Guide to The Advanced Open-Source DBMS

Research And Development

Software Laboratory Center Binus University

Table of Content

Glimpse About PostgreSQL

"The most advanced, SQL-compliant and open-source objective-RDBMS"

-O.S Tezer-

Chapter I

Basic and Database Administration in PostgreSQL

PostgreSQL is an open source RDBMS developed by a worldwide team of volunteers. PostgreSQL is not controlled by any corporation or other private entity and the source code is available free of charge. It was originally under the BSD license, but is now called the PostgreSQL License (TPL). It has enterprise class features such as SQL windowing functions, the ability to create your own aggregate functions, common table and recursive common table expressions, and others. These features are rarely found in other open source database platforms, but commonly found in proprietary DBMS such as Oracle, SQL Server, and IBM DB2.

What Makes PostgreSQL So Unique?

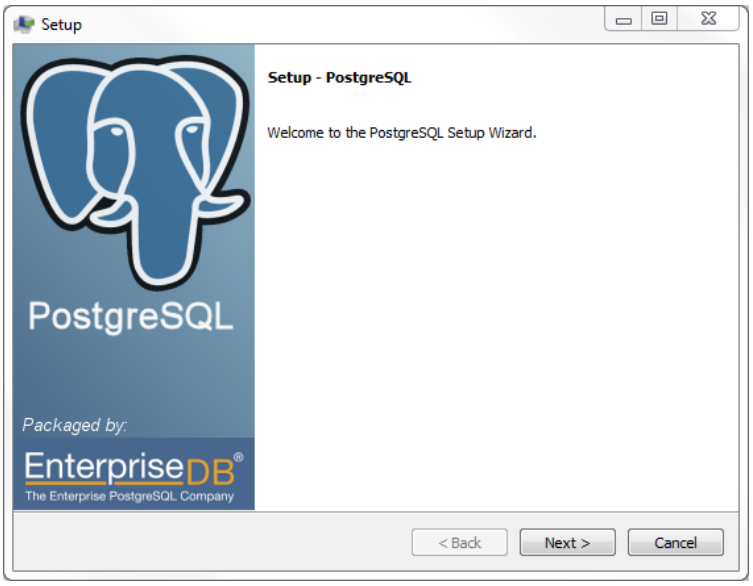
PostgreSQL allows you to write stored procedures and functions in several programming language, such as: SQL (built-in), PL/pgSQL (built-in), PL/Perl, PL/Python, PL/Java, and PL/R. The custom type support of PostgreSQL is sophisticated and very easy to use. You can define new data types in PostgreSQL that can then be used as a table column. Every data type has a companion array type so that you can store an array of a type in a data column or use it in an SQL statement.

In addition to the ability of defining new types, you can also define operators, functions, and index bindings to work with these. If building your own types and functions is not your thing, you have a wide variety of extensions to choose from, many of which are packaged with PostgreSQL distros. PostgreSQL 9.1 introduced a new SQL construct, CREATE EXTENSION, which allows you to install the many available extensions with a single SQL statement for each in a specific database. Here's the example:

