and zero data loss (minimal Recovery Point Objective or RPO

and minimal Recovery Time Objective or RTO).

Replication is the process of sharing transactional data to ensure consistency between redundant database nodes. This improves fault tolerance, which leads to better reliability of the overall system.
Replication is simply the process of copying data from one location to another, but there are many ways in which this can be done and this article will explore some of these ways. Backup is still the cornerstone of a solid disaster recovery strategy, but since backups are usually run only once a day there is a high risk of losing large amounts of data and also having systems being offline for long periods of time (the Recovery Point Objective and Recovery Time Objective for backup is usually days, whereas with replication it is usually minutes or seconds).

With replication, none of these risks exist, but a comprehensive disaster recovery strategy should not only use replication as the sole means of protecting data as there are some disadvantages of doing so. For example, if a virus enters the system and only replication is used to protect data then that virus will most likely be replicated along with all the rest of the data to the secondary location.
That is why backup should still be used as it is a point-in-time copy of data on tape in a safe location that can be restored without viruses or data corruption. Using replication (especially in real-time) for 100% of data in an environment is unrealistic and cost-prohibitive. The bandwidth required for most businesses wanting to replicate all data would be astronomical, besides the fact that all data is not created equal so it would not make sense to replicate everything.

- Customer-centric systems: All data stored in systems being accessed by customers is mission-critical and should be protected at all costs.
- Partner-centric systems: Most of the data stored in partner-centric systems is business-critical, but not all. Carefully consider what should be replicated and what should only be backed up.
- Internal systems: Certain internal systems are critical for keeping the business going, such as e-mail, and should be protected at all costs. Other functions might accept a day or more worth of data loss.

Different Ways of Replicating
When it comes to replicating data there are many ways in which this can be done and many technologies used to do so. When choosing the right method of replication, carefully consider the data that is being protected and how much of that data can be afforded to lose. The different methods are:

• Synchronous Replication: Data is copied in real-time from system to system, so if the primary system was to fail the secondary system has an exact copy of data and can take over instantaneously. Synchronous replication is the only method that guarantees zero data loss.

• Asynchronous Replication: Data is copied from system to system with a small lag time, usually measured in milliseconds. This form of replication is less dependent on available bandwidth than synchronous replication, but if the primary system was to fail some data loss would occur.

• Periodic Replication: Data is copied from system to system on a set schedule, for example once a day or twice a day. Periodic replication is not ideal for critical systems as large amounts of data loss would occur if the primary system was to fail. Periodic replication is ideal for remote office backup.
After selecting a method of replication, comes the selection of what technology to use. The two most common ways of replicating data are server-based replication and storage array-based replication.

Server-based replication has the advantage of not locking users into using a certain storage array from a certain vendor. The replication solution from VERITAS for example, can replicate from any array to any array regardless of vendor, such as from EMC to Hitachi, or IBM to Serial ATA disk.
This ultimately lowers costs and provides users with the flexibility to choose what is right for their environment. Most server-based replication solutions can also replicate data over IP networks natively, so users do not have to buy expensive extra hardware to achieve this functionality.
Storage array-based replication has the advantage of being operating system independent, since replication is done from array to array. The downside of storage array-based replication is that vendors often demand users only replicate from and to similar arrays.
This can be very costly, especially if you are using high-performance disk at your primary site and now have to use the same at your secondary site. Also, array-based solutions often require extra hardware to send data over IP networks, further increasing costs.

Replication Tools Overview
Currently there are half-dozen different replication tools, depending on the user's purpose and platform. This is limited to master-slave replication in mature open source projects, including built-in PITR and Slony-I. Multi-master replication is available in the new project Bucardo as well as in various clustering tools. Built-in simple replication is planned for version 8.5, due in 2010.

An important aspect of an enterprise database engineering is replication to ensure maximum availability. PostgreSQL provides four options for replication. Each offers it's own features, benefits and issues. This is only an overview, hopefully leading the reader to the more relevant packages for their needs.
One new alternative to Slony is a project known as RubyRep, which is designed to avoid some of the limitations of Slony. RubyRep provides both master-slave and master-master replication, and it works for PostgreSQL as well as MySQL.

"Bucardo" -Trigger-based, asynchronous, multi-master or master-slave, written using plperl. Bucardo is an asynchronous PostgreSQL replication system, allowing for both multi-master and multi-slave operations. It was developed at Backcountry.com by Jon Jensen and Greg Sabino Mullane of End Point Corporation, and is now in use at many other organizations.

In many cases asynchronous replication is just not enough to model a certain business case. Therefore Cybertec offers a synchronous multimaster replication solution for PostgreSQL called Cybercluster. Cybertec is a PostgreSQL replication solution which makes sure that the database cluster is consistent at every point in time.

