



PostgreSQL Fundamental

YODESAYA TIMPROM (P'MON)

INSTRUCTOR

Part I

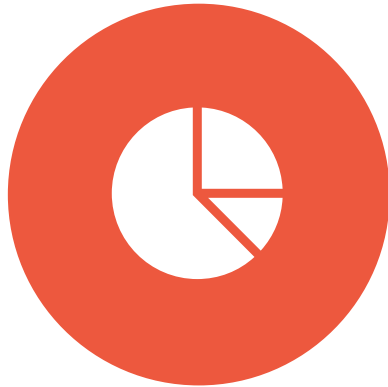
- ☐ Introduction & Environment Setup
- ☐ Syntax/Data Types
- ☐ Import & Export/Create/Drop Databases
- ☐ Schema/Create/Drop Tables

Part II

- ☐ Insert/Update/Delete Query
- ☐ Operators/Expressions/Functions
- ☐ Where Clauses/ARRAY & Full Text Search/Indexes
- ☐ Order By/Group By Clause/JOIN TABLE&VIEW



The Introduction of PostgreSQL, Installation & Environment Setup



COMPARING THE EDB
POSTGRES PLATFORM AND
POSTGRESQL



REQUIREMENTS SPEC



POSTGRES ENTERPRISE
MANAGER (PEM)



The Introduction of PostgreSQL

PostgreSQL ในปัจจุบันเป็นทั้ง open-source ที่ถือว่าเป็น object-relational database management system (ORDBMS) ที่ based on ผู้บุกเบิกมันขึ้นมาคือ [POSTGRES](#) และถูกพัฒนาใน Version 4.2 เป็นเวอร์ชันสุดท้ายที่ The University of California at Berkeley Computer Science Department จึงส่งให้ในปัจจุบันนำมาทำเป็น version ที่ available ในหลายๆ commercial database systems

SQL standard and offers many modern features:

- complex queries
- foreign keys
- triggers
- updatable views
- transactional integrity
- multiversion concurrency control

PostgreSQL can be extended by the user in many ways, for example by adding new

- data types
- functions
- operators
- aggregate functions
- index methods
- procedural languages




PostgreSQL Installation Pages

[Download the installer](#) certified by EDB for all supported PostgreSQL versions.

Platform support

The installers are tested by EDB on the following platforms. They can generally be expected to run on other comparable versions, for example, desktop releases of Windows:

PostgreSQL Version	64 Bit Windows Platforms	32 Bit Windows Platforms
15	2019, 2016	
14	2019, 2016	
13	2019, 2016	
12	2019, 2016, 2012 R2	
11	2019, 2016, 2012 R2	
10	2016, 2012 R2 & R1, 7, 8, 10	



Upcoming Webinar: What's New in PostgreSQL 15 • Dec 7 • Register Now

Solutions

Products

Services

Resources

Partners

Company

Contact us

Careers









Docs

Sign in

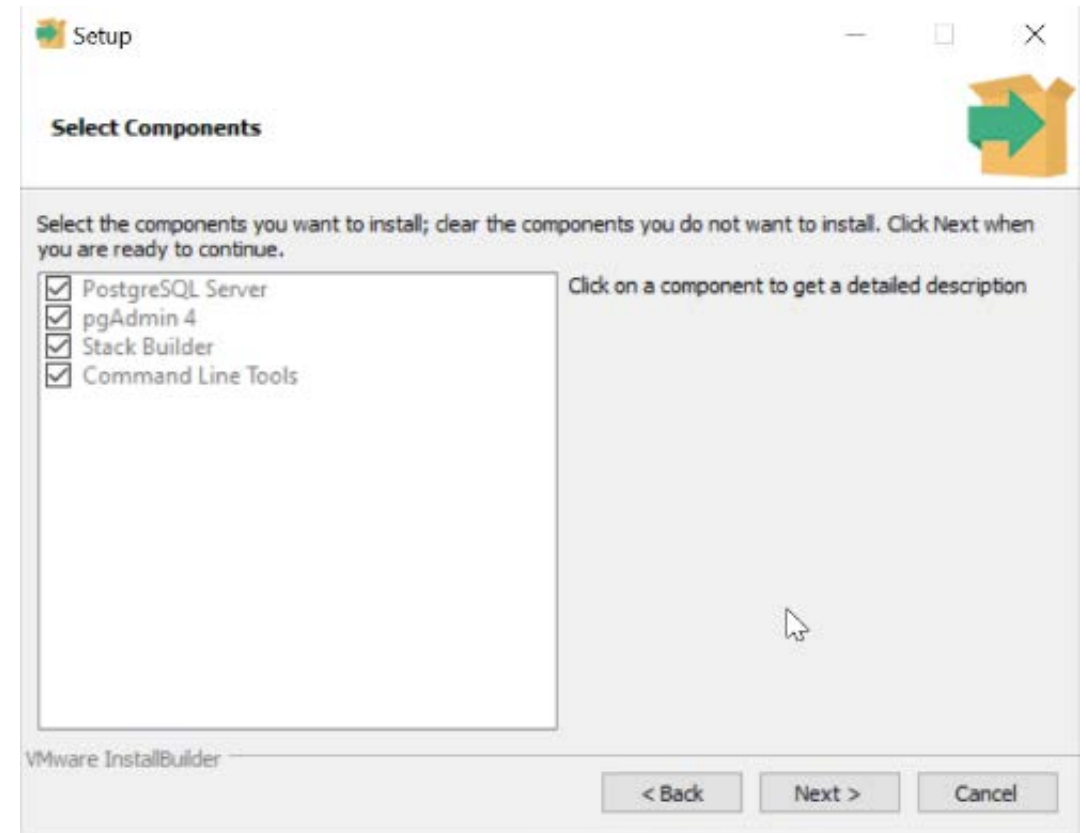
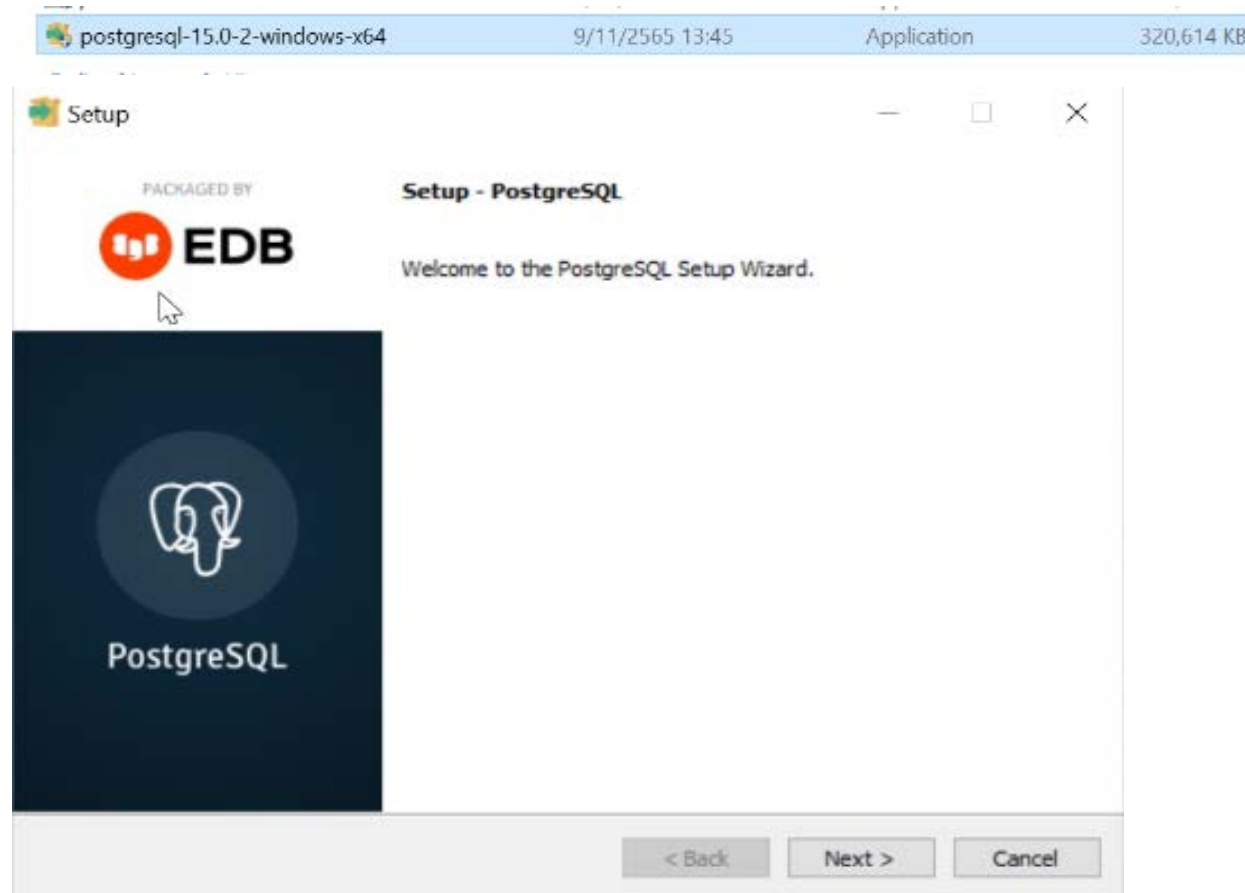
Get started

Download PostgreSQL

Open source PostgreSQL packages and installers from EDB

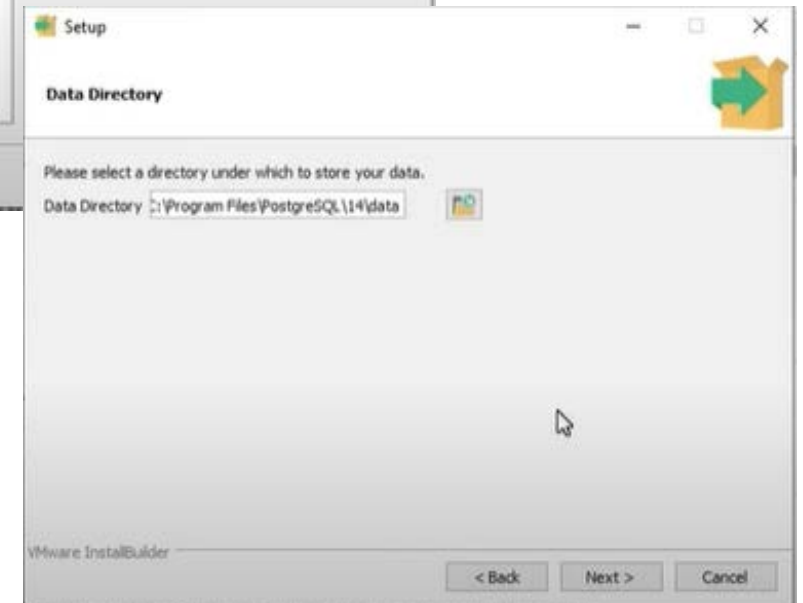
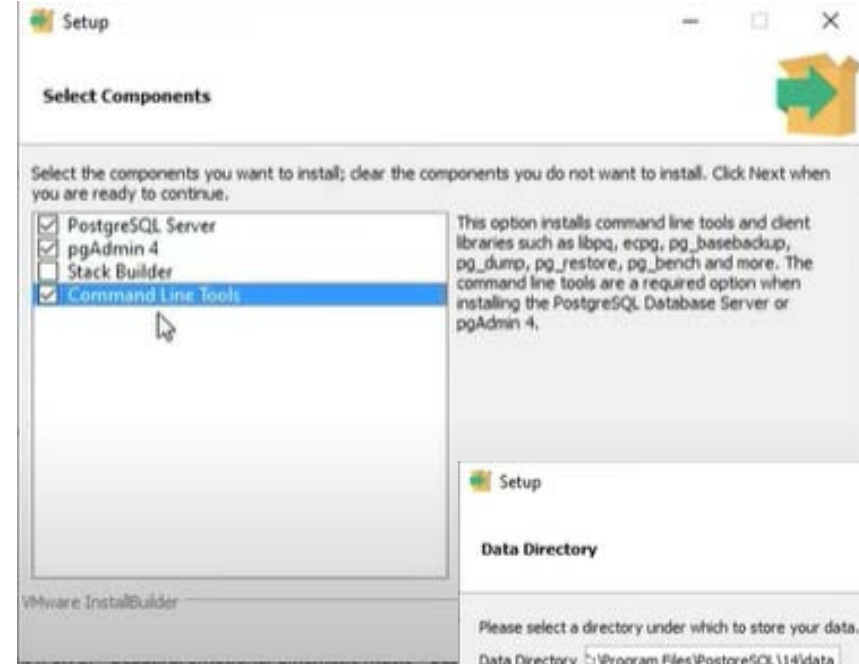
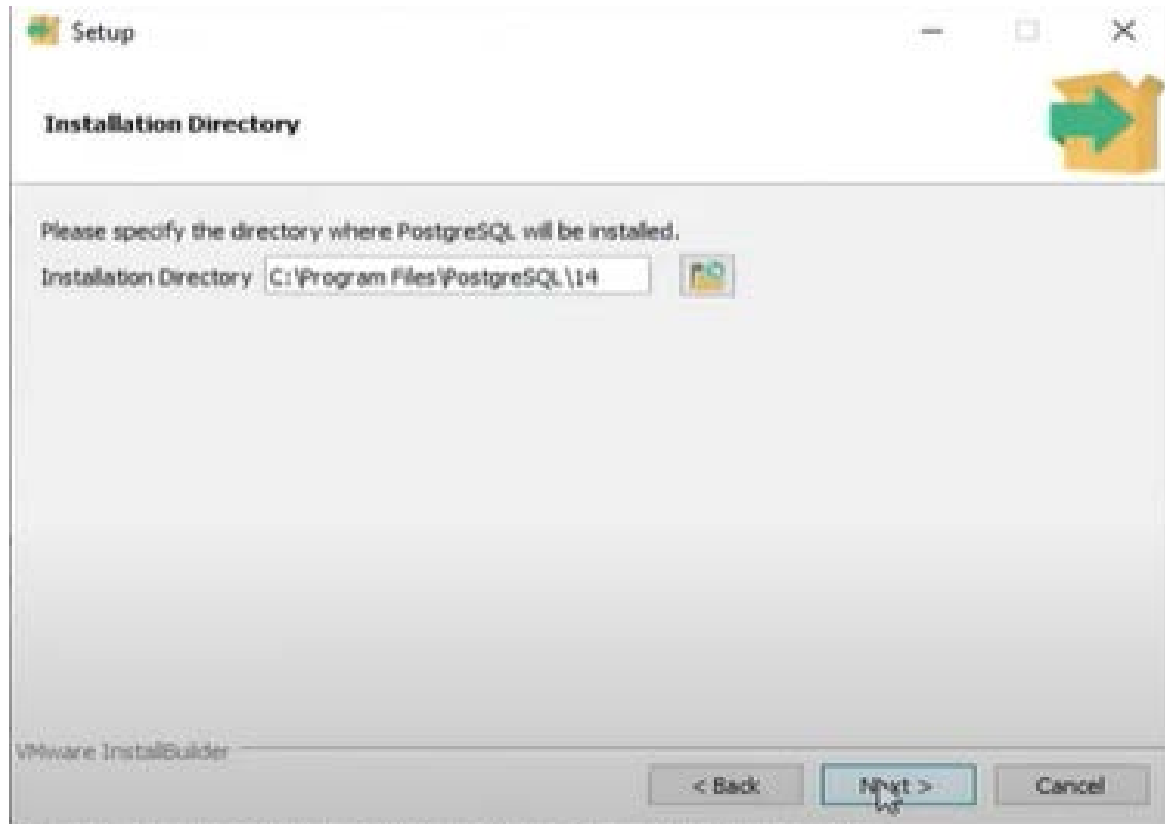
PostgreSQL Version	Linux x86-64	Linux x86-32	Mac OS X	Windows x86-64	Windows x86-32
15.1	postgresql.org	postgresql.org			Not supported
14.6	postgresql.org	postgresql.org			Not supported
13.9	postgresql.org	postgresql.org			Not supported
12.13	postgresql.org	postgresql.org			Not supported

PostgreSQL Installation Package and Wizard



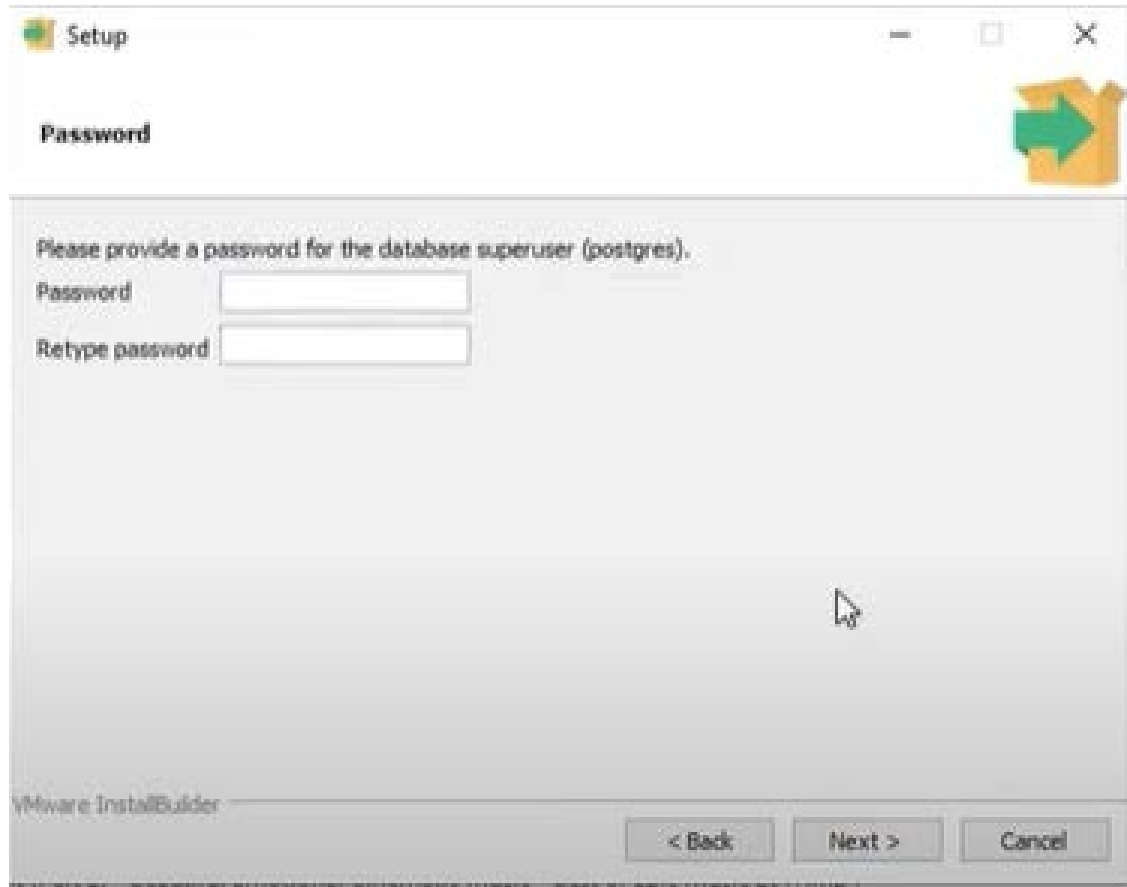
**Tick เลือกทุกช่องยกเว้น Stack Builder

PostgreSQL Installation Package and Wizard (ต่อ)

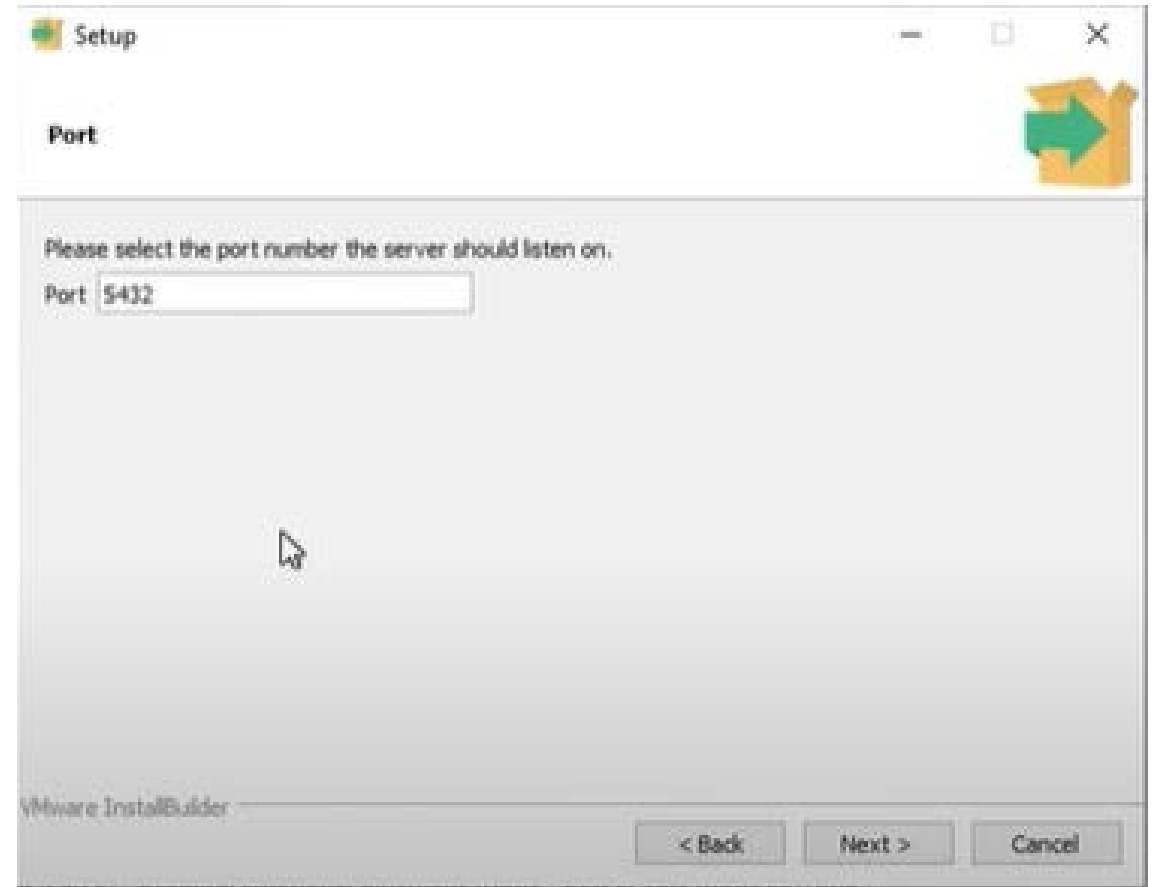


**ในรูปเป็นตัวอย่าง Path แต่เป็น Version 14 แต่ที่เราติดตั้งกันคือ version 15 (C:\Program Files\PostgreSQL\15)

PostgreSQL Installation Package and Wizard (ต่อ)