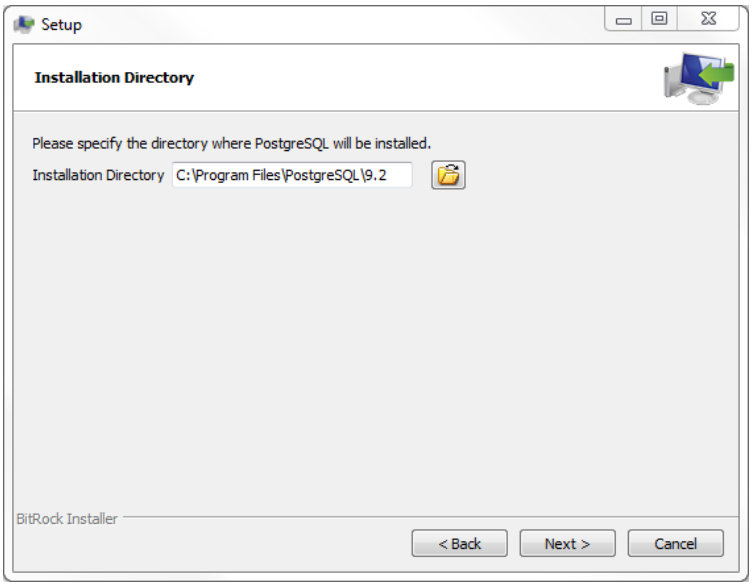
```
CREATE EXTENSION hstore;
```

Installation

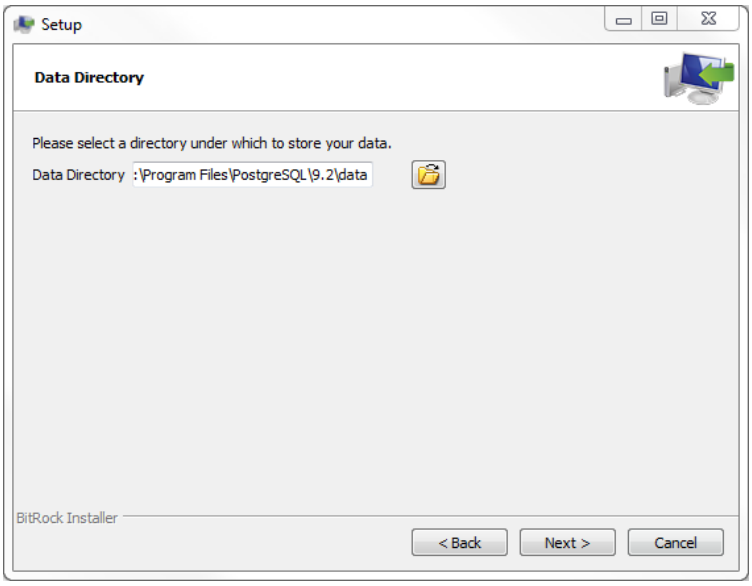
You need to download the installer from PostgreSQL Official website (**http://www.postgresql.org/download/windows/).** The following illustrates each step and its options for installation. If you install a different version, you may get additional steps.



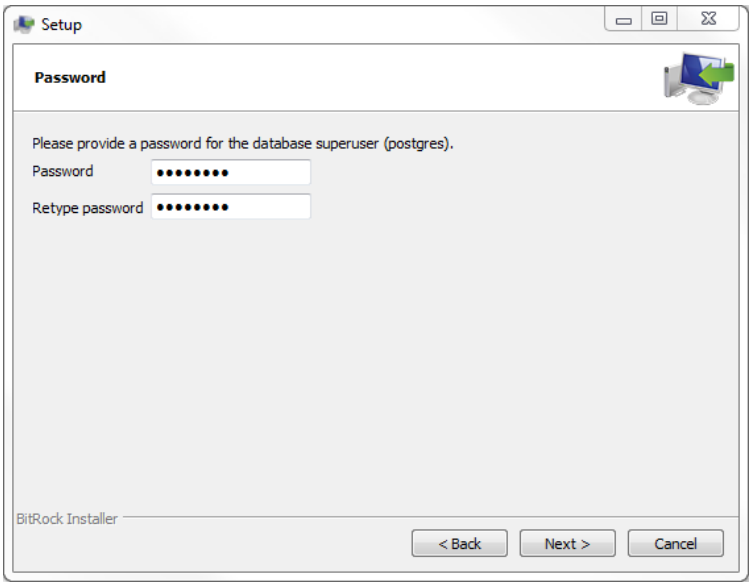
Start Installing PostgreSQL



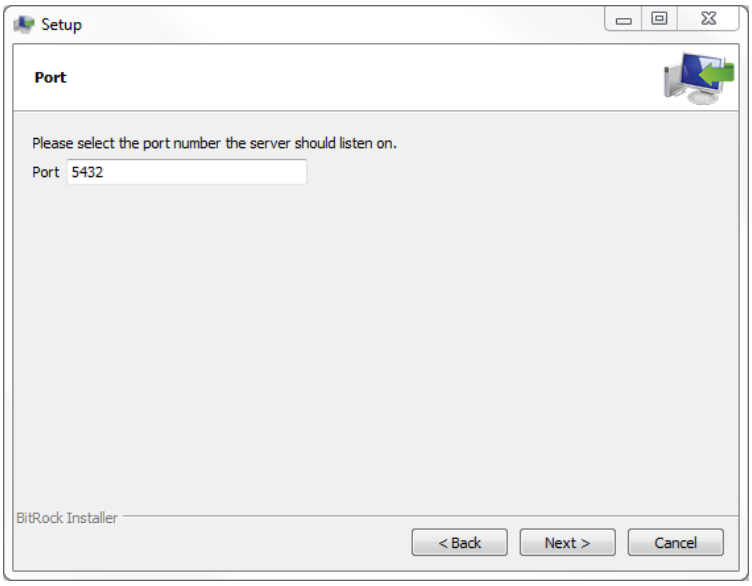
Specify installation directory, choose your own or keep the default