Daffodil Replicator is a powerful Open Source Java tool for data integration, data migration and data protection in real time. It allows bi-directional data replication and synchronization between homogeneous / heterogeneous databases including Oracle and MySQL.

pg_comparator" Perl-based, table-level async master-slave "diff" and "patch" method of replication. Low configuration overhead. This tool provides a network and time efficient PostgreSQL table content comparison. It may work with other database as well. It is free, open-source and distributed under the BSD License.

Important PostgreSQL Replication Packages

PGCluster - Multi-master no delay synchronus replication for load sharing or HA. Large objects are now supported.

- Slony I - Master to multi-slave cascading and almost-failover.
- Pgpool - Connection pooling front end with synchronous replication
- PostgreSQL Table Comparator - rsync for PostgreSQL.

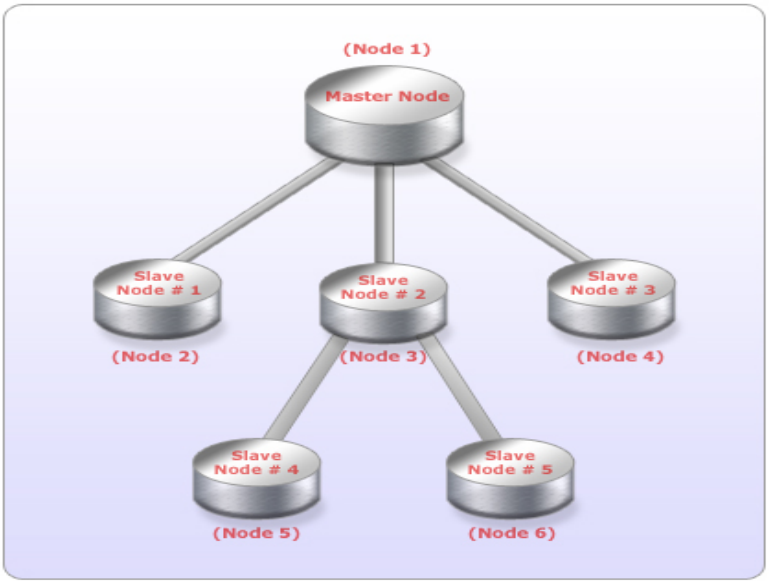
Slony-I

Slony-I is a "master to multiple slaves" replication system supporting cascading (e.g. - a node can feed another node which feeds another node...) and failover. If you need multiple instances of your database for high availability, backup or for a no-downtime migration to a new version, this versatile tool will keep all of them in sync. Slony is suitable for use in envrionments that require high-availability. PostgreSQL is an advanced, object-relational database management system that is frequently used to provide services of Database Management System. Although this database management system has proven to be stable for many years, the two available open-source replication solutions, rserv and ERServer, had serious limitations and needed replacement.

Fortunately, such a replacement recently became available. Slony-I is a trigger-based master to multiple slaves replication system for PostgreSQL being developed by Jan Wieck. This enterprise-level replication solution works asynchronously and offers all key features required by data centers. Among the key Slony-I usage scenarios are:

- Database replication from the head office to various branches to reduce bandwidth usage or speed up database requests. For improved performance for geographically dispersed users.
- Database replication to offer load balancing in all instances. This can be particularly useful for report generators or dynamic Web sites.
- Database replication to offer high availability of database services.
- Hot backup using a standby server or upgrades to a new release of PostgreSQL.

This lets you through the steps required to install Slony-I and replicate a simple database located on the same machine.



Replication Limitations

- Replicated tables must have a unique or primary key
- It does not support replication of large objects
- Schema changes are not propagated (though they can be coordinated) i.e Does not detect nor propagate table definition changes.
- It does not support synchronizing databases outside of replication
- There are limitations on version compatability i.e for example: You cannot replicate from PostgreSQL 8.2 to PostgreSQL 8.4.
- It is more difficult to set up than many other replication solutions.
- Does not allow multiple masters
- Cannot detect a node failure

Replication Components

Replication cluster – A set of database instances between which replication is to take place
Master node – The primary database with which applications interact
Slave node – All nodes that are part of the replication cluster except the master node
Replication set – The tables and sequences that are to be replicated
Cascaded replication – When a slave node becomes a master node for another slave node
Switchover – A planned reversal of a master and slave node
Failover – Replacing the master node with a slave node when an unplanned event takes the master node offline.

replication
: The main process that handles the communication between nodes

slonik
: The command processor application that is used to set up and modify configurations of replication clusters

replication schema
: Holds all necessary information for the replication cluster such as configuration information, pending transactions, status, etc.

