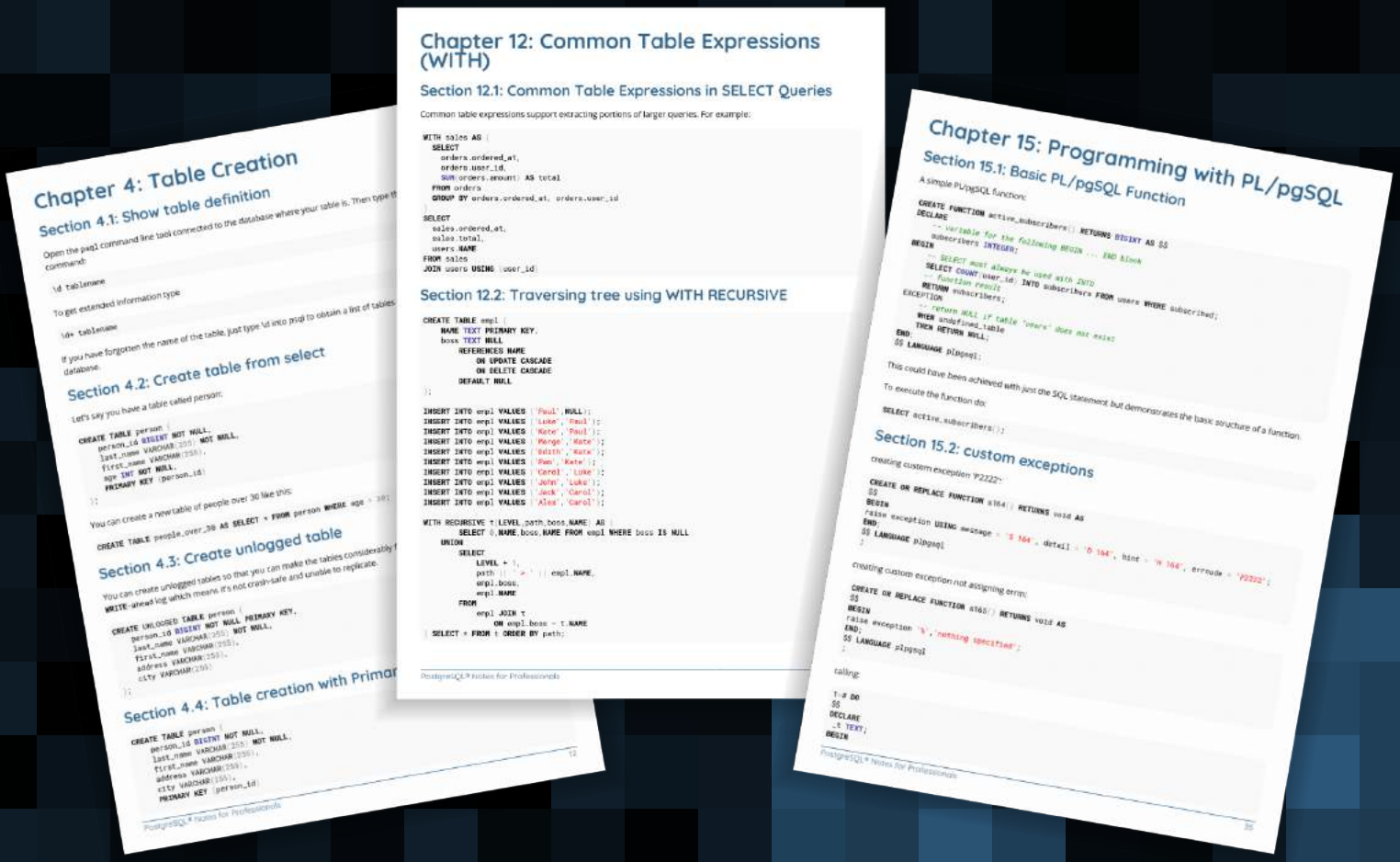
PostgreSQL®

Notes for Professionals



60+ pages

of professional hints and tricks

Contents

Chapter 1: Getting started with PostgreSQL

Version	Release date	EOL date
10.0	2017-10-05	2022-10-01
9.6	2016-09-29	2021-09-01
9.5	2016-01-07	2021-01-01
9.4	2014-12-18	2019-12-01
9.3	2013-09-09	2018-09-01
9.2	2012-09-10	2017-09-01
9.1	2011-09-12	2016-09-01
9.0	2010-09-20	2015-09-01
8.4	2009-07-01	2014-07-01

Section 1.1: Installing PostgreSQL on Windows

While it's good practice to use a Unix based operating system (ex. Linux or BSD) as a production server you can easily install PostgreSQL on Windows (hopefully only as a development server).

Download the Windows installation binaries from EnterpriseDB: http://www.enterprisedb.com/products-services-training/pgdownload This is a third-party company started by core contributors to the PostgreSQL project who have optimized the binaries for Windows.

Select the latest stable (non-Beta) version (9.5.3 at the time of writing). You will most likely want the Win x86-64 package, but if you are running a 32 bit version of Windows, which is common on older computers, select Win x86-32 instead.

Note: Switching between Beta and Stable versions will involve complex tasks like dump and restore. Upgrading within beta or stable version only needs a service restart.

You can check if your version of Windows is 32 or 64 bit by going to Control Panel -> System and Security -> System -> System type, which will say "##-bit Operating System". This is the path for Windows 7, it may be slightly different on other versions of Windows.

In the installer select the packages you would like to use. For example:

- pgAdmin (https://www.pgadmin.org) is a free GUI for managing your database and I highly recommend it. In 9.6 this will be installed by default .
- PostGIS (http://postgis.net) provides geospatial analysis features on GPS coordinates, distances etc. very popular among GIS developers.
- The Language Package provides required libraries for officially supported procedural language PL/Python, PL/Perl and PL/Tcl.
- Other packages like pgAgent, pgBouncer and Slony are useful for larger production servers, only checked as needed.

All those optional packages can be later installed through "Application Stack Builder".

Note: There are also other non-officially supported language such as PL/V8, PL/Lua PL/Java available.

Open pgAdmin and connect to your server by double clicking on its name, ex. "PostgreSQL 9.5 (localhost:5432).

From this point you can follow guides such as the excellent book PostgreSQL: Up and Running, 2nd Edition (http://shop.oreilly.com/product/0636920032144.do).

Optional: Manual Service Startup Type

PostgreSQL runs as a service in the background which is slightly different than most programs. This is common for databases and web servers. Its default Startup Type is Automatic which means it will always run without any input from you.

Why would you want to manually control the PostgreSQL service? If you're using your PC as a development server some of the time and but also use it to play video games for example, PostegreSQL could slow down your system a bit while its running.

Why wouldn't you want manual control? Starting and stopping the service can be a hassle if you do it often.

If you don't notice any difference in speed and prefer avoiding the hassle then leave its Startup Type as Automatic and ignore the rest of this guide. Otherwise...

Go to Control Panel -> System and Security -> Administrative Tools.

Select "Services" from the list, right click on its icon, and select Send To -> Desktop to create a desktop icon for more convenient access.

Close the Administrative Tools window then launch Services from the desktop icon you just created.

Scroll down until you see a service with a name like postgresql-x##-9.# (ex. "postgresql-x64-9.5").

Right click on the postgres service, select Properties -> Startup type -> Manual -> Apply -> OK. You can change it back to automatic just as easily.

If you see other PostgreSQL related services in the list such "pgbouncer" or "PostgreSQL Scheduling Agent - pgAgent" you can also change their Startup Type to Manual because they're not much use if PostgreSQL isn't running. Although this will mean more hassle each time you start and stop so it's up to you. They don't use as many resources as PostgreSQL itself and may not have any noticeable impact on your systems performance.

If the service is running its Status will say Started, otherwise it isn't running.

To start it right click and select Start. A loading prompt will be displayed and should disappear on its own soon after. If it gives you an error try a second time. If that doesn't work then there was some problem with the installation, possibly because you changed some setting in Windows most people don't change, so finding the problem might require some sleuthing.

To stop postgres right click on the service and select Stop.

If you ever get an error while attempting to connect to your database check Services to make sure its running.

For other very specific details about the EDB PostgreSQL installation, e.g. the python runtime version in the official language pack of a specific PostgreSQL version, always refer to the official EBD installation guide , change the version in link to your installer's major version.

Section 1.2: Install PostgreSQL from Source on Linux

Dependencies:

- GNU Make Version > 3.80
- an ISO/ ANSI C-Compiler (e.g. gcc)
- an extractor like tar or gzip
- zlib-devel

- readline-devel oder libedit-devel

Sources: Link to the latest source (9.6.3)

Now you can extract the source files:

```
tar -xzvf postgresql-9.6.3.tar.gz
```

There are a large number of different options for the configuration of PostgreSQL:

Full Link to the full installation procedure

Small list of available options:

- `--prefix=PATH` path for all files
- `--exec-prefix=PATH` path for architectur-dependet file
- `--bindir=PATH` path for executable programs
- `--sysconfdir=PATH` path for configuration files
- `--with-pgport=NUMBER` specify a port for your server
- `--with-perl` add perl support
- `--with-python` add python support
- `--with-openssl` add openssl support
- `--with-ldap` add ldap support
- `--with-blocksize=BLOCKSIZE` set pagesize in KB
  - `BLOCKSIZE` must a power of 2 and between 1 and 32
- `--with-wal-segsize=SEGSIZE` set size of WAL-Segment size in MB
  - `SEGSIZE` must be a power of 2 between 1 and 64

Go into the new created folder and run the cofigure script with the desired options:

```
./configure --exec=/usr/local/pgsql
```

Run `make` to create the objectfiles

Run `make install` to install PostgreSQL from the built files

Run `make clean` to tidy up

For the extension switch the directory `cd contrib`, run `make` and `make install`

Section 1.3: Installation on GNU+Linux

On most GNU+Linux operating systems, PostgreSQL can easily be installed using the operating system package manager.

Red Hat family

Respositories can be found here: https://yum.postgresql.org/repopackages.php

Download the repository to local machine with the command

```
yum -y install https://download.postgresql.org/pub/repos/yum/X.X/redhat/rhel-7-x86_64/pgdg-
redhatXX-X.X-X.noarch.rpm
```

View available packages:

```
yum list available | grep postgres*
```

Neccesary packages are: postgresqlXX postgresqlXX-server postgresqlXX-libs postgresqlXX-contrib

These are installed with the following command: yum -y install postgresqlXX postgresqlXX-server postgresqlXX-libs postgresqlXX-contrib

Once installed you will need to start the database service as the service owner (Default is postgres). This is done with the pg_ctl command.

```
sudo -su postgres
./usr/pgsql-X.X/bin/pg_ctl -D /var/lib/pgsql/X.X/data start
```

To access the DB in CLI enter `psql`

Debian family

On Debian and derived operating systems, type:

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

This will install the PostgreSQL server package, at the default version offered by the operating system's package repositories.

If the version that's installed by default is not the one that you want, you can use the package manager to search for specific versions which may simultaneously be offered.

You can also use the Yum repository provided by the PostgreSQL project (known as PGDG) to get a different version. This may allow versions not yet offered by operating system package repositories.

Section 1.4: How to install PostgreSQL via MacPorts on OSX

In order to install PostgreSQL on OSX, you need to know which versions are currently supported.