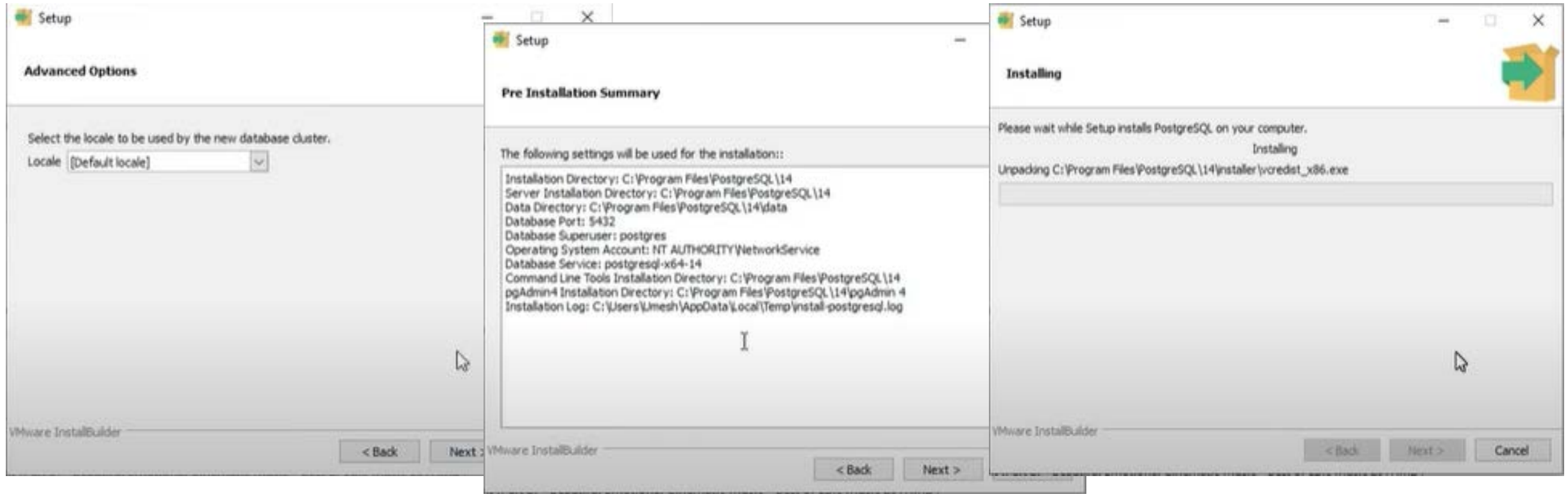


**ในรูปให้เราตั้ง password ที่เราต้องจำได้บันทึกเอาไว้ด้วยเพราะถ้าไม่เก็บ password เอาไว้แล้วลืมจะต้องถอนโปรแกรมทิ้งเลยแล้วไปลงใหม่ทั้งหมด



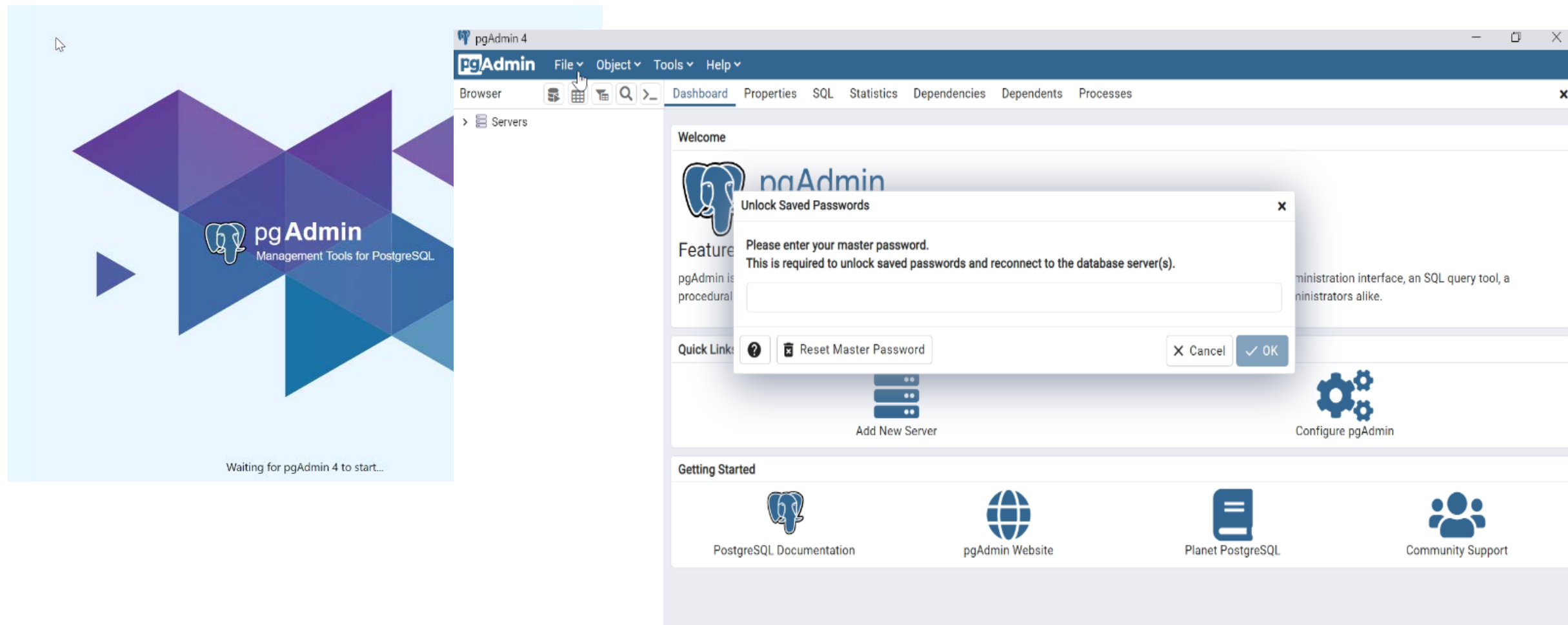
**default port คือ 5432

PostgreSQL Installation Package and Wizard (ต่อ)



**ถัดมาเลือก Default Locale > Next >> Next >> Next จนกระทั่งติดตั้งสำเร็จจะขึ้นหน้าจอสุดท้ายว่า Completing the PostgreSQL Setup Wizard Setup has finished installing PostgreSQL on your computer. ก็กดปุ่ม Finish เป็นอันเสร็จเรียบร้อย

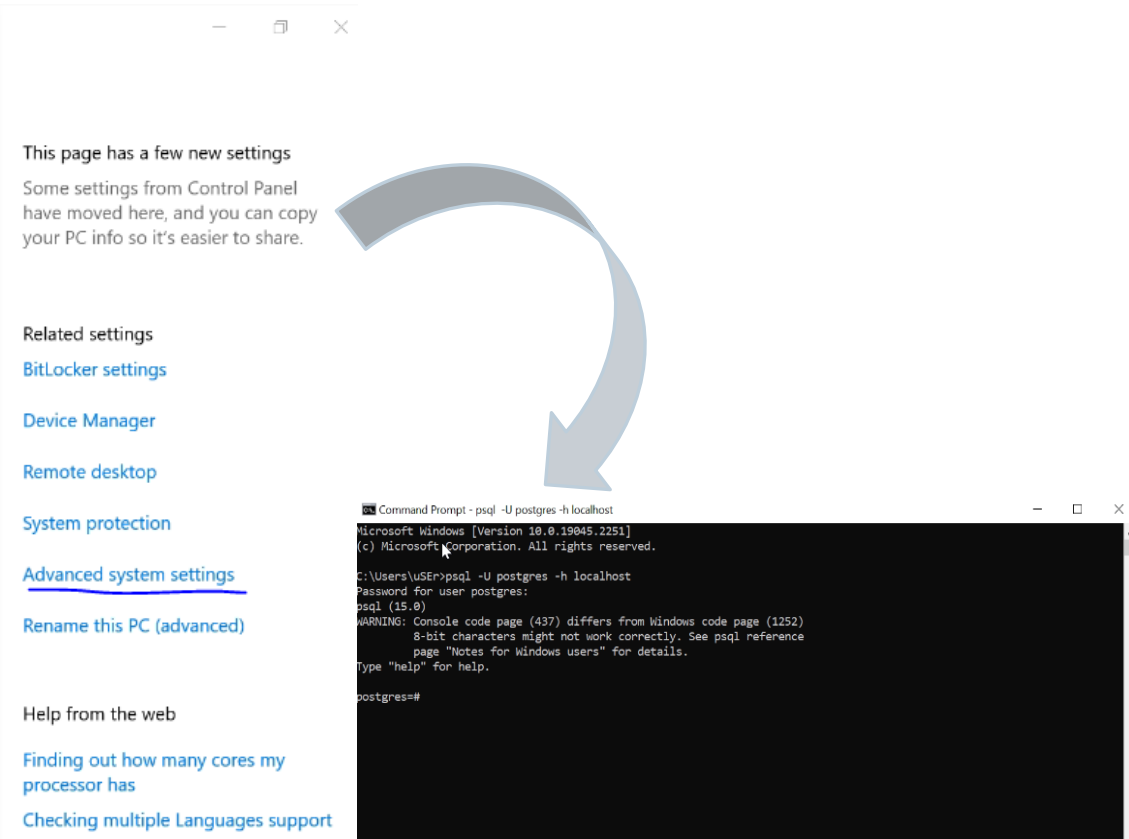
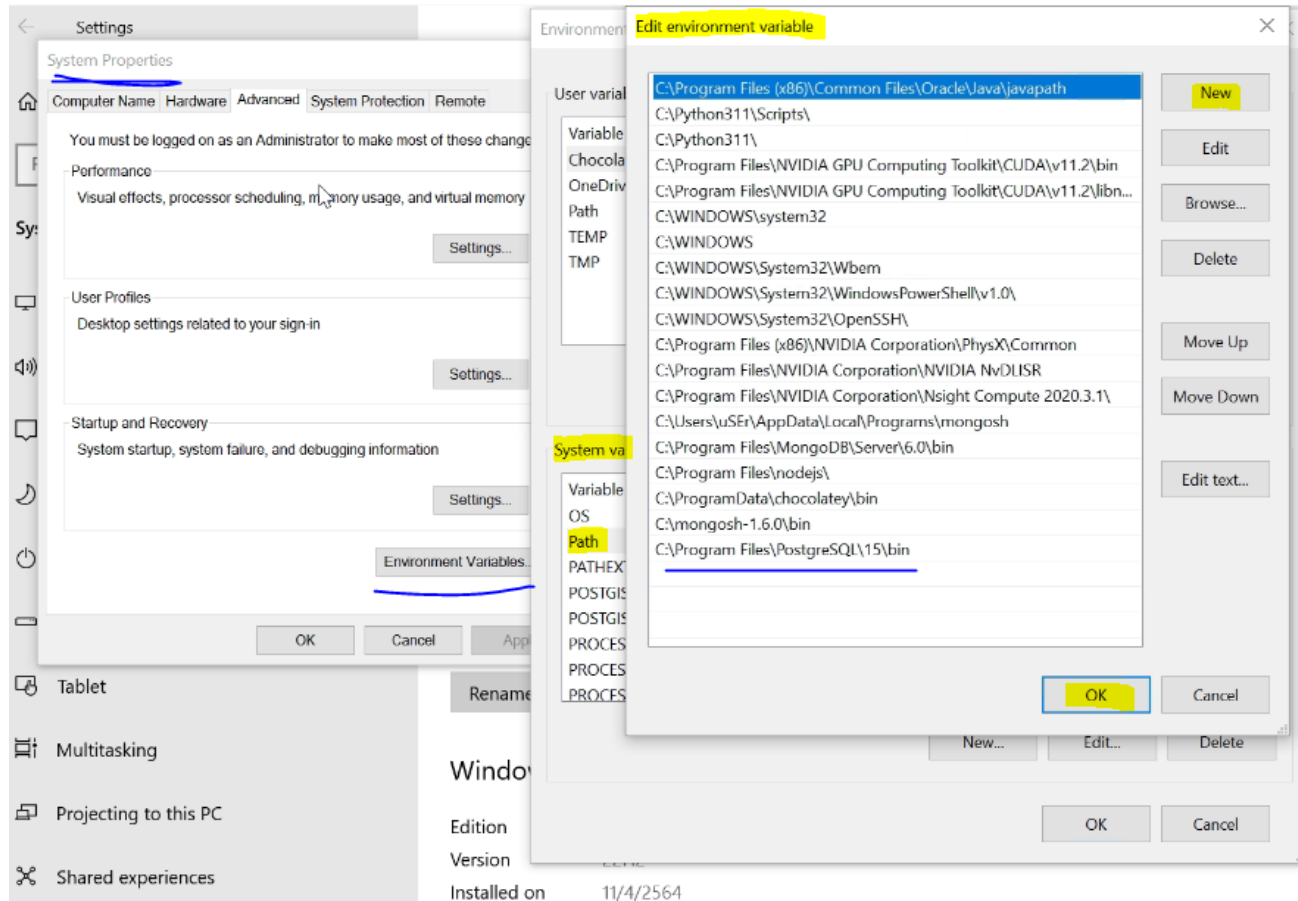
PostgreSQL and pgAdmin 4 Starting



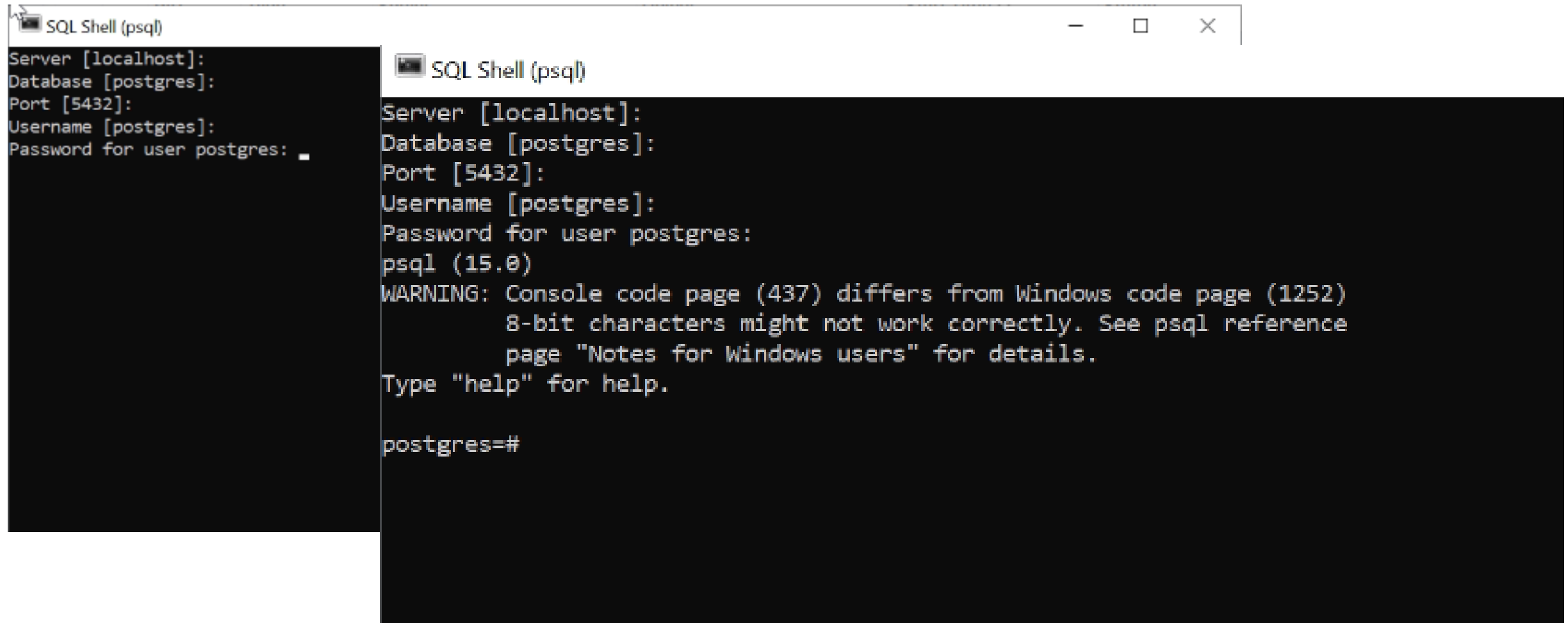
**เปิด pgAdmin 4 ขึ้นมารอโหลดขึ้นมาเสร็จในหน้าแรกจะมี pop-up ให้เราใส่ password แล้วกด > OK

PostgreSQL and env path call psql Shell command windows

**เปิด Right Click ที่ This PC > เลือก Properties menu > ไปที่ System เลือก Advanced system settings > จะมี pop-up ขึ้นมาเป็น System Properties >> ไปดูที่ Environment Variables... menu >> Click เข้าไปตรง System variables >> เลือก Variable ที่มีค่าเป็น Path >> จากนั้น Double Click เข้าไปทำการเพิ่ม New >> ใส่ค่าใหม่ C:\Program Files\PostgreSQL\15\bin >> กด OK จนกระทั่งปิด Pop-up ไปทั้งหมด



PostgreSQL and psql Shell command windows



The image displays two side-by-side terminal windows, both titled "SQL Shell (psql)". The left window shows the initial connection prompts: "Server [localhost]:", "Database [postgres]:", "Port [5432]:", "Username [postgres]:", and "Password for user postgres:". The right window shows the same prompts, followed by the psql version "psql (15.0)", a warning about console code page differences, and the prompt "postgres=#".

```
SQL Shell (psql)
Server [localhost]:
Database [postgres]:
Port [5432]:
Username [postgres]:
Password for user postgres: _

SQL Shell (psql)
Server [localhost]:
Database [postgres]:
Port [5432]:
Username [postgres]:
Password for user postgres:
psql (15.0)
WARNING: Console code page (437) differs from Windows code page (1252)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
Type "help" for help.

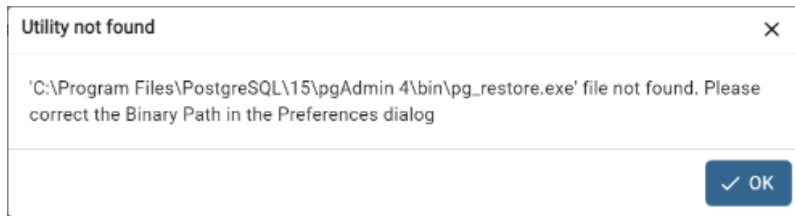
postgres=#
```

**เปิด psql ขึ้นมา.ให้เรากด enter ผ่านไปเรื่อยๆจนกระทั่งถึงบรรทัด Password for user postgres: ให้เราใส่ password

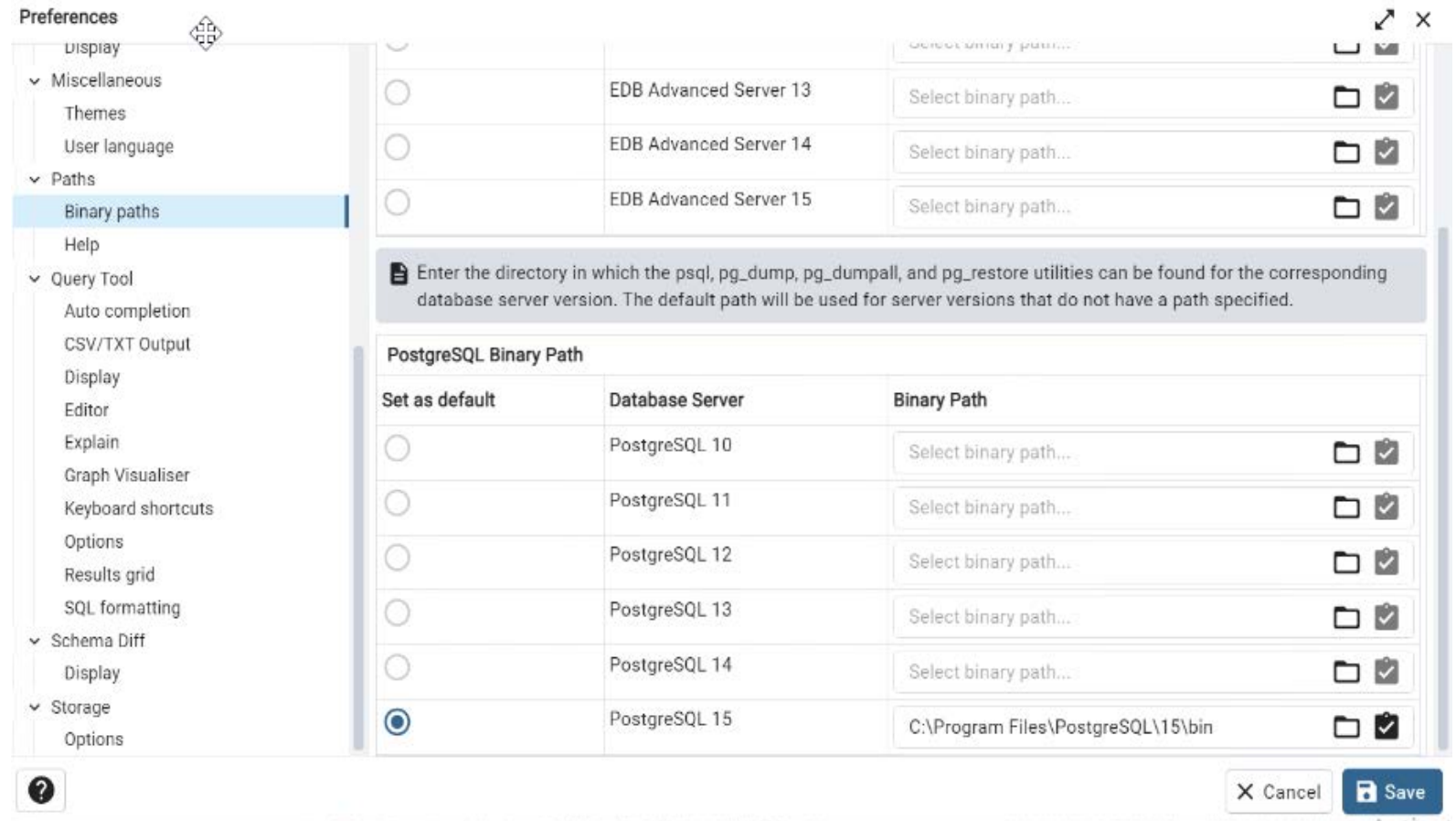
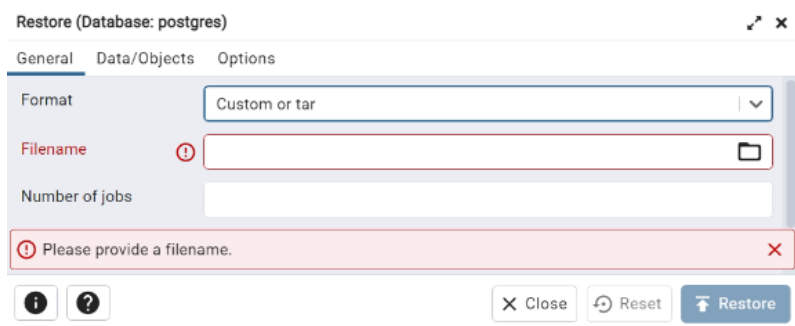
PostgreSQL and Environment Setup Binary Path to Support Restore Option