To be used by embedding into shell scripts and reading commands from files or stdin. Typical uses are:

- Creating clusters
- Adding tables
- Switchover
- Failover

Replication Configuration

- Design the replication set
- Create an empty schema on all slave databases
- Create the replication set
- Start the Replication Daemon/Process
- Subscribe to the replication set

SLONY REPLICATION SETUP

SLONY-I INSTALLATION

Slony-I Source Code:Slony-I replication engine is available in Source code, RPM, win32 format and can be obtained from following site: http://slony.info/

Installing Slony-I on Linux

Slony-I can be installed on Linux using Binaries or from Source code.

a) Installing Slony-I from Source Code:

- Unpacking Slony-I source code.
- If its .gz file then
  gunzip slony.tar.gz;
  tar -xf slony.tar
- If its .bz2 file then
  gunzip slony.tar.bz2;
  tar -xf slony.tar

This will create a directory under the current directory with the Slony-I sources. Head into that directory for the rest of the installation procedure. After this , run following commands

$. /configure --with-pgconfigdir=$PGMAIN/bin (path to the pg_config file usually /usr/local/pgsql/bin) –with-perltools=<path to pgsql bin directory where all maintenance perl script will be installed>

$ gmake all

$ gmake install

b) Installing Slony-I from RPM:

- Installing Slony-I using these RPMs is as easy as installing any RPM.
  rpm -ivh postgresql-slony1-engine-....rpm
  If you want to upgrade the previous version, just use.
  rpm -Uvh postgresql-slony1-engine-....rpm

But please remember to follow the usual upgrade procedure, too. The RPM installs the files into their usual places. The configuration files are installed under /etc, the binary files are installed in /usr/bin, libraries are installed in /usr/lib/pgsql, and finally the docs are installed in /usr/share/doc/postgresql-slony1-engine.

This will install files into the PostgreSQL install directory as specified by the configure --prefix option used in the PostgreSQL installation. Make sure you have appropriate permissions to write into that area. Commonly you need to do this either as root.

3 Slony Configuration

Replicating PostgreSQL using Slony –I involves following steps (Slony-I should be installed on Master and Slave machine): Also verify all the tables have primary or unique key.

- Creating slave database
- Configuring Database
- Starting Slon Process
- Subscribing the Replication set on Master machine

Important Slony Commands

slonik_create_set - This requires SLONYSET to be set as well as SLONYNODES; it is used to generate the slonik script to set up a replication set consisting of a set of tables and sequences that are to be replicated.
slonik_drop_node - Generates Slonik script to drop a node from a Slony-I cluster.
slonik_drop_set - Generates Slonik script to drop a replication set (e.g. - set of tables and sequences) from a Slony-I cluster.
slonik_drop_table - Generates Slonik script to drop a table from replication. Requires, as input, the ID number of the table (available from table sl_table) that is to be dropped.
slonik_execute_script - Generates Slonik script to push DDL changes to a replication set.
slonik_failover - Generates Slonik script to request failover from a dead node to some new origin
slonik_init_cluster - Generates Slonik script to initialize a whole Slony-I cluster, including setting up the nodes, communications paths, and the listener routing.
slonik_merge_sets - Generates Slonik script to merge two replication sets together.
slonik_move_set - Generates Slonik script to move the origin of a particular set to a different node.
slonik_restart_node - Generates Slonik script to request the restart of a node. This was particularly useful pre-1.0.5 when nodes could get snarled up when slon daemons died.
slonik_restart_nodes - Generates Slonik script to restart all nodes in the cluster. Not particularly useful.
slony_show_configuration - Displays an overview of how the environment (e.g. - SLONYNODES) is set to configure things.
slon_kill - Kills slony watchdog and all slon daemons for the specified set. It only works if those processes are running on the local host, of course!
slon_start - This starts a slon daemon for the specified cluster and node, and uses slon_watchdog to keep it running.
slon_watchdog- Used by slon_start.
slonik_store_node - Adds a node to an existing cluster.
slonik_subscribe_set- Generates Slonik script to subscribe a particular node to a particular replication set.
slonik_uninstall_nodes - This goes through and drops the Slony-I schema from each node; use this if you want to destroy replication throughout a cluster. This is a VERY unsafe script!
slonik_unsubscribe_set - Generates Slonik script to unsubscribe a node from a replication set.