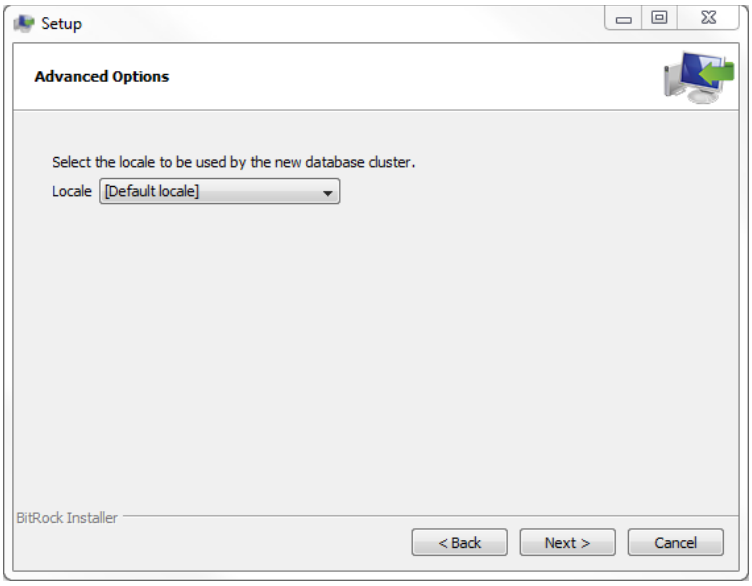
Specify the directory where you want to store the data



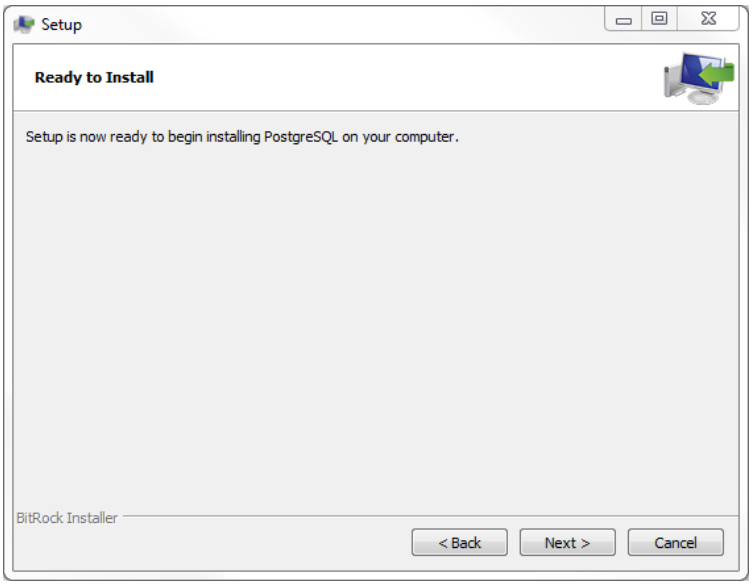
Enter the password for the database superuser and service account.



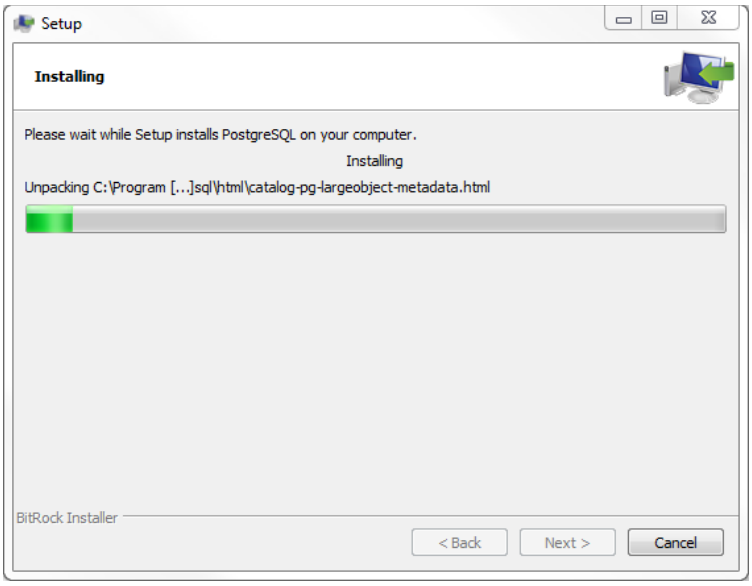
Enter the port for PostgreSQL. Make sure that no other applications are using this port. Leave it as default if you are unsure.