PostgreSQL Binary Path : C:\Program Files\PostgreSQL\15\bin

Before

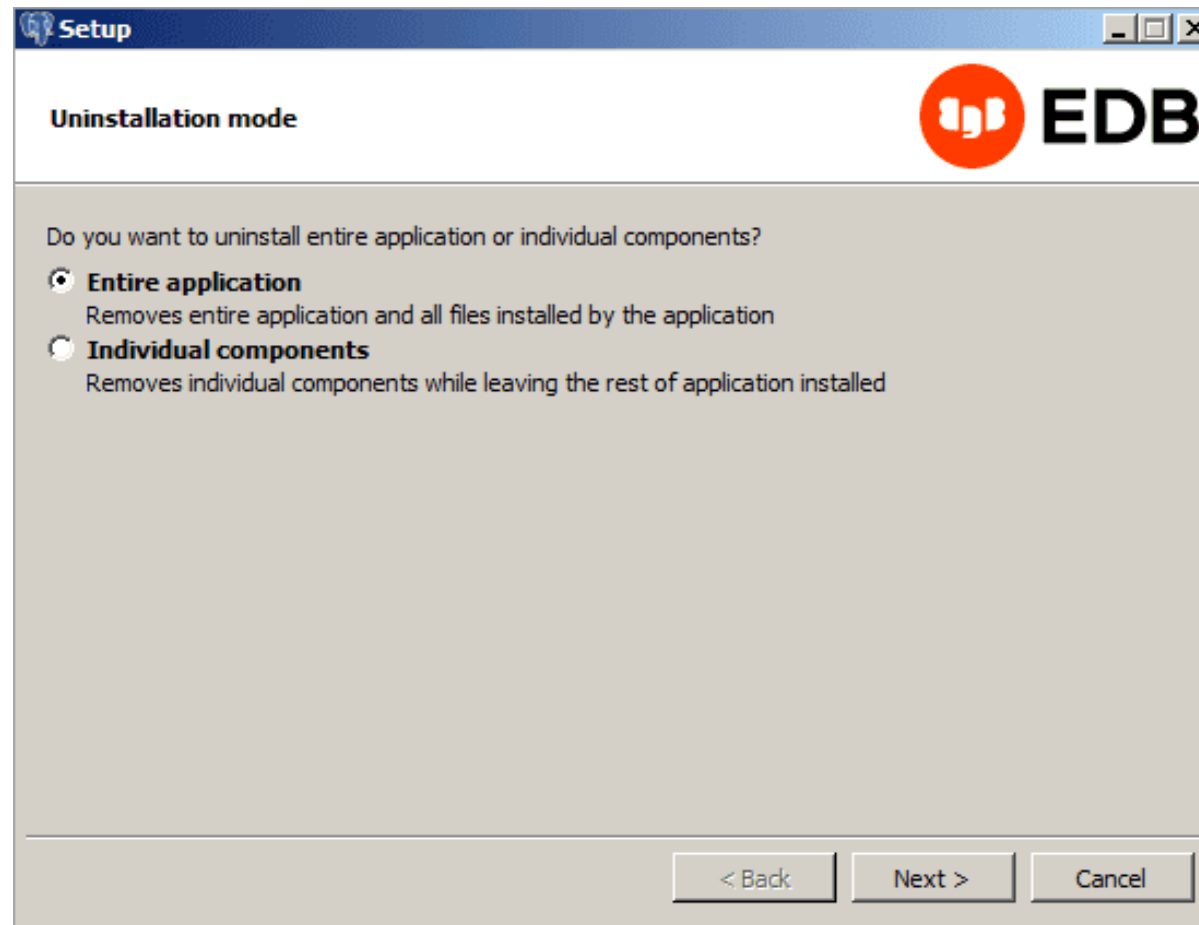


After



**เปิด pgAdmin 4 และไปที่ File > Preferences หรือ ไปที่ Server(1) > Configure pgAdmin > Binary Paths

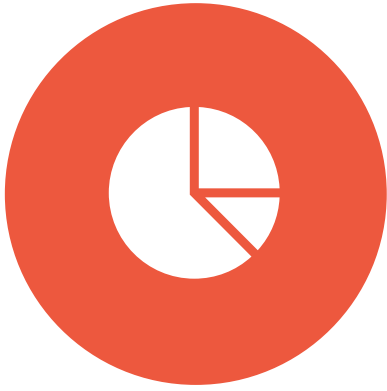
PostgreSQL Uninstalling EDB Postgres Advanced Server on Windows



**เปิด Uninstall ใน Windows ของเรา



Data Types & Syntax



POSTGRES SQL CHEATSHEET



DATA TYPES
& TYPE CONVERSION



SYNTAX



PostgreSQL Data Types

PostgreSQL แบ่งประเภทของ native data types ออกมาได้เป็น 21 ประเภทได้แก่ Numeric Types, Character Types, Binary Data Types, Date/Time Types, Boolean Type, Enumerated Types, Arrays, Bit String Types, XML Type เป็นต้น ไม่นับรวมจำพวกที่เป็น Custom Data Type ซึ่งเมื่อเราต้องการสร้างขึ้นมาเองเราจะใช้ syntax ดังต่อไปนี้

```
CREATE TYPE name AS
```

```
( [ attribute_name data_type [ COLLATE collation ] [, ... ] ] )
```

```
CREATE TYPE name AS ENUM
```

```
( [ 'label' [, ... ] ] )
```

```
CREATE TYPE name AS RANGE (
```

```
    SUBTYPE = subtype
```

```
    [ , SUBTYPE_OPCLASS = subtype_operator_class ]
```

```
    [ , COLLATION = collation ]
```

```
    [ , CANONICAL = canonical_function ]
```

```
    [ , SUBTYPE_DIFF = subtype_diff_function ]
```

```
    [ , MULTIRANGE_TYPE_NAME = multirange_type_name ]
```

```
)
```




PostgreSQL Data Types : SERIAL

ประเภท SERIAL column ใน PostgreSQL จะใช้สำหรับทำ key ที่เป็น running number หรือ identity หรือ auto_increment ตัวอย่างเช่น :

```
CREATE TABLE fruits(  
    id SERIAL PRIMARY KEY,  
    name varchar(50),  
    price int  
);
```

```
INSERT INTO fruits(name, price) VALUES ('coconut', 15);
```

```
INSERT INTO fruits(name, price) VALUES ('mango', 20), ('papaya', 25), ('durian', 120);
```

จะไม่สามารถ reset ค่า identity number ให้เป็น 1 กลับมาใช้ใหม่ได้จะไปเริ่มนับตัวเลขถัดจากที่เราลบออกไป

```
DELETE FROM fruits;
```

เวลาจะ clear ค่า identity number ออกใหม่ทั้งหมดเพื่อเริ่มต้นนับ 1 ใหม่

```
TRUNCATE TABLE fruits RESTART IDENTITY;
```



PostgreSQL Type Conversion

ในการ query ข้อมูลจำเป็นต้องมีการผสมประเภทข้อมูลที่แตกต่างกันในนิพจน์เดียว PostgreSQL มี function และวิธีการอำนวยความสะดวกมากมายสำหรับการแปลงข้อมูลให้เราสามารถปรับแต่งได้ตามที่ต้องการ

Operators

การ CAST >> SELECT |/ CAST(40 AS double precision) AS "square root of 40";

>> SELECT ~ CAST('20' AS int8) AS "negation";

การทำ String Concatenation Operator Type Resolution >> SELECT text 'abc' || 'def' AS "text and unknown";

การทำ Array Inclusion Operator Type Resolution >> SELECT array[1,2] <@ '{1,2,3}' as "is subset";

การทำ type CAST ด้วย syntax เพื่อ convert ค่าของ type หนึ่งไปยังอีก type หนึ่ง >> expression::type เช่น SELECT '100'::INTEGER, '01-OCT-2022'::DATE;

Functions

การใช้ TO_CHAR(expression,format), to_timestamp(timestamp, format), to_date(text,format), to_number(string, format)

การใช้ Rounding Function Argument Type Resolution >> SELECT round(4, 4);

การทำ Variadic Function Resolution >> CREATE FUNCTION public.variadic_example(VARIADIC numeric[]) RETURNS int

LANGUAGE sql AS 'SELECT 1';

CREATE FUNCTION

การทำ Substring Function Type Resolution >> SELECT substr(varchar '1234', 3);

Value Storage

การทำ character Storage Type Conversion >> CREATE TABLE vv (v character(20));

INSERT INTO vv SELECT 'abc' || 'def';

SELECT v, octet_length(v) FROM vv;



PostgreSQL Syntax : Identifiers and Key Words

คำสำคัญและตัวระบุ (*Key words and identifiers*) ที่ไม่ใช่เครื่องหมายอัฒภาคมันจะไม่คำนึงถึงตัวพิมพ์เล็กและใหญ่เลย ดังนั้นเราสามารถใช้ :

```
SELECT * FROM CATEGORIES LIMIT 5;
```

```
select * from categories limit 5;
```

```
Select * FRom public.Orders limit 2;
```

แต่ที่นิยมใช้เป็นสากลก็คือการเขียน Key words เป็นตัวพิมพ์ใหญ่และชื่อเป็นตัวพิมพ์เล็ก เช่น: `SELECT * FROM public.employees ORDER BY employee_id ASC;`

นอกจากนั้นสามารถใช้แบบ delimited identifier or quoted identifier เช่น `SELECT * FROM "orders" ORDER BY "employee_id" ASC;`

Reserved Key Words คือ คำสำคัญที่สงวนไว้ซึ่งไม่ได้รับอนุญาตให้เป็น identifiers และไม่ได้รับอนุญาตในการใช้งานใดๆ นอกจากเป็นโทเค็นพื้นฐานใน SQL statements, โดย Keywords จำพวกนี้ไม่ได้รับอนุญาตให้ใช้เป็นชื่อคอลัมน์หรือตาราง แม้ว่าในบางกรณี คำเหล่านี้ได้รับอนุญาตให้เป็นป้ายชื่อคอลัมน์ (เช่น ใน AS clause)

Non-reserved Keywords คือ Keywords ที่ไม่ได้สงวนไว้ซึ่งมีความหมายที่กำหนดในภาษา แต่อนุญาตให้ใช้เป็น identifiers ได้เช่นกัน Postgres มี additional keywords ซึ่งอนุญาตการใช้งานที่ไม่จำกัดที่คล้ายกัน โดยเฉพาะอย่างยิ่ง keywords เหล่านี้ได้รับอนุญาตให้ใช้เป็นชื่อคอลัมน์หรือตารางได้



PostgreSQL Syntax : Identifiers and Key Words

เราสามารถกำหนดการเขียน syntax ใน SQL statements ได้หลายแบบ ;

DDL (Data Definition Language) : ภาษาสำหรับการจัดการ และนิยามโครงสร้างของฐานข้อมูล เป็นภาษาที่มีไว้สำหรับการจัดการฐานข้อมูลโดยเฉพาะไม่ว่าจะเป็นการ สร้างฐานข้อมูล, แก้ไขหรือลบฐานข้อมูล โดยในภาษา DDL นั้นประกอบไปด้วยภาษาคำสั่งต่าง ๆ ก็คือ CREATE, ALTER, DROP ยกตัวอย่างเช่น

```
uPDaTE my_Table SeT a = 5;
```

DML (Data Manipulation Language) คือภาษาสำหรับการจัดการข้อมูลที่จัดเก็บอยู่ในตารางข้อมูล ซึ่งในกลุ่มภาษา DML นั้นจะครอบคลุมการจัดการข้อมูลทั้งหมด เช่น การเพิ่ม, แก้ไข, ค้นหา และลบข้อมูล โดยคำสั่งต่าง ๆ ก็คือ SELECT, INSERT, UPDATE, DELETE เป็นต้น

DCL (Data Control Language) : DCL เป็นกลุ่มคำสั่งที่จะช่วยให้ผู้บริหารฐานข้อมูล (DBA) สามารถควบคุมฐานข้อมูลเพื่อกำหนดสิทธิการอนุญาต (Grant) หรือการยกเลิกการเข้าใช้ (Revoke) ฐานข้อมูล ซึ่งเป็นกระบวนการป้องกันความปลอดภัยในฐานข้อมูล รวมทั้งการจัดการทรานแซกชัน (Transaction Management) ต่างๆได้ เป็นต้น

PostgreSQL Syntax : Constants

การ define ค่า constant ใน pgSQL เราสามารถใช้ syntax ตามนี้: `constant_name CONSTANT data_type := expression;`

```
DO $$ DECLARE
```

```
    VAT CONSTANT NUMERIC := 0.1;
```

```
    net_price  NUMERIC := 20.5;
```

```
BEGIN  RAISE NOTICE 'The selling price is %', net_price * ( 1 + VAT );
```

```
END $$;
```



PostgreSQL Syntax : Special Characters

หรือ ตัวอักษรพิเศษ : อักขระบางตัวที่ไม่ใช่ alphanumeric (อักษรและตัวเลข) มีความหมายพิเศษที่แตกต่างจาก operator รายละเอียดเกี่ยวกับการใช้งาน

dollar sign (\$) ; ตามด้วยตัวเลขจะใช้เพื่อแสดงพารามิเตอร์ตำแหน่งใน body ของนิยามฟังก์ชันหรือคำสั่งที่เตรียมไว้ และเครื่องหมายดอลลาร์สามารถเป็นส่วนหนึ่งของ identifier หรือ string constant ที่เสนอราคาเป็นดอลลาร์ไว้อีกด้วย

Parentheses (()) : วงเล็บ มีความหมายตามปกติในการจัดกลุ่มนิพจน์ (group expressions) และบังคับใช้ลำดับความสำคัญ ในบางกรณี จำเป็นต้องใช้วงเล็บเป็นส่วนหนึ่งของ syntax ในบาง SQL command โดยเฉพาะ

Brackets ([]) ; วงเล็บ ([]) ใช้เพื่อเลือกองค์ประกอบของอาร์เรย์ เพิ่มเติมเกี่ยวกับ [Array](#)

Commas (,) : เครื่องหมายจุลภาค (,) ใช้ในโครงสร้างของบาง syntactical เพื่อให้แยกองค์ประกอบของรายการ

semicolon (;) ; เครื่องหมายอัฒภาค (;) ใช้กำหนดการยุติคำสั่ง SQL

colon (:) : เครื่องหมายทวิภาค (:) ใช้เพื่อเลือก ส่วนของ data แยกจากอาร์เรย์

asterisk (*) : เครื่องหมายดอกจัน (*) ใช้ในบางบริบทเพื่อแสดงฟิลด์ทั้งหมดของแถวตาราง หรือ composite value (ค่าผสม) นอกจากนี้ยังมีความหมายพิเศษเมื่อใช้เป็นอาร์กิวเมนต์ ของ ฟังก์ชันการรวม (argument of an aggregate function) คือหมายถึง aggregate ไม่ต้องการใช้งาน explicit parameter นั้นเอง

period (.) : จุด (.) ใช้ในค่าคงที่ตัวเลข และเพื่อแยกชื่อ schema, table, และชื่อ column ออกจากกันเป็นต้น

PostgreSQL Syntax : Comments

การใช้ comment ใน SQL มี 2 แบบ

1. Single line -- This is a standard SQL comment
2. Multiple lines /* multiline comment

* with nesting: /* nested block comment */

*/



Import & Export/ Create/Drop Databases



IMPORT & EXPORT
SQL SYNTAX SELECT



SQL SYNTAX CREATE
DATABASE



SQL SYNTAX DROP
DATABASE

PostgreSQL : Import & Export

Exporting a PostgreSQL database

1. การใช้ pg_dump program รูปแบบคำสั่งเป็นดังนี้

```
pg_dump -U dbusername dbname > dbexport.pgsql
```

2. การใช้ phpPgAdmin

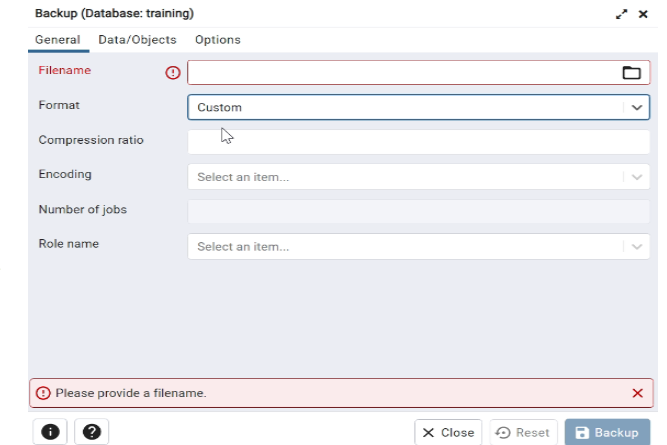
ทำการ click ขวาที่ Database ที่ต้องการแล้วเลือกเมนู Backup...

3. การ Export ด้วย CSV files

เลือกที่ table และเลือก View/Edit Data จะทำการแสดงข้อมูลเลือก icon เพื่อ export CSV file นำไปใช้งานต่อไป



4. การ Export ด้วย 3rd Party Tools อื่นๆ



Importing a PostgreSQL database

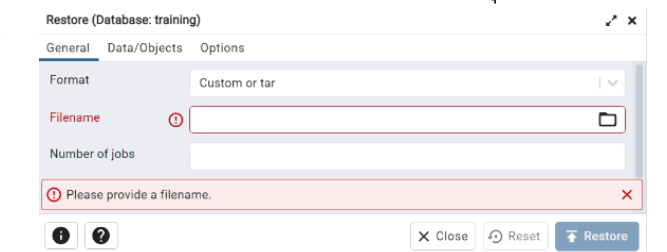
1. การใช้ psql program รูปแบบคำสั่งเป็นดังนี้

```
psql -U username dbname < dbexport.pgsql
```

2. การใช้ phpPgAdmin

ทำการ CREATE DATABASE เปรลาขึ้นมาก่อน จากนั้นไปที่ database click ขวาเลือกเมนู Restore... จากนั้นเลือก browse เลือกไฟล์ที่ต้องการจาก Filename และกดปุ่ม Restore ข้อมูลในตารางก็จะนำเข้ามาไปยัง schema > Tables ที่เราใช้งานอยู่

3. การ Import ด้วย INSERT command ที่ Query Tool





PostgreSQL : CREATE Databases

Create database with SQL command

```
CREATE DATABASE <database_name> WITH OWNER <username>;
```

ในการสร้างฐานข้อมูล เราต้องเป็น superuser หรือมีสิทธิ์พิเศษ CREATEDB

ในการ create role > [PostgreSQL: Documentation: 15: CREATE ROLE](#)

นอกจากนี้ยังมีการใช้ Parameters เพื่อระบุใช้ในการสร้าง Database อีกด้วย

นอกจากนั้นเรายังสามารถใช้ shell command ในการสร้าง Database โดย PostgreSQL provided psql command เอาไว้ให้ใน bin เรียบร้อยแล้วสามารถเรียกใช้งานได้
รูปแบบคือ createdb [connection-option...] [option...] [dbname [description]]

ตัวอย่าง :

```
$ createdb -p 5000 -h eden -T template0 -e demo
```

ก็คือ

```
CREATE DATABASE demo TEMPLATE template0;
```

PostgreSQL : Character Set Support

PostgreSQL provided the character sets available for use in PostgreSQL. สามารถตรวจสอบใน psql command line ด้วย \l หรือ psql -l ก็ได้

**ส่วนเพิ่มเติม > <https://www.postgresql.org/docs/current/multibyte.html>



PostgreSQL : DROP Databases

Drop database with SQL command

```
DROP DATABASE IF EXISTS <database_name>;
```

คำสั่งจะทำการลบรายการแคตตาล็อกสำหรับฐานข้อมูลและลบไดเรกทอรีที่มีข้อมูล สามารถดำเนินการได้โดยเจ้าของฐานข้อมูลเท่านั้น ไม่สามารถดำเนินการได้ในขณะที่เราเชื่อมต่อกับฐานข้อมูลเป้าหมายอยู่ในขณะนั้น ในการลบ database มี parameters เพิ่มเติมอีก > [PostgreSQL: Documentation: 15: DROP DATABASE](#)
นอกจากนี้ หากบุคคลอื่นเชื่อมต่อกับฐานข้อมูลเป้าหมาย คำสั่งนี้จะล้มเหลวเว้นแต่เราจะใช้ตัวเลือก FORCE ดังนี้ DROP DATABASE mydb WITH (FORCE);

นอกจากนั้นเรายังสามารถใช้ shell command ในการลบ Database โดย PostgreSQL provided psql command เอาไว้ให้ใน bin เรียบร้อยแล้วสามารถเรียกใช้งานได้
รูปแบบคือ dropdb [connection-option...] [option...] dbname หากสั่ง force ดังนี้ dropdb mydb -force หรือ dropdb mydb -f
ตัวอย่าง :

```
$ dropdb -p 5000 -h eden -i -e demo
```

ก็คือ

```
DROP DATABASE demo;
```

PostgreSQL : Rename database [PostgreSQL: Documentation: 15: ALTER DATABASE](#)

```
ALTER DATABASE <old_name> RENAME TO <new_name>;
```

```
ALTER DATABASE name OWNER TO { new_owner | CURRENT_ROLE | CURRENT_USER | SESSION_USER }
```

```
ALTER DATABASE name [ [ WITH ] option [ ... ] ]
```

ตัวอย่างเช่น ALTER DATABASE test SET enable_indexscan TO off;