Implementation

Step 1 - Download the soly-I source (slonyI 2.0.2.tar.bz2) from below url and untar it.
http://main.slony.info/downloads/2.0/source/
tar -xjf slony1-2.0.2.tar.bz2
Step 2. - Move to the untar directory location and configure the slony.
#cd slony1-2.0.2
./configure --prefix=/usr/local/pgsql/bin --with-pgconfigdir=/usr/local/pgsql/bin --with-perltools=/usr/local/pgsql/bin
make
make install

Step 3 - Go to the slony soruce directory/tools/altperl/ and copy the sample slony.conf to the postgres bin location
#cd /<slonysource Location>/tools/altper/
#cp slony_tools.conf-sample /usr/local/pgsql/bin/slon.conf

Step 4 - Go to the postgres bin location and under it you find another bin location of slony, so go to that directory and copy all the files to postgres bin location

```
#cd /usr/local/pgsql/bin/bin/
#cp *  /usr/local/pgsql/bin/
```

Step 5 - Now give the postgres ownership to all the files which we have copied to the postgres bin location.

```
# chown -R postgres:postgres slon*
```

Step 6 - Create a log directory where slony errors are any messages to store and give the postgres ownership for that directory

```
#cd /var/log
#mkdir slony
# chown -R postgres:postgres /var/log/slony
```

Step 7 - Create Two clusters with different ports using initdb command, in our example we take 5432 and 5433. After creating the two cluster stop the clusters and start doing the below process one by one starting each cluster according to the steps.. First start 5432 port cluster and do the things as mentioned below:-

Create the Master database and a primary key table in one cluster (5432) as master, and also insert few records in it to see the replication

```
postgres=# create database master;
```

Connect to the database created

```
postgres=#\c master
```

Create the table for replication

```
master=#create table MasterTable(ecode integer primary key);
master=# insert into MasterTable values(100),(101),(102),(103),(104);
```

Come out of the master

```
master=#\q
```

Step 8 - Now start the another cluster(5433) and do the following steps

Create the Slave database and same templete of master database table in another cluster (5433)

```
postgres=# create database slave;
```

connect to

```
postgres=#\c slave
slave=#create table MasterTable(ecode integer primary key);
```

Come out of the slave

```
slave=#\q
```

Step 9 - Now the slony setups will start. Edit the copied slon.conf file from postgres bin location and change the nodes and append the tables as shown below.

Slon.conf file

Vi slon.conf

```
if ($ENV{"SLONYNODES"}) {
   require $ENV{"SLONYNODES"};
} else {
$CLUSTER_NAME = 'replication';
$LOGDIR = '/var/log/slony';
```



```
        $MASTERNODE = 1;
        $DEBUGLEVEL = 2;
        add_node(node     => 1,
                 host     => 'localhost',
                 dbname   => 'master',
                 port     => 5432,
                 user     => 'postgres',
                 password => '');
        add_node(node     => 2,
                 host     => 'localhost',
                 dbname   => '8.4slave',
                 port     => 5455,
                 user     => 'postgres',
                  password => '');
        }
        $SLONY_SETS = {
           "set1" => {
                        "set_id" => 1,
                        "table_id"    => 1,
                        "pkeyedtables" => [
                                    'MasterTable', ],
                     },
        };
        if ($ENV{"SLONYSET"}) {
                 require $ENV{"SLONYSET"};
        }
        1;
:wq
```

Step 10 Now, run the init cluster script:-

```
        $ ./slonik_init_cluster -c slon.conf
```

Step 11

```
        $ ./slonik_init_cluster -c slon.conf | ./slonik
```

Note:- You may get error as create language plpgsql. Create language in both master and slave directory.
Example:-

```
        master=#create language plpgsql;
```

Step 12 Create the set of both the nodes mentioned in the slon.conf file

```
        $ ./slonik_create_set -c slon.conf 1
```

Step 13 Configure the Setup now

```
        $ ./slonik_create_set -c slon.conf 1 | ./slonik
```

Step 14 Now, start the first node by running the slon_start script.

```
        $ ./slon_start -c slon.conf 1
```

Step 15 Now, Start the second node by running the slon_start script.

```
        $ ./slon_start -c slon.conf 2
```

Step 16 - Subscribe the slave nodes to master node by running the following script

```
        $ ./slonik_subscribe_set -c slon.conf 1 2
```

Step 17 -Subscribe both the sets which we created by issuing this command


```
$ ./slonik_subscribe_set -c slon.conf 1 2 | ./slonik
```

Now you have a successfully completed the setup, for verification to know the replication sync, connect to the slave and check.

13. Connection Pooling

Overview of PGPOOL-II

By default, Postgres doesn't come with a database connection pooling feature. This functionality is provided by PGPOOL-II, an open source project. For high traffic databases, you typically use database connection pooling to reduce the amount of time it takes to get a database connection and speed up access to your database.

How to install pgpool-II?
Below are the steps for installing pgpool in default /usr/local directory.