Use this command to see what versions you have available.

```
sudo port list | grep "^postgresql[[:digit:]]\{2\}[[:space:]]"
```

You should get a list that looks something like the following:

```
postgresql80                 @8.0.26         databases/postgresql80
postgresql81                 @8.1.23         databases/postgresql81
postgresql82                 @8.2.23         databases/postgresql82
postgresql83                 @8.3.23         databases/postgresql83
postgresql84                 @8.4.22         databases/postgresql84
postgresql90                 @9.0.23         databases/postgresql90
postgresql91                 @9.1.22         databases/postgresql91
postgresql92                 @9.2.17         databases/postgresql92
postgresql93                 @9.3.13         databases/postgresql93
postgresql94                 @9.4.8          databases/postgresql94
postgresql95                 @9.5.3          databases/postgresql95
postgresql96                 @9.6beta2       databases/postgresql96
```

In this example, the most recent version of PostgreSQL that is supported in 9.6, so we will install that.

```
sudo port install postgresql96-server postgresql96
```

You will see an installation log like this:

```
--->  Computing dependencies for postgresql96-server
--->  Dependencies to be installed: postgresql96
--->  Fetching archive for postgresql96
--->  Attempting to fetch postgresql96-9.6beta2_0.darwin_15.x86_64.tbz2 from
https://packages.macports.org/postgresql96
--->  Attempting to fetch postgresql96-9.6beta2_0.darwin_15.x86_64.tbz2.rmd160 from
https://packages.macports.org/postgresql96
--->  Installing postgresql96 @9.6beta2_0
--->  Activating postgresql96 @9.6beta2_0

To use the postgresql server, install the postgresql96-server port

--->  Cleaning postgresql96
--->  Fetching archive for postgresql96-server
--->  Attempting to fetch postgresql96-server-9.6beta2_0.darwin_15.x86_64.tbz2 from
https://packages.macports.org/postgresql96-server
--->  Attempting to fetch postgresql96-server-9.6beta2_0.darwin_15.x86_64.tbz2.rmd160 from
https://packages.macports.org/postgresql96-server
--->  Installing postgresql96-server @9.6beta2_0
--->  Activating postgresql96-server @9.6beta2_0

To create a database instance, after install do
sudo mkdir -p /opt/local/var/db/postgresql96/defaultdb
sudo chown postgres:postgres /opt/local/var/db/postgresql96/defaultdb
sudo su postgres -c '/opt/local/lib/postgresql96/bin/initdb -D
/opt/local/var/db/postgresql96/defaultdb'

--->  Cleaning postgresql96-server
--->  Computing dependencies for postgresql96
--->  Cleaning postgresql96
--->  Updating database of binaries
--->  Scanning binaries for linking errors
--->  No broken files found.
```

The log provides instructions on the rest of the steps for installation, so we do that next.

```
sudo mkdir -p /opt/local/var/db/postgresql96/defaultdb
sudo chown postgres:postgres /opt/local/var/db/postgresql96/defaultdb
sudo su postgres -c '/opt/local/lib/postgresql96/bin/initdb -D
/opt/local/var/db/postgresql96/defaultdb'
```

Now we start the server:

```
sudo port load -w postgresql96-server
```

Verify that we can connect to the server:

```
su postgres -c psql
```

You will see a prompt from postgres:

```
psql (9.6.1)
Type "help" for help.
```

```
postgres=#
```

Here you can type a query to see that the server is running.

```
postgres=#SELECT setting FROM pg_settings WHERE NAME='data_directory';
```

And see the response:

```
                setting
----------------------------------------
/opt/local/var/db/postgresql96/defaultdb
(1 row)
postgres=#
```

Type \q to quit:

```
postgres=#\q
```

And you will be back at your shell prompt.

Congratulations! You now have a running PostgreSQL instance on OS/X.

Section 1.5: Install postgresql with brew on Mac

Homebrew calls itself '*the missing package manager for macOS*'. It can be used to build and install applications and libraries. Once installed, you can use the `brew` command to install PostgreSQL and it's dependencies as follows:

```
brew UPDATE
brew install postgresql
```

Homebrew generally installs the latest stable version. If you need a different one then `brew SEARCH postgresql` will list the versions available. If you need PostgreSQL built with particular options then `brew info postgresql` will list which options are supported. If you require an unsupported build option, you may have to do the build yourself, but can still use Homebrew to install the common dependencies.

Start the server:

```
brew services START postgresql
```

Open the PostgreSQL prompt

```
psql
```

If psql complains that there's no corresponding database for your user, run **CREATEDB**.

Section 1.6: Postgres.app for Mac OSX

An extremely simple tool for installing PostgreSQL on a Mac is available by downloading Postgres.app.
You can change preferences to have PostgreSQL run in the background or only when the application is running.

Chapter 2: Data Types

PostgreSQL has a rich set of native data types available to users. Users can add new types to PostgreSQL using the CREATE TYPE command.

https://www.postgresql.org/docs/9.6/static/datatype.html

Section 2.1: Numeric Types

Name	Storage Size	Description	Range
SMALLINT	2 bytes	small-range integer	-32768 to +32767
INTEGER	4 bytes	ypical choice for integer	-2147483648 to +2147483647
BIGINT	8 bytes	large-range integer	-9223372036854775808 to +9223372036854775807
DECIMAL	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
NUMERIC	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
REAL	4 bytes	variable-precision, inexact	6 decimal digits precision
DOUBLE PRECISION	8 bytes	variable-precision, inexact	15 decimal digits precision
smallserial	2 bytes	small autoincrementing integer	1 to 32767
serial	4 bytes	autoincrementing integer	1 to 2147483647
BIGSERIAL	8 bytes	large autoincrementing integer	1 to 9223372036854775807
int4range		Range of integer	
int8range		Range of bigint	
numrange		Range of numeric	

Section 2.2: Date/ Time Types

Name	Storage Size	Description	Low Value	High Value	Resolution
TIMESTAMP (without time zone)	8 bytes	both date and time (no time zone)	4713 BC	294276 AD	1 microsecond / 14 digits
TIMESTAMP (with time zone)	8 bytes	both date and time, with time zone	4713 BC	294276 AD	1 microsecond / 14 digits
DATE	4 bytes	date (no time of day)	4713 BC	5874897 AD	1 day
TIME (without time zone)	8 bytes	time of day (no date)	00:00:00	24:00:00	1 microsecond / 14 digits
TIME (with time zone)	12 bytes	times of day only, with time zone	00:00:00+1459	24:00:00-1459	1 microsecond / 14 digits
INTERVAL	16 bytes	time interval	-178000000 years	178000000 years	1 microsecond / 14 digits
tsrange		range of timestamp without time zone			
tstzrange		range of timestamp with time zone			
daterange		range of date			

Section 2.3: Geometric Types

Name	Storage Size	Description	Representation
`point`	16 bytes	Point on a plane	(x,y)
`line`	32 bytes	Infinite line	{A,B,C}
`lseg`	32 bytes	Finite line segment	((x1,y1),(x2,y2))
`BOX`	32 bytes	Rectangular box	((x1,y1),(x2,y2))
`path`	16+16n bytes	Closed path (similar to polygon)	((x1,y1),...)
`path`	16+16n bytes	Open path	[(x1,y1),...]
`polygon`	40+16n bytes	Polygon (similar to closed path)	((x1,y1),...)
`CIRCLE`	24 bytes	Circle	<(x,y),r> (center point and radius)

Section 2.4: Network Adress Types

Name	Storage Size	Description
`CIDR`	7 or 19 bytes	IPv4 and IPv6 networks
`INET`	7 or 19 bytes	IPv4 and IPv6 hosts and networks
`macaddr`	6 bytes	MAC addresses

Section 2.5: Character Types

Name	Description
`CHARACTER varying(n)`, `varchar(n)`	variable-length with limit
`character(n)`, `char(n)`	fixed-length, blank padded
`TEXT`	variable unlimited length

Section 2.6: Arrays

In PostgreSQL you can create Arrays of any built-in, user-defined or enum type. In default there is no limit to an Array, but you *can* specify it.

Declaring an Array

```
SELECT INTEGER[];
SELECT INTEGER[3];
SELECT INTEGER[][];
SELECT INTEGER[3][3];
SELECT INTEGER ARRAY;
SELECT INTEGER ARRAY[3];
```

Creating an Array

```
SELECT '{0,1,2}';
SELECT '{{0,1},{1,2}}';
SELECT ARRAY[0,1,2];
SELECT ARRAY[ARRAY[0,1],ARRAY[1,2]];
```

Accessing an Array

By default PostgreSQL uses a one-based numbering convention for arrays, that is, an array of n elements starts with `ARRAY[1]` and ends with `ARRAY[n]`.

```
--accesing a spefific element
```

```
WITH arr AS (SELECT ARRAY[0,1,2] int_arr) SELECT int_arr[1] FROM arr;

int_arr
---------
        0
(1 ROW)

--sclicing an array
WITH arr AS (SELECT ARRAY[0,1,2] int_arr) SELECT int_arr[1:2] FROM arr;

int_arr
---------
    {0,1}
(1 ROW)
```

Getting information about an array

```
--array dimensions (as text)
WITH arr AS (SELECT ARRAY[0,1,2] int_arr) SELECT ARRAY_DIMS(int_arr) FROM arr;

array_dims
------------
       [1:3]
(1 ROW)

--length of an array dimension
 WITH arr AS (SELECT ARRAY[0,1,2] int_arr) SELECT ARRAY_LENGTH(int_arr,1) FROM arr;

 array_length
 --------------
              3
 (1 ROW)

--total number of elements across all dimensions
 WITH arr AS (SELECT ARRAY[0,1,2] int_arr) SELECT cardinality(int_arr) FROM arr;

 cardinality
 -------------
             3
 (1 ROW)
```

Array functions

> will be added

Chapter 3: Dates, Timestamps, and Intervals

Section 3.1: SELECT the last day of month

You can select the last day of month.

```
SELECT (DATE_TRUNC('MONTH', ('201608'||'01')::DATE) + INTERVAL '1 MONTH - 1 day')::DATE;
```

`201608` is replaceable with a variable.

Section 3.2: Cast a timestamp or interval to a string

You can convert a `TIMESTAMP` or `INTERVAL` value to a string with the `TO_CHAR()` function:

```
SELECT TO_CHAR('2016-08-12 16:40:32'::TIMESTAMP, 'DD Mon YYYY HH:MI:SSPM');
```

This statement will produce the string "12 Aug 2016 04:40:32PM". The formatting string can be modified in many different ways; the full list of template patterns can be found here.

Note that you can also insert plain text into the formatting string and you can use the template patterns in any order:

```
SELECT TO_CHAR('2016-08-12 16:40:32'::TIMESTAMP,
               '"Today is "FMDay", the "DDth" day of the month of "FMMonth" of "YYYY');
```

This will produce the string "Today is Saturday, the 12th day of the month of August of 2016". You should keep in mind, though, that any template patterns - even the single letter ones like "I", "D", "W" - are converted, unless the plain text is in double quotes. As a safety measure, you should put all plain text in double quotes, as done above.

You can localize the string to your language of choice (day and month names) by using the TM (translation mode) modifier. This option uses the localization setting of the server running PostgreSQL or the client connecting to it.

```
SELECT TO_CHAR('2016-08-12 16:40:32'::TIMESTAMP, 'TMDay, DD" de "TMMonth" del año "YYYY');
```

With a Spanish locale setting this produces "Sábado, 12 de Agosto del año 2016".

Section 3.3: Count the number of records per week

```
SELECT DATE_TRUNC('week', <>) AS "Week" , COUNT(*)
FROM <>
GROUP BY 1
ORDER BY 1;
```

Chapter 4: Table Creation

Section 4.1: Show table definition

Open the `psql` command line tool connected to the database where your table is. Then type the following command:

```
\d tablename
```

To get extended information type

```
\d+ tablename
```

If you have forgotten the name of the table, just type \d into psql to obtain a list of tables and views in the current database.

Section 4.2: Create table from select

Let's say you have a table called person:

```
CREATE TABLE person (
    person_id BIGINT NOT NULL,
    last_name VARCHAR(255) NOT NULL,
    first_name VARCHAR(255),
    age INT NOT NULL,
    PRIMARY KEY (person_id)
);
```

You can create a new table of people over 30 like this:

```
CREATE TABLE people_over_30 AS SELECT * FROM person WHERE age > 30;
```

Section 4.3: Create unlogged table

You can create unlogged tables so that you can make the tables considerably faster. Unlogged table skips writing **WRITE**-ahead log which means it's not crash-safe and unable to replicate.

```
CREATE UNLOGGED TABLE person (
    person_id BIGINT NOT NULL PRIMARY KEY,
    last_name VARCHAR(255) NOT NULL,
    first_name VARCHAR(255),
    address VARCHAR(255),
    city VARCHAR(255)
);
```

Section 4.4: Table creation with Primary Key

```
CREATE TABLE person (
    person_id BIGINT NOT NULL,
    last_name VARCHAR(255) NOT NULL,
    first_name VARCHAR(255),
    address VARCHAR(255),
    city VARCHAR(255),
    PRIMARY KEY (person_id)
```

```
);
```

Alternatively, you can place the `PRIMARY KEY` constraint directly in the column definition:

```
CREATE TABLE person (
    person_id BIGINT NOT NULL PRIMARY KEY,
    last_name VARCHAR(255) NOT NULL,
    first_name VARCHAR(255),
    address VARCHAR(255),
    city VARCHAR(255)
);
```

It is recommended that you use lower case names for the table and as well as all the columns. If you use upper case names such as `Person` you would have to wrap that name in double quotes (`"Person"`) in each and every query because PostgreSQL enforces case folding.