PostgreSQL : Table Inheritance

การสืบทอดตาราง จากตารางหลักที่ต้องการไปยังตารางที่ต้องการสร้างขึ้นใหม่ ตัวอย่างเช่น

```
Create table employee (  
    id serial primary key,  
    fname text,  
    lname text  
);
```

```
Create table permanent_emp (  
    join_date date,  
    salary int  
) inherits (employee);
```

```
Alter table employee add column gender char(1);
```

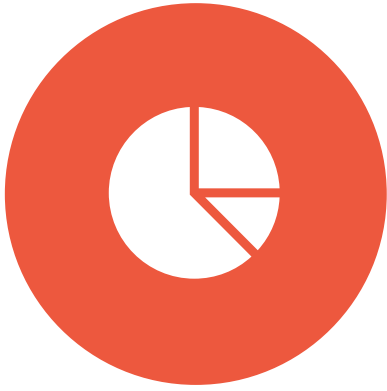
```
Alter table employee add column dob date;
```

```
Alter table permanent_emp rename column salary to salaries; /*การแก้ไขตารางความสัมพันธ์ที่ inherit object table มาจะไม่มีผลต่อตารางหลัก*/
```

Noted : ข้อควรระวังทุกครั้งที่เรามีการ insert ข้อมูลไปยังตารางใดตารางหนึ่งที่มี inherits ต่อกันข้อมูลจะถูกเก็บลงไปทั้ง 2 หรือทุกตารางที่มันทำ inheritance กันมา

และทุกครั้งที่เราทำการแก้ไขตารางหลัก (alter table) ในที่นี้คือ employee ตัวตารางที่สืบทอดคุณสมบัติของมันไปก็จะมีผลในการแก้ไขตารางของตัวมันเองแต่ละตารางไปด้วยโดยอัตโนมัติ เช่นกัน

Schema, Create/Drop Tables



SCHEMA



CREATE TABLE



DROP TABLES



PostgreSQL : Schema

คือ namespace ที่เก็บ objects ของ database ที่เราอ้างถึงหรือเรียกใช้ อย่างเช่น tables, views, indexes, data types, functions, stored procedures และ operators เป็นต้น การเข้าถึง object ที่อยู่ใน schema จำเป็นต้องเรียกใช้ syntax : schema_name.object_name

มีเหตุผลข้อดีหลายประการที่เราอาจต้องการใช้สคีมา:

1. เพื่อให้ผู้ใช้หลายคนใช้ฐานข้อมูลเดียวโดยไม่รบกวนซึ่งกันและกัน
2. เพื่อจัดระเบียบวัตถุฐานข้อมูลเป็นกลุ่มตรรกะเพื่อให้จัดการได้มากขึ้น
3. แอ็พพลิเคชันของบุคคลที่สามารถใส่ใน schema แยกกันได้ เพื่อไม่ให้ชนกับชื่อของอ็อบเจกต์อื่น
4. Schema นั้นคล้ายคลึงกับไดเรกทอรีในระดับระบบปฏิบัติการ ยกเว้นแต่ว่า schema นั้นไม่สามารถซ้อนกันได้

การ Creating a Schema

รูปแบบ >> CREATE SCHEMA myschema;

เวลาเราอ้างถึงหรือเรียกใช้ ดังนี้ : schema.table หรือ database.schema.table

และเมื่อต้องการเข้าถึงในการสร้าง table CREATE TABLE myschema.mytable (...);

เมื่อต้องการสร้าง schema ที่เป็นของคนอื่น หรือ เป็นการสร้าง schema ที่ระบุผู้มีสิทธิ์เข้าถึง

รูปแบบ >> CREATE SCHEMA schema_name AUTHORIZATION user_name;

การ Drop a schema

รูปแบบ >> DROP SCHEMA myschema;

หากต้องการลบ schema รวมถึงวัตถุที่มีอยู่ทั้งหมด ให้ใช้:

รูปแบบ >> DROP SCHEMA myschema CASCADE;



PostgreSQL : Create Tables

Creating a New Table ใช้ syntax CREATE TABLE เพื่อ define a new table
ยกตัวอย่างเช่น :

```
CREATE TABLE films (  
    code      char(5) CONSTRAINT firstkey PRIMARY KEY,  
    title     varchar(40) NOT NULL,  
    did       integer NOT NULL,  
    date_prod date,  
    kind      varchar(10),  
    len       interval hour to minute  
);
```

หรือ

```
CREATE TABLE films (  
    code      char(5),  
    title     varchar(40),  
    did       integer,  
    date_prod date,  
    kind      varchar(10),  
    len       interval hour to minute,  
    CONSTRAINT code_title PRIMARY KEY(code,title)  
);
```



PostgreSQL : Drop Tables

Drop a Table ใช้ syntax ดังนี้

```
DROP TABLE [IF EXISTS] table_name  
[CASCADE | RESTRICT];
```

ยกตัวอย่างเช่น :

```
DROP TABLE IF EXISTS authors;
```

หรือ DROP TABLE author;

ในกรณีที่ ต้องการลบอบเจกต์ที่อ้างอิงทั้งหมดออกก่อนที่จะทิ้งตาราง สามารถใช้ตัวเลือก CASCADE ดังนี้:

```
DROP TABLE authors CASCADE;
```

Drop multiple tables ใช้การระบุชื่อ table ต่อกันด้วยเครื่องหมาย comma ดังนี้ :

```
DROP TABLE tablename1, tablename2;
```



PostgreSQL : Privileges (Tables)

การสร้าง user และกำหนดสิทธิการใช้งาน (privileges) ให้กับตารางเรากำหนดสร้างได้จาก psql command line หรือจะทำที่ GUI pgAdmin ก็ทำได้ดังนี้

CREATE USER รูปแบบ : CREATE USER name [[WITH] option [...]]

ตัวอย่างเช่น : create user alpha with encrypted password 'test122022';

GRANT PRIVILEGES รูปแบบ : GRANT group_role TO role1, ...; หรือ GRANT SELECT ON TABLE <schema.tablename> TO user;

ตัวอย่างเช่น : grant select on table public.benjerry to alpha;

หรือให้สิทธิอื่นๆในการเข้าถึง จัดการ Table ได้ด้วยเช่น :

grant select, insert, update, delete on table public.menu to alpha;

หรือให้สิทธิกับทุก Tables grant select, insert, update, delete on all tables in schema public to alpha;

หรือให้สิทธิทั้งหมดไปยัง Table ทั้งหมดของ Schema ที่ต้องการไปให้ user จะเขียน command เป็น :

grant all privileges on all tables in schema public to alpha;

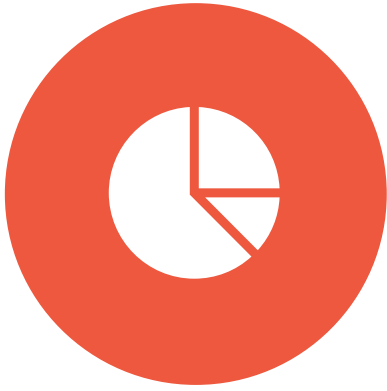
สำหรับการยกเลิกให้สิทธิสามารถใช้ REVOKE command รูปแบบ : REVOKE group_role FROM role1, ... ;

หรือ REVOKE [GRANT OPTION FOR] { USAGE | ALL [PRIVILEGES] } ON TABLE IN SCHEMA <schemaname> TO user;

ตัวอย่างเช่น : revoke all privileges on all tables in schema public from alpha;

Noted: ในส่วน GUI ไปที่ pgAdmin เลือกที่ table ที่อยู่ใน schema แล้ว Right Click ไป Menu Properties... จากนั้นไปดูยัง Tab > Security > Privileges ที่นี้เราจะสามารถจัดการให้สิทธิกับ User ที่เราสร้างขึ้นมาแล้วได้ด้วยการจัดการและบันทึกการแก้ไขไว้นั่นเอง

Insert/Update/Delete Query



INSERT TABLES



UPDATE TABLES



DELETE TABLES



PostgreSQL : INSERT Query

INSERT — create new rows in a table

รูปแบบ

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
```

```
INSERT INTO table_name [ AS alias ] [ ( column_name [, ...] ) ]
```

```
    [ OVERRIDING { SYSTEM | USER } VALUE ]
```

```
    { DEFAULT VALUES | VALUES ( { expression | DEFAULT } [, ...] ) [, ...] | query }
```

```
    [ ON CONFLICT [ conflict_target ] conflict_action ]
```

```
    [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]
```

where conflict_target can be one of:

```
    ( { index_column_name | ( index_name ) column_name }
```

ตัวอย่าง การ insert ตามค่า default ของ table ดังนี้ INSERT INTO films DEFAULT VALUES;

ตัวอย่าง : การ insert ไปยัง table กับ column ที่ required และต้องระบุตำแหน่ง value ที่จะเพิ่มไปใน column ตามลำดับให้ถูกต้อง

```
INSERT INTO films VALUES
```

```
    ('UA502', 'Bananas', 105, '1971-07-13', 'Comedy', '82 minutes');
```

ตัวอย่าง การ insert ไปยัง table กับบาง column ที่ required และไม่จำเป็นต้องเรียงตาม column ใน table ข้อมูลจะเพิ่มให้ตาม mapping ของ query

```
INSERT INTO films (code, title, did, date_prod, kind)
```

```
    VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drama');
```

ตัวอย่าง การ insert multiple rows

```
INSERT INTO films (code, title, did, date_prod, kind) VALUES ('B6717', 'Tampopo', 110, '1985-02-10', 'Comedy'), ('HG120', 'The Dinner Game', 140, DEFAULT, 'Comedy');
```



PostgreSQL : MERGE

MERGE — conditionally insert, update, or delete rows of a table

รูปแบบ [WITH with_query [, ...]]

MERGE INTO target_table_name [[AS] target_alias]

USING data_source ON join_condition

when_clause [...]

where data_source is:

{ source_table_name | (source_query) } [[AS] source_alias]

and when_clause is:

{ WHEN MATCHED [AND condition] THEN { merge_update | merge_delete | DO NOTHING } |

WHEN NOT MATCHED [AND condition] THEN { merge_insert | DO NOTHING } }

and merge_insert is:

INSERT [(column_name [, ...])]

[OVERRIDING { SYSTEM | USER } VALUE]

{ VALUES ({ expression | DEFAULT } [, ...]) | DEFAULT VALUES }

and merge_update is:

UPDATE SET { column_name = { expression | DEFAULT } |

(column_name [, ...]) = ({ expression | DEFAULT } [, ...]) } [, ...]

and merge_delete is:

DELETE

Example

MERGE INTO customer_account ca

USING (SELECT customer_id, transaction_value FROM
recent_transactions) AS t

ON t.customer_id = ca.customer_id

WHEN MATCHED THEN

UPDATE SET balance = balance + transaction_value

WHEN NOT MATCHED THEN

INSERT (customer_id, balance)

VALUES (t.customer_id, t.transaction_value);

หรือ

MERGE INTO wines w

USING wine_stock_changes s

ON s.wine_name = w.wine_name

WHEN NOT MATCHED AND s.stock_delta > 0 THEN

INSERT VALUES(s.wine_name, s.stock_delta)

WHEN MATCHED AND w.stock + s.stock_delta > 0 THEN

UPDATE SET stock = w.stock + s.stock_delta

WHEN MATCHED THEN

DELETE;



PostgreSQL : INSERT Query (insert into select)

รูปแบบ

```
INSERT INTO table(column1, column2, ...)  
VALUES (value1, value2, ....)  
SELECT  
[columns]  
FROM  
[table]  
WHERE  
[condition];
```

ตัวอย่าง

```
INSERT INTO products (product_no, name, price)  
SELECT product_no, name, price FROM new_products  
WHERE release_date = 'today';
```

หรือ

```
insert into items_ver  
select * from items where item_id=2;
```

Noted : คำสั่ง INSERT INTO SELECT จะคัดลอกข้อมูลจากตารางหนึ่งและแทรกลงในอีกตารางหนึ่ง คำสั่ง INSERT INTO SELECT กำหนดให้ชนิดข้อมูลในตารางต้นทางและตารางเป้าหมายต้องตรงกัน เหตุผล: records ที่มีอยู่ในตารางเป้าหมายอยู่แล้วจะไม่ได้รับผลกระทบ



PostgreSQL : INSERT Query (select into)

เป็นรูปแบบการสร้างตารางใหม่และเติมข้อมูลที่คำนวณมาจากผลของคำสั่ง query โดยที่ตารางใหม่มีชื่อและประเภทข้อมูลที่เชื่อมโยงกับคอลัมน์เอาต์พุตของ SELECT command

รูปแบบ `[WITH [RECURSIVE] with_query [, ...]]`

`SELECT [ALL | DISTINCT [ON (expression [, ...])]]`

`* | expression [[AS] output_name] [, ...]`

`INTO [TEMPORARY | TEMP | UNLOGGED] [TABLE] new_table`

`[FROM from_item [, ...]]`

`[WHERE condition]`

`[GROUP BY expression [, ...]]`

`[HAVING condition]`

`[WINDOW window_name AS (window_definition) [, ...]]`

`[`

ตัวอย่าง

```
SELECT * INTO films_recent FROM films WHERE date_prod >= '2022-01-01';
```

Noted : คำสั่ง SELECT INTO จะสร้าง ตารางใหม่และแทรกแถวจากแบบสอบถามเข้าไป หากต้องการคัดลอกข้อมูลบางส่วนจากตารางต้นฉบับ ให้ใช้ คำสั่ง WHERE เพื่อระบุแถวที่จะคัดลอก



PostgreSQL : UPDATE Query

UPDATE — update rows of a table

รูปแบบ `[WITH [RECURSIVE] with_query [, ...]]`

`UPDATE [ONLY] table_name [*] [[AS] alias]`

`SET { column_name = { expression | DEFAULT } |`

`(column_name [, ...]) = [ROW] ({ expression | DEFAULT } [, ...]) |`

`(column_name [, ...]) = (sub-SELECT)`

`} [, ...]`

`[FROM from_item [, ...]]`

`[WHERE condition | WHERE CURRENT OF cursor_name]`

`[RETURNING`

ตัวอย่าง

```
UPDATE weather SET temp_lo = temp_lo+1, temp_hi = temp_lo+15, prcp = DEFAULT
WHERE city = 'San Francisco' AND date = '2003-07-03';
```

หรือ

```
UPDATE weather SET temp_lo = temp_lo+1, temp_hi = temp_lo+15, prcp = DEFAULT
WHERE city = 'San Francisco' AND date = '2003-07-03'
RETURNING temp_lo, temp_hi, prcp;
```

```
UPDATE summary s SET (sum_x, sum_y, avg_x, avg_y) =
(SELECT sum(x), sum(y), avg(x), avg(y) FROM data d
WHERE d.group_id = s.group_id);
```



PostgreSQL : DELETE Query

DELETE — delete rows of a table

รูปแบบ `[WITH [RECURSIVE] with_query [, ...]]`

`DELETE FROM [ONLY] table_name [*] [[AS] alias]`

`[USING from_item [, ...]]`

`[WHERE condition | WHERE CURRENT OF cursor_name]`

`[RETURNING * | output_expression [[AS] output_name] [, ...]]`

ตัวอย่าง

`DELETE FROM films;`

หรือใช้ USING relationship table

`DELETE FROM films USING producers WHERE producer_id = producers.id AND producers.name = 'foo';`

หรือใช้ เงื่อนไขใน sub-query

`DELETE FROM films WHERE producer_id IN (SELECT id FROM producers WHERE name = 'foo');`

หรือจะใช้ระบุ cursor ที่ชี้อยู่ ณ. ปัจจุบัน (Delete the row of tasks on which the cursor c_tasks is currently positioned)

`DELETE FROM tasks WHERE CURRENT OF c_tasks;`

TRUNCATE — empty a table or set of tables และนอกจากนี้ยังมีคำสั่งในการ clear data ใน table ออกทั้งหมดด้วยการใช้ TRUNCATE TABLE command

รูปแบบ `TRUNCATE [TABLE] [ONLY] name [*] [, ...]`

`[RESTART IDENTITY | CONTINUE IDENTITY] [CASCADE | RESTRICT]`

ตัวอย่าง `TRUNCATE othertable CASCADE;`

หรือ `TRUNCATE bigtable, fattable RESTART IDENTITY;`



Operators, Expressions



OPERATORS
& OPERATOR PRECEDENCE



VALUE EXPRESSIONS



CALLING FUNCTIONS



PostgreSQL : Using NoSQL in Postgres

ข้อมูล data ใน Postgres Tables สำหรับ column ข้อมูลของ data type ที่ support การใช้งาน NoSQL data มีให้ทั้ง json, jsonb, jsonpath

วิธีการนำ (import) data ไปยัง column ประเภท json (JavaScript Object Notation) หรือ jsonb (json Binary สามารถทำ index ด้วย GIN indexing)

หากแค่นำเข้าบันทึกควรใช้ JSON นำเข้าได้เร็วกว่าเมื่อเทียบกับ JSONB อย่างไรก็ตาม หากทำการประมวลผลเพิ่มเติม JSONB จะเร็วกว่า และกรณีต้องการเพิ่ม index ที่ใช้ค้นหาข้อมูล JSONB เป็นตัวเลือกที่ดีกว่า นอกจากนี้ JSONB ส่งผลให้พื้นที่จัดเก็บมีขนาดใหญ่ขึ้นไปด้วย เพิ่มเติม >> [When To Avoid JSONB In A PostgreSQL Schema](https://www.konbert.com/blog/import-json-into-postgres-using-copy)

1 Import JSON into Postgres using COPY command

คำสั่ง COPY สามารถใช้เพื่อคัดลอกเนื้อหาจากตารางหนึ่งไปยังอีกตารางหนึ่ง หรือเพื่อคัดลอกข้อมูลจากไฟล์ลงในตาราง มันเป็นเครื่องมือที่ทรงพลังมากและทำงานได้เร็วมากเช่นกัน! ทำให้เราสามารถนำเข้าไฟล์ขนาดหลาย GB ได้ภายในไม่กี่วินาทีหากทำอย่างถูกต้อง วิธีการเพิ่มเติม >> <https://konbert.com/blog/import-json-into-postgres-using-copy>

2 insert script ตัวอย่างเช่น >> INSERT INTO orders (info) VALUES('{ "customer": "John Doe", "items": {"product": "Beer","qty": 6}}');

Noted: For more : [JSON](#)

3 Using CTE และ ใช้ function row_to_json() มาช่วยเพื่อสร้าง custom table / row and data ที่เราทำขึ้นใน select query statement

ยกตัวอย่างเช่น :

```
WITH data(col1,col2,cola,colb) AS (
    VALUES (1,2,'fred','bob')
)
SELECT row_to_json(data) FROM data;
```

หรือ

```
SELECT row_to_json(data)
FROM (VALUES (1,2,'fred','bob')) data(col1,col2,cola,colb);
```

3.1 Using jsonb_agg(expression) : ทำหน้าที่รวบรวมค่าอินพุตทั้งหมด รวมถึงค่า Null เป็นอาร์เรย์ JSON ค่าจะถูกแปลงเป็น JSON ตาม to_json หรือ to_jsonb



PostgreSQL : Using NoSQL in Postgres (cont.)

4 Import by Integration tools or 3rd party Import data tools

ยกตัวอย่างเช่นการใช้ Talend (data integration tool) - Extract Json data to database (Postgres)

และเมื่อเราทำการ export data จาก table ที่มีโครงสร้างผสมระหว่าง Jsonb, json และ data types อื่นๆ ออกไปยัง csv file สิ่งที่ต้องตระหนักเมื่อเวลาเรานำข้อมูลนั้น import ด้วย csv file เดิมกลับเข้ามาจะไม่สามารถทำได้ ด้วย copy command หรือแม้แต่ทำใน pgAdmin Import menu และเจอ “ERROR: invalid input syntax for type json” วิธีการแก้ไขเมื่อต้องการนำ csv file import data กลับเข้ามาต้องทำการดัดแปลงข้อมูลก่อนนำเข้าสักเล็กน้อย ด้วยวิธีการที่เรียกว่า “Escape the inner quotes” ดังต่อไปนี้

ยกตัวอย่างเช่น

เมื่อเปิดดู data ใน csv file ที่ export ออกมาจะมีหน้าตาแบบนี้

```
# COPY baz TO STDOUT DELIMITER ',' CSV;
```

```
1,"{""a"": ""b""}"
```

และเมื่อต้องการนำ import กลับไปใส่ใน table ที่มี json, jsonb data types ต้องแปลงเป็น

```
#COPY baz TO STDOUT DELIMITER ',' CSV ESCAPE E'\\';
```

```
1,"{\"a\": \"b\"}"
```

Noted : มันคือการแทนที่ "" ด้วย \“ นั่นเอง

แต่วิธีที่ดีที่สุดคือ ทำ export เป็น json file type แล้วทำการ import เข้าจะดีกว่าใช้ csv file type



PostgreSQL : Export PostgreSQL data to a JSON file

การแปลงแถวข้อมูลให้เป็น JSON ด้วย row_to_json() ด้วยการ export select query ไปยัง json file รูปแบบ : row_to_json(record [, pretty_bool]) --โดยที่ option pretty

BOOLEAN คือ จะเพิ่มการขึ้นบรรทัดใหม่ระหว่าง top-level elements เพื่อจัดรูปแบบสวยงามตาม json ค่า defaultของมันเป็น True อยู่แล้ว

หรือ SELECT row_to_json(<table_name>) FROM <table_name> WHERE ...

ตัวอย่าง : SELECT row_to_json(row(1,'foo')); -- ผลลัพธ์ที่ได้ {"f1":1,"f2":"foo"}

หรือ SELECT row_to_json(province) from province;

สำหรับการ export ไปแค่บาง column ให้ใช้คำสั่งดังนี้ : SELECT row_to_json(row(field1, field2, field...)) FROM <table_name> WHERE ...

ตัวอย่าง : SEELCT row_to_json(row(p_id, name)) from province;

หรือใช้ CTE มาช่วยจัด column name ให้ดูเหมือนชื่อจากใน table ดังนี้ :

```
WITH cte as (  
    select p_id, name from province  
)
```

```
SELECT row_to_json(cte) from cte;
```

ส่วนการ export ออกไปยัง file ทำด้วย copy command ดังนี้ : copy(<statement>) to '<path>/<json_file>' ตัวอย่างเช่น :