Choose the default locale



You've finished providing information for the PostgreSQL installer. Click the Next button to let PostgreSQL installer to install PostgreSQL



The installation may take few minutes to complete

Click Finish button to complete the installation

There are several ways to verify the installation. You can try to connect to the PostgreSQL database server from any client application e.g., psql, pgAdmin, etc.

Main Configuration Files

There are three main configuration files control basic operations of a PostgreSQL server instance. These files are all located in the default PostgreSQL data folder. You can edit them using your text editor of choice, or using the admin pack that comes with pgAdmin:

• **postgresql.conf** controls general settings, such as how much memory to allocate, default storage location for new databases, which IPs PostgreSQL listens on, where logs are stored, and so forth.

• **pg_hba.conf** controls security. It manages access to the server, dictating which users can login into which databases, which IPs or groups of IPs are permitted to connect and the authentication scheme expected.

• **pg_ident.conf** is the mapping file that maps an authenticated OS login to a PostgreSQL user. This file is used less often, but allows you to map a server account to a PostgreSQL account. For example, people sometimes map the OS root account to the postgre's super user account. Each authentication line in pg_hba.conf can use a different pg_ident.conf file.

You can use this query to find the file location:

```
SELECT name, setting
FROM pg_settings
WHERE category = 'File Locations';
```

	name text	setting text
1	config file	C:/Program Files/PostgreSQL/9.4/data/postgresql.conf
2	data directory	C:/Program Files/PostgreSQL/9.4/data
3	external pid file	
4	hba file	C:/Program Files/PostgreSQL/9.4/data/pg hba.conf
5	ident file	C:/Program Files/PostgreSQL/9.4/data/pg ident.conf

- **The postgresql.conf file**

Postgresql.conf controls the core settings of the PostgreSQL server instance as well as default settings for new databases. Here is the example query to check your current settings:

```
SELECT name, context, setting , boot_val , reset_val
FROM pg_settings
WHERE name
in('listen_addresses','max_connections','shared_buffers','work_mem')
```

	name text	context text	setting text	boot_val text	reset_val text
1	listen addresses	postmaster	*	localhost	*
2	max connections	postmaster	100	100	100
3	shared buffers	postmaster	16384	1024	16384
4	work mem	user	4096	4096	4096

Here is the explanation about the query:

1. Context is the required action if you change its value. If it is set to postmaster, it means changing this parameter requires a restart of the postgresql service. Otherwise, changes require at least a reload.
2. Setting is the currently running setting in effect.
3. boot_val is the default setting.
4. reset_val is the new value if you were to restart or reload.

5. listen_addresses is the IP that PostgreSQL use to listen on. This usually defaults to localhost, but it can also be * (all available IPs).
6. max_connections is the maximum number of concurrent connections allowed.
7. work_mem controls the maximum amount of memory allocated for each operation such as sorting, hash join, and others.

- **The pg_hba.conf File**

The pg_hba.conf controls which and how users can connect to PostgreSQL databases. Changes to the pg_hba.conf require a reload or a server restart to take effect. Here is my pg_hba.conf looks like:

```
# TYPE  DATABASE        USER            ADDRESS                 METHOD
# IPv4 local connections:
host    all             all             127.0.0.1/32            md5
# IPv6 local connections:
host    all             all             ::1/128                 md5
# Allow replication connections from localhost, by a user with the
# replication privilege.
#host    replication     postgres        127.0.0.1/32            md5
#host    replication     postgres        ::1/128                 md5
```

::1/128 means IPv6 syntax for defining localhost. This only applies to servers with IPv6 support and may cause the configuration file to not load if you have it and don't have IPv6.