Section 4.5: Create a table that references other table

In this example, User Table will have a column that references the Agency table.

```
CREATE TABLE agencies ( -- first create the agency table
  id SERIAL PRIMARY KEY,
  NAME TEXT NOT NULL
)

CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  agency_id NOT NULL INTEGER REFERENCES agencies(id) DEFERRABLE INITIALLY DEFERRED -- this is going
to references your agency table.
)
```

Chapter 5: SELECT

Section 5.1: SELECT using WHERE

In this topic we will base on this table of users :

```
CREATE TABLE sch_test.user_table
(
  id serial NOT NULL,
  username CHARACTER VARYING,
  pass CHARACTER VARYING,
  first_name CHARACTER varying(30),
  last_name CHARACTER varying(30),
  CONSTRAINT user_table_pkey PRIMARY KEY (id)
)
```

```
+----+------------+-----------+----------+------+
| id | first_name | last_name | username | pass |
+----+------------+-----------+----------+------+
| 1  | hello      | world     | hello    | word |
+----+------------+-----------+----------+------+
| 2  | root       | me        | root     | toor |
+----+------------+-----------+----------+------+
```

Syntax

Select every thing:

```
SELECT * FROM schema_name.table_name WHERE <condition>;
```

Select some fields :

```
SELECT field1, field2 FROM schema_name.table_name WHERE <condition>;
```

Examples

```
-- SELECT every thing where id = 1
SELECT * FROM schema_name.table_name WHERE id = 1;

-- SELECT id where username = ? and pass = ?
SELECT id FROM schema_name.table_name WHERE username = 'root' AND pass = 'toor';

-- SELECT first_name where id not equal 1
SELECT first_name FROM schema_name.table_name WHERE id != 1;
```

Chapter 6: Find String Length / Character Length

To get length of "character varying", "text" fields, Use char_length() or character_length().

Section 6.1: Example to get length of a character varying field

Example 1, Query: `SELECT CHAR_LENGTH('ABCDE')`

Result:

> 5

Example 2, Query: `SELECT CHARACTER_LENGTH('ABCDE')`

Result:

> 5

Chapter 7: COALESCE

Coalesce returns the first none null argument from a set of arguments. Only the first non null argument is return, all subsequent arguments are ignored. The function will evaluate to null if all arguments are null.

Section 7.1: Single non null argument

```
PGSQL> SELECT COALESCE(NULL, NULL, 'HELLO WORLD');

COALESCE
--------
'HELLO WORLD'
```

Section 7.2: Multiple non null arguments

```
PGSQL> SELECT COALESCE(NULL, NULL, 'first non null', NULL, NULL, 'second non null');

coalesce
--------
'first non null'
```

Section 7.3: All null arguments

```
PGSQL> SELECT COALESCE(NULL, NULL, NULL);

COALESCE
--------
```

Chapter 8: INSERT

Section 8.1: Insert data using COPY

COPY is PostgreSQL's bulk-insert mechanism. It's a convenient way to transfer data between files and tables, but it's also far faster than `INSERT` when adding more than a few thousand rows at a time.

Let's begin by creating sample data file.

```
cat > samplet_data.csv

1,Yogesh
2,Raunak
3,Varun
4,Kamal
5,Hari
6,Amit
```

And we need a two column table into which this data can be imported into.

```
CREATE TABLE copy_test(id INT, NAME varchar(8));
```

Now the actual copy operation, this will create six records in the table.

```
COPY copy_test FROM '/path/to/file/sample_data.csv' DELIMITER ',';
```

Instead of using a file on disk, can insert data from **STDIN**

```
COPY copy_test FROM STDIN DELIMITER ',';
Enter DATA TO be copied followed BY a newline.
END WITH a backslash AND a period ON a line BY itself.
>> 7,Amol
>> 8,Amar
>> \.
TIME: 85254.306 ms

SELECT * FROM copy_test ;
```

```
id |  name
----+--------
1 | Yogesh
3 | Varun
5 | Hari
7 | Amol
2 | Raunak
4 | Kamal
6 | Amit
8 | Amar
```

Also you can copy data from a table to file as below:

```
COPY copy_test TO 'path/to/file/sample_data.csv'  DELIMITER ',';
```

For more details on COPY you can check here

Section 8.2: Inserting multiple rows

You can insert multiple rows in the database at the same time:

```
INSERT INTO person (NAME, age) VALUES
  ('john doe', 25),
  ('jane doe', 20);
```

Section 8.3: INSERT data and RETURING values

If you are inserting data into a table with an auto increment column and if you want to get the value of the auto increment column.

Say you have a table called `my_table`:

```
CREATE TABLE my_table
(
id serial NOT NULL, -- serial data type is auto incrementing four-byte integer
NAME CHARACTER VARYING,
contact_number INTEGER,
CONSTRAINT my_table_pkey PRIMARY KEY (id)
);
```

If you want to insert data into `my_table` and get the id of that row:

```
INSERT INTO my_table(NAME, contact_number) VALUES ( 'USER', 8542621) RETURNING id;
```

Above query will return the id of the row where the new record was inserted.

Section 8.4: Basic INSERT

Let's say we have a simple table called person:

```
CREATE TABLE person (
    person_id BIGINT,
    NAME VARCHAR(255).
    age INT,
    city VARCHAR(255)
);
```

The most basic insert involves inserting all values in the table:

```
INSERT INTO person VALUES (1, 'john doe', 25, 'new york');
```

If you want to insert only specific columns, you need to explicitly indicate which columns:

```
INSERT INTO person (NAME, age) VALUES ('john doe', 25);
```

Note that if any constraints exist on the table , such as NOT NULL, you will be required to include those columns in either case.

Section 8.5: Insert from select

You can insert data in a table as the result of a select statement:

```
INSERT INTO person SELECT * FROM tmp_person WHERE age < 30;
```

Note that the projection of the select must match the columns required for the insert. In this case, the `tmp_person` table has the same columns as `person`.

Section 8.6: UPSERT - INSERT ... ON CONFLICT DO UPDATE..

since version 9.5 postgres offers `UPSERT` functionality with **`INSERT`** statement.

Say you have a table called my_table, created in several previous examples. We insert a row, returning PK value of inserted row:

```
b=# INSERT INTO my_table (name,contact_number) values ('one',333) RETURNING id;
id
----
2
(1 row)

INSERT 0 1
```

Now if we try to insert row with existing unique key it will raise an exception:

```
b=# INSERT INTO my_table VALUES (2,'one',333);
ERROR:  duplicate KEY VALUE violates UNIQUE CONSTRAINT "my_table_pkey"
DETAIL:  KEY (id)=(2) already EXISTS.
```

Upsert functionality offers ability to insert it anyway, solving the conflict:

```
b=# INSERT INTO my_table values (2,'one',333) ON CONFLICT (id) DO UPDATE SET name =
my_table.name||' changed to: "two" at '||now() returning *;
id |                        name                              | contact_number
----+----------------------------------------------------------------------------------------------
--------------+----------------
2 | one changed to: "two" at 2016-11-23 08:32:17.105179+00 |            333
(1 row)

INSERT 0 1
```

Section 8.7: SELECT data into file

You can COPY table and paste it into a file.

```
postgres=# select * from my_table;
c1 | c2 | c3
----+----+----
1 |  1 |  1
2 |  2 |  2
3 |  3 |  3
4 |  4 |  4
5 |  5 |
(5 rows)

postgres=# copy my_table to '/home/postgres/my_table.txt' using delimiters '|' with null as
'null_string' csv header;
COPY 5
```

```
postgres=# \! cat my_table.txt
c1|c2|c3
1|1|1
2|2|2
3|3|3
4|4|4
5|5|null_string
```

Chapter 9: UPDATE

Section 9.1: Updating a table based on joining another table

You can also update data in a table based on data from another table:

```
UPDATE person
SET state_code = cities.state_code
FROM cities
WHERE cities.city = city;
```

Here we are joining the `person` `city` column to the `cities` `city` column in order to get the city's state code. This is then used to update the `state_code` column in the `person` table.

Section 9.2: Update all rows in a table

You update all rows in table by simply providing a `column_name = VALUE`:

```
UPDATE person SET planet = 'Earth';
```

Section 9.3: Update all rows meeting a condition

```
UPDATE person SET state = 'NY' WHERE city = 'New York';
```

Section 9.4: Updating multiple columns in table

You can update multiple columns in a table in the same statement, separating `col=val` pairs with commas:

```
UPDATE person
   SET country = 'USA',
       state = 'NY'
WHERE city = 'New York';
```

Chapter 10: JSON Support

JSON - Java Script Object Notation , Postgresql support JSON Data type since 9.2 version. There are some predefined function and operators to access the JSON data. The `->` operator returns the key of JSON column. The `->>` operator returns the value of JSON Column.

Section 10.1: Using JSONb operators

Creating a DB and a Table

```
DROP DATABASE IF EXISTS books_db;
CREATE DATABASE books_db WITH ENCODING='UTF8' TEMPLATE template0;

DROP TABLE IF EXISTS books;

CREATE TABLE books (
  id SERIAL PRIMARY KEY,
  client TEXT NOT NULL,
  DATA JSONb NOT NULL
);
```

Populating the DB

```
INSERT INTO books(client, DATA) VALUES (
    'Joe',
    '{ "title": "Siddhartha", "author": { "first_name": "Herman", "last_name": "Hesse" } }'
),(
    'Jenny',
    '{ "title": "Dharma Bums", "author": { "first_name": "Jack", "last_name": "Kerouac" } }'
),(
    'Jenny',
    '{ "title": "100 años de soledad", "author": { "first_name": "Gabo", "last_name": "Marquéz" }
}'
);
```

Lets see everything inside the table books:

```
SELECT * FROM books;
```

Output:

id integer	client character varying	data jsonb
1	Joe	{"title": "Siddhartha", "author": {"last name": "Hesse", "first name": "Herman"}}
2	Jenny	{"title": "Dharma Bums", "author": {"last name": "Kerouac", "first name": "Jack"}}
3	Jenny	{"title": "100 años de soledad", "author": {"last name": "Marquéz", "first name": "Gabo"}}

`->` operator returns values out of JSON columns

Selecting 1 column:

```
SELECT client,
    DATA->'title' AS title
    FROM books;
```

Output:

	client character varying	title jsonb
1	Joe	"Siddhartha"
2	Jenny	"Dharma Bums"
3	Jenny	"100 años de soledad"

Selecting 2 columns:

```
SELECT client,
   DATA->'title' AS title, DATA->'author' AS author
   FROM books;
```

Output:

client character varying	title jsonb	author jsonb
Joe	"Siddhartha"	{"last_name": "Hesse", "first_name": "Herman"}
Jenny	"Dharma Bums"	{"last name": "Kerouac", "first name": "Jack"}
Jenny	"100 años de soledad"	{"last name": "Marquéz", "first name": "Gabo"}

`->` vs `->>`

The `->` operator returns the original JSON type (which might be an object), whereas `->>` returns text.