```
copy (  
    WITH cte as (  
        select p_id, name from province where region = 'S'  
    )  
    SELECT row_to_json(cte) from cte  
) to 'c:/tmp/south.json';
```

Note: นอกจากนี้ยังมี function ที่ Postgres มีให้ใช้งานสำหรับข้อมูล json ที่ต้องการอีกจำนวนหนึ่ง เช่น array_to_json(), json_build_array(), jsonb_build_object(), to_jsonb() เป็นต้น



PostgreSQL : Table column store text, array and json data

การเก็บ string หรือ ค่า data object value ลงไปใน text , array และ json data type ตัวอย่างเช่น

```
create table staff( id int primary key, name varchar(50), skills text, skills_a text[], skills_j jsonb );
```

```
insert into staff(id, name, skills) values
```

```
(1, 'Peter', 'C|Java|SQL'),
```

```
(2, 'Jane', 'C++|Python'),
```

```
(3, 'Ann', 'Word|Excel|PowerPoint'),
```

```
(4, 'Bruce', 'Swift|Objective-C|Word|Excel|PowerPoint');
```

```
update staff set skills_a = regexp_split_to_array(skills, '|');
```

```
update staff set skills_j = array_to_json(regexp_split_to_array(skills, '|'));
```

```
/*การเช็ค query result จาก array data type*/
```

```
select * from staff where 'Python' = any(skills_a);
```

```
/*การเช็ค query result จาก json data type*/
```

```
select * from staff where skills_j ? 'Python';
```

```
/*การเช็ค query result จาก text string data type*/
```

```
select * from staff where skills like '%C%';
```

```
/*การแสดงค่าจาก json data ไปยัง query result*/
```

```
select name, skills_a, skills_a[1], skills_j->>0, skills_j->>0 from staff;
```

Noted: .ใน Postgres Array จะเริ่มนับ Index position จาก 1 ส่วน Json จะเริ่มนับ Index position จาก 0 ไป



PostgreSQL : JSON Functions and Operators

json and jsonb Operators

Operator	Right Operand Type	Description	Example	Example Result
->	int	Get JSON array element (indexed from zero, negative integers count from the end)	'[{"a":"foo"}, {"b":"bar"}, {"c":"baz"}]::json->2	{"c":"baz"}
->	text	Get JSON object field by key	'{"a": {"b":"foo"}}::json->'a'	{"b":"foo"}
->>	int	Get JSON array element as text	'[1,2,3]::json->>2	3
->>	text	Get JSON object field as text	'{"a":1,"b":2}::json->>'b'	2
#>	text[]	Get JSON object at specified path	'{"a": {"b":{"c": "foo"}}}'::json#>{'a,b'}	{"c": "foo"}
#>>	text[]	Get JSON object at specified path as text	'{"a":[1,2,3],"b":[4,5,6]}::json#>>{'a,2}'	3

Noted: สำหรับการใช้งานเพิ่มเติมแนะนำว่าควร refer to online manual [JSON Functions and Operators](#)
และส่วนเพิ่มเติม [JSON Support Functions](#)



PostgreSQL : JSON Functions and Operators (cont.)

Additional jsonb Operators

Operator	Right Operand Type	Description	Example
@>	jsonb	Does the left JSON value contain the right JSON path/value entries at the top level?	'{"a":1, "b":2} '::jsonb @> '{"b":2} '::jsonb
<@	jsonb	Are the left JSON path/value entries contained at the top level within the right JSON value?	'{"b":2} '::jsonb <@ '{"a":1, "b":2} '::jsonb
?	text	Does the string exist as a top-level key within the JSON value?	'{"a":1, "b":2} '::jsonb ? 'b'
?	text[]	Do any of these array strings exist as top-level keys?	'{"a":1, "b":2, "c":3} '::jsonb ? array['b', 'c']
?&	text[]	Do all of these array strings exist as top-level keys?	'{"a", "b"} '::jsonb ?& array['a', 'b']
	jsonb	Concatenate two jsonb values into a new jsonb value	'{"a", "b"} '::jsonb ['c', 'd'] '::jsonb
-	text	Delete key/value pair or string element from left operand. Key/value pairs are matched based on their key value.	'{"a": "b"} '::jsonb - 'a'
-	text[]	Delete multiple key/value pairs or string elements from left operand. Key/value pairs are matched based on their key value.	'{"a": "b", "c": "d"} '::jsonb - '{a,c} '::text[]
-	integer	Delete the array element with specified index (Negative integers count from the end). Throws an error if top level container is not an array.	'{"a", "b"} '::jsonb - 1
#-	text[]	Delete the field or element with specified path (for JSON arrays, negative integers count from the end)	'{"a", {"b":1}} '::jsonb #- '{1,b}'
@?	jsonpath	Does JSON path return any item for the specified JSON value?	'{"a":[1,2,3,4,5]} '::jsonb @? '\$a[*] ? (@ > 2)'
@@	jsonpath	Returns the result of JSON path predicate check for the specified JSON value. Only the first item of the result is taken into account. If the result is not Boolean, then null is returned.	'{"a":[1,2,3,4,5]} '::jsonb @@ '\$a[*] > 2'



PostgreSQL : UNION, INTERSECT and EXCEPT Operator

การใช้ Operator ทั้ง UNION, INTERSECT และ EXCEPT กับ postgresql statements โดย UNION ทำหน้าที่รวมชุดผลลัพธ์ของ query หลายๆ ชุดยังช่วยกำจัดแถวที่ซ้ำกันออกจากผลลัพธ์ด้วยวิธีเดียวกับ DISTINCT (หากต้องการเก็บแถวที่ซ้ำกัน โดยไม่คำนึงค่า NULL ใน table ให้ใช้ UNION ALL)

ส่วน INTERSECT ส่งคืนจุดตัดของ query 2 ชุดขึ้นไป ชุดข้อมูลแต่ละชุดถูกกำหนดโดยคำสั่ง SELECT ต้องมีอยู่ร่วมกันทั้งหมดจึงนำมาแสดง (จะต่างจาก INTERSECT ALL)

และส่วน EXCEPT ส่งคืนแถวทั้งหมดที่อยู่ในผลลัพธ์ของ query ซ้ายแต่ไม่ได้อยู่ในผลลัพธ์ของ query อื่น (ความแตกต่างระหว่าง queries) ข้อมูลซ้ำถูกตัดออก (ต่างจาก EXCEPT ALL)

รูปแบบ :

query1 UNION [ALL] query2	--เป็นการ UNION
query1 INTERSECT [ALL] query2	--เป็นการ INTERSECT
query1 EXCEPT [ALL] query2	--เป็นการ EXCEPT

ตัวอย่าง UNION เช่น :

```
SELECT a FROM b UNION SELECT x FROM y LIMIT 10
```

ส่วน UNION ALL เช่น :

```
SELECT * FROM top_rated_films UNION ALL SELECT * FROM most_popular_films ORDER BY title;
```

ตัวอย่าง INTERSECT เช่น :

```
SELECT a FROM b INTERSECT SELECT x FROM y LIMIT 10
```

ส่วน INTERSECT ALL เช่น :

```
SELECT * FROM most_popular_films INTERSECT ALL SELECT * FROM top_rated_films;
```

ตัวอย่าง EXCEPT เช่น :

```
SELECT a FROM b EXCEPT SELECT x FROM y LIMIT 10
```

ส่วน EXCEPT ALL เช่น :

```
SELECT * FROM top_rated_films EXCEPT ALL SELECT * FROM most_popular_films ORDER BY title;
```



PostgreSQL : Subquery Expressions

ใน PostgreSQL ใช้นิพจน์แบบสอบถามย่อยที่สอดคล้องกับ SQL ที่มีอยู่ใน PostgreSQL แบบฟอร์มนิพจน์ทั้งหมดในส่วนนี้จะส่งคืนผลลัพธ์ Boolean (true/false) ซึ่งประกอบด้วย:

EXISTS : รูปแบบ EXISTS (subquery) ตัวอย่างเช่น : SELECT col1 FROM tab1 WHERE EXISTS (SELECT * FROM tab2 WHERE col2 = tab1.col2);

หากได้ผลลัพธ์จากการส่งคืนอย่างน้อยหนึ่งแถวของ EXISTS จะให้ค่าเป็น True หากข้อความค้นหาย่อยไม่ส่งคืนแถว (no row) ค่าจะเป็น False เราสามารถใช้หลักการเขียนทดสอบโดยใช้รูปแบบ EXISTS(SELECT 1 WHERE ...) นำมาช่วยลดเวลาในการจัดค้น subquery ลงได้ด้วย (ควรหลีกเลี่ยงในการใช้กับ INTERSECT ที่มีใน subquery)

IN : รูปแบบ expression IN (subquery) หรือ row_constructor IN (subquery)

ส่วน subquery ทางด้านขวาต้องส่งค่าคืนอย่างน้อย 1 แถวที่เท่ากันจึงได้ค่า True และคืนค่า False เมื่อผลของ subquery ไม่เท่ากับ ด้านซ้ายและรวมถึง No row ก็ได้ค่า False ด้วย แต่ถ้าผลลัพธ์ทางซ้ายให้ค่า NULL หรือถ้าค่าทางขวามีให้ค่าอย่างน้อย 1 ค่าเป็น NULL แล้วจะได้ค่าเป็น NULL

NOT IN : รูปแบบ expression NOT IN (subquery) หรือ row_constructor NOT IN (subquery)

ค่าที่ได้จะเป็น True ก็ต่อเมื่อพบเฉพาะแถวข้อความ subquery ที่ไม่เท่ากันหรือ No row ก็ได้ด้วยและหากพบผลลัพธ์ของแถวที่เท่ากันจะให้ค่า False แต่หากนิพจน์ทางซ้ายมีให้ค่า NULL หรือถ้ามีค่าทางขวามีไม่เท่ากัน และแถวขวามีอย่างน้อยหนึ่งค่าให้ค่า NULL ค่าที่ได้ออกมาจะเป็น Null

ANY/SOME : รูปแบบ expression operator ANY (subquery) และ row_constructor operator ANY (subquery) หรือ expression operator SOME (subquery) และ row_constructor operator SOME (subquery) โดย ANY มีค่าเท่ากับ IN แต่ ANY มีความหลากหลายมากกว่า เนื่องจากสามารถใช้ร่วมกับตัวดำเนินการต่างๆโดยนำหน้าด้วยตัวดำเนินการเปรียบเทียบอย่างใดอย่างหนึ่งต่อไปนี้ =, <=, >, <, > และ <> หาก subquery ตรงกันจะส่งคืนค่า True หากไม่ตรงจะส่งคืนค่า False (ตัว SOME ใช้ได้เหมือน ANY)

ALL : รูปแบบ expression operator ALL (subquery) หรือ row_constructor operator ALL (subquery) โดย <> ALL มีค่าเท่ากับ NOT IN

จะเป็น True ถ้าทุกแถวให้ผลลัพธ์เป็น True รวมทั้ง No row ด้วยแต่จะให้ค่า False ถ้าผลลัพธ์เป็น False ส่วนหากไม่มีการเปรียบเทียบกับแถว subquery ที่ให้ค่า False อย่างน้อย 1 รายการจะให้ค่า NULL

Single-Row Comparison : รูปแบบ row_constructor operator (subquery)

ด้านขวาหรือ subquery ต้องส่งกลับจำนวนคอลัมน์เท่าที่มีนิพจน์ในแถวด้านซ้ายมือและตัว subquery ต้องส่งผลลัพธ์ได้แค่ 1 row และต้องเท่ากันเท่านั้น หากส่งคืน No row จะคืนค่าเป็น Null



PostgreSQL : String Functions and Operators

ในการใช้งานตัวแปรประเภทข้อมูลแบบ String Text หรือ Varchar ตัวอักขระใดๆ เราจำเป็นต้องใช้งาน Function และ Operator หลายๆ ตัวในชุดคำสั่ง query เพื่อลดเวลาและเพิ่มประสิทธิภาพการใช้งานได้สะดวก ทั้งยังสามารถเรียกใช้ Operator ได้เหมาะสม อีกทั้งยังสามารถประยุกต์หลายๆ Function ได้ด้วย ยกตัวอย่างดังนี้

การทำ substr() เป็นการเลือกช่วงข้อมูลออกมายัง output หรือ ไปใช้งานต่อ กรณีใช้งานกับ text string

รูปแบบ : substring (string text [FROM start integer] [FOR count integer])

ตัวอย่าง : substring('Thomas' from 2 for 3) --ผลลัพธ์→ hom
 substring('Thomas' from 3) --ผลลัพธ์→ omas
 substring('Thomas' for 2) --ผลลัพธ์→ Th

Noted : ส่วนเพิ่มเติมกรณี [Binary String Functions and Operators](#) และ กรณี [Bit String Functions and Operators](#)

การทำ concat() หรือ concatenation คือการเชื่อมข้อความเข้าด้วยกัน 2 หรือมากกว่า 2 ก็ได้ โดยจะใช้เป็นการเชื่อมข้อความกับค่าคงที่ต่างๆ ได้ด้วยเช่นกัน

รูปแบบ : CONCAT(str_1, str_2, ...number, NULL ...)

หรือ <string1> || <string2> || <string3> -- concatenation with operator ||

ในส่วนของ select query นิยมนำมาทำ concatenate column อย่างเช่น select *,concat(city,state) as city_state from zipcodes

กรณีที่เรานำไปเชื่อมกับค่า NULL อย่างเช่น : SELECT 'Concat with ' || NULL AS result_string; --ผลลัพธ์ที่ได้ออกมาจะเป็น NULL value. เว้นแต่เราใช้ concat() function ค่าที่ได้จะไม่คืนค่า NULL แต่จะ ignores NULL เป็นค่าว่างไป

รวมทั้ง Operator ต่างๆ ยกตัวอย่างเช่น

การเปรียบเทียบ text รูปแบบ text ^@ text ตัวอย่างเช่น : 'alphabet' ^@ 'alph' --จะคืนค่ามาเป็น Boolean if the first string starts with

การตรวจสอบ quote และเติม quote รูปแบบ quote_nullable (text) หรือ quote_nullable (anyelement) ตัวอย่างเช่น : quote_nullable(42.5) --ผลลัพธ์ '42.5'
 quote_nullable(NULL) --ผลลัพธ์จะคืนค่า NULL



PostgreSQL : String Functions and Operators (cont.)

การทำ OVERLAY() เพื่อจะช่วย replace ส่วนของ String ใช้ในส่วน of select statement

รูปแบบ : overlay(string PLACING replacement FROM start [FOR count])

ตัวอย่างการใช้งาน : SELECT overlay('Hello Tim' PLACING 'Hi' FROM 1 FOR 5)

ผลลัพธ์ที่ได้ :
overlay

Hi Tim

การทำ repeat() เพื่อใช้ในการทำซ้ำ string character ที่ต้องการตามจำนวนที่เราระบุเอาไว้ ในส่วนของ select statement

รูปแบบ : repeat(string text, number int)

ตัวอย่างการใช้งาน : SEELCT repeat('Pg', 4);

ผลลัพธ์ที่ได้ : PgPgPgPg

การใช้ replace() เพื่อใช้แทนที่ค่าในชุดของ string ด้วย string ย่อยจากข้อมูลที่เรากำหนดความต้องการใน parameter ของมัน

รูปแบบ : replace(string text, from text, to text)

ตัวอย่างการใช้งาน : SELECT replace('abcdefabcdef', 'cd', 'XX');

ผลลัพธ์ที่ได้ : abXXefabXXef

สามารถเอามาดัดแปลงเพื่อให้แสดงผลจากตารางโดยทำเป็น OVERLAY() function ซ้อนกัน เพื่อนำไปประยุกต์ใช้งานกับการแสดงผล บัตร ATM เพื่อซ่อนเลขบัตรลูกค้าได้ด้วย เช่น

```
select credit_card, overlay(credit_card placing repeat('*', 5) from 3 for 5),  
       overlay(overlay(credit_card placing repeat('*', 5) from 3 for 5) placing repeat('-',3) from 12 for 3)  
from customer;
```

Noted : นอกจากนี้อแล้วยังมี string function ที่เราใช้งานกันบ่อยๆและมีความสะดวกในการนำมาช่วยประยุกต์ใช้งานร่วมกันกับชุด select statement อีกมากมาย เช่น substring(), concat(), format(), initcap(), length(), strops(), convert(), char_length(), trim(), lower(), upper(), chr(), decode(), encode(), ltrim(), rtrim(), to_hex(), split_part() left(), right()

เป็นต้น

```
1 select credit_card,  
2 overlay(credit_card placing repeat('*', 5) from 3 for 5),  
3 overlay(overlay(credit_card placing repeat('*', 5) from 3 for 5) placing repeat('-', 3) from 12 for 3)  
4 from customer
```

(10 rows affected)
Total execution time: 00:00:00.006

	credit_card	overlay	overlay
1	3584706095398384	35*****095398384	35*****0953---84
2	6706574872498320	67*****072498320	67*****0724---20
3	5315907330526398	53*****330526398	53*****3305---98
4	5602215575267365	56*****575267365	56*****5752---65
5	3587441607904084	35*****607904084	35*****6079---84
6	5838736841040421	58*****841040421	58*****8410---21
7	3581310782654267	35*****782654267	35*****7826---67
8	3540550384985318	35*****384985318	35*****3849---18
9	3537927281544536	35*****281544536	35*****2815---36
10	3742805109458261	37*****109458261	37*****1094---61



PostgreSQL : String Functions and Operators (cont.)

การทำ SPLIT_PART() เพื่อใช้ในการแยกข้อความออกจากกันเป็นส่วนๆ เช่น แยกชื่อและนามสกุลออกจากกัน หรือ แยก email user

รูปแบบ : `split_part(string text, delimiter text, field int)`

ตัวอย่างการใช้งาน : `SELECT split_part('abc~@~def~@~ghi', '~@~', 2);` --ผลลัพธ์ที่ได้คือ def

หรือ `SELECT split_part(email, '@', 1) emailname, split_part(email, '@', 2) host from freelancer;`

การทำ regexp_replace() เพื่อนำมาใช้ค้นหาข้อความแล้วแทนที่ข้อความตาม pattern ที่กำหนด

รูปแบบ : `regexp_replace(string text, pattern text, replacement text [, flags text])`

ตัวอย่างการใช้งาน : `SELECT regexp_replace('Thomas', '[mN]a.', 'M');` -- ผลลัพธ์ที่ได้คือ ThM

การทำ regexp_split_to_array() เพื่อนำมาใช้ แบ่งข้อความที่มีตัวคั่นเป็น array

รูปแบบ : `regexp_split_to_array(string text, pattern text [, flags text])`

ตัวอย่างการใช้งาน : `SELECT regexp_split_to_array('hello world', 'E'\s+');` -- ผลลัพธ์ที่ได้คือ {hello,world}

การทำ regexp_split_to_table() เพื่อนำมาใช้ แบ่งข้อความที่มีตัวคั่นออกมาเป็นแถวลงไปยัง table (มีความคล้ายกับการทำ UNNEST() function ใน Array)

รูปแบบ : `regexp_split_to_table(string text, pattern text [, flags text])`

ตัวอย่างการใช้งาน : `SELECT regexp_split_to_table('hello world', 'E'\s+');` -- ผลลัพธ์ที่ได้คือ hello

world

หรืออีกตัวอย่าง : `SELECT regexp_split_to_table('AbcdefghabCDefGh', 'cd.', 'i');` -- ซึ่ง flags i คือ insensitive-case

ผลลัพธ์ที่ได้ `regexp_split_to_table`

Ab

fghab

fGh

Noted: ทดสอบนำ `SELECT regexp_split_to_table('ab,cd', ',');` และ `SELECT regexp_split_to_table('ab cd ef gh', '\s+');` ผลลัพธ์ที่ได้เป็นอย่างไร



PostgreSQL : Comparison Functions and Operators (and, or, not, in, between)

เป็นคำสั่งในการตรวจสอบเงื่อนไขด้วย and, or, not, in, between ในชุดของ select query โดย operator เหล่านี้นำมาใช้ร่วมกับตัวอื่นๆได้ในแต่ละชุดคำสั่ง

AND มีความหมายและ

ตัวอย่างเช่น :

```
SELECT * FROM employees  
WHERE (city = 'Miami' AND first_name = 'Sarah')  
OR (employee_id <= 2000);
```

OR มีความหมายหรือ

ตัวอย่างเช่น :

```
SELECT employee_id, last_name, first_name FROM employees  
WHERE (last_name = 'Smith') OR (last_name = 'Anderson' AND state = 'Florida') OR (last_name = 'Ferguson' AND status = 'Active' AND state = 'California');
```

NOT มีความหมายไม่ เช่น NOT NULL

ตัวอย่างเช่น :

```
SELECT * FROM suppliers WHERE supplier_name NOT IN ('Apple', 'Samsung', 'RIM');
```

IN มีความหมายอยู่ใน

ตัวอย่างเช่น :

```
SELECT * FROM employees WHERE employee_id IN (300, 301, 500, 501);
```

BETWEEN ... AND (ในภาษา pgsqll หมายถึงตั้งแต่ค่าเริ่มต้นจนถึงค่าสุดท้ายในช่วงข้อมูล ไม่ได้มีความหมายว่าระหว่าง)

ตัวอย่างเช่น :

```
SELECT * FROM employees WHERE start_date BETWEEN '2022-04-01' AND '2022-04-30';
```

หรือ

```
SELECT * FROM employees WHERE employee_id NOT BETWEEN 500 AND 599;
```

Noted : ตัวอย่าง between คำสั่งแรกมีค่าเหมือนกับ