Extensions

Extensions is the add-ons that you can install in a PostgreSQL database to extend its base functionality. It provides the best feature of open source software: people collaborating, building, and freely sharing new features. Extensions in PostgreSQL are installed separately in each database. If you want to see which extensions you have already installed, here is the example query:

```
SELECT * FROM pg_available_extensions WHERE installed_version IS NOT NULL ORDER BY name;
```

	name name	default_version text	installed_version text	comment text
1	btree gist	1.0	1.0	support for indexing common datatypes in GiST
2	plpgsql	1.0	1.0	PL/pgSQL procedural language

If you have downloaded and installed the new extension, you can use this syntax:

```
CREATE EXTENSION [Extension Name];
```

Tablespaces

PostgreSQL uses tablespaces to describe a logical names for physical locations on disk. PostgreSQL have two default tablespaces: pg_default, which stores for all user data and pg_global, which stores all system data. These are located in the same folder as your default data cluster. You are also free to create tablespaces and set them on any server disks. To create a tablespace, you just need to denote a logical name and a physical folder with using this query:

```
CREATE TABLESPACE [Tablespace Name] LOCATION '[Disk Location]';
```

You can shuffle database objects among different tablespaces. To move all objects in the database to our secondary tablespace:

```
ALTER DATABASE mydb SET TABLESPACE secondary;
```

To move just a table:

```
ALTER TABLE mytable SET TABLESPACE secondary;
```

Chapter II

PostgreSQL Table and Data Types

Series Function

PostgreSQL has an useful function called generate_series() that we have yet to find in other leading databases. It's actually used to creating sequential rows automatically like the FOR .. LOOP behavior in SQL. Here is the example:

```
SELECT x FROM generate_series(0,50,10) As x;
```

	x integer
1	0
2	10
3	20
4	30
5	40
6	50

Arrays

Arrays play an important role in PostgreSQL because they are particularly useful in building aggregate functions, forming IN and ANY clauses, etc. The most common way to create an array is to simply type the elements:

```
SELECT ARRAY[1,2,3] As number;
```

	number integer[]
1	{1,2,3}

You also can use the constructor function: array() to fill the elements in array from the extracted query.

```
SELECT array(SELECT DISTINCT date_part('year', log_ts)
FROM logs ORDER BY date_part('year', log_ts));
```

You can convert delimited strings to an array with the string_to_array() function as follows:

```
SELECT string_to_array('abc.123.!@#', '.') As x;
```

	x text[]
1	{abc,123,!@#}

If you want to take a set of any data type and convert it to an array, PostgreSQL provides the array_agg() function.

```
SELECT array_agg(log_ts ORDER BY log_ts) As x
FROM logs WHERE log_ts BETWEEN '2011-01-01'::timestamptz AND '2011-01-15'::timestamptz;
```

	x timestamp with time zone[]
1	{"2011-01-01 00:00:00+07","2011-01-10 00:00:00+07","2011-01-15 00:00:00+07"}

Elements in arrays are most commonly referenced using the index of the element. PostgreSQL array index starts at 1. If you try to access an element above the upper bound, you won't get an error—only NULL will be returned. The next example grabs the first and last element of our array column.

```
SELECT fact[array_upper(fact, 1)] As firstFact
 FROM facts;
```

The array_upper() is used to get the upper bound of the array and the second parameter of the function indicates the dimension. PostgreSQL also supports array slicing using the start:end syntax and splice two arrays together with the concatenation operator. Here is the example:

```
SELECT fact [1:2] || fact[5:6] FROM facts;
```

Characters

There are tree basic types of character types in PostgreSQL: char, varchar, and text. Text in PostgreSQL is not stored any differently from varchar, and no performance difference for the same size data so it isn't divide into mediumtext, bigtext, and so forth. PostgreSQL's can also manipulate the regular expression with cool way. You can return the regex matches as tables, arrays, etc. Back-referencing and other fairly advanced search patterns are also supported. Our example shows you how to format phone numbers stored simply as contiguous digits:

```
SELECT regexp_replace('081291785304', '([0-9]{4})([0-9]{4})([0-9]{4})',
E'\(\\1\) \\2-\\3')
As x;
```

	x text
1	(0812) 9178-5304

The \\1, \\2, etc. refers to the elements in our pattern expression. We use the reverse solidus \(to escape the parenthesis. The E' is PostgreSQL syntax for denoting that a string is an expression so that special characters like \ would be treated literally.