1. Download pgpool from the following link:
   http://pgfoundry.org/projects/pgpool/
2. In the directory which you have extracted the source tar ball, execute the following commands.
   $ ./configure
   $ make
   $ make install
   Note: pgpool-II requires libpq library in PostgreSQL 7.4 or later (version 3 protocol). If configure script displays the following error message, libpq. library may not be installed, or it is not of version 3.
   configure: error: libpq is not installed or libpq is old

If the library is version 3, but above message is still displayed, your libpq library is probably not recognized by configure script. configure script searches for libpq library under /usr/local/pgsql libaray. If you have installed PostgreSQL to a directory other than /usr/local/pgsql, use --with-pgsql, or --with-pgsql-includedir and --with-pgsql-libdir command line options when you execute configure.

How to configure pgpool?
pgpool-II configuration parameters are saved in pgpool.conf file. The file is in "parameter = value" per line format. When you install pgpool-II, pgpool.conf.sample is automatically created. It is recommend to copy and rename it to pgpool.conf, and edit it.

```
$ cp /usr/local/etc/pgpool.conf.sample /usr/local/etc/pgpool.conf
```

pgpool-II only accepts connections from the local host using port 9999. If you wish to receive conenctions from other hosts, set listen_addresses to '*'.
listen_addresses = 'localhost'
port = 9999

How to start and stop pgpool?
Navigate to the directory where pgpool is installed and execute the following file to start the application:

```
$pgpool
```


The above command, however, prints no log messages because pgpool detaches the terminal. If you want to show pgpool log messages, you pass –n option to pgpool command. pgpool-II is executed as non-daemon process, and the terminal will not be detached.

```
$ pgpool -n &
```

The log messages are printed on the terminal, so the recommended options to use are like the following.

```
$ pgpool -n -d > /tmp/pgpool.log 2>&1 &
```

-d option enables debug messages to be generated.

The above command keeps appending log messages to /tmp/pgpool.log. If you need to rotate log files, pass the logs to a external command which have log rotation function. For example, cronolog helps you.

```
$ pgpool -n 2>&1 | /usr/sbin/cronolog \ --hardlink=/var/log/pgsql/pgpool.log \ '/var/log/pgsql/%Y-%m-%d-pgpool.log' &
```

To stop pgpool-II process, execute the following command.

```
$ pgpool stop
```

If any client is still connected, pgpool-II waits for them to disconnect, and then terminate itself. Execute the following command instead if you want to shutdown pgpool-II forcibly.

```
$ pgpool -m fast stop
```

How to configure pgpool for replication?

To enable the database replication function, set replication_mode to true in pgpool.conf file.

```
replication_mode = true
```

When replication_mode is set to true, pgpool-II will send a copy of a received query to all the database nodes. When load_balance_mode is set to true, pgpool-II will distribute SELECT queries among the database nodes.

```
load_balance_mode = true
```

For more details kindly refer to the following link under Your first replication :
http://pgpool.projects.postgresql.org/pgpool-II/doc/tutorial-en.html

Pgbouncer

What is pgbouncer?

pgbouncer is a PostgreSQL connection pooler. Any target application can be connected to pgbouncer as if it were a PostgreSQL server, and pgbouncer will create a connection to the actual server, or it will reuse one of its existing connections. The aim of pgbouncer is to lower the performance impact of opening new connections to PostgreSQL.

How to install pgbouncer with PostgreSQL?

1. Download tar of pgbouncer from link :
   http://pgfoundry.org/frs/download.php/1873/pgbouncer-1.2.3.tgz
2. Download tar of libevent from link:
   http://monkey.org/~provos/libevent-1.4.8-stable.tar.gz
3. Install libevent.
4. Install pgbouncer.

5. if you get the error "error while loading shared libraries: libevent-1.4.so.2: cannot open shared object file: No such file or directory"
while running pgbouncer then do the following steps:

vi /etc/ld.so.conf.d/libevent-i386.conf

Note : If it is not created then create a new file libevent-i386.conf in the same folder, then enter /usr/local/lib/ Write and quit (:wq!)

The path in the libevent-i386.conf is the path where the actual .so files are located at. The path is set when we run the ./configure –prefix=/usr/local/
during the libevent compilation. Reloading the ld configuration with
ldconfig (Configure Dynamic Linker Run Time Bindings.).

How to configure pgbouncer?

Once pgbouncer is installed on the system:
1. copy sample pgbouncer.ini file from /usr/local/share/doc/pgbouncer/ to any other location that you want.
2. Modify the parameters of pgbouncer.ini file like :
   [databases] auth_type auth_file (we will have to create auth_file in the mentioned path).
3. Give the db port ,username,password,etc. Sample auth_file : /usr/local/pgsql/userlist.txt
   "postgres" "postgres"
4. Start pgbouncer :
   pgbouncer –d pgbouncer.ini
5. Check the log file : pgbouncer.log in the same path.
6. Check the pgbouncer listener port.(netstat –tan | grep "6000") if 6000 is your default port for pgbouncer.