```
SELECT * FROM employees WHERE start_date >= '2022-04-01' AND start_date <= '2022-04-30';
```



PostgreSQL : Aggregate Functions

ฟังก์ชันการรวมจะคำนวณผลลัพธ์ได้จากชุดของค่าที่ป้อนเข้าไปใน select query ในภาษา pgsql มีอยู่หลายกลุ่มหลายประเภท ดังนี้

General-Purpose Aggregate Functions (สำหรับวัตถุประสงค์ทั่วไป) ยกตัวอย่างเช่น

`array_agg (anyonarray) → anyarray`

`string_agg (value text, delimiter text) → text`

`json_agg (anyelement) → json`

`count (*) → bigint`, `avg (integer) → numeric`, `sum (bigint) → numeric`, `min (see text) → same as input type`, `max (see text) → same as input type`

`bit_and (bit) → bit`

`xmlagg (xml) → xml`

Aggregate Functions for Statistics (สำหรับการรวมทางสถิติ) ยกตัวอย่างเช่น

`regr_count (Y double precision, X double precision) → bigint`

`stddev (numeric_type) → double precision for real or double precision, otherwise numeric`

Ordered-Set Aggregate Functions (สำหรับสั่ง-ตั้งค่ารวมฟังก์ชัน) ยกตัวอย่างเช่น

`mode () WITHIN GROUP (ORDER BY anyelement) → anyelement`

`percentile_disc (fractions double precision[]) WITHIN GROUP (ORDER BY anyelement) → anyarray`

Hypothetical-Set Aggregate Functions (สำหรับตั้งค่าสมมุติฐาน) ยกตัวอย่างเช่น

`rank (args) WITHIN GROUP (ORDER BY sorted_args) → bigint`

`dense_rank (args) WITHIN GROUP (ORDER BY sorted_args) → bigint`

Grouping Operations (สำหรับการดำเนินการจัดกลุ่ม) ยกตัวอย่างเช่น

`GROUPING (group_by_expression(s)) → integer`



PostgreSQL : Aggregate Functions COUNT, SUM, AVG, MIN, MAX

คำสั่ง COUNT เป็นการนับจำนวนผลลัพธ์ในชุดคำสั่ง select query

คำสั่ง SUM เป็นการรวมผลลัพธ์ในชุดคำสั่ง select query

คำสั่ง AVG เป็นการหาผลลัพธ์ค่าเฉลี่ยในชุดคำสั่งของ select query

คำสั่ง MIN เป็นการหาผลลัพธ์ข้อมูลที่มีค่าน้อยที่สุดในชุดคำสั่งของ select query

คำสั่ง MAX เป็นการหาผลลัพธ์ข้อมูลที่มีค่ามากที่สุดในชุดคำสั่งของ select query

โดยปกติแล้วชุด Function ที่ใช้เหล่านี้จำเป็นต้องใช้คู่กับ Group by ที่เป็นการจัดกลุ่มข้อมูล syntax เขียนต่อจาก Where Clauses ในกรณีที่เราเรียกดูข้อมูลมากกว่า 1 column หรือถ้าต้องการเรียกดูข้อมูลออกมาทั้งกลุ่ม ส่วนกรณีที่ไม่จำเป็นต้องใช้กับ Group by นั่นคือการใช้ select query field เพียง 1 column จาก table นั้นเอง

PostgreSQL : DISTINCT

คำสั่ง DISTINCT เป็นการเลือกแสดงผลลัพธ์ในชุดคำสั่ง select query ที่ซ้ำกันออกมาเพียงแค่อะอย่างละ 1 รายการ

รูปแบบ
SELECT
DISTINCT column1, column2
FROM
table_name;

Additional : [SELECT DISTINCT](#)



PostgreSQL : RANK and DENSE_RANK

เป็นคำสั่งที่ช่วยในการค้นหาลำดับที่ตามที่ต้องการ DENSE_RANK() กำหนดอันดับให้กับทุกแถวในแต่ละพาร์ติชันของชุดผลลัพธ์ แตกต่างจากฟังก์ชัน RANK() ตรงที่จะกำหนดอันดับให้กับทุกแถวภายในพาร์ติชันของชุดผลลัพธ์ สำหรับแต่ละพาร์ติชัน อันดับของแถวแรกคือ 1 และมันจะทำการเชื่อมโยงกันเพื่อคำนวณอันดับของแถวถัดไป ส่วน ฟังก์ชัน DENSE_RANK() จะส่งคืนอันดับเดียวกันสำหรับแถวที่มีค่าเท่ากันแต่ละพาร์ติชัน

Noted : สำหรับ RANK() อันดับอาจไม่เรียงตามลำดับกัน นอกจากนี้ แถวที่มีค่าเท่ากันจะได้รับอันดับเดียวกันอีกด้วย

รูปแบบ :
 RANK() OVER (
 [PARTITION BY partition_expression, ...]
 ORDER BY sort_expression [ASC | DESC], ...
)

และ
 DENSE_RANK() OVER (
 [PARTITION BY partition_expression, ...]
 ORDER BY sort_expression [ASC | DESC], ...
)

ตัวอย่างเช่น :
 select name, critic_score, rank() over (order by critic_score desc) as rank from ps4;
 select name, critic_score, dense_rank() over (order by critic_score desc) as dense_rank from ps4;
 select year_of_release, genre, name, critic_score,
 row_number() over w as row_number,
 rank() over w as rank,
 dense_rank() over w as dense_rank from ps4
 window w as (partition by year_of_release, genre order by critic_score desc);



PostgreSQL : LIMIT and OFFSET

คำสั่ง **LIMIT** จะดึงข้อมูลตามจำนวน records ที่ระบุหลังคำสั่งหลัก LIMIT เท่านั้น เว้นแต่ว่า query จะส่งคืน record น้อยกว่าจำนวนที่ระบุโดย LIMIT (การใช้ LIMIT ALL กับ LIMIT NULL ก็คือการไม่ใช้ LIMIT เป็นตัวควบคุมผลลัพธ์ที่ได้ออกมา) เมื่อใช้ LIMIT สิ่งสำคัญคือต้องใช้ ORDER BY clause ที่จำกัดแถวผลลัพธ์ให้เป็นลำดับเฉพาะ ไม่งั้นผลที่ได้จะเป็น unpredictable subset ของ query's rows (ไม่ทราบลำดับของแถวที่ได้)

คำสั่ง **OFFSET** เป็นการบอกให้ข้ามหลายแถวก่อนที่จะ return rows ให้ output ออกมา OFFSET 0 เหมือนกับ OFFSET ที่มีอาร์กิวเมนต์ NULL จะเหมือนกับเป็นการละเว้นส่วนคำสั่ง OFFSET (ค่า default เป็น ละเว้นแถวที่มาก่อน)

Noted : หากทั้ง OFFSET และ LIMIT ปรากฏขึ้น (นำมาใช้งานด้วยกัน) แถว OFFSET จะถูกข้ามไปก่อนที่จะเริ่มนับแถว LIMIT แล้วค่อยส่ง return row กลับมาที่ output การ rows skipped ด้วย OFFSET ยังคงต้องคำนวณภายในเซิร์ฟเวอร์ ดังนั้น OFFSET ขนาดใหญ่อาจไม่ได้ช่วยเรื่องประสิทธิภาพของการ query data ออกมา การใช้ LIMIT 0 สามารถใช้ในสถานการณ์ที่เราต้องการทราบว่ามียอดล้นในตาราง นั้นเอง

รูปแบบคือ

```
SELECT select_list
FROM table_expression
[ ORDER BY ... ]
[ LIMIT { number | ALL } ] [ OFFSET number ]
```

ตัวอย่างเช่น

SELECT film_id, title, release_year FROM film ORDER BY film_id LIMIT 5 OFFSET 6;

หรือ SELECT * FROM province order by area_km2 desc offset 5 rows fetch first 3 rows only;

Noted: เราสามารถใช้คำสั่งจาก limit <number> เป็น fetch first <number> rows only; ได้เหมือนกัน

```
dvdrental=# SELECT
dvdrental=# film_id,
dvdrental=# title,
dvdrental=# release_year
dvdrental=# FROM
dvdrental=# film
dvdrental=# ORDER BY
dvdrental=# film_id
dvdrental=# LIMIT 5 OFFSET 6;
 film_id |      title      | release_year
-----+-----+-----
       7 | Airplane Sierra |      2006
       8 | Airport Pollock |      2006
       9 | Alabama Devil   |      2006
      10 | Aladdin Calendar |      2006
      11 | Alamo Videotape |      2006
(5 rows)
```



PostgreSQL : Conditional Expressions

เงื่อนไข CASE : เป็นเงื่อนไขทั่วไป คล้ายกับคำสั่ง if/else ในภาษาโปรแกรมอื่นๆ

รูปแบบ CASE WHEN condition THEN result

[WHEN ...]

[ELSE result]

END

หรือ CASE expression

WHEN value THEN result

[WHEN ...]

[ELSE result]

END

ตัวอย่างเช่น : SELECT ... WHERE CASE WHEN x <> 0 THEN y/x > 1.5 ELSE false END;

Function **COALESCE** ทำหน้าที่ ค้นค่าอาร์กิวเมนต์แรกที่ไม่ใช่ค่า Null โดยทั่วไปจะใช้กับคำสั่ง SELECT เพื่อจัดการกับค่า Null อย่างมีประสิทธิภาพ สามารถรับอาร์กิวเมนต์ได้ไม่จำกัดจำนวน จะส่งกลับอาร์กิวเมนต์แรกที่ไม่เป็น not null หากอาร์กิวเมนต์ทั้งหมดเป็นค่าว่างจะคืนค่าเป็น null โดยพิจารณาจากซ้ายไปขวาจนกว่าจะพบอาร์กิวเมนต์แรกที่ไม่ใช่ค่าว่าง

รูปแบบ COALESCE(value [, ...])

ตัวอย่างเช่น : select day, coalesce(tickets, 0) from stats;

หรือ SELECT coalesce(null,null, 1, 2, 3, null, 4);



PostgreSQL : Conditional Expressions (with Range Data Types cont.)

การใช้ numrange data type มากำหนดค่าช่วงข้อมูลแบบ Approximate match เพื่อให้ Look up value ใน range และประยุกต์ใช้กับ CASE .. WHEN สามารถใช้งานดังตัวอย่างนี้ :

```
SELECT student_id, score,
CASE
    WHEN '[85,)'::numrange @> score THEN 'A'
    WHEN '[75,85)'::numrange @> score THEN 'B'
    WHEN '[65,75)'::numrange @> score THEN 'C'
    WHEN '[50,65)'::numrange @> score THEN 'D'
    WHEN '[0,50)'::numrange @> score THEN 'F'
END grade
FROM scores;
```

Noted: โดย [หมายถึง ตั้งแต่

) หมายถึง จากตำแหน่ง current ขึ้นไป

@> หมายถึง contains อยู่ใน range

ส่วนการสร้าง Table ที่ใช้เก็บค่า numrange ดังตัวอย่างนี้ :

```
CREATE TABLE grade_lookup (
    grade character varying(2) NOT NULL,
    score_range numrange
);
```



PostgreSQL : Conditional Expressions (cont.)

Function **NULLIF** ส่งกลับค่า Null ถ้าค่า 1 เท่ากับค่า 2; มิฉะนั้นจะส่งกลับค่า 1 โดยอาร์กิวเมนต์ทั้งสองที่ใช้เปรียบเทียบกับกันต้องเป็นประเภทที่เปรียบเทียบกับกันได้

รูปแบบ NULLIF(value1, value2)

ตัวอย่างเช่น : SELECT NULLIF(value, '(none)') ...

Noted: ในตัวอย่างนี้ ถ้าค่าเป็น (none) จะส่งกลับค่า null มิฉะนั้นจะส่งกลับค่า value แทนถ้าอีกอันไม่ใช่ none หรือ null

และเราสามารถใช้อย่าง NULLIF to prevent division-by-zero error ได้ด้วย

```
SELECT NULLIF (1, 1); -- return NULL
```

```
SELECT NULLIF (1, 0); -- return 1
```

```
SELECT NULLIF ('A', 'B'); -- return A
```

Function **GREATEST** and **LEAST** เป็นการ เลือกค่าที่มากที่สุดหรือน้อยที่สุดจาก list ของจำนวนทั้งหมดที่มีใน expression ส่วนค่า NULL ในรายการจะถูกละเว้น ผลลัพธ์จะเป็น NULL ก็ต่อเมื่อใน expression ทั้งหมดประเมินค่าเป็น NULL

รูปแบบ GREATEST(value [, ...])

LEAST(value [, ...])

ตัวอย่างเช่น : SELECT GREATEST(3, 15, 7); -- Result: 15

SELECT GREATEST('Bear', 'Zebra', 'Ant'); -- Result: Zebra

SELECT LEAST(25, 6, 7, 10, 20, 54); -- returns 6

SELECT LEAST('D','A', 'B', 'C','d','e','E','a'); -- returns 'a'

SELECT LEAST(6, NULL); -- result: 6

Noted : SELECT GREATEST(3, 'Fifteen', 7); --จะได้ Error: invalid input syntax for type integer: "Fifteen"

SELECT GREATEST(); -- ERROR: syntax error at or near ")"



PostgreSQL : Pattern Matching

แบ่งได้ 3 แบบใหญ่คือ

LIKE : นิพจน์ LIKE ส่งคืนค่า TRUE หาก String ตรงกับรูปแบบที่ให้มา (ตามที่คาดไว้ นิพจน์ NOT LIKE ส่งคืนค่า FALSE หาก LIKE ส่งคืนค่า TRUE

```
ตัวอย่างเช่น :      'abc' LIKE 'abc'           --true
                   'abc' LIKE 'a%'          --true
                   'abc' LIKE '_b_'         --true
                   'abc' LIKE 'c'           --false
```

SIMILAR TO Regular Expressions : ตัวดำเนินการ SIMILAR TO คืนค่า TRUE หรือ FALSE ขึ้นอยู่กับว่ารูปแบบตรงกับสตริงที่กำหนดหรือไม่ ซึ่งคล้ายกับ LIKE ยกเว้นว่าจะตีความรูปแบบโดยใช้ SQL standard's definition ของ regular expression ซึ่งอย่างที่เรารู้กันดีในภาษา SQL regular expressions จะผสมผสานกันได้หลากหลาย

```
ตัวอย่างเช่น :      'abc' SIMILAR TO 'abc'           --true
                   'abc' SIMILAR TO 'a'             --false
                   'abc' SIMILAR TO '%(b|d)%'       --true
                   'abc' SIMILAR TO '(b|c)%'        --false
                   '-abc-' SIMILAR TO '%\mabc\M%'   --true
                   ' xabcy' SIMILAR TO '%\mabc\M%' --false
```

POSIX Regular Expressions : นิพจน์ทั่วไปของ POSIX เป็นวิธีการจับคู่รูปแบบที่มีประสิทธิภาพมากกว่าตัวดำเนินการ LIKE และ SIMILAR TO เครื่องมือ Unix จำนวนมาก เช่น egrep, sed หรือ awk ใช้ภาษาการจับคู่รูปแบบที่คล้ายกับแบบนี้ รูปแบบดังต่อไปนี้

text ~ text	--Boolean	String matches regular expression, case sensitively	'thomas' ~ 't.*ma'	-- t
text ~* text	--Boolean	String matches regular expression, case insensitively	'thomas' ~* 'T.*ma'	-- t
text !~ text	--Boolean	String does not match regular expression, case sensitively	'thomas' !~ 't.*max'	--t
text !~* text	--Boolean	String does not match regular expression, case insensitively	'thomas' !~* 'T.*ma'	--f



PostgreSQL : OPERATORS

ใน Postgres มี Operator อยู่หลายแบบมากๆแต่ละแบบขึ้นอยู่กับประเภทของข้อมูล การเปรียบเทียบ การใช้งานทางคณิตศาสตร์ล้วนแล้วแต่ต้องมี ส่วนประกอบของ Operator อยู่ทั้งสิ้น เพื่อให้การทำงานของ Function ที่มีให้เรียกใช้งานรวมถึงที่เราสร้างขึ้นใหม่เองทำให้เราสามารถทำการจัดการ data ได้อย่างถูกต้องแม่นยำ ลดเวลาและพลิกแพลงประยุกต์ใช้ให้ตรงกับ ลักษณะ Output หรือ File ที่เราจะนำไปใช้งานต่อได้นั่นเอง เนื่องจากแบ่งเป็นหัวข้อหลักๆรวบรวมไว้ดังนี้

- 1. Logical Operators
- 2. Comparison Operators
- 3. Mathematical Operators
- 4. String Operators
- 5. Binary String Operators
- 6. Bit String Operators
- 7. Date/Time Operators
- 8. Geometric Operators
- 9. Network Address Operators
- 10. Text Search Operators
- 11. Array Operators

OPERATOR	DESCRIPTION
~	This is used to match Regular Expression and it is CASE Sensitive.
~*	This is also used to match Regular Expression and it is CASE Insensitive.
!~	This is used to filter the unmatched Regular Expression and it is CASE Sensitive.
!~*	This is also used to filter the unmatched Regular Expression and it is CASE Insensitive.

Noted: ส่วนที่นำไปใช้บ่อยๆจะเป็น Mathematical and Comparison operators สามารถดูเพิ่มเติมที่ [Chapter 9. Functions and Operators](#) และ [Version 15](#)



PostgreSQL : OPERATOR PRECEDENCE

แสดงลำดับความสำคัญและการเชื่อมโยงของตัวดำเนินการใน PostgreSQL และถ้ามีการเพิ่มวงเล็บใช้ในกรณีที่ต้องการให้นิพจน์ที่มี operators หลายตัวถูกแยกวิเคราะห์
Operator Precedence (highest to lowest)

Operator/Element	Associativity	Description
.	left	table/column name separator
::	left	PostgreSQL-style typecast
[]	left	array element selection
+ -	right	unary plus, unary minus
^	left	exponentiation
* / %	left	multiplication, division, modulo
+ -	left	addition, subtraction
(any other operator)	left	all other native and user-defined operators
BETWEEN IN LIKE ILIKE SIMILAR		range containment, set membership, string matching
< > = <= >= <>		comparison operators
IS ISNULL NOTNULL		IS TRUE, IS FALSE, IS NULL, IS DISTINCT FROM, etc.
NOT	right	logical negation
AND	left	logical conjunction
OR	left	logical disjunction



PostgreSQL : Regular expression

เราสามารถใช้งานผ่าน REGEXP_MATCHES Function โดยมันทำหน้าที่จับคู่นิพจน์ทั่วไปกับ String และส่งคืน String ย่อยที่ตรงกัน รูปแบบ :

```
REGEXP_MATCHES(source_string, pattern [, flags])
```

โดย pattern คือ นิพจน์ POSIX ทั่วไปสำหรับการจับคู่ และ flags คือ ตัวที่ควบคุมลักษณะการทำงานของฟังก์ชันตัวอย่างเช่น ตัว i อนุญาตให้จับคู่โดยไม่คำนึงถึงตัวพิมพ์เล็กและใหญ่

ตัวอย่างการใช้งาน :

```
SELECT REGEXP_MATCHES('Learning #PostgreSQL #REGEXP_MATCHES', '#([A-Za-z0-9_]+)', 'g');
```

ผลลัพธ์ที่ได้จะได้มาเป็น (2 rows) regexp_matches

```
-----  
{PostgreSQL}  
{REGEXP_MATCHES}
```

อีกหนึ่งรูปแบบการใช้ Operators ตัวอย่างเช่น :

```
select * from benjerry where flavor ilike '%banana%';
```

```
select * from benjerry where flavor ~* 'banana';      -- contains ~* (case insensitive)
```

```
select * from benjerry where flavor ~* '^chocolate';      -- begins with ^
```

```
select * from benjerry where flavor ~* 'fudge$';      -- ends with $
```

```
select * from benjerry where flavor ~* 'chocolate|fudge|cookie';      -- chocolate or fudge or cookie
```

```
select * from benjerry where flavor ~* 'chocolate.*fudge|fudge.*chocolate';      -- both chocolate and fudge
```

```
-- using lookahead
```

```
select * from benjerry where flavor ~* '(?=.*chocolate)(?=.*fudge)';
```

```
select * from benjerry where flavor ~* '(?=.*fudge)(?=.*chocolate)';
```

```
-- word boundary: \ycake\y not cheesecake, shortcake but only cake word
```

```
select * from benjerry where flavor ~* '\ycake\y';
```

Noted: การใส่ string quote ใน function หลายตัวจะเกิด error ขึ้นได้แนะนำให้ทำการใส่ (E' select statement='\.\') ในวงเล็บกรณีมี Where clauses ที่ = string ภายใต้ single quote



PostgreSQL : Calling Functions

นอกจาก Function ที่ Postgres มีให้แล้วเรายังสามารถสร้าง Function ขึ้นมาใช้งานเอง รูปแบบเป็น ดังนี้

CREATE [OR REPLACE] FUNCTION

name ([[argmode] [argname] argtype [{ DEFAULT | = } default_expr] [, ...])

[RETURNS rettype

| RETURNS TABLE (column_name column_type [, ...])]

{ LANGUAGE lang_name

| TRANSFORM { FOR TYPE type_name } [, ...]

| WINDOW

| { IMMUTABLE | STABLE | VOLATILE }

| [NOT] LEAKPROOF

| { CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT }

| { [EXTERNAL] SECURITY INVOKER | [EXTERNAL] SECURITY DEFINER }

| PARALLEL { UNSAFE | RESTRICTED | SAFE }

| COST execution_cost

| ROWS result_rows

| SUPPORT support_function

| SET configuration_parameter { TO value | = value | FROM CURRENT }

| AS 'definition'

| AS 'obj_file', 'link_symbol'

| sql_body

} ...

ตัวอย่าง template เช่น :

```
create [or replace] function function_name(param_list)
```

```
returns return_type
```

```
language plpgsql
```

```
as
```

```
$$
```

```
declare
```

```
-- variable declaration
```

```
begin
```

```
-- logic
```

```
end;
```

```
$$
```



PostgreSQL : Calling Functions (cont.)

วิธีการสร้างอาจใช้ตัวช่วยสร้าง script ใน pgAdmin tool หรือควรสร้าง function template file ขึ้นมาเมื่อต้องการใช้งานก็หยิบรูปแบบการสร้าง function นั้นมาประกอบเป็น code ก็ได้

ตัวอย่าง function เช่น :

```
create or replace function bmi(h real, w real) returns real language 'plpgsql' as $$
begin
    return w / (h * h);
end;
$$;
```

การเรียกใช้งาน :

```
select bmi(1.7, 70.0);
```

หรือ

```
select *, bmi(height_m, weight_kg) body_mass_index from employee;
```

หรือ

```
select emp_id, weight_kg, height_m, bmi(height_m, weight_kg) body_mass_index from employee;
```

การ Drop Function รูปแบบ :

```
DROP FUNCTION [ IF EXISTS ] name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] [, ...]
[ CASCADE | RESTRICT ]
```

ตัวอย่างเช่น

```
drop function if exists bmi; -- drop existing functions
```

```
drop function if exists rectangle;
```

```
drop function if exists square;
```

```
drop function if exists cylinder;
```

```
drop function if exists randbetween;
```

Noted: เพิ่มเติม PL/pgSQL [IF Statement](#)



PostgreSQL : Calling Functions (cont.)

Creating a function that returns value (create table-valued function) วิธีพัฒนาฟังก์ชัน PL/pgSQL ที่ส่งคืนตาราง (เหมือนเราสร้าง View Table แต่มีความ flexible กว่า) รูปแบบคือ