Return NESTED objects

You can use the `->` to return a nested object and thus chain the operators:

```
SELECT client,
   DATA->'author'->'last_name' AS author
   FROM books;
```

Output:

client character varying	author jsonb
Joe	"Hesse"
Jenny	"Kerouac"
Jenny	"Marquéz"

Filtering

Select rows based on a value inside your JSON:

```
SELECT
client,
DATA->'title' AS title
FROM books
 WHERE DATA->'title' = '"Dharma Bums"';
```

Notice WHERE uses `->` so we must compare to JSON `'"Dharma Bums"'`

Or we could use `->>` and compare to `'Dharma Bums'`

Output:

client character varying	title jsonb
Jenny	"Dharma Bums"

Nested filtering

Find rows based on the value of a nested JSON object:

```
SELECT
 client,
 DATA->'title' AS title
 FROM books
  WHERE DATA->'author'->>'last_name' = 'Kerouac';
```

Output:

client character varying	title jsonb
Jenny	"Dharma Bums"

A real world example

```
CREATE TABLE events (
  NAME varchar(200),
  visitor_id varchar(200),
  properties json,
  browser json
);
```

We're going to store events in this table, like pageviews. Each event has properties, which could be anything (e.g. current page) and also sends information about the browser (like OS, screen resolution, etc). Both of these are completely free form and could change over time (as we think of extra stuff to track).

```
INSERT INTO events (NAME, visitor_id, properties, browser) VALUES
(
  'pageview', '1',
  '{ "page": "/" }',
  '{ "name": "Chrome", "os": "Mac", "resolution": { "x": 1440, "y": 900 } }'
),(
  'pageview', '2',
  '{ "page": "/" }',
  '{ "name": "Firefox", "os": "Windows", "resolution": { "x": 1920, "y": 1200 } }'
),(
  'pageview', '1',
  '{ "page": "/account" }',
  '{ "name": "Chrome", "os": "Mac", "resolution": { "x": 1440, "y": 900 } }'
),(
  'purchase', '5',
  '{ "amount": 10 }',
  '{ "name": "Firefox", "os": "Windows", "resolution": { "x": 1024, "y": 768 } }'
),(
  'purchase', '15',
  '{ "amount": 200 }',
  '{ "name": "Firefox", "os": "Windows", "resolution": { "x": 1280, "y": 800 } }'
),(
  'purchase', '15',
  '{ "amount": 500 }',
  '{ "name": "Firefox", "os": "Windows", "resolution": { "x": 1280, "y": 800 } }'
);
```

Now lets select everything:

```
SELECT * FROM events;
```

Output:

name character varying(200)	visitor_id character varying(200)	properties json	browser json
pageview	1	{ "page": "/" }	{ "name": "Chrome", "os": "Mac", "resolution": { "x": 1440, "y": 900 } }
pageview	2	{ "page": "/" }	{ "name": "Firefox", "os": "Windows", "resolution": { "x": 1920, "y": 1200 } }
pageview	1	{ "page": "/account" }	{ "name": "Chrome", "os": "Mac", "resolution": { "x": 1440, "y": 900 } }
purchase	5	{ "amount": 10 }	{ "name": "Firefox", "os": "Windows", "resolution": { "x": 1024, "y": 768 } }
purchase	15	{ "amount": 200 }	{ "name": "Firefox", "os": "Windows", "resolution": { "x": 1280, "y": 800 } }
purchase	15	{ "amount": 500 }	{ "name": "Firefox", "os": "Windows", "resolution": { "x": 1280, "y": 800 } }

JSON operators + PostgreSQL aggregate functions

Using the JSON operators, combined with traditional PostgreSQL aggregate functions, we can pull out whatever we want. You have the full might of an RDBMS at your disposal.

- Lets see browser usage:

```
SELECT browser->>'name' AS browser,
  COUNT(browser)
  FROM events
  GROUP BY browser->>'name';
```

Output:

browser text	count bigint
Firefox	4
Chrome	2

- Total revenue per visitor:

```
SELECT visitor_id, SUM(CAST(properties->>'amount' AS INTEGER)) AS total
FROM events
WHERE CAST(properties->>'amount' AS INTEGER) > 0
GROUP BY visitor_id;
```

Output:

visitor_id character varying(200)	total bigint
5	10
15	700

- Average screen resolution

```
SELECT AVG(CAST(browser->'resolution'->>'x' AS INTEGER)) AS width,
  AVG(CAST(browser->'resolution'->>'y' AS INTEGER)) AS height
FROM events;
```

Output:

width numeric	height numeric
1397.3333333333333333	894.6666666666666667

More examples and documentation here and here.

Section 10.2: Querying complex JSON documents

Taking a complex JSON document in a table:

```
CREATE TABLE mytable (DATA JSONB NOT NULL);
CREATE INDEX mytable_idx ON mytable USING gin (DATA jsonb_path_ops);
INSERT INTO mytable VALUES($$
{
    "name": "Alice",
    "emails": [
        "alice1@test.com",
        "alice2@test.com"
    ],
    "events": [
        {
            "type": "birthday",
            "date": "1970-01-01"
        },
        {
            "type": "anniversary",
            "date": "2001-05-05"
        }
    ],
    "locations": {
        "home": {
            "city": "London",
            "country": "United Kingdom"
        },
        "work": {
            "city": "Edinburgh",
            "country": "United Kingdom"
        }
    }
}
$$);
```

Query for a top-level element:

```
SELECT DATA->>'name' FROM mytable WHERE DATA @> '{"name":"Alice"}';
```

Query for a simple item in an array:

```
SELECT DATA->>'name' FROM mytable WHERE DATA @> '{"emails":["alice1@test.com"]}';
```

Query for an object in an array:

```
SELECT DATA->>'name' FROM mytable WHERE DATA @> '{"events":[{"type":"anniversary"}]}';
```

Query for a nested object:

```
SELECT DATA->>'name' FROM mytable WHERE DATA @> '{"locations":{"home":{"city":"London"}}}';
```

Performance of `@>` compared to `->` and `->>`

It is important to understand the performance difference between using `@>`, `->` and `->>` in the **`WHERE`** part of the query. Although these two queries appear to be broadly equivalent:

```
SELECT DATA FROM mytable WHERE DATA @> '{"name":"Alice"}';
SELECT DATA FROM mytable WHERE DATA->'name' = '"Alice"';
SELECT DATA FROM mytable WHERE DATA->>'name' = 'Alice';
```

the first statement will use the index created above whereas the latter two will not, requiring a complete table scan.

It is still allowable to use the `->` operator when obtaining resultant data, so the following queries will also use the index:

```
SELECT DATA->'locations'->'work' FROM mytable WHERE DATA @> '{"name":"Alice"}';
SELECT DATA->'locations'->'work'->>'city' FROM mytable WHERE DATA @> '{"name":"Alice"}';
```

Section 10.3: Creating a pure JSON table

To create a pure JSON table you need to provide a single field with the type `JSONB`:

```
CREATE TABLE mytable (DATA JSONB NOT NULL);
```

You should also create a basic index:

```
CREATE INDEX mytable_idx ON mytable USING gin (DATA jsonb_path_ops);
```

At this point you can insert data in to the table and query it efficiently.

Chapter 11: Aggregate Functions

Section 11.1: Simple statistics: min(), max(), avg()

In order to determine some simple statistics of a value in a column of a table, you can use an aggregate function.

If your `individuals` table is:

Name	Age
Allie	17
Amanda	14
Alana	20

You could write this statement to get the minimum, maximum and average value:

```
SELECT MIN(age), MAX(age), AVG(age)
FROM individuals;
```

Result:

min	max	avg
14	20	17

Section 11.2: regr_slope(Y, X) : slope of the least-squares-fit linear equation determined by the (X, Y) pairs

To illustrate how to use regr_slope(Y,X), I applied it to a real world problem. In Java, if you don't clean up memory properly, the garbage can get stuck and fill up the memory. You dump statistics every hour about memory utilization of different classes and load it into a postgres database for analysis.

All memory leak candidates will have a trend of consuming more memory as more time passes. If you plot this trend, you would imagine a line going up and to the left:

```
    ^
|
s   |  Legend:
i   |  *  - data point
z   |  -- - trend
e   |
(   |
b   |                  *
y   |                      --
t   |                   --
e   |              * --    *
s   |           --
)   |       *--      *
|     --    *
|  -- *
---------------------------------------->
time
```

Suppose you have a table containing heap dump histogram data (a mapping of classes to how much memory they consume):

```
CREATE TABLE heap_histogram (
    -- when the heap histogram was taken
    histwhen TIMESTAMP WITHOUT TIME ZONE NOT NULL,
    -- the object type bytes are referring to
    -- ex: java.util.String
    CLASS CHARACTER VARYING NOT NULL,
    -- the size in bytes used by the above class
    bytes INTEGER NOT NULL
);
```

To compute the slope for each class, we group by over the class. The HAVING clause > 0 ensures that we get only candidates with a positive slop (a line going up and to the left). We sort by the slope descending so that we get the classes with the largest rate of memory increase at the top.

```
-- epoch returns seconds
SELECT CLASS, REGR_SLOPE(bytes,EXTRACT(epoch FROM histwhen)) AS slope
    FROM public.heap_histogram
    GROUP BY CLASS
    HAVING REGR_SLOPE(bytes,EXTRACT(epoch FROM histwhen)) > 0
    ORDER BY slope DESC ;
```

Output:

```
         class              |        slope
----------------------------+---------------------
java.util.ArrayList         |     71.7993806279174
java.util.HashMap           |     49.0324576155785
java.lang.String            |     31.7770770326123
joe.schmoe.BusinessObject   |     23.2036817108056
java.lang.ThreadLocal       |     20.9013528767851
```

From the output we see that java.util.ArrayList's memory consumption is increasing the fastest at 71.799 bytes per second and is potentially part of the memory leak.

Section 11.3: string_agg(expression, delimiter)

You can concatenate strings separated by delimiter using the `STRING_AGG()` function.

If your `individuals` table is:

Name	Age	Country
Allie	15	USA
Amanda	14	USA
Alana	20	Russia

You could write `SELECT ... GROUP BY` statement to get names from each country:

```
SELECT STRING_AGG(NAME, ', ') AS NAMES, country
FROM individuals
GROUP BY country;
```

Note that you need to use a `GROUP BY` clause because `STRING_AGG()` is an aggregate function.

Result:

names	country
Allie, Amanda	USA
Alana	Russia

More PostgreSQL aggregate function described here

Chapter 12: Common Table Expressions (WITH)

Section 12.1: Common Table Expressions in SELECT Queries

Common table expressions support extracting portions of larger queries. For example:

```
WITH sales AS (
  SELECT
    orders.ordered_at,
    orders.user_id,
    SUM(orders.amount) AS total
  FROM orders
  GROUP BY orders.ordered_at, orders.user_id
)
SELECT
  sales.ordered_at,
  sales.total,
  users.NAME
FROM sales
JOIN users USING (user_id)
```

Section 12.2: Traversing tree using WITH RECURSIVE

```
CREATE TABLE empl (
    NAME TEXT PRIMARY KEY,
    boss TEXT NULL
        REFERENCES NAME
            ON UPDATE CASCADE
            ON DELETE CASCADE
        DEFAULT NULL
);

INSERT INTO empl VALUES ('Paul',NULL);
INSERT INTO empl VALUES ('Luke','Paul');
INSERT INTO empl VALUES ('Kate','Paul');
INSERT INTO empl VALUES ('Marge','Kate');
INSERT INTO empl VALUES ('Edith','Kate');
INSERT INTO empl VALUES ('Pam','Kate');
INSERT INTO empl VALUES ('Carol','Luke');
INSERT INTO empl VALUES ('John','Luke');
INSERT INTO empl VALUES ('Jack','Carol');
INSERT INTO empl VALUES ('Alex','Carol');

WITH RECURSIVE t(LEVEL,path,boss,NAME) AS (
        SELECT 0,NAME,boss,NAME FROM empl WHERE boss IS NULL
    UNION
        SELECT
            LEVEL + 1,
            path || ' > ' || empl.NAME,
            empl.boss,
            empl.NAME
        FROM
            empl JOIN t
                ON empl.boss = t.NAME
) SELECT * FROM t ORDER BY path;
```