Pgbouncer Source installation on Windows

Step 1.Download the pgbouncer for windows from http://winpg.jp/~saito/pg_work/plproxy/pgbouncer/

Step 2. Extract the zip file downloaded to the /bin directory of the PostgreSQL
   Example
   c:\program file\postgresql\8.4\bin

Step 3. Create the pgbouncer.ini file manually in the \bin directory with the following contents.
   Example: (pgbouncer.ini file)

```
[databases]
template1 = host=127.0.0.1 port=5432 dbname=template1
[pgbouncer]
listen_port = 6543
listen_addr = 127.0.0.1
auth_type = md5
auth_file = users.txt
logfile = pgbouncer.log
pidfile = pgbouncer.pid
admin_users = someuser
```

Step 4.After Creating the pgbouncer.ini file, create users.txt file with the user name and password of the server as mentioned in example
   Example:- (users.txt)

"username" "password of the server"

Step 5. Register the pgbouncer service by the following command

\bin> pgbouncer.exe -regservice pgbouncer.ini

Note: Service will be registered with this command. For further check we can see in services panel.

Step 6. After the service registeration, start the pgbouncer with the following command

\bin> pgbouncer.exe -v pgbouncer.ini

Step 7 Open the other Windows terminal and check for the pgbouncer

Example

bin>psql -p 6543 -U postgres pgbouncer

now we can see the prompt as

pgbouncer=#

14. Advance Monitoring Techniques

Overview of Monitoring PostgreSQL

What monitor?
The first question we should ask ourselves in the development of a monitoring concept, what information, data and values should be usefully monitored.

Database Activity
First, of course, should the activity of the database system itself monitors will. Among other things, the following points are relevant here:

- Does the database system at all?
- current sessions / connections
- currently running commands
- previously executed commands, statistics
- Size of tables, indexes, tablespaces

PostgreSQL provides such information on views and functions, and some may also on this analysis of the log files and operating system functionality be determined.

Locks
By Multi Version Concurrency Control (MVCC) locks (locks) generally play in PostgreSQL have a lesser role than in many other database systems with a more traditional architecture. A large-scale monitoring of restricted activities therefore not common. Nevertheless, there are ways to view the current list of locks what the analysis of certain situations, mostly from bad to errors stem-programmed applications that can be helpful.

Logs
PostgreSQL requires every server process information in the course of several log files, For example, error messages and timers, as well as suggestions for improvement for configuration. This information contains important information about whether to run the PostgreSQL server and client applications correctly. But this helps only If this log information also notes be taken and processed. Now it is almost impossible to ever read all the log files themselves, so there It sure tools that automatically analyze the log files and the most interesting Can present information clearly. Another condition is that the Logging is set and the most useful information, and only those to detect, refer to Chapter 2

OS
Many operating parameters are highly relevant to the functionality and performance a PostgreSQL database system. Therefore, it is also advisable to become familiar with the tools of the operating system, so we Watch these values and then to optimize the PostgreSQL installation. This include:

- Process List
- Memory usage

• CPU load
• Context switches
• I / O load
• Disk Space

In addition, a general supervision of the other operating system functionality, For example, the network or the time recommended to.

Backup

Not least, it is also important to monitor the operation of the backup, as a backup, not running or is not usable, is not a backup. The monitoring of data backup includes two aspects. First, should examine be whether the backup is running, or run, depending on how they configure is. Secondly, it should also be regularly assessed whether the usable data is, therefore, can be recorded again. Error in restoring Data sometimes indicate that the initial monitoring is not was complete. Monitoring is therefore also as a process, not only as a configuration setting to understand.

How to monitor?

If it is decided what should be properly monitored, we now come to which interfaces and tools here can help. Are mainly To do this, the built-in options such as the PostgreSQL statistics collector but use the simple tools of the operating system should not be neglected will. Unix Tools The clearest and quickest monitoring results obtained with some well-known Unix utilities. The information, although sometimes only crude, but still include these tools for many PostgreSQL administrators as standard.

ps

The first and easiest tool to get an overview of the activities of a Gain PostgreSQL server on a computer, is ps, the output of a process list. PostgreSQL server processes to fit their Displaying the ps command in such a way that one can recognize what the process. Sample output is

```
$ Ps-f-U postgres
UID PID PPID C STIME TTY TIME CMD
postgres 2972 1 0 Mar30? 00:00:09 / usr / bin / postgres-D / var / lib / postgresql /
data
postgres 2975 2972 0 Mar30? 00:00:56 postgres: writer process
postgres 2976 2972 0 Mar30? 00:00:40 postgres: wal writer process
postgres 2977 2972 0 Mar30? 00:00:16 postgres: autovacuum launcher process
postgres 2978 2972 0 Mar30? 00:00:11 postgres: stats collector process
postgres 23175 2972 0 22:18? 00:00:00 postgres: peter peter [local] SELECT
postgres 23164 2972 0 22:18? 00:00:00 postgres: peter testdb 127.0.0.1 (56330) idle
```