```
DROP FUNCTION IF EXISTS function_name;
```

```
create or replace function function_name (  
    parameter_list  
)
```

```
returns table ( column_list )
```

```
language plpgsql
```

```
as $$
```

```
declare
```

```
-- variable declaration
```

```
begin
```

```
-- body
```

```
end; $$
```

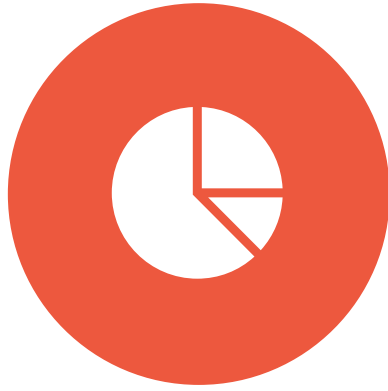
```
LANGUAGE 'plpgsql';
```

Noted: เพิ่มเติม [Control Structures](#)

```
create or replace function get_film (  
    p_pattern varchar  
)  
    returns table (  
        film_title varchar,  
        film_release_year int  
    )  
    language plpgsql  
as $$  
begin  
    return query  
        select  
            title,  
            release_year::integer  
        from  
            film  
        where  
            title ilike p_pattern;  
end; $$
```



Where Clauses and Indexes



WHERE CLAUSES



ARRAY AND
FULL TEXT SEARCH



INDEXES
CREATE, RENAME & DROP



PostgreSQL : Where Clauses and HAVING

เป็นส่วนของชุดคำสั่งที่ใช้ในการตรวจสอบเงื่อนไขของ select query

รูปแบบ (เฉพาะ where) : WHERE search_condition

ตัวอย่างเช่น : SELECT ... FROM fdt WHERE c1 BETWEEN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10) AND 100

SELECT ... FROM fdt WHERE EXISTS (SELECT c1 FROM t2 WHERE c2 > fdt.c1)

SELECT ... FROM fdt WHERE c1 IN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10)

หลังจากผ่านตัวกรอง WHERE ตารางอินพุตที่ได้รับอาจถูกจัดกลุ่ม โดยใช้ส่วนย่อย GROUP BY และกำจัดแถวกลุ่มโดยใช้ส่วนคำสั่ง HAVING เพื่กรองผลลัพธ์ออกมา

รูปแบบ : SELECT column1, aggregate_function (column2)

FROM table_name

GROUP BY column1

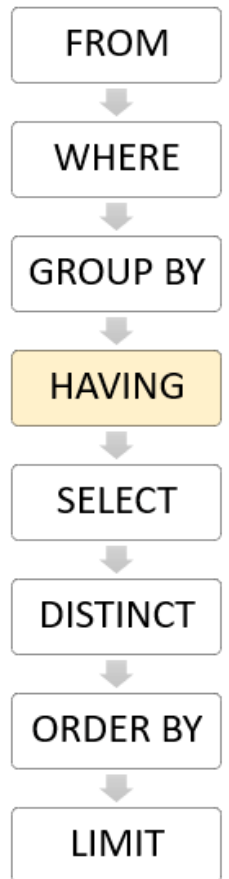
HAVING condition;

ตัวอย่างเช่น : SELECT store_id, COUNT (customer_id)

FROM customer

GROUP BY store_id

HAVING COUNT (customer_id) > 300;





PostgreSQL : ARRAY

เป็นส่วนของ Data Type ประเภทหนึ่งที่จะสามารถช่วยให้เราสามารถเก็บค่าได้หลายค่าภายใน column เดียว สำหรับการประกาศตัวแปรประเภท Array

ตัวอย่าง :

```
CREATE TABLE contacts (  
    id serial PRIMARY KEY,  
    name VARCHAR (100),  
    phones TEXT []  
);
```

รูปแบบการป้อนค่า Array ดังตัวอย่างนี้ : '{{1,2,3},{4,5,6},{7,8,9}}'

Insert PostgreSQL array values : _ INSERT INTO contacts (name, phones) VALUES('John Doe', ARRAY ['(408)-589-5846','(408)-589-5555']);

โดยนอกจากใช้ตัว ARRAY constructor แล้วยังสามารถใช้ วงเล็บปีกกา ดังนี้: INSERT INTO contacts (name, phones) VALUES('Lily Bush','{(408)-589-5841}'),
('William Gate','{(408)-589-5842}','(408)-589-58423}');

Noted : อาร์เรย์หลายมิติต้องมีขอบเขตที่ตรงกันสำหรับแต่ละมิติ ความไม่ตรงกันทำให้เกิดข้อผิดพลาด ตัวอย่างเช่น:

```
INSERT INTO sal_emp  
VALUES ('Bill', '{10000, 10000, 10000, 10000}',  
'{"meeting", "lunch"}, {"meeting"}');
```

ERROR: multidimensional arrays must have array expressions with matching dimensions

ส่วนวิธีการเรียกใช้ และเข้าถึงสมาชิกใน Array (Accessing Arrays) ดังนี้ : SELECT name FROM sal_emp WHERE pay_by_quarter[1] <> pay_by_quarter[2];

ผลลัพธ์ที่ได้ก็คือ : column Name แสดงผลทั้งหมดออกมาจาก Table โดยเปรียบเทียบค่าที่ไม่เท่ากันข้างในของ คู่ Array ใน Where Clauses นั้นเอง

มีอีก function ที่น่าสนใจการใช้ **string_to_array()** เพื่อช่วยในการ split string ออกมาด้วย character ที่อยู่ใน data ของ column เช่น ช่องว่างหรือ space เช่น

```
SELECT director, string_to_array(director, ' ') from movie1000; --ผลลัพธ์ที่ได้แสดง data type มาเป็น type เดิมอย่างนี้คือ text[] และเป็น element คั่นด้วย comma
```

Noted : ข้อควรระวังในภาษา pgsql สมาชิกของ Array position จะเริ่มนับจาก 1 ไม่ใช่ 0



PostgreSQL : ARRAY (cont.)

การสืบค้นข้อมูล Querying data in an array

เรายังสามารถเข้าถึง multidimensional arrays ใน subarrays ได้โดยเรียกใช้รูปแบบ lower-bound:upper-bound ตัวอย่างเช่น

```
SELECT schedule[1:2][1:1] FROM sal_emp WHERE name = 'Bill';
```

การใช้ ANY function ในการสืบค้นข้อมูล Array เป็นการหาคำที่คั่นอยู่ตัวตำแหน่งใดตัวหนึ่งใน Array ก็ได้ ตัวอย่างเช่น :

```
SELECT * FROM applicant WHERE 'SQL' = any(skills) and 'Python' = any(skills);
```

การใช้ @> contain function ตัวอย่างเช่น : `SELECT * FROM applicant WHERE skills @> array['SQL', 'Python']::varchar[];`

unnest ขยายหลายอาร์เรย์ออกมา ในชุดของแถว องค์ประกอบของอาร์เรย์จะถูกอ่านตามลำดับการจัดเก็บ หากอาร์เรย์มีความยาวไม่เท่ากัน ตัวที่สั้นกว่าจะถูกเสริมด้วย NULL เช่น

```
unnest(ARRAY[['foo','bar'],['baz','quux']]) → foo
                                             bar
                                             baz
                                             quux
```

ขนาดปัจจุบันของค่าอาร์เรย์ใดๆ สามารถเรียกค้นได้ด้วยฟังก์ชัน array_dims: ตัวอย่างเช่น :

```
SELECT array_dims(schedule) FROM sal_emp WHERE name = 'Carol';
```

นอกจากนี้ยังสามารถดึงข้อมูลขนาดได้ด้วย array_upper และ array_lower ซึ่งส่งคืนขอบเขตบนและล่างของมิติอาร์เรย์ที่ระบุเอาไว้ (ค่าที่ได้เป็นจำนวนตัวเลข) ตัวอย่างเช่น :

```
SELECT array_upper(schedule, 1) FROM sal_emp WHERE name = 'Carol';
```

array_length จะคืนค่าความยาวของขนาดอาร์เรย์ที่ต้องการทราบ (ค่าที่ได้เป็นจำนวนตัวเลข) ตัวอย่างเช่น :

```
SELECT array_length(schedule, 1) FROM sal_emp WHERE name = 'Carol';
```

cardinality ส่งคืนจำนวนของ elements ที่มีอยู่ทั้งหมดใน Array ในทุกมิติ (dimensions) (ค่าที่ได้เป็นจำนวนตัวเลข) ตัวอย่างเช่น :

```
SELECT cardinality(schedule) FROM sal_emp WHERE name = 'Carol';
```



PostgreSQL : ARRAY (cont.)

การ Modifying Arrays

ด้วยการ UPDATE, INSERT และ DELETE ข้อมูลจาก ARRAY ได้ด้วยรูปแบบ array function และ UPDATE & DELETE command ที่เรารู้กันเคย ตัวอย่างเช่น :

```
UPDATE sal_emp SET pay_by_quarter = '{25000,25000,27000,27000}' WHERE name = 'Carol';
```

หรือจะใช้ ARRAY expression syntax เช่น :

```
UPDATE sal_emp SET pay_by_quarter = ARRAY[25000,25000,27000,27000] WHERE name = 'Carol';
```

หรือ อัปเดตแบบ single element เช่น :

```
UPDATE sal_emp SET pay_by_quarter[4] = 15000 WHERE name = 'Bill';
```

หรือแม้แต่ อัปเดตใน slice เช่น :

```
UPDATE sal_emp SET pay_by_quarter[1:2] = '{27000,27000}' WHERE name = 'Carol';
```

หรือใช้ array_replace function ตัวอย่างเช่น :

```
UPDATE applicant SET skills = array_replace(skills, 'SQL', 'PostgreSQL') WHERE id = 1;
```

ในการ INSERT เราสามารถใช้ array_append function มาช่วยในการเติม data เข้าไปได้ ตัวอย่างเช่น :

```
UPDATE applicant SET skills = array_append(skills, 'Go') WHERE id = 1;
```

ในการ DELETE เราสามารถใช้ array_remove function มาช่วยในการลบ data ออกไปได้ ตัวอย่างเช่น

```
UPDATE applicant SET skills = array_remove(skills, 'Java') WHERE id = 1;
```

หรือใช้ DELETE command ตัวอย่างเช่น

```
DELETE from applicant WHERE 'C++' = any(skills);
```



PostgreSQL : ARRAY | ENUM enumerated type (cont.)

เป็นประเภทข้อมูลที่ประกอบด้วยชุดของค่าที่เรียงลำดับแบบคงที่ (static) ตัวอย่างของประเภท enum อาจเป็นวันในสัปดาห์ หรือชุดของค่าสถานะสำหรับชิ้นส่วนของข้อมูลในรูป string

Declaration of Enumerated Types

Enum types ถูกสร้างขึ้นโดยใช้คำสั่ง CREATE TYPE ตัวอย่างเช่น:

```
CREATE TYPE rainbow AS ENUM ('red', 'orange', 'yellow', 'green', 'blue', 'purple');
```

เมื่อสร้างแล้ว ชนิด enum สามารถใช้ในตารางและนิยามฟังก์ชันได้เหมือนกับ ข้อมูลชนิดอื่นๆ อีกตัวอย่างของการสร้าง (ประกาศ) และการเรียกใช้งาน:

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
```

```
CREATE TABLE person (
```

```
    name text,
```

```
    current_mood mood
```

```
);
```

```
INSERT INTO person VALUES ('Moe', 'happy');
```

```
SELECT * FROM person WHERE current_mood = 'happy';
```

ผลลัพธ์ที่ได้คือ :

```
name | current_mood
```

```
-----+-----
```

```
Moe | happy
```

สามารถใช้คำสั่ง order by มากำหนดผล query อย่างเช่น : `SELECT * FROM person WHERE current_mood > 'sad' ORDER BY current_mood;`

Noted : ข้อควรระวังต้องระวังการนำค่าใน enum มาใช้งานโดยต้องเป็น data สมาชิกใน enum หากไม่มีอยู่จะเกิด error นอกจากนั้น column data type แบบ enum สามารถเป็นค่า Null ได้



PostgreSQL : Full Text Search

เป็นความสามารถในการระบุเอกสารแบบ natural-language ที่ตรงกับข้อความค้นหา และเลือกที่จะจัดเรียงตามความเกี่ยวข้องกับข้อความค้นหา และส่งคืนตามลำดับความคล้ายคลึงกับคำค้นหา PostgreSQL มีตัว operators อย่าง ~, ~*, LIKE และ ILIKE สำหรับประเภทข้อมูลที่เป็นข้อความ สำหรับการค้นหาภายใน PostgreSQL โดยปกติแล้ว เอกสารจะเป็นฟิลด์ข้อความภายในแถวของตารางฐานข้อมูล หรืออาจเป็นการรวมกัน (เชื่อมข้อมูล) ของฟิลด์ดังกล่าว อาจจัดเก็บไว้ในหลายตารางหรือเก็บแบบ dynamically กล่าวอีกนัยหนึ่ง เอกสารสามารถสร้างขึ้นจากส่วนต่างๆ สำหรับการจัดทำดัชนี ความเป็นไปได้อีกอย่างคือการจัดเก็บเอกสารเป็นไฟล์ข้อความ การดำเนินการค้นหา และสามารถใช้ตัวระบุเฉพาะบางตัว (unique identifier) เพื่อดึงเอกสารจากระบบไฟล์ นอกจากนี้ PostgreSQL ยัง support การเก็บทุกอย่างไว้ในฐานข้อมูล เพื่อช่วยให้เข้าถึงข้อมูลเมตาของเอกสารได้ง่าย และช่วยในการจัดทำดัชนี และแสดงผลอีกด้วย

การค้นหาข้อความ Full text searching ใช้ตัว match operator รูปแบบ @@ ซึ่งจะคืนค่า True (หรือ t) หาก tsvector (เอกสาร) ตรงกับ tsquery (ข้อความค้นหา) ตัวอย่างเช่น :

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector @@ 'cat & rat'::tsquery;
```

หรือ

```
SELECT 'fat & cow'::tsquery @@ 'a fat cat sat on a mat and ate a fat rat'::tsvector;
```

เราอาจใช้การรวมคำหลายคำโดยใช้ AND, OR, NOT และตามด้วย operator รวมทั้งมี function ที่ใช้ support การค้นหายกตัวอย่างเช่น : to_tsquery, plainto_tsquery และ Phraseto_tsquery เป็นต้น มีฟังก์ชัน to_tsquery, plainto_tsquery และ Phraseto_tsquery ที่เป็นประโยชน์ในการแปลงข้อความ user-written text ไปเป็น tsquery ที่เหมาะสม โดยหลักแล้วจะทำให้คำที่ปรากฏในข้อความเป็น normalizing ในทำนองเดียวกัน to_tsvector ใช้เพื่อแยกวิเคราะห์และช่วยทำให้ normalize document string การจับคู่ค้นหาข้อความเป็นดังนี้ :

```
SELECT to_tsvector('fat cats ate fat rats') @@ to_tsquery('fat & rat');
```

Noted : สามารถใช้วงเล็บเพื่อควบคุมการซ้อนตัวดำเนินการ tsquery

การ Configuration :

ระหว่างการติดตั้ง จะมีการเลือกการกำหนดค่าที่เหมาะสม และตั้งค่า default_text_search_config ตามนั้นใน postgresql.conf หากใช้การกำหนดค่าการค้นหาข้อความเดียวกันสำหรับคลัสเตอร์ทั้งหมด สามารถใช้ค่าใน postgresql.conf หากต้องการใช้การกำหนดค่าที่แตกต่างกันทั่วทั้งคลัสเตอร์แต่มีการกำหนดค่าเดียวกันภายในฐานข้อมูลใดฐานข้อมูลหนึ่ง ให้ใช้ ALTER DATABASE ... SET เพื่อจะทำให้กำหนดขีดความสามารถในแต่ละเซสชันได้ โดยไปตั้งค่า default_text_search_config นั้นเอง



PostgreSQL : Indexes (CREATE, ALTER & DROP)

การสร้าง index แบบ non unique และ unique โดยเราสามารถสร้างดัชนีบนคอลัมน์ที่ระบุของความสัมพันธ์ที่ระบุ ซึ่งสามารถใช้กับตารางหรือ materialized view ได้ ดัชนีจะใช้เพื่อเพิ่มประสิทธิภาพฐานข้อมูลเป็นหลัก (แม้ว่าการใช้งานที่ไม่เหมาะสมอาจส่งผลให้ประสิทธิภาพการทำงานช้าลงได้)

รูปแบบ : `CREATE [UNIQUE] INDEX [CONCURRENTLY] [[IF NOT EXISTS] name] ON [ONLY] table_name [USING method]
({ column_name | (expression) } [COLLATE collation] [opclass [(opclass_parameter = value [, ...])]] [ASC | DESC] [NULLS { FIRST | LAST }] [, ...])
[INCLUDE (column_name [, ...])]
[NULLS [NOT] DISTINCT]
[WITH (storage_parameter [= value] [, ...]`

PostgreSQL มี index methods แบบ B-tree, hash, GiST, SP-GiST, GIN และ BRIN นอกจากนี้ผู้ใช้อย่างยังสามารถกำหนดวิธีการสร้างดัชนีของตนเองได้ (แต่ไม่เป็นที่นิยมนัก)

สำหรับ Parameters UNIQUE จะใช้เมื่อต้องการทำให้ระบบตรวจสอบค่าที่ซ้ำกันในตาราง (หากมีข้อมูลอยู่แล้ว) เมื่อสร้าง Index ขึ้นมาใช้งานมีประโยชน์ทุกครั้งที่เพิ่มข้อมูล ความพยายามที่จะ INSERT หรือ UPDATE ข้อมูลซึ่งจะส่งผลให้เกิดรายการที่ซ้ำกันก็จะทำไม่ได้และเกิดข้อผิดพลาดขึ้น

ตัวอย่างเช่น : `CREATE UNIQUE INDEX title_idx ON films (title);` -- To create a unique B-tree index on the column title in the table films

หรือ `CREATE INDEX gin_idx ON documents_table USING GIN (locations) WITH (fastupdate = off)` -- To create a GIN index with fast updates disabled

การลบ Index รูปแบบ : `DROP INDEX [CONCURRENTLY] [IF EXISTS] name [, ...] [CASCADE | RESTRICT]`

ตัวอย่างเช่น : `DROP INDEX title_idx;` -- This command will remove the index title_idx:

Noted: เราสามารถทำการ delete ผ่าน GUI ใน pgAdmin ด้วยการ click ขวาที่ Index นั้นๆเลือก Menu > Delete/Drop

การแก้ไขเปลี่ยนแปลง Index รูปแบบ : `ALTER INDEX [IF EXISTS] name RENAME TO new_name`

หรือ `ALTER INDEX ALL IN TABLESPACE name [OWNED BY role_name [, ...]]
SET TABLESPACE new_tablespace [NOWAIT]`

ตัวอย่างเช่น : `ALTER INDEX distributors RENAME TO suppliers;`

หรือ `ALTER INDEX distributors SET TABLESPACE fasttablespace;` -- To move an index to a different tablespace



PostgreSQL : Indexes (cont.)

การนำ Index ไปใช้งานกับ table มีส่วนของการนำไปแสดงผล performance ได้ชัดเจนด้วยการที่เราใช้คำสั่ง EXPLAIN ดำเนินการแสดงการสแกนตารางที่อ้างอิงโดยคำสั่งร่วม

รูปแบบ : EXPLAIN [(option [, ...])] statement

หรือ EXPLAIN [ANALYZE] [VERBOSE] statement

ตัวอย่างเช่น : EXPLAIN SELECT * FROM address WHERE phone = '223664661973';

Noted: หากต้องการวิเคราะห์คำสั่งใด ๆ เช่น INSERT, UPDATE หรือ DELETE โดยไม่ส่งผลกระทบต่อข้อมูล ควรรวม EXPLAIN ANALYZE ไว้ใน block syntax ดังตัวอย่างนี้

BEGIN;

EXPLAIN ANALYZE sql_statement;

ROLLBACK;

Index Types ดัชนีแต่ละประเภทใช้อัลกอริทึมที่แตกต่างกันซึ่งเหมาะสมที่สุดสำหรับข้อความค้นหาประเภทต่างๆ คำสั่ง CREATE INDEX ตามค่า Default จะสร้างดัชนี B-tree มาใช้ซึ่งเหมาะกับสถานการณ์ทั่วไปที่สุด เลือกประเภทดัชนีอื่นๆ โดยใช้ USING ตามด้วยชื่อประเภทดัชนี

operators: < <= = >= >

Index ประเภท Hash : เก็บรหัส Hash 32 บิตที่ได้มาจากค่าของคอลัมน์ที่จัดทำดัชนี ดังนั้น ดัชนีดังกล่าวสามารถจัดการได้เฉพาะการเปรียบเทียบความเท่าเทียมกันอย่างง่ายเท่านั้นคือ

operators: =

Index ประเภท GiST : เป็นแบบดัชนีที่ไม่ใช่ Single แต่เป็นโครงสร้างพื้นฐาน (infrastructure) ที่สามารถใช้กลยุทธ์การจัดทำดัชนีต่างๆ ได้เหมาะกับ ประเภทข้อมูลทางเรขาคณิตสองมิติหลายประเภท ซึ่งสนับสนุนการสืบค้นที่จัดทำดัชนี

operators: << &< &> >> <<| &<| |&> |>> @> <@ ~= &&

Index ประเภท GIN : หรือเรียกว่า inverted indexes เหมาะสำหรับค่าข้อมูลที่ประกอบด้วยค่าคอมโพเนนต์หลายค่า เช่น Array

operators: <@ @> = &&

Index ประเภท BRIN : (ย่อมาจาก Block Range INdices) เก็บข้อมูลสรุปเกี่ยวกับค่าที่จัดเก็บไว้ในช่วงบล็อกทางกายภาพที่ต่อเนื่องกันของตารางมีประสิทธิภาพมากที่สุดสำหรับคอลัมน์ที่มีค่าสัมพันธ์กันพวกชนิดข้อมูลที่มีลำดับการจัดเรียงเชิงเส้น

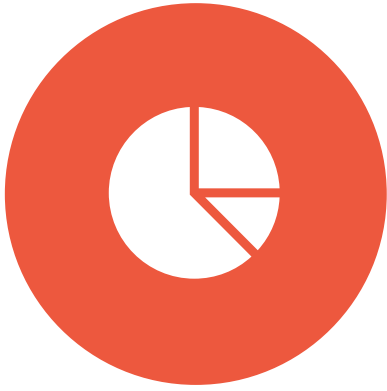
operators: < <= = >= >

Noted: มี Index อีกประเภทหนึ่งซึ่งไม่ได้กล่าวถึงคือ SP-GiST จะเหมือนกับ GiST ใช้กับ quadtrees, k-d trees, และ radix trees (tries).

operators: << >> ~= <@ <<| |>>

QUERY PLAN
► Index Scan using idx_address_phone on address (cost=0.28..8.29 rows=1 width=61)
Index Cond: ((phone)::text = '223664661973'::text)

Order By, Group By Clause



ORDER BY



GROUP BY



JOIN TABLE
& VIEW



PostgreSQL : ORDER BY

หลังจากคิวรีสร้างตารางเอาต์พุต (หลังจากประมวลผลรายการที่เลือกแล้ว) เราสามารถเลือกจัดเรียงได้ หากไม่ได้เลือกการเรียงลำดับ แถวจะถูกส่งคืนออกมาตามลำดับที่ไม่ได้ระบุ แต่ละ query สามารถตามด้วยคีย์เวิร์ด ASC หรือ DESC ที่เป็นทางเลือกเพื่อกำหนดทิศทางการเรียงลำดับเป็นจากน้อยไปหามาก (Ascending) หรือจากมากไปน้อย (Descending) โดยแบบเรียงจากน้อยไปหามาก (ASC) จะเป็นรูปแบบ default ของการ query ในชุดคำสั่งแต่ละครั้ง นอกจากนั้นยัง สามารถใช้ตัวเลือก NULLS FIRST และ NULLS LAST เพื่อระบุว่าค่า Null ปรากฏก่อนหรือหลังค่าที่ไม่เป็น Null ในการเรียงลำดับ การใช้งานแบบ NULLS FIRST ค่า NULL จะขึ้นมาก่อนต่างจาก NULL LAST ค่า NULL จะเรียงไว้สุดท้ายของผลลัพธ์