Timestamp With Time Zone

Timestamp With Time Zone is a variant of TIMESTAMP that includes a time zone offset or time zone region name in its value. The time zone offset is the difference (in hours and minutes) between local time zone and UTC (Coordinated Universal Time, formerly Greenwich Mean Time). Here is the example using Jakarta / Bangkok time zone:

```
SELECT '2012-02-28 10:00 PM America/Los_Angeles'::timestamptz,'2012-02-28 10:00 PM America/Los_Angeles'::timestamp;
```

	timestamptz timestamp with time zone	timestamp timestamp without time zone
1	2012-02-29 13:00:00+07	2012-02-28 22:00:00

Operators and Functions for Time Data Types

We can use the versatile generate_series function for interval steps in PostgreSQL 8.3 or above. Here is the example:

```
SELECT (dt - interval '1 day')::date As eom
FROM generate_series('2/1/2012', '6/30/2012', interval '1 month') As dt;
```

	eom date
1	2012-01-31
2	2012-02-29
3	2012-03-31
4	2012-04-30
5	2012-05-31

Another popular activity is extracting or formatting parts of a complete date time. We can use the date_part() and to_char() functions for these problems. Here is the next example with the time zone aware data type.

```
SELECT dt, date_part('hour',dt) As mh, to_char(dt, 'HH12:MI AM') As formtime
 FROM generate_series('2012-03-11 12:30 AM', '2012-03-11 3:00 AM', interval
'30 minutes')
As dt
```

	dt timestamp with time zone	mh double precision	formtime text
1	2012-03-11 00:30:00+07	0	12:30 AM
2	2012-03-11 01:00:00+07	1	01:00 AM
3	2012-03-11 01:30:00+07	1	01:30 AM
4	2012-03-11 02:00:00+07	2	02:00 AM
5	2012-03-11 02:30:00+07	2	02:30 AM
6	2012-03-11 03:00:00+07	3	03:00 AM

By default, generate_series() will assume timestamp with time zone if you don’t explicitly cast to timestamp.

Building You Own Custom Type

PostgreSQL provides the feature if you wish to build your own custom type. For example, let’s build a complex number data type as follows:

```
CREATE TYPE ComplexNumber AS (r double precision, i double precision);
```

We can then use this type as a column type:

```
CREATE TABLE circuits(id text PRIMARY KEY, volt ComplexNumber);
```

We can then query our table with statements such as:

```
SELECT id, (volt).* FROM circuits;
```

Tables

In this section, we'll demonstrate some common table creation examples. Most are similar to or exactly what you'll find in other databases.

```
CREATE TABLE logs(
 log_id serial PRIMARY KEY , user_name varchar(50)
 , description text
 , log_ts timestamp with time zone NOT NULL DEFAULT current_timestamp);
CREATE INDEX idx_logs_log_ts ON logs USING btree(log_ts);
```

PostgreSQL is the only database that i know it offers the table inheritance. When you specify that a child table inherit from a parent, the child will be created with all the columns of the parent in addition to its own columns. All structural changes made to the parent will automatically propagate its child tables. Whenever you query the parent, all rows in the children are included as well. Noted that not every trait is passed down, like indexes and primary key constraints. Here is the example:

```
CREATE TABLE logs_2015(PRIMARY KEY(log_id)) INHERITS (logs);

ALTER TABLE logs_2015

 ADD CONSTRAINT chk_y2015

 CHECK (log_ts BETWEEN '2015-01-01'::timestamptz AND '2016-01-
01'::timestamptz);
```

PostgreSQL with version 9.0 or above provides another way of table creation whereby the column structure is defined by a composite data type. When using this method, you can't add additional columns directly to the table. The advantage of this approach is that if you have many tables sharing the same structure and you need to alter the column, you can do so by simply changing the underlying type.

We'll demonstrate this by creating the specific type:

```
CREATE TYPE user AS (username varchar(50), password varchar(50));
```

Then, we can then create a table that has rows of user instance type:

```
CREATE TABLE admin OF user (CONSTRAINT pk_admin PRIMARY KEY
(username));
```

If we want to add an email address to admin table, we can use the alter command as follows:

```
 ALTER TYPE admin ADD ATTRIBUTE email varchar(50) CASCADE;
```

Normally, you can't change the definition of a type if tables depend on that type. The CASCADE modifier allows you to override this restriction.