Chapter 13: Window Functions

Section 13.1: generic example

Preparing data:

```
CREATE TABLE wf_example(i INT, t TEXT,ts timestamptz,b BOOLEAN);
INSERT INTO wf_example SELECT 1,'a','1970.01.01',TRUE;
INSERT INTO wf_example SELECT 1,'a','1970.01.01',FALSE;
INSERT INTO wf_example SELECT 1,'b','1970.01.01',FALSE;
INSERT INTO wf_example SELECT 2,'b','1970.01.01',FALSE;
INSERT INTO wf_example SELECT 3,'b','1970.01.01',FALSE;
INSERT INTO wf_example SELECT 4,'b','1970.02.01',FALSE;
INSERT INTO wf_example SELECT 5,'b','1970.03.01',FALSE;
INSERT INTO wf_example SELECT 2,'c','1970.03.01',TRUE;
```

Running:

```
SELECT *
  , DENSE_RANK() OVER (ORDER BY i) dist_by_i
  , LAG(t) OVER () prev_t
  , NTH_VALUE(i, 6) OVER () nth
  , COUNT(TRUE) OVER (PARTITION BY i) num_by_i
  , COUNT(TRUE) OVER () num_all
  , NTILE(3) over() ntile
FROM wf_example
;
```

Result:

```
 i | t |           ts           | b | dist_by_i | prev_t | nth | num_by_i | num_all | ntile
---+---+------------------------+---+-----------+--------+-----+----------+---------+-------
1 | a | 1970-01-01 00:00:00+01 | f |         1 |        |   3 |        3 |       8 |     1
1 | a | 1970-01-01 00:00:00+01 | t |         1 | a      |   3 |        3 |       8 |     1
1 | b | 1970-01-01 00:00:00+01 | f |         1 | a      |   3 |        3 |       8 |     1
2 | c | 1970-03-01 00:00:00+01 | t |         2 | b      |   3 |        2 |       8 |     2
2 | b | 1970-01-01 00:00:00+01 | f |         2 | c      |   3 |        2 |       8 |     2
3 | b | 1970-01-01 00:00:00+01 | f |         3 | b      |   3 |        1 |       8 |     2
4 | b | 1970-02-01 00:00:00+01 | f |         4 | b      |   3 |        1 |       8 |     3
5 | b | 1970-03-01 00:00:00+01 | f |         5 | b      |   3 |        1 |       8 |     3
(8 rows)
```

Explanation:

dist_by_i: `DENSE_RANK() OVER (ORDER BY i)` is like a row_number per distinct values. Can be used for the number of distinct values of i (`COUNT(DISTINCT i)` wold not work). Just use the maximum value.

prev_t: `LAG(t) OVER ()` is a previous value of t over the whole window. mind that it is null for the first row.

nth: `NTH_VALUE(i, 6) OVER ()` is the value of sixth rows column i over the whole window

num_by_i: `COUNT(TRUE) OVER (PARTITION BY i)` is an amount of rows for each value of i

num_all: `COUNT(TRUE) OVER ()` is an amount of rows over a whole window

ntile: `NTILE(3) over()` splits the whole window to 3 (as much as possible) equal in quantity parts

Section 13.2: column values vs dense_rank vs rank vs row_number

here you can find the functions.

With the table wf_example created in previous example, run:

```
SELECT i
  , DENSE_RANK() OVER (ORDER BY i)
  , ROW_NUMBER() OVER ()
  , RANK() OVER (ORDER BY i)
FROM wf_example
```

The result is:

```
 i | dense_rank | row_number | rank
---+------------+------------+------
1 |          1 |          1 |    1
1 |          1 |          2 |    1
1 |          1 |          3 |    1
2 |          2 |          4 |    4
2 |          2 |          5 |    4
3 |          3 |          6 |    6
4 |          4 |          7 |    7
5 |          5 |          8 |    8
```

- *dense_rank* orders **VALUES** of **i** by appearance in window. `i=1` appears, so first row has dense_rank, next and third i value does not change, so it is `dense_rank` shows *1* - FIRST value not changed. fourth row `i=2`, it is second value of *i* met, so `dense_rank` shows 2, andso for the next row. Then it meets value `i=3` at 6th row, so it show 3. Same for the rest two values of *i*. So the last value of `dense_rank` is the number of distinct values of *i*.
- *row_number* orders **ROWS** as they are listed.
- *rank* Not to confuse with `dense_rank` this function orders **ROW NUMBER** of **i** values. So it starts same with three ones, but has next value 4, which means `i=2` (new value) was met at row 4. Same `i=3` was met at row 6. Etc..

Chapter 14: Recursive queries

There are no real recursive querys!

Section 14.1: Sum of Integers

```
WITH RECURSIVE t(n) AS (
    VALUES (1)
  UNION ALL
    SELECT n+1 FROM t WHERE n < 100
)
SELECT SUM(n) FROM t;
```

Link to Documentation

Chapter 15: Programming with PL/pgSQL

Section 15.1: Basic PL/pgSQL Function

A simple PL/pgSQL function:

```
CREATE FUNCTION active_subscribers() RETURNS BIGINT AS $$
DECLARE
    -- variable for the following BEGIN ... END block
    subscribers INTEGER;
BEGIN
    -- SELECT must always be used with INTO
    SELECT COUNT(user_id) INTO subscribers FROM users WHERE subscribed;
    -- function result
    RETURN subscribers;
EXCEPTION
    -- return NULL if table "users" does not exist
    WHEN undefined_table
    THEN RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

This could have been achieved with just the SQL statement but demonstrates the basic structure of a function.

To execute the function do:

```
SELECT active_subscribers();
```

Section 15.2: custom exceptions

creating custom exception 'P2222':

```
CREATE OR REPLACE FUNCTION s164() RETURNS void AS
$$
BEGIN
raise exception USING message = 'S 164', detail = 'D 164', hint = 'H 164', errcode = 'P2222';
END;
$$ LANGUAGE plpgsql
;
```

creating custom exception not assigning errm:

```
CREATE OR REPLACE FUNCTION s165() RETURNS void AS
$$
BEGIN
raise exception '%','nothing specified';
END;
$$ LANGUAGE plpgsql
;
```

calling:

```
t=# DO
$$
DECLARE
 _t TEXT;
BEGIN
```

```
  perform s165();
  exception WHEN SQLSTATE 'P0001' THEN raise info '%','state P0001 caught: '||SQLERRM;
  perform s164();

END;
$$
;
INFO:  state P0001 caught: NOTHING specified
ERROR:  S 164
DETAIL:  D 164
HINT:  H 164
CONTEXT:  SQL STATEMENT "SELECT s164()"
PL/pgSQL FUNCTION inline_code_block line 7 AT PERFORM
```

here custom P0001 processed, and P2222, not, aborting the execution.

Also it makes huge sense to keep a table of exceptions, like here: http://stackoverflow.com/a/2700312/5315974

Section 15.3: PL/pgSQL Syntax

```
CREATE [OR REPLACE] FUNCTION functionName (someParameter 'parameterType')
RETURNS 'DATATYPE'
AS $_block_name_$
DECLARE
    --declare something
BEGIN
    --do something
    --return something
END;
$_block_name_$
LANGUAGE plpgsql;
```

Section 15.4: RETURNS Block

Options for returning in a PL/pgSQL function:

- `Datatype` List of all datatypes
- `Table(column_name column_type, ...)`
- `SETOF 'Datatype' OR 'table_column'`

Chapter 16: Inheritance

Section 16.1: Creating children tables

```
CREATE TABLE users (username TEXT, email TEXT);
CREATE TABLE simple_users () INHERITS (users);
CREATE TABLE users_with_password (PASSWORD TEXT) INHERITS (users);
```

Our three tables look like this:

users

Column	Type
username	text
email	text

simple_users

Column	Type
username	text
email	text

users_with_password

Column	Type
username	text
email	text
password	text

Chapter 17: Export PostgreSQL database table header and data to CSV file

From Adminer management tool it's has export to csv file option for mysql database But not available for postgresql database. Here I will show the command to export CSV for postgresql database.

Section 17.1: copy from query

```
COPY (SELECT oid,relname FROM pg_class LIMIT 5) TO STDOUT;
```

Section 17.2: Export PostgreSQL table to csv with header for some column(s)

```
COPY products(is_public, title, discount) TO 'D:\csv_backup\products_db.csv' DELIMITER ',' CSV
HEADER;

COPY categories(NAME) TO 'D:\csv_backup\categories_db.csv' DELIMITER ',' CSV HEADER;
```

Section 17.3: Full table backup to csv with header

```
COPY products TO 'D:\csv_backup\products_db.csv' DELIMITER ',' CSV HEADER;

COPY categories TO 'D:\csv_backup\categories_db.csv' DELIMITER ',' CSV HEADER;
```

Chapter 18: Triggers and Trigger Functions

The trigger will be associated with the specified table or view and will execute the specified function function_name when certain events occur.

Section 18.1: Type of triggers

Trigger can be specified to fire:

- **`BEFORE`** the operation is attempted on a row - insert, update or delete;
- **`AFTER`** the operation has completed - insert, update or delete;
- **`INSTEAD OF`** the operation in the case of inserts, updates or deletes on a view.

Trigger that is marked:

- **`FOR EACH ROW`** is called once for every row that the operation modifies;
- **`FOR EACH STATEMENT`** is called onde for any given operation.

Preparing to execute examples

```
CREATE TABLE company (
    id          SERIAL PRIMARY KEY NOT NULL,
    NAME        TEXT NOT NULL,
    created_at  TIMESTAMP,
    modified_at TIMESTAMP DEFAULT NOW()
)

CREATE TABLE log (
    id          SERIAL PRIMARY KEY NOT NULL,
    table_name  TEXT NOT NULL,
    table_id    TEXT NOT NULL,
    description TEXT NOT NULL,
    created_at  TIMESTAMP DEFAULT NOW()
)
```

Single insert trigger

Step 1: create your function

```
CREATE OR REPLACE FUNCTION add_created_at_function()
  RETURNS TRIGGER AS $BODY$
BEGIN
  NEW.created_at := NOW();
  RETURN NEW;
END $BODY$
LANGUAGE plpgsql;
```

Step 2: create your trigger

```
CREATE TRIGGER add_created_at_trigger
BEFORE INSERT
ON company
FOR EACH ROW
EXECUTE PROCEDURE add_created_at_function();
```

Step 3: test it

```
INSERT INTO company (NAME) VALUES ('My company');
SELECT * FROM company;
```

Trigger for multiple purpose

Step 1: create your function

```
CREATE OR REPLACE FUNCTION add_log_function()
  RETURNS TRIGGER AS $BODY$
DECLARE
  vDescription TEXT;
  vId INT;
  vReturn RECORD;
BEGIN
    vDescription := TG_TABLE_NAME || ' ';
     IF (TG_OP = 'INSERT') THEN
        vId := NEW.id;
        vDescription := vDescription || 'added. Id: ' || vId;
        vReturn := NEW;
    ELSIF (TG_OP = 'UPDATE') THEN
        vId := NEW.id;
        vDescription := vDescription || 'updated. Id: ' || vId;
        vReturn := NEW;
    ELSIF (TG_OP = 'DELETE') THEN
        vId := OLD.id;
        vDescription := vDescription || 'deleted. Id: ' || vId;
        vReturn := OLD;
    END IF;

    RAISE NOTICE 'TRIGER called on % - Log: %', TG_TABLE_NAME, vDescription;

    INSERT INTO log
        (table_name, table_id, description, created_at)
        VALUES
        (TG_TABLE_NAME, vId, vDescription, NOW());

    RETURN vReturn;
END $BODY$
  LANGUAGE plpgsql;
```

Step 2: create your trigger

```
CREATE TRIGGER add_log_trigger
AFTER INSERT OR UPDATE OR DELETE
ON company
FOR EACH ROW
EXECUTE PROCEDURE add_log_function();
```

Step 3: test it

```
INSERT INTO company (NAME) VALUES ('Company 1');
INSERT INTO company (NAME) VALUES ('Company 2');
INSERT INTO company (NAME) VALUES ('Company 3');
UPDATE company SET NAME='Company new 2' WHERE NAME='Company 2';
DELETE FROM company WHERE NAME='Company 1';
SELECT * FROM log;
```

Section 18.2: Basic PL/pgSQL Trigger Function

This is a simple trigger function.