Note that ps depending on your operating system supports different options and also generate different output formats. The example above shows an expanded Output (-f) of all processes of the user postgres (postgres-U) on a Linux system.

```
ps ax | grep postgres
or
ps-ef | grep postgres
```


It is best thing is to try something with the options around to find a usable format for themselves. The first process shown is the main process, also "postmaster" (called for its former name). The column "CMD" here shows the arguments with which the Process was started. (This can, for example from an init.d script together have been.) The other process entries are subprocesses of the main process. These processes their command line entry in "CMD" column, so that we recognize change is what these processes. You can also go through the process list grep 'postgres' to send to see only the subprocesses to.
These first four processes are automatically by the main process for some (background) Tasks started. Depending on the current situation, configuration and PostgreSQLVersion Here you can sometimes see other processes. The last two processes are active database connections. The format of the entry process is

postgres: user database host activity client

The first three fields are set for the duration of the connection. Activity indicator change when a new SQL command is executed or been terminated. This Only the general command displays, for example, "SELECT" or "CREATE TABLE", but not the full command because the process list so accessible for everyone, and these details may Not everyone is familiar. The above item "idle" means that the Database Session currently executing a command. Possible would be a Augabe the following manner:

postgres 23175 2972 0 22:18? 00:00:00 postgres: peter testdb [local] SELECT waiting
postgres 23164 2972 0 22:18? 00:00:00 postgres: peter testdb 127.0.0.1 (56330) idle in transaction

The output of "idle in transaction" means that the current session does nothing, but a transaction is open (ie has a BEGIN or START TRANSACTION was executed). The Issue "waiting" for a command means that the session is waiting for a lock. One can conclude, in this example very simple, that process 23,175 to process 23,164 waiting for, since this is the only candidate. For a more detailed analysis of the effects of prolonged PostgreSQL process list can view a more detailed block list, see below. Using this process overview, you can already draw some preliminary conclusions. In general, would be avoided, of course, the state "waiting" or not see, at least for long. At heaped appearance of "idle in transaction" should set alarm bells ringing. Transactions to keep open at idle, causing problems such as the one illustrated Block other meetings and has typically on errors or shortcomings in the Client application through. Updating the process may permit the configuration parameters update_off process_title. On some platforms, this brings a performance gain.
In most cases it is irrelevant.

Note: On Solaris, dealing with the ps command is slightly more complicated. To PostgreSQL adapted and updated command lines see, proceed as follows: One has the command / usr / ucb / ps take instead of / bin / ps. When calling you use something like auxww ps | grep postgres exercise and it is important to specify two "w" 's. In addition, the proper command of the main process to be shorter than the adjusted Command line of the sub-processes (ie, omit rather have a few options and) adding to the configuration file. If that's not all fits together, it will be just the right command line arguments see no updated information. One can show with ps also get information about memory usage of processes. Here, for example, an excerpt from the output of ps aux on a

Linux system:
USER PID% CPU% MEM VSZ RSS TTY STAT START TIME COMMAND
...
postgres 13530 4.4 1.3 152780 6808? S 18:26 0:01 / usr/lib/postgresql/8.3 / bin / postgres-D / var / lib / postgresql /

postgres 13539 0.0 0.2 152780 1408? Ss 18:26 0:00 postgres: writer process
postgres 13540 0.0 0.2 152780 1272? Ss 18:26 0:00 postgres: wal writer process
postgres 13541 0.0 0.3 152916 1548? Ss 18:26 0:00 postgres: autovacuum launcher process
postgres 13542 0.0 0.2 15716 1232? Ss 18:26 0:00 postgres: stats collector process

Here, the column "VSZ" is the virtual size and "RSS" to find the Resident Set Size (Details To the appropriate man page for ps). One might be appalled by this issue, as the PostgreSQL processes here all alone even when doing nothing and spending more 150 MB of memory. This issue has However, the mistake that all processes that use shared memory, that in their Memory usage count, although, of course, the shared memory exists only once. In this example, 128 Mbytes shared_buffers were configured. In fact, evidence the processes in this situation, therefore, very little own memory.116In general, the memory usage figures in ps (and other programs that same values in the kernel to query and output) from this and similar reasons always relatively doubtful and should be approached with caution. The man page for ps can sometimes give information on reservations, but is probably not completely or current.