Exclusion Constraint

If you want to create an online reservation system, the first requirement is to make sure that no two reservations may overlap (i.e. no schedule conflicts). How can we make a constraint to do that?. PostgreSQL 9.0 provides a new constraint named Exclusion Constraint. It offers constraint enforcement mechanism more advanced that UNIQUE constraint. Its performance has not much different with UNIQUE constraint and can specify that no two tuples contain values that overlap with each other. For the example, we have to install the Temporal PostgreSQL first. Here is the example query that return error because the second data is overlap with the previous data:

```
CREATE TABLE reservation(during PERIOD);
ALTER TABLE reservation
   ADD EXCLUDE USING gist
   (during WITH &&);
INSERT INTO reservation VALUES('[2009-01-05, 2009-01-10]');
INSERT INTO reservation VALUES('[2009-01-07, 2009-01-12]');//Error
```

Chapter III

The PostgreSQL Unique Ways.

Window Functions

Window functions are a common ANSI-SQL feature supported in PostgreSQL since 8.4. Window function has an unusual ways to see and use the data without using joins or subqueries. Here is the example to obtain the average value for all records from facts:

```
SELECT tract_id, val, AVG(val) OVER () as val_avg FROM facts
WHERE SUBSTR(tract_id,1,3) = '000';
```

	tract_id character(10)	val double precision	val_avg double precision
1	00001	200	275
2	00002	250	275
3	00010	150	275
4	00020	500	275

The OVER () converted our conventional AVG() function into a window function. When PostgreSQL sees a window function in a particular row, it will actually scan all rows fitting the WHERE clause, perform the aggregation, and output the value as part of the row. You can also use PARTITION BY to subdivide the window into smaller panes and then to take the aggregate over those panes instead of over the entire set of rows. The result is then output along with the row depending on which pane it belongs to. In this next example, we repeat what we did before with a partition by tract_id's first three characters:

```
SELECT tract_id, val, AVG(val) OVER (PARTITION BY left(tract_id,3))
As val_avg_part
FROM facts ORDER BY tract_id;
```

	tract_id character(10)	val double precision	val_avg_part double precision
1	00001	200	275
2	00002	250	275
3	00010	150	275
4	00020	500	275
5	00100	300	300
6	00200	400	400

You can combine ORDER BY with PARTITION BY. It will restart the ordering for each partition. Here is the example using the previous query:

```
SELECT tract_id, val, AVG(val) OVER (PARTITION BY left(tract_id,3) ORDER
BY val) As val_avg_part
FROM facts ORDER BY tract_id;
```

	tract_id character(10)	val double precision	val_avg_part double precision
1	00001	200	175
2	00002	250	200
3	00010	150	150
4	00020	500	275
5	00100	300	300
6	00200	400	400

PostgreSQL provides the Window function named LEAD() and LAG() to take a forward or backward step over the ROW_NUMBER() function. ROW_NUMBER() is used to ordering the rows based on some condition.

If the retrieved rows are outside the window partition, both LEAD() and LAG() will return NULL.

Common Table Expressions (CTE)

Common Table Expressions (CTE) is the PostgreSQL feature that allow user to assign a temporary variable name to a query definition so that it can be reused in a larger query. This features doesn't exist in MySQL of any version. There are three different ways to use CTEs:

1. Standard CTE

   Standard CTE is a non-recursive and non-writable CTE that used for the purpose of readability of your SQL. Here is the example:

```
WITH cte as(
        SELECT AVG(val) as rata FROM facts
)
SELECT * FROM facts,cte
WHERE val > cte.rata;
```

	tract_id character(10)	tract_name character varying(50)	val double precision	rata double precision
1	00200	123123	400	300
2	00020	child	500	300

2. Writeable CTE

The writeable CTE was introduced in PostgreSQL 9.1 and extends the CTE to allow for update, delete, insert statements. Here is the example:

- First, we insert the dummy data into logs_2015 table.

```
INSERT INTO logs_2015(
        log_id, user_name, description, log_ts)
  VALUES (1, 'SJ', 'SJ Desc', '2015-04-04'),
  (2, 'HS', 'HS Desc', '2015-05-05'),
  (3, 'KS', 'KS Desc', '2015-06-06'),
  (4, 'RD', 'RD Desc', '2015-03-03')
```