```
CREATE OR REPLACE FUNCTION my_simple_trigger_function()
RETURNS TRIGGER AS
$BODY$

BEGIN
    -- TG_TABLE_NAME :name of the table that caused the trigger invocation
```

```
IF (TG_TABLE_NAME = 'users') THEN

    --TG_OP : operation the trigger was fired
  IF (TG_OP = 'INSERT') THEN
    --NEW.id is holding the new database row value (in here id is the id column in users table)
    --NEW will return null for DELETE operations
    INSERT INTO log_table (date_and_time, description) VALUES (NOW(), 'New user inserted. User ID:
'|| NEW.id);
    RETURN NEW;

  ELSIF (TG_OP = 'DELETE') THEN
    --OLD.id is holding the old database row value (in here id is the id column in users table)
    --OLD will return null for INSERT operations
    INSERT INTO log_table (date_and_time, description) VALUES (NOW(), 'User deleted.. User ID: ' ||
OLD.id);
    RETURN OLD;

  END IF;

RETURN NULL;
END IF;

END;
$BODY$
LANGUAGE plpgsql VOLATILE
COST 100;
```

Adding this trigger function to the `users` table

```
CREATE TRIGGER my_trigger
AFTER INSERT OR DELETE
ON users
FOR EACH ROW
EXECUTE PROCEDURE my_simple_trigger_function();
```

Chapter 19: Event Triggers

Event Triggers will be fired whenever event associated with them occurs in database.

Section 19.1: Logging DDL Command Start Events

Event Type-

- `DDL_COMMAND_START`
- `DDL_COMMAND_END`
- SQL_DROP

This is example for creating an Event Trigger and logging `DDL_COMMAND_START` events.

```
CREATE TABLE TAB_EVENT_LOGS(
  DATE_TIME TIMESTAMP,
  EVENT_NAME TEXT,
  REMARKS TEXT
);

CREATE OR REPLACE FUNCTION FN_LOG_EVENT()
  RETURNS EVENT_TRIGGER
  LANGUAGE SQL
  AS
  $main$
    INSERT INTO TAB_EVENT_LOGS(DATE_TIME,EVENT_NAME,REMARKS)
      VALUES(NOW(),TG_TAG,'Event Logging');
  $main$;

CREATE EVENT TRIGGER TRG_LOG_EVENT ON DDL_COMMAND_START
  EXECUTE PROCEDURE FN_LOG_EVENT();
```

Chapter 20: Role Management

Section 20.1: Create a user with a password

Generally you should avoid using the default database role (often `postgres`) in your application. You should instead create a user with lower levels of privileges. Here we make one called `niceusername` and give it a password `very-strong-PASSWORD`

```
CREATE ROLE niceusername WITH PASSWORD 'very-strong-password' LOGIN;
```

The problem with that is that queries typed into the `psql` console get saved in a history file `.psql_history` in the user's home directory and may as well be logged to the PostgreSQL database server log, thus exposing the password.

To avoid this, use the `\PASSWORD` command to set the user password. If the user issuing the command is a superuser, the current password will not be asked. (Must be superuser to alter passwords of superusers)

```
CREATE ROLE niceusername WITH LOGIN;
\PASSWORD niceusername
```

Section 20.2: Grant and Revoke Privileges

Suppose, that we have three users :

1. The Administrator of the database > admin
2. The application with a full access for her data > read_write
3. The read only access > read_only

```
--ACCESS DB
REVOKE CONNECT ON DATABASE nova FROM PUBLIC;
GRANT  CONNECT ON DATABASE nova  TO USER;
```

With the above queries, untrusted users can no longer connect to the database.

```
--ACCESS SCHEMA
REVOKE ALL     ON SCHEMA public FROM PUBLIC;
GRANT  USAGE   ON SCHEMA public  TO USER;
```

The next set of queries revoke all privileges from unauthenticated users and provide limited set of privileges for the `read_write` user.

```
--ACCESS TABLES
REVOKE ALL ON ALL TABLES IN SCHEMA public FROM PUBLIC ;
GRANT SELECT                         ON ALL TABLES IN SCHEMA public TO read_only ;
GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA public TO read_write ;
GRANT ALL                            ON ALL TABLES IN SCHEMA public TO ADMIN ;

--ACCESS SEQUENCES
REVOKE ALL   ON ALL SEQUENCES IN SCHEMA public FROM PUBLIC;
GRANT SELECT ON ALL SEQUENCES IN SCHEMA public TO read_only; -- allows the use of CURRVAL
GRANT UPDATE ON ALL SEQUENCES IN SCHEMA public TO read_write; -- allows the use of NEXTVAL and
SETVAL
GRANT USAGE  ON ALL SEQUENCES IN SCHEMA public TO read_write; -- allows the use of CURRVAL and
NEXTVAL
GRANT ALL    ON ALL SEQUENCES IN SCHEMA public TO ADMIN;
```

Section 20.3: Create Role and matching database

To support a given application, you often create a new role and database to match.

The shell commands to run would be these:

```
$ CREATEUSER -P blogger
Enter PASSWORD FOR the NEW ROLE: ********
Enter it again: ********

$ CREATEDB -O blogger blogger
```

This assumes that `pg_hba.conf` has been properly configured, which probably looks like this:

```
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    sameuser        ALL             localhost               md5
LOCAL   sameuser        ALL                                     md5
```

Section 20.4: Alter default search_path of user

With the below commands, user's default search_path can be set.

1. Check search path before set default schema.

```
postgres=# \c postgres user1
You are now connected TO DATABASE "postgres" AS USER "user1".
postgres=> SHOW search_path;
  search_path
----------------
 "$user",public
(1 ROW)
```

2. Set `search_path` with `ALTER USER` command to append a new schema `my_schema`

```
postgres=> \c postgres postgres
You are now connected TO DATABASE "postgres" AS USER "postgres".
postgres=# ALTER USER user1 SET search_path='my_schema, "$user", public';
ALTER ROLE
```

3. Check result after execution.

```
postgres=# \c postgres user1
PASSWORD FOR USER user1:
You are now connected TO DATABASE "postgres" AS USER "user1".
postgres=> SHOW search_path;
 search_path
-------------
 my_schema, "$user", public
(1 ROW)
```

Alternative:

```
postgres=# SET ROLE user1;
postgres=# SHOW search_path;
 search_path
-------------
 my_schema, "$user", public
(1 ROW)
```

Section 20.5: Create Read Only User

```
CREATE USER readonly WITH ENCRYPTED PASSWORD 'yourpassword';
GRANT CONNECT ON DATABASE <database_name> TO readonly;

GRANT USAGE ON SCHEMA public TO readonly;
GRANT SELECT ON ALL SEQUENCES IN SCHEMA public TO readonly;
GRANT SELECT ON ALL TABLES IN SCHEMA public TO readonly;
```

Section 20.6: Grant access privileges on objects created in the future

Suppose, that we have `three users` :

1. The Administrator of the database > **`ADMIN`**
2. The application with a full access for her data > `read_write`
3. The read only access > `read_only`

With below queries, you can set access privileges on objects created in the future in specified schema.

```
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema GRANT SELECT                    ON TABLES TO
read_only;
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema GRANT SELECT,INSERT,DELETE,UPDATE ON TABLES TO
read_write;
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema GRANT ALL                       ON TABLES TO ADMIN;
```

Or, you can set access privileges on objects created in the future by specified user.

```
ALTER DEFAULT PRIVILEGES FOR ROLE ADMIN GRANT SELECT  ON TABLES TO read_only;
```

Chapter 21: Postgres cryptographic functions

In Postgres, cryptographic functions can be unlocked by using pgcrypto module. CREATE EXTENSION pgcrypto;

Section 21.1: digest

`DIGEST()` functions generate a binary hash of the given data. This **can** be used to create a random hash.

Usage: `digest(DATA TEXT, TYPE TEXT) RETURNS BYTEA`

Or: `digest(DATA BYTEA, TYPE TEXT) RETURNS BYTEA`

Examples:

- `SELECT DIGEST('1', 'sha1')`
- `SELECT DIGEST(CONCAT(CAST(CURRENT_TIMESTAMP AS TEXT), RANDOM()::TEXT), 'sha1')`

Chapter 22: Comments in PostgreSQL

COMMMENT main purpose is to define or change a comment on database object.

Only a single comment(string) can be given on any database object. COMMENT will help us to know what for the particular database object has been defined whats its actual purpose is.

The rule for ***COMMENT ON ROLE*** is that you must be superuser to comment on a superuser role, or have the ***CREATEROLE*** privilege to comment on non-superuser roles. Of course, a *superuser can comment on anything*

Section 22.1: COMMENT on Table

```
COMMENT ON TABLE table_name IS 'this is student details table';
```

Section 22.2: Remove Comment

```
COMMENT ON TABLE student IS NULL;
```

Comment will be removed with above statement execution.

Chapter 23: Backup and Restore

Section 23.1: Backing up one database

```
pg_dump -Fc -f DATABASE.pgsql DATABASE
```

The `-Fc` selects the "custom backup format" which gives you more power than raw SQL; see `pg_restore` for more details. If you want a vanilla SQL file, you can do this instead:

```
pg_dump -f DATABASE.sql DATABASE
```

or even

```
pg_dump DATABASE > DATABASE.sql
```

Section 23.2: Restoring backups

```
psql < backup.sql
```

A safer alternative uses `-1` to wrap the restore in a transaction. The `-f` specifies the filename rather than using shell redirection.

```
psql -1f backup.sql
```

Custom format files must be restored using `pg_restore` with the `-d` option to specify the database:

```
pg_restore -d DATABASE DATABASE.pgsql
```

The custom format can also be converted back to SQL:

```
pg_restore backup.pgsql > backup.sql
```

Usage of the custom format is recommended because you can choose which things to restore and optionally enable parallel processing.

You may need to do a pg_dump followed by a pg_restore if you upgrade from one postgresql release to a newer one.

Section 23.3: Backing up the whole cluster

```
$ pg_dumpall -f backup.sql
```

This works behind the scenes by making multiple connections to the server once for each database and executing `pg_dump` on it.

Sometimes, you might be tempted to set this up as a cron job, so you want to see the date the backup was taken as part of the filename:

```
$ postgres-backup-$(DATE +%Y-%m-%d).sql
```

However, please note that this could produce large files on a daily basis. Postgresql has a much better mechanism for regular backups - WAL archives

The output from pg_dumpall is sufficient to restore to an identically-configured Postgres instance, but the configuration files in `$PGDATA` (`pg_hba.conf` and `postgresql.conf`) are not part of the backup, so you'll have to back them up separately.