Top

Ps Besides, there are other programs that issue with which one court dockets can be. It is interesting especially the top utility, that the list regularly and updated according to various criteria such as CPU load and memory usage can be ordered to spend. Here's a sample call with arguments to be recommended:

```
top-c-u postgres
```

The options are-c to display the complete command line, so that as shown above is replaced by the current command and other information, and-u postgres to processes the list to restrict the user to postgres.
In addition to top and the program is recommended to htop. It's basically the same, but has a somewhat more friendly to Midnight Commander ajar, interactive Surface. Again, there is the option-u. The-c option does not exist, and it will also not required, as shown in the default, the entire command line is. A possible appeal would be htop-u postgres The disadvantage of the top programs from ps is that the output constantly hopping " '. This type of output is more likely to monitor the current status and Suitable system utilization, while the output of ps for better static analysis as illustrated above, the breakdown of barriers and long-running transactions can be used.

ptop

There is also an extra for PostgreSQL conceived top-like program. The package is named ptop, the program is, however pg_top. It connects to a PostgreSQLDatenbankinstanz and has also the usual options for Host, Port, User and so on. There is a list of all the processes included in this database instance out, supplemented by additional information from top known as memory and CPU utilization. Besides pg_top like top has a self-updating output and various interactive features. ptop is something new and unknown, but a look at it might be worth it. The Web site is ptop http://ptop.projects.postgresql.org/. There one also finds Downloads Has source code and RPMs for Red Hat / Fedora. In Debian, a package is included.

vmstat

vmstat is the name produces a tool for monitoring the memory (VM = virtual memory). It shows in addition to, but also statistics on the I / O system and CPU and is therefore very well suited to become general about

the system load an image. See here a sample output:

$ Vmstat 5
procs ----------- memory ---------- --- swap - ----- io ---- - system - ---- cpu ---
rb swpd free buff cache si so bi bo in cs us sy id wa
1 0 0 42760 22428 273268 0 0 156 58 1044 452 9 2 85 4
0 0 0 42760 22436 273268 0 0 0 12 1002 246 0 0 99 0
0 0 0 42752 22436 273268 0 0 0 1 1003 243 1 0 99 0
0 0 0 42760 22436 273268 0 0 0 15 1056 242 0 0 99 0
0 0 0 42760 22436 273268 0 0 0 0 1092 242 1 0 99 0
0 0 0 42760 22436 273268 0 0 0 0 1002 244 0 0 99 0

Argument 5 indicates that the output will be refreshed every five seconds. The first line of output indicates the system load since the boats, the following Rows in each case the last five seconds. If no argument is specified, only the first line of output. With an interval specification can be so good vmstat incidentally run to the system's behavior as under load or when the system analysis to observe. The columns under "memory" type of memory consumption. Usually only very little memory really "free" because the free memory for the file system cache (Column "cache") is used. This is from the perspective of PostgreSQL good and meaningful. If the column "cache" but also goes to zero, there is too little memory. Then they can usually be observed under "swap" activity. On a powerful Database server should not occur but for the swapping.
The columns under "io" mean: "bi" = blocks in, so read, "bo" = blocks out, then written. A block is 1024 bytes big on Linux. Here, you can then the I / ODurchsatz the system bar. To do more, see the following section on

Pgfouine
pgFouine is a PostgreSQL log analyzer used to generate detailed reports from a PostgreSQL log file. pgFouine can help you to determine which queries you should optimize to speed up your PostgreSQL
based application.

Pgfouine setup information as follows:
1. Download/unpack “pgfouine” source from the below link::
http://pgfoundry.org/frs/download.php/2575/pgfouine-1.2.tar.g

2. untar the pgFouine tarball anywhere you want.

3. Change the below parameters in the postgresql.conf file.
log_destination = 'stderr'◎
log_directory = 'pg_log'◎
log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log'◎log_rotation_age = 1d◎
log_min_messages = info◎
log_min_error_statement = notice◎
log_min_duration_statement = 100◎
log_line_prefix = '%t [%p]: [%l-1] '◎
log_error_verbosity = verbose
◎log_checkpoints = on
◎log_duration = on◎
log_statement = 'all'

I prefer these settings in testing/development environment to create a very detailed output (on a productive system the log_min_XY parameters should be adjusted to e.g. error) and get one logfile per day in the directory $PGDATA/pg_log. The parameterlog_line_prefix is adapted to the needs of pgfouine, THE log file analyzer for postgres logs.

4.Restart the Postgres service using pg_ctl utility.

5.Generate the log file analyzer using the below command::

```
pgfouine.php -logtype stderr -file "/Library/PostgresPlus/8.4SS/data/pg_log/log_postgresql-2010-07-08_010653.log_old" > 2010-11-10_select_only.html
```

NOTE:: pgFouine is a PostgreSQL log analyzer written in PHP. It's requires PHP must be installed in your system