- Create table logs_2015_01 that inherits the logs_2015 and have a constraint to check the first quarter of 2015.

```
CREATE TABLE logs_2015_01(PRIMARY KEY(log_id)
, CONSTRAINT chk_y2015_01 CHECK(log_ts >= '2015-01-01' AND log_ts <
'2015-05-01'))
INHERITS (logs_2015)
```

- Create the Writeable CTE to delete and return the data from logs_2015 where log_ts is before 1 May 2015.

```
WITH wCTE AS (DELETE FROM ONLY logs_2015
 WHERE log_ts < '2015-05-01' RETURNING *)
INSERT INTO logs_2015_01 SELECT * FROM wCTE;
```

	log_id integer	user_name character varying(50)	description text	log_ts timestamp with time zone
1	1	SJ	SJ Desc	2015-04-04 00:00:00+07
2	4	RD	RD Desc	2015-03-03 00:00:00+07

3. Recursive CTE

   PostgreSQL can also make the recursive CTEs using UNION ALL. To turn a basic CTE to a recursive one, add the RECURSIVE modifier after the WITH. With recursive CTE, you can have a mix of recursive and non-recursive table expressions like a tree structure. Here is the example if we want to select all table with its descendants:

   - Get list of all Table ID and table name that have child tables but have no parent table.

```
WITH RECURSIVE
 tbls AS (
 SELECT c.oid as tableoid,c.relname AS tablenamet
 FROM pg_class c
 LEFT JOIN pg_inherits As th ON th.inhrelid = c.oid
 WHERE th.inhrelid IS NULL AND c.relkind = 'r'::"char" AND
c.relhassubclass = true
```

- Gets all data in previous tbls recursively using UNION ALL. Then, join the data with pg_inherits and pg_class to get all related children using pg_class relname.

```
UNION ALL
SELECT  c.oid, tbls.tablename || '->' || c.relname AS tablename
FROM tbls INNER JOIN pg_inherits As th ON th.inhparent = tbls.tableoid
INNER JOIN pg_class c ON th.inhrelid = c.oid)
```

- Show the tbls data order by table name.

```
SELECT * FROM tbls ORDER BY tablename;
```

	tableoid oid	tablename name
1	25201	facts
2	25204	facts->facts child
3	25207	facts->facts child2
4	28624	logs
5	28676	logs->logs 2015
6	28687	logs->logs 2015->logs 2015 01

DISTINCT ON

DISTINCT ON is like the common DISTINCT, except that it allows you to define what columns to consider distinct, and order the data by the preferred column. Here is the example:

```
SELECT DISTINCT ON(left(tract_id, 3)) left(tract_id, 3) As part_id
 , tract_id, tract_name
FROM facts ORDER BY part_id;
```

	part_id text	tract_id character(10)	tract_name character varying(50)
1	000	00001	asdasd
2	001	00100	zxczxc
3	002	00200	123123

The ON modifier can take on multiple columns and they will determine the uniqueness of the preview data.

Composite Data Type

PostgreSQL provides a lot of flexibility for data types, we can make the custom data type as follows:

```
SELECT X FROM facts As X
```

	x facts
1	("00001 ",asdasd,200)
2	("00002 ",qweqwe,250)
3	("00010 ",asdjkl,150)
4	("00100 ",zxczxc,300)
5	("00200 ",123123,400)
6	("00020 ",child,500)

You can also take the advantage of JavaScript Object Notation (JSON) support in PostgreSQL 9.2 or above to use a combination of array_agg and array_to_json to return a single JSON object that will be used in other AJAX apps.

```
SELECT array_to_json(array_agg(X) ) As json_data
FROM (SELECT X FROM facts As X LIMIT 2) test;
```

	json_data json
1	[{"tract id":"00001 ","tract name":"asdasd","val":200},{"tract id":"00002 ","tract name":"qweqwe","val":250}]

Afterword

In conclusion, PostgreSQL is not only the common relational DBMS. PostgreSQL provides its unique ways, such as exclusion constraint, table inheritance, custom data types, JSON output, etc and all of them have a purpose to solve our common problem in database and its related apps. In the end, I apologize if there were any mistake in my training module. Thank you so much and see you next time.

References

https://it-ebooks.info/book/4784/