```
postgres=# SELECT pg_start_backup('my-backup');
postgres=# SELECT pg_stop_backup();
```

To take a filesystem backup, you must use these functions to help ensure that Postgres is in a consistent state while the backup is prepared.

Section 23.4: Using psql to export data

Data can be exported using copy command or by taking use of command line options of psql command.

To Export csv data from table user to csv file:

```
psql -p \<port> -U \<username> -d \<DATABASE> -A -F<DELIMITER> -c\<sql TO EXECUTE> \> \<output
filename WITH path>

psql -p 5432 -U postgres -d test_database -A -F, -c "select * from user" > /home/USER/user_data.CSV
```

Here combination of -A and -F does the trick.

`-F` is to specify delimiter

```
-A OR --no-align
```

Switches to unaligned output mode. (The default output mode is otherwise aligned.)

Section 23.5: Using Copy to import

To Copy Data from a CSV file to a table

```
COPY <tablename> FROM '<filename with path>';
```

To insert into table `USER` from a file named `user_data.CSV` placed inside `/home/USER/`:

```
COPY USER FROM '/home/user/user_data.csv';
```

To Copy data from pipe separated file to table

```
COPY USER FROM '/home/user/user_data' WITH DELIMITER '|';
```

Note: In absence of the option **`WITH DELIMITER`**, the default delimiter is comma `,`

To ignore header line while importing file

Use the Header option:

```
COPY USER FROM '/home/user/user_data' WITH DELIMITER '|' HEADER;
```

Note: If data is quoted, by default data quoting characters are double quote. If the data is quoted using any other character use the **`QUOTE`** option; however, this option is allowed only when using CSV format.

Section 23.6: Using Copy to export

To Copy table to standard o/p

```
COPY <tablename> TO STDOUT (DELIMITER '|');
```

To export table user to Standard output:

```
COPY USER TO STDOUT (DELIMITER '|');
```
To Copy table to file

```
COPY USER FROM '/home/user/user_data' WITH DELIMITER '|';
```
To Copy the output of SQL statement to file

```
COPY (sql STATEMENT) TO '<filename with path>';
COPY (SELECT * FROM USER WHERE user_name LIKE 'A%')
TO '/home/user/user_data';
```
To Copy into a compressed file

```
COPY USER TO PROGRAM 'gzip > /home/user/user_data.gz';
```

Here program gzip is executed to compress user table data.

Chapter 24: Backup script for a production DB

parameter	details
save_db	The main backup directory
dbProd	The secondary backup directory
DATE	The date of the backup in the specified format
dbprod	The name of the database to be saved
/opt/postgres/9.0/bin/pg_dump	The path to the pg_dump binary
-h	Specifies the host name of the machine on which the server is running, Example : localhost
-p	Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections, Example 5432
-U	User name to connect as.

Section 24.1: saveProdDb.sh

In general, we tend to back up the DB with the pgAdmin client. The following is a sh script used to save the database (under linux) in two formats:

- **SQL file**: for a possible resume of data on any version of PostgreSQL.
- **Dump file**: for a higher version than the current version.

```
#!/bin/sh
cd /save_db
#rm -R /save_db/*
DATE=$(date +%d-%m-%Y-%Hh%M)
echo -e "Sauvegarde de la base du ${DATE}"
mkdir prodDir${DATE}
cd prodDir${DATE}

#dump file
/opt/postgres/9.0/bin/pg_dump -i -h localhost -p 5432 -U postgres -F c -b -w -v -f
"dbprod${DATE}.backup" dbprod

#SQL file
/opt/postgres/9.0/bin/pg_dump -i -h localhost -p 5432 -U postgres --format plain --verbose  -f
"dbprod${DATE}.sql" dbprod
```

Chapter 25: Accessing Data Programmatically

Section 25.1: Accessing PostgreSQL with the C-API

The C-API is the most powerful way to access PostgreSQL and it is surprisingly comfortable.

Compilation and linking

During compilation, you have to add the PostgreSQL include directory, which can be found with `pg_config --includedir`, to the include path.
You must link with the PostgreSQL client shared library (`libpq.so` on UNIX, `libpq.dll` on Windows). This library is in the PostgreSQL library directory, which can be found with `pg_config --libdir`.

Note: For historical reason, the library is called `libpq.so`and *not* `libpg.so`, which is a popular trap for beginners.

Given that the below code sample is in file `coltype.c`, compilation and linking would be done with

```
gcc -Wall -I "$(pg_config --includedir)" -L "$(pg_config --libdir)" -o coltype coltype.c -lpq
```

with the GNU C compiler (consider adding `-Wl,-rpath,"$(pg_config --libdir)"` to add the library search path) or with

```
cl /MT /W4 /I <include directory> coltype.c <path TO libpq.lib>
```

on Windows with Microsoft Visual C.

Sample program

```
/* necessary for all PostgreSQL client programs, should be first */
#include <libpq-fe.h>

#include <stdio.h>
#include <string.h>

#ifdef TRACE
#define TRACEFILE "trace.out"
#endif

int main(int argc, char **argv) {
#ifdef TRACE
    FILE *trc;
#endif
    PGconn *conn;
    PGresult *res;
    int rowcount, colcount, i, j, firstcol;
    /* parameter type should be guessed by PostgreSQL */
    const Oid paramTypes[1] = { 0 };
    /* parameter value */
    const char * const paramValues[1] = { "pg_database" };

    /*
     * Using an empty connectstring will use default values for everything.
     * If set, the environment variables PGHOST, PGDATABASE, PGPORT and
     * PGUSER will be used.
     */
    conn = PQconnectdb("");
```

```
    /*
     * This can only happen if there is not enough memory
     * to allocate the PGconn structure.
     */
    if (conn == NULL)
    {
        fprintf(stderr, "Out of memory connecting to PostgreSQL.\n");
        return 1;
    }

    /* check if the connection attempt worked */
    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "%s\n", PQerrorMessage(conn));
        /*
         * Even if the connection failed, the PGconn structure has been
         * allocated and must be freed.
         */
        PQfinish(conn);
        return 1;
    }

#ifdef TRACE
    if (NULL == (trc = fopen(TRACEFILE, "w")))
    {
        fprintf(stderr, "Error opening trace file \"%s\"!\n", TRACEFILE);
        PQfinish(conn);
        return 1;
    }

    /* tracing for client-server communication */
    PQtrace(conn, trc);
#endif

    /* this program expects the database to return data in UTF-8 */
    PQsetClientEncoding(conn, "UTF8");

    /* perform a query with parameters */
    res = PQexecParams(
        conn,
        "SELECT column_name, data_type "
            "FROM information_schema.columns "
            "WHERE table_name = $1",
        1,            /* one parameter */
        paramTypes,
        paramValues,
        NULL,        /* parameter lengths are not required for strings */
        NULL,        /* all parameters are in text format */
        0            /* result shall be in text format */
    );

    /* out of memory or sever communication broken */
    if (NULL == res)
    {
        fprintf(stderr, "%s\n", PQerrorMessage(conn));
        PQfinish(conn);
#ifdef TRACE
        fclose(trc);
#endif
        return 1;
    }
```

```
    /* SQL statement should return results */
    if (PGRES_TUPLES_OK != PQresultStatus(res))
    {
        fprintf(stderr, "%s\n", PQerrorMessage(conn));
        PQfinish(conn);
#ifdef TRACE
        fclose(trc);
#endif
        return 1;
    }

    /* get count of result rows and columns */
    rowcount = PQntuples(res);
    colcount = PQnfields(res);

    /* print column headings */
    firstcol = 1;

    printf("Description of the table \"pg_database\"\n");

    for (j=0; j<colcount; ++j)
    {
        if (firstcol)
            firstcol = 0;
        else
            printf(": ");

        printf(PQfname(res, j));
    }

    printf("\n\n");

    /* loop through rosult rows */
    for (i=0; i<rowcount; ++i)
    {
        /* print all column data */
        firstcol = 1;

        for (j=0; j<colcount; ++j)
        {
            if (firstcol)
                firstcol = 0;
            else
                printf(": ");

            printf(PQgetvalue(res, i, j));
        }

        printf("\n");
    }

    /* this must be done after every statement to avoid memory leaks */
    PQclear(res);
    /* close the database connection and release memory */
    PQfinish(conn);
#ifdef TRACE
    fclose(trc);
#endif
    return 0;
}
```

Section 25.2: Accessing PostgreSQL from python using psycopg2

You can find description of the driver here.

The quick example is:

```
import psycopg2

db_host = 'postgres.server.com'
db_port = '5432'
db_un = 'user'
db_pw = 'password'
db_name = 'testdb'

conn = psycopg2.connect("dbname={} host={} user={} password={}".format(
                        db_name,  db_host, db_un, db_pw),
                        cursor_factory=RealDictCursor)
cur = conn.cursor()
sql = 'select * from testtable where id > %s and id < %s'
args = (1, 4)
cur.execute(sql, args)

print(cur.fetchall())
```

Will result:

```
[{'id': 2, 'fruit': 'apple'}, {'id': 3, 'fruit': 'orange'}]
```

Section 25.3: Accessing PostgreSQL from .NET using the Npgsql provider

One of the more popular .NET providers for Postgresql is Npgsql, which is ADO.NET compatible and is used nearly identically as other .NET database providers.

A typical query is performed by creating a command, binding parameters, and then executing the command. In C#:

```
var connString = "Host=myserv;Username=myuser;Password=mypass;Database=mydb";
using (var conn = new NpgsqlConnection(connString))
{
    var querystring = "INSERT INTO data (some_field) VALUES (@content)";

    conn.Open();
    // Create a new command with CommandText and Connection constructor
    using (var cmd = new NpgsqlCommand(querystring, conn))
    {
        // Add a parameter and set its type with the NpgsqlDbType enum
        var contentString = "Hello World!";
        cmd.Parameters.Add("@content", NpgsqlDbType.Text).Value = contentString;

        // Execute a query that returns no results
        cmd.ExecuteNonQuery();


        /* It is possible to reuse a command object and open connection instead of creating new ones
*/

        // Create a new query and set its parameters
```

```
        int keyId = 101;
        cmd.CommandText = "SELECT primary_key, some_field FROM data WHERE primary_key = @keyId";
        cmd.Parameters.Clear();
        cmd.Parameters.Add("@keyId", NpgsqlDbType.Integer).Value = keyId;

        // Execute the command and read through the rows one by one
        using (NpgsqlDataReader reader = cmd.ExecuteReader())
        {
            while (reader.Read())   // Returns false for 0 rows, or after reading the last row of
the results
            {
                // read an integer value
                int primaryKey = reader.GetInt32(0);
                // or
                primaryKey = Convert.ToInt32(reader["primary_key"]);

                // read a text value
                string someFieldText = reader["some_field"].ToString();
            }
        }
    }
}    // the C# 'using' directive calls conn.Close() and conn.Dispose() for us
```

Section 25.4: Accessing PostgreSQL from PHP using Pomm2

On the shoulders of the low level drivers, there is pomm. It proposes a modular approach, data converters, listen/notify support, database inspector and much more.

Assuming, Pomm has been installed using composer, here is a complete example:

```
<?php
use PommProject\Foundation\Pomm;
$loader = require __DIR__ . '/vendor/autoload.php';
$pomm = new Pomm(['my_db' => ['dsn' => 'pgsql://user:pass@host:5432/db_name']]);

// TABLE comment (
// comment_id uuid PK, created_at timestamptz NN,
// is_moderated bool NN default false,
// content text NN CHECK (content !~ '^\s+$'), author_email text NN)
$sql = <<<SQL
SELECT
  comment_id,
  created_at,
  is_moderated,
  content,
  author_email
FROM comment
  INNER JOIN author USING (author_email)
WHERE
  age(now(), created_at) < $*::interval
ORDER BY created_at ASC
SQL;

// the argument will be converted as it is cast in the query above
$comments = $pomm['my_db']
    ->getQueryManager()
    ->query($sql, [DateInterval::createFromDateString('1 day')]);

if ($comments->isEmpty()) {
    printf("There are no new comments since yesterday.");
} else {
```

```
    foreach ($comments as $comment) {
        printf(
                "%s has posted at %s. %s\n",
                $comment['author_email'],
                $comment['created_at']->format("Y-m-d H:i:s"),
                $comment['is_moderated'] ? '[OK]' : '');
    }
}
```

Pomm's query manager module escapes query arguments to prevent SQL injection. When the arguments are cast, it also converts them from a PHP representation to valid Postgres values. The result is an iterator, it uses a cursor internally. Every row is converted on the fly, booleans to booleans, timestamps to \DateTime etc.

Chapter 26: Connect to PostgreSQL from Java

The API to use a relational database from Java is JDBC.

This API is implemented by a JDBC driver.

To use it, put the JAR-file with the driver on the JAVA class path.

This documentation shows samples how to use the JDBC driver to connect to a database.

Section 26.1: Connecting with java.sql.DriverManager

This is the simplest way to connect.

First, the driver has to be *registered* with `java.sql.DriverManager` so that it knows which class to use.
This is done by loading the driver class, typically with `java.lang.Class.forname(`;driver class name>).

```
/**
 * Connect to a PostgreSQL database.
 * @param url the JDBC URL to connect to; must start with "jdbc:postgresql:"
 * @param user the username for the connection
 * @param password the password for the connection
 * @return a connection object for the established connection
 * @throws ClassNotFoundException if the driver class cannot be found on the Java class path
 * @throws java.sql.SQLException if the connection to the database fails
 */
private static java.sql.Connection connect(String url, String user, String password)
    throws ClassNotFoundException, java.sql.SQLException
{
    /*
     * Register the PostgreSQL JDBC driver.
     * This may throw a ClassNotFoundException.
     */
    Class.forName("org.postgresql.Driver");
    /*
     * Tell the driver manager to connect to the database specified with the URL.
     * This may throw an SQLException.
     */
    return java.sql.DriverManager.getConnection(url, user, password);
}
```

Not that user and password can also be included in the JDBC URL, in which case you don't have to specify them in the `getConnection` method call.

Section 26.2: Connecting with java.sql.DriverManager and Properties

Instead of specifying connection parameters like user and password (see a complete list here) in the URL or a separate parameters, you can pack them into a `java.util.Properties` object:

```
/**
 * Connect to a PostgreSQL database.
 * @param url the JDBC URL to connect to. Must start with "jdbc:postgresql:"
 * @param user the username for the connection
 * @param password the password for the connection
```

```
 * @return a connection object for the established connection
 * @throws ClassNotFoundException if the driver class cannot be found on the Java class path
 * @throws java.sql.SQLException if the connection to the database fails
 */
private static java.sql.Connection connect(String url, String user, String password)
    throws ClassNotFoundException, java.sql.SQLException
{
    /*
     * Register the PostgreSQL JDBC driver.
     * This may throw a ClassNotFoundException.
     */
    Class.forName("org.postgresql.Driver");
    java.util.Properties props = new java.util.Properties();
    props.setProperty("user", user);
    props.setProperty("password", password);
    /* don't use server prepared statements */
    props.setProperty("prepareThreshold", "0");
    /*
     * Tell the driver manager to connect to the database specified with the URL.
     * This may throw an SQLException.
     */
    return java.sql.DriverManager.getConnection(url, props);
}
```

Section 26.3: Connecting with javax.sql.DataSource using a connection pool

It is common to use `javax.sql.DataSource` with JNDI in application server containers, where you register a data source under a name and look it up whenever you need a connection.

This is code that demonstrates how data sources work:

```
/**
 * Create a data source with connection pool for PostgreSQL connections
 * @param url the JDBC URL to connect to. Must start with "jdbc:postgresql:"
 * @param user the username for the connection
 * @param password the password for the connection
 * @return a data source with the correct properties set
 */
private static javax.sql.DataSource createDataSource(String url, String user, String password)
{
    /* use a data source with connection pooling */
    org.postgresql.ds.PGPoolingDataSource ds = new org.postgresql.ds.PGPoolingDataSource();
    ds.setUrl(url);
    ds.setUser(user);
    ds.setPassword(password);
    /* the connection pool will have 10 to 20 connections */
    ds.setInitialConnections(10);
    ds.setMaxConnections(20);
    /* use SSL connections without checking server certificate */
    ds.setSslMode("require");
    ds.setSslfactory("org.postgresql.ssl.NonValidatingFactory");

    return ds;
}
```

Once you have created a data source by calling this function, you would use it like this:

```
/* get a connection from the connection pool */
java.sql.CONNECTION conn = ds.getConnection();
```

```
/* do some work */

/* hand the connection back to the pool - it will not be closed */
conn.close();
```

Chapter 27: PostgreSQL High Availability

Section 27.1: Replication in PostgreSQL

- **Configuring the Primary Server**
  - **Requirements:**
    - Replication User for replication activities
    - Directory to store the WAL archives
  - **Create Replication user**

```
CREATEUSER -U postgres replication -P -c 5 --replication
```

```
+ OPTION -P will prompt you FOR NEW PASSWORD
+ OPTION -c IS FOR maximum connections. 5 connections are enough FOR replication
+ -replication will GRANT replication PRIVILEGES TO the USER
```

  - **Create a archive directory in data directory**

```
mkdir $PGDATA/archive
```

  - **Edit the pg_hba.conf file**

    This is host base authentication file, contains the setting for client autherntication. Add below entry:

```
     #hosttype    database_name     user_name       hostname/IP      method
      host        replication       replication     <slave-IP>/32    md5
```

  - **Edit the postgresql.conf file**

    This is the configuration file of PostgreSQL.

    `wal_level = hot_standby`

    This parameter decides the behavior of slave server.

```
   `hot_standby` logs what IS required TO accept READ ONLY queries ON slave SERVER.

   `streaming` logs what IS required TO just apply the WAL's on slave.

   `archive` which logs what is required for archiving.
```

    `archive_mode=ON`

    This parameters allows to send WAL segments to archive location using `archive_command` parameter.

    `archive_command = 'test ! -f /path/to/archivedir/%f && cp %p /path/to/archivedir/%f'`

    Basically what above `archive_command` does is it copies the WAL segments to archive directory.

    `wal_senders = 5` This is maximum number of WAL sender processes.

    Now restart the primary server.

- **Backing up the primay server to the slave server**

  Before making changes on the server stop the primary server.

> Important: Don't start the service again until all configuration and backup steps are complete. You must bring up the standby server in a state where it is ready to be a backup server. This means that all configuration settings must be in place and the databases must be already synchronized. Otherwise, streaming replication will fail to start`

- **Now run the pg_basebackup utility**

  `pg_basebackup` utility copies the data from primary server data directory to slave data directory.

  ```
  $ pg_basebackup -h <PRIMARY IP> -D /var/lib/postgresql/<VERSION>/main -U replication -v -P --xlog-method=stream
  ```

```
    -D: This IS tells pg_basebackup WHERE TO the initial backup

    -h: Specifies the SYSTEM WHERE TO look FOR the PRIMARY SERVER

    -xlog-method=stream: This will FORCE the pg_basebackup TO open another CONNECTION AND stream
enough xlog WHILE backup IS running.
                         It ALSO ensures that fresh backup can be started WITHOUT failing back TO
USING an archive.
```

- **Configuring the standby server**

  To configure the standby server, you'll edit postgresql.conf and create a new configuration file named recovery.conf.

  ```
  hot_standby = ON
  ```

  This specifies whether you are allowed to run queries while recovering

  - **Creating recovery.conf file**

    ```
    standby_mode = ON
    ```

    Set the connection string to the primary server. Replace with the external IP address of the primary server. Replace with the password for the user named replication

    `primary_conninfo = 'host= port=5432 user=replication password='

    (Optional) Set the trigger file location:

    ```
    trigger_file = '/tmp/postgresql.trigger.5432'
    ```

    The `trigger_file` path that you specify is the location where you can add a file when you want the system to fail over to the standby server. The presence of the file "triggers" the failover. Alternatively, you can use the pg_ctl promote command to trigger failover.

- **Start the standby server**

  You now have everything in place and are ready to bring up the standby server

Attribution

This article is substantially derived from and attributed to How to Set Up PostgreSQL for High Availability and Replication with Hot Standby, with minor changes in formatting and examples and some text deleted. The source was published under the Creative Commons Public License 3.0, which is maintained here.

Chapter 28: EXTENSION dblink and postgres_fdw

Section 28.1: Extention FDW

FDW is an implimentation of dblink it is more helpful, so to use it:

1. Create an extention:

```
CREATE EXTENSION postgres_fdw;
```

2. Create SERVER:

```
CREATE SERVER name_srv FOREIGN DATA WRAPPER postgres_fdw OPTIONS (host 'hostname',
dbname 'bd_name', port '5432');
```

3. Create user mapping for postgres server

```
CREATE USER MAPPING FOR postgres SERVER name_srv OPTIONS(USER 'postgres', PASSWORD 'password');
```

4. Create foreign table:

```
CREATE FOREIGN TABLE table_foreign (id INTEGER, code CHARACTER VARYING)
SERVER name_srv OPTIONS(schema_name 'schema', table_name 'table');
```

5. use this foreign table like it is in your database:

```
SELECT * FROM table_foreign;
```

Section 28.2: Foreign Data Wrapper

To access complete schema of server db instead of single table. Follow below steps:

1. Create EXTENSION :

```
CREATE EXTENSION postgres_fdw;
```

2. Create SERVER :

```
CREATE SERVER server_name FOREIGN DATA WRAPPER postgres_fdw OPTIONS (host 'host_ip',
dbname 'db_name', port 'port_number');
```

3. Create USER MAPPING:

```
CREATE USER MAPPING FOR CURRENT_USER
SERVER server_name
OPTIONS (USER 'user_name', PASSWORD 'password');
```

4. Create new schema to access schema of server DB:

```
CREATE SCHEMA schema_name;
```

5. Import server schema:

```
IMPORT FOREIGN SCHEMA schema_name_to_import_from_remote_db
FROM SERVER server_name
INTO schema_name;
```

6. Access any table of server schema:

```
SELECT * FROM schema_name.table_name;
```

This can be used to access multiple schema of remote DB.

Section 28.3: Extention dblink

dblink EXTENSION is a technique to connect another database and make operation of this database so to do that you need:

1-Create a dblink extention:

```
CREATE EXTENSION dblink;
```

2-Make your operation:

For exemple Select some attribute from another table in another database:

```
SELECT * FROM
dblink ('dbname = bd_distance port = 5432 host = 10.6.6.6 user = username
password = passw@rd', 'SELECT id, code FROM schema.table')
AS newTable(id INTEGER, code CHARACTER VARYING);
```

Chapter 29: Postgres Tip and Tricks

Section 29.1: DATEADD alternative in Postgres

- `SELECT CURRENT_DATE + '1 day'::INTERVAL`
- `SELECT '1999-12-11'::TIMESTAMP + '19 days'::INTERVAL`
- `SELECT '1 month'::INTERVAL + '1 month 3 days'::INTERVAL`

Section 29.2: Comma separated values of a column

```
SELECT
    STRING_AGG(<TABLE_NAME>.<COLUMN_NAME>, ',')
FROM
    <SCHEMA_NAME>.<TABLE_NAME> T
```

Section 29.3: Delete duplicate records from postgres table

```
DELETE
    FROM <SCHEMA_NAME>.<Table_NAME>
WHERE
    ctid NOT IN
        (
        SELECT
            MAX(ctid)
        FROM
            <SCHEMA_NAME>.<TABLE_NAME>
        GROUP BY
            <SCHEMA_NAME>.<TABLE_NAME>.*
        )
;
```

Section 29.4: Update query with join between two tables alternative since Postresql does not support join in update query

```
    UPDATE <SCHEMA_NAME>.<TABLE_NAME_1> AS A
    SET <COLUMN_1> = TRUE
    FROM <SCHEMA_NAME>.<TABLE_NAME_2> AS B
    WHERE
        A.<COLUMN_2> = B.<COLUMN_2> AND
        A.<COLUMN_3> = B.<COLUMN_3>
```

Section 29.5: Difference between two date timestamps month wise and year wise

Monthwise difference between two dates(timestamp)

```
SELECT
    (
        (DATE_PART('year', AgeonDate) - DATE_PART('year', tmpdate)) * 12
        +
        (DATE_PART('month', AgeonDate) - DATE_PART('month', tmpdate))
    )
FROM dbo."Table1"
```

Yearwise difference between two dates(timestamp)

```
SELECT (DATE_PART('year', AgeonDate) - DATE_PART('year', tmpdate)) FROM dbo."Table1"
```

Section 29.6: Query to Copy/Move/Transafer table data from one database to other database table with same schema

First Execute

```
CREATE EXTENSION DBLINK;
```

Then

```
INSERT INTO
    <SCHEMA_NAME>.<TABLE_NAME_1>
SELECT *
FROM
    DBLINK(
    'HOST=<IP-ADDRESS> USER=<USERNAME> PASSWORD=<PASSWORD> DBNAME=<DATABASE>',
    'SELECT * FROM <SCHEMA_NAME>.<TABLE_NAME_2>')
    AS <TABLE_NAME>
    (
    <COLUMN_1> <DATATYPE_1>,
    <COLUMN_1> <DATATYPE_2>,
    <COLUMN_1> <DATATYPE_3>
    );
```