```
รูปแบบ SELECT select_list
        FROM table_expression
        ORDER BY sort_expression1 [ASC | DESC] [NULLS { FIRST | LAST }]
        [, sort_expression2 [ASC | DESC] [NULLS { FIRST | LAST }] ...]
```

ตัวอย่าง

```
SELECT a, b FROM table1 ORDER BY a + b, c;
-- NULLs are first now
SELECT name FROM cities ORDER BY name NULLS FIRST;
หรือ NULLS LAST >> SELECT actor_id, first_name, last_name FROM actor ORDER BY last_name NULLS LAST;
-- เราสามารถใช้ทั้ง DESC และ ASC กับ multiple columns ในแต่ละ query
SELECT title, release_year, rating FROM film ORDER BY release_year DESC, rating ASC;
-- การใช้ function มาเป็นการเรียงลำดับ
SELECT column FROM table ORDER BY random();
-- การ Sorting with GROUP BY and ORDER BY
SELECT rating, COUNT(*) "number" FROM film GROUP BY rating ORDER BY COUNT(*);
```

	PostgreSQL
Ascending order	NULLs last
Descending order	NULLs first



PostgreSQL : GROUP BY

GROUP BY clause ใช้ร่วมกับคำสั่ง SELECT เพื่อจัดกลุ่มแถวจากในตารางที่มีข้อมูลที่เหมือนกัน สิ่งนี้ทำเพื่อขจัดความซ้ำซ้อนในเอาต์พุตและ/หรือการรวม output จากการประมวลผลที่ใช้กับกลุ่ม data เหล่านี้ ตำแหน่งการวาง query command จะใช้ GROUP BY clause ตามหลัง WHERE clause ในคำสั่ง SELECT และนำหน้า ORDER BY clause

รูปแบบ

```
SELECT column-list  
FROM table_name  
WHERE [ conditions ]  
GROUP BY column1, column2....columnN  
ORDER BY column1, column2....column
```

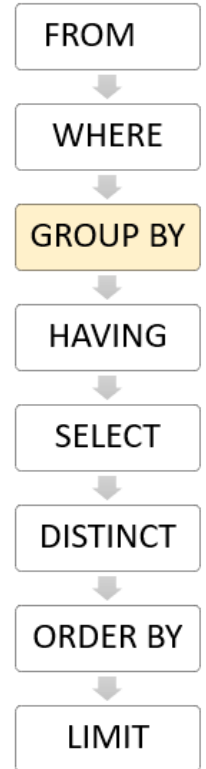
ตัวอย่าง

```
SELECT NAME, SUM(SALARY) FROM COMPANY GROUP BY NAME;
```

และเรายังใช้ GROUP BY กับ HAVING โดยจะกรองแถวที่จัดกลุ่มเหล่านี้โดยใช้ HAVING และกรองจำนวนเอาต์พุตโดยใช้ FILTER ตัวอย่างเช่น

```
SELECT city, max(temp_lo), count(*) FILTER (WHERE temp_lo < 30)  
FROM weather  
GROUP BY city  
HAVING max(temp_lo) < 40;
```

กล่าวโดยสรุปเราสามารถประยุกต์ใช้ GROUP BY กับ Aggregate Functions ได้มากมายจะ group รวมผลลัพธ์ด้วยการจัดกลุ่มหลาย column ก็สามารทำได้





PostgreSQL : GROUPING SETS and ROLLUP

GROUPING SETS คำสั่งในการรวมข้อมูล หาผลรวมย่อยหลาย ๆ กลุ่มย่อยพร้อม ๆ กันแบบเป็นชั้นๆและหลายๆกลุ่ม

ตัวอย่างเช่น : `select country, avg(lifeexp) from gapminder`

`group by grouping sets (country, ())`

Noted : จะได้แถวสรุปค่าเฉลี่ยของของทุก country มาคำนวณใน level แรก จะเหมือนกับการใช้งานแบบ UNION ข้อมูลของผลลัพธ์ที่ได้จาก 2 table เข้ามารวมไว้ด้วยกัน ดังนี้

`select country, avg(lifeexp) from gapminder`

`group by grouping sets (country, ())`

UNION

`select null, avg(lifeexp) from gapminder`

`order by country;`

หรือ

`select continent, country, avg(lifeexp) from gapminder`

`group by grouping sets (`

`country,`

`continent, ())`

`order by continent, country;`

หรือ

`select year, continent, avg(lifeexp) from gapminder`

`group by grouping sets (`

`(year, continent),`

`year,`

`continent, ())`

`HAVING year > 2000 order by year;`



PostgreSQL : GROUPING SETS and ROLLUP (cont.)

ROLLUP คำสั่งในการหาผลรวมย่อยหลาย ๆ ชั้น เช่นหาผลรวมย่อยหรือผลรวมทั้งหมดสำหรับรายงาน ของยอดขายตาม เดือน ไตรมาส และปี เป็นต้น

Noted : แตกต่างจาก CUBE subclause เนื่องจาก ROLLUP จะไม่สร้างชุดการจัดกลุ่มที่เป็นไปได้ทั้งหมดตามคอลัมน์ที่ระบุ มันสร้างแค่ subset ของข้อมูลสิ่งๆนั้น

รูปแบบ :
SELECT <column1>, <column2>
FROM <table_name>
GROUP BY
 ROLLUP(<column1>,<column2>...)
[ORDER BY <column_list>];

ตัวอย่างเช่น :
select branch, date_part('year', dt)::int [year],
 date_part('quarter', dt)::int [quarter],
 date_part('month', dt)::int [month], sum(revenue)
from sales
group by rollup(branch, date_part('year', dt), date_part('quarter', dt), date_part('month', dt))
order by branch, [year], [quarter], [month];

หรือเขียนแบบ grouping set ได้อีกด้วยดังนี้

```
select date_part('year', dt)::int [year], date_part('quarter', dt)::int [quarter], date_part('month', dt)::int [month], sum(revenue)
from sales
group by grouping sets ( (date_part('year', dt), date_part('quarter', dt), date_part('month', dt)),
                        (date_part('year', dt), date_part('quarter', dt)),
                        date_part('quarter', dt), () )
order by [year], [quarter], [month];
```



PostgreSQL : JOIN TABLE

การเชื่อมแบบ FULL JOIN : ทำการรวมแบบเต็มของ 2 Tables จะรวมผลลัพธ์ของการรวมด้านซ้ายและการรวมด้านขวาทั้งหมด ถ้าแถวในตารางที่รวมไม่ตรงกันผลที่ได้มันจะ set ค่า NULL สำหรับทุกคอลัมน์ของตารางที่ไม่มีแถวที่ตรงกัน ถ้าแถวจากตารางหนึ่งตรงกับแถวในอีกตารางหนึ่ง แถวผลลัพธ์จะมีคอลัมน์ที่สร้างจากคอลัมน์ของแถวจากทั้งสองตาราง

ตัวอย่างเช่น : `SELECT * FROM A FULL [OUTER] JOIN B on A.id = B.id;` --(OUTER keyword is optional)

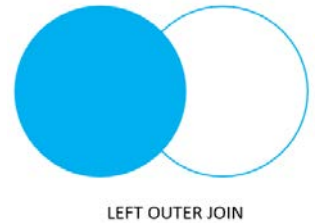
Noted: เป็นการ JOIN แบบที่เอา data ของทั้ง 2 Tables มาโชว์ที่ผลลัพธ์ทั้งหมดไม่มีข้อยกเว้น

และในบาง select query หากเราใช้ LEFT JOIN UNION RIGHT JOIN แบบไม่มี Where clauses จะได้ค่าเท่าๆกันกับ FULL JOIN ได้เช่นกัน



การเชื่อมแบบ LEFT OUTER JOIN : เป็นผลลัพธ์รวมโดยเลือกข้อมูลจากตารางด้านซ้ายหากเท่ากันกับตารางด้านขวา สำหรับแต่ละแถวในตารางด้านซ้ายตามเงื่อนไข โดยเติมคอลัมน์ที่มาจากตารางด้านขวาด้วย NULL

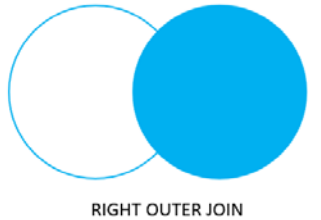
ตัวอย่างเช่น : `SELECT f.film_id, title, inventory_id FROM film f LEFT JOIN inventory i USING (film_id)`
`WHERE i.film_id IS NULL ORDER BY title;`



การเชื่อมแบบ RIGHT OUTER JOIN : เลือกข้อมูลจากตารางด้านขวาหากเท่ากันกับตารางด้านซ้าย ประกอบด้วยคอลัมน์จากทั้งสองตารางและรวมแถวใหม่ในชุดผลลัพธ์และเติมคอลัมน์จากตารางด้านซ้ายด้วย NULL (จะเลือกแถวทั้งหมดจากตารางด้านขวาไม่ว่าจะมีแถวที่ตรงกันจากตารางด้านซ้ายหรือไม่)

ตัวอย่างเช่น : `SELECT review, title FROM films RIGHT JOIN film_reviews using (film_id) WHERE title IS NULL;`

Noted: ให้ระวังการนำตัวเลขมากระทำทางคณิตศาสตร์ด้วยเครื่องหมาย +,-,*/ หรืออื่นๆกับค่า NULL ในผลลัพธ์กับตารางจะได้ค่าออกมาเป็น NULL ควรใช้ Coalesce() function มาช่วยในการแปลง NULL เป็น 0 ก่อนนำไปคำนวณ



การเชื่อมภายในตารางเดียวกัน (SELF JOIN หรือ INNER JOIN) : เป็นการรวมปกติที่รวมตารางเข้ากับตัวเองมักจะใช้เพื่อสืบค้นข้อมูลแบบลำดับขั้นหรือเพื่อเปรียบเทียบแถวภายในตารางเดียวกันโดยจะต้องระบุตารางเดียวกัน 2 ครั้งโดยใช้ table aliases ที่แตกต่างกันใน query (การทำตารางเสมือนด้วยตารางเดิมเป็นตารางใหม่เพื่อ JOIN กัน)

ตัวอย่างเช่น : `SELECT e.first_name || ' ' || e.last_name employee, m.first_name || ' ' || m.last_name manager FROM employee e`
`INNER JOIN employee m ON m.employee_id = e.manager_id ORDER BY manager;`

PostgreSQL : JOIN TABLE (cont.)

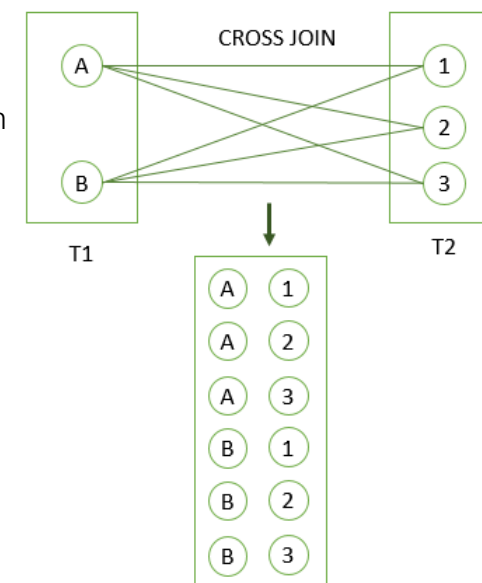
การเชื่อมแบบการทำงานของ CROSS JOIN : เป็นการเชื่อมแบบไขว้มองแบบ Table ว่าเป็น 2 set นำมาทำได้เป็นผลลัพธ์แบบผลคูณ Cartesian แตกต่างจากส่วนคำสั่งการรวมอื่นๆ เช่น LEFT JOIN, RIGHT JOIN หรือ INNER JOIN เพราะ CROSS JOIN ไม่มี join predicate (คือ ON table1.field_key = table2.field_key)

รูปแบบ : `SELECT column-lists FROM Table1 CROSS JOIN Table2;`

หรือ `SELECT [column_list[*]] FROM Table1, Table2;`

หรือ `SELECT * FROM Table1 INNER JOIN Table2 ON true;`

ตัวอย่างเช่น : `SELECT * FROM "EMPLOYMENT" CROSS JOIN "DEPARTMENT";`
`Select (*) from T1 CROSS JOIN T2`



Noted: เราสามารถประยุกต์ใช้ CROSS JOIN ในการข้อมูลแบบการจัดการแข่งขันพบกันหมด (round-robin tournament) ได้ด้วย

ข้อควรระวังในการใช้ CROSS JOIN กับตารางที่มี JOIN อื่นอย่าง INNER JOIN รวมอยู่ใน 1 ชุด query ตัวอย่างเช่น เงื่อนไข FROM T1 CROSS JOIN T2 INNER JOIN T3 ON ไม่เหมือนกับเงื่อนไข FROM T1, T2 INNER JOIN T3 ON เนื่องจากเงื่อนไขสามารถอ้างอิง T1 ในกรณีแรกได้ แต่ไม่ใช่กรณีที่สอง เป็นต้น



PostgreSQL : View

คือ virtual table ทำหน้าที่แสดงถึงผลลัพธ์ของแบบสอบถามไปยังตารางพื้นฐานอย่างน้อยหนึ่งตารางใน Postgres และ view ถูกใช้เพื่อลดความซับซ้อนของแบบสอบถามที่ซับซ้อน เนื่องจากแบบสอบถามเหล่านี้ถูกกำหนดครั้งเดียวจากใน view และจากนั้นสามารถสอบถามโดยตรงผ่านการ select

วิธีการเรียกสดง view ใน psql ใช้ดังนี้ : \dv หรือ \sv [view_name] ซึ่ง 2 command ให้ผลต่างกันว่า \sv จะแสดงรูปแบบคำสั่ง sql และโครงสร้างของ view table ออกมา ส่วน \dv เป็นการสืบค้น view ใน databaseที่เราเลือกใช้อยู่

CREATE VIEW — define a new view

```
รูปแบบคือ CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] [ RECURSIVE ] VIEW name [ ( column_name [, ...] ) ]
[ WITH ( view_option_name [= view_option_value] [, ... ] ) ]
AS query
[ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

ยกตัวอย่างเช่น :

```
create view vw_alcohol as
select country,beer_servings,wine_servings,total_litres_of_pure_alcohol from public.alcohol
where total_litres_of_pure_alcohol <> 0;
```

นอกจากนั้นยังมีการสร้างแบบ recursive view หรือ Common Table Expression (CTE)

รูปแบบ CREATE RECURSIVE VIEW [schema .] view_name (column_names) AS SELECT ...;

จะเหมือนกับการใช้ WITH expression_name [(column_name [,...n])] AS (CTE_query_definition)



PostgreSQL : View (cont.)

ตัวอย่างของ Recursive view

```
CREATE RECURSIVE VIEW fact(n, factorial) AS
(
  SELECT 1 as n, 5 as factorial
  union all
  SELECT n+1, factorial*n FROM fact where n < 5
);
select * from fact;
```

หากเราต้องการแก้ไข view เฉพาะที่เป็นเงื่อนไขหรือ query สามารถใช้ CREATE OR REPLACE ได้ แต่ถ้าต้องการแก้ไข column ใน view ให้มีการเพิ่มหรือลบออกต้องทำการ ALTER VIEW หรือ DROP VIEW แล้วสร้างใหม่

ALTER VIEW — change the definition of a view

รูปแบบ ALTER VIEW [IF EXISTS] name ALTER [COLUMN] column_name SET DEFAULT expression

หรือ ALTER VIEW [IF EXISTS] name ALTER [COLUMN] column_name DROP DEFAULT

ALTER VIEW [IF EXISTS] name OWNER TO { new_owner | CURRENT_ROLE | CURRENT_USER | SESSION_USER }

ALTER VIEW [IF EXISTS] name RENAME [COLUMN] column_name TO new_column_name

ALTER VIEW [IF EXISTS] name RENAME TO new_name

ALTER VIEW [IF EXISTS] name SET SCHEMA new_schema

ALTER VIEW [IF EXISTS] name SET (view_opti



PostgreSQL : View (cont.)

DROP VIEW — remove a view

รูปแบบ DROP VIEW [IF EXISTS] name [, ...] [CASCADE | RESTRICT]

ตัวอย่าง DROP VIEW IF EXISTS view_name;

และการกำหนด option CASCADE เพื่อที่จะ drop the dependent objects ไปด้วย

นอกจากนี้ยังมีการกำหนดสร้าง MATERIALIZED VIEW เป็น view cache ที่จะมีการนำข้อมูลใน view query นี้ไปเก็บไว้ในดิสก์ เก็บผลลัพธ์ของการค้นหาที่ซับซ้อน อนุญาตให้เราสามารถรีเฟรชผลลัพธ์นี้เป็นระยะๆ ช่วยในการเข้าถึงข้อมูลอย่างรวดเร็ว โดยเฉพาะอย่างยิ่งสำหรับแอปพลิเคชัน BI และ data warehouses

CREATE MATERIALIZED VIEW — define a new materialized view

รูปแบบ CREATE MATERIALIZED VIEW [IF NOT EXISTS] table_name [(column_name [, ...])] AS query [WITH [NO] DATA]

ALTER MATERIALIZED VIEW — change the definition of a materialized view

รูปแบบ ALTER MATERIALIZED VIEW [IF EXISTS] name action [, ...]

DROP MATERIALIZED VIEW — remove a materialized view

รูปแบบ DROP MATERIALIZED VIEW [IF EXISTS] name [, ...] [CASCADE | RESTRICT]

REFRESH MATERIALIZED VIEW — replace the contents of a materialized view

รูปแบบ REFRESH MATERIALIZED VIEW [CONCURRENTLY] name [WITH [NO] DATA]



PostgreSQL : VALUES and CTE (Common Table Expressions)

เป็นคำสั่งที่ช่วยนำมาสร้างตารางชั่วคราวโดยใช้ VALUES ใน PostgreSQL โดยจะเป็นการส่งคืนชุดของหนึ่งแถวขึ้นไป (เป็นแบบ standalone SQL statement)

รูปแบบ : VALUES (expression [, ...]) [, ...]

[ORDER BY sort_expression [ASC | DESC | USING operator] [, ...]]

[LIMIT { count | ALL }]

[OFFSET start [ROW | ROWS]]

[FETCH { FIRST | NEXT } [count] { ROW | ROWS } ONLY]

ตัวอย่างเช่น : VALUES

(1, 'Peter', 'Griffin'),

(2, 'Homer', 'Simpson'),

(3, 'Ned', 'Flanders'),

(4, 'Barney', 'Rubble'),

(5, 'George', 'Costanza')

FETCH FIRST 3 ROWS ONLY;

หรือเรียกใช้งานแบบ : SELECT FirstName, LastName FROM

(VALUES

(1, 'Peter', 'Griffin'),

(2, 'Homer', 'Simpson'),

(3, 'Ned', 'Flanders')

) AS Idiots(IdiotId, FirstName, LastName) WHERE IdiotId = 2;

Noted : VALUES list ทั้งหมดต้องมีความยาวเท่ากันไม่เช่นนั้นจะเกิด Error . ในการสร้างตารางได้ เช่นนี้เป็นตัวอย่าง list m มีไม่เท่ากันเกิด Error ขึ้น VALUES (1, 2), (3);



PostgreSQL : VALUES and CTE (Common Table Expressions) (cont.)

เป็นคำสั่งที่ช่วยนำมาใช้สร้างตารางทั่วไปเป็นแบบชั่วคราวในแง่ที่ว่ามีการอยู่ในระหว่างการดำเนินการค้นหาหรือในคำสั่ง SQL อื่น เช่น INSERT, UPDATE และ DELETE ในตอนนั้นนั่นเอง

รูปแบบ : WITH cte_name (column_list) AS (
 CTE_query_definition
)
 statement;

ตัวอย่างเช่น : with cte as (
 select * from users where beta is true
)
 select events.* from events
 inner join cte on cte.id = events.user_id;

หรือเรียกใช้ RECURSIVE กับ WITH สามารถอ้างถึงผลลัพธ์ของมันเอง ตัวอย่างง่ายๆ query นี้เพื่อรวมจำนวนเต็มตั้งแต่ 1 ถึง 100 ดังนี้ :

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

Noted : เรายังสามารถใช้ WITH เพื่อสร้างตารางชั่วคราวต่อกันหลายๆตารางเพื่อนำมาเรียกใช้ร่วมกันได้อีกด้วย เมื่อต้องการ join table หลายๆอันร่วมกันเป็นต้น

Online Documentation

<https://www.postgresql.org/docs/>

Cloud Postgres : EDB BigAnimal

<https://www.enterprisedb.com/>

EDB Postgres Migration

<https://www.enterprisedb.com/products/migration>

How to import and export data using CSV files in PostgreSQL

<https://www.enterprisedb.com/postgres-tutorials/how-import-and-export-data-using-csv-files-postgresql>

PostgreSQL Database Backup Solutions

<https://www.enterprisedb.com/products/backup-recovery>

Introduction to JSON Functions in PostgreSQL 9.5


<https://www.youtube.com/watch?v=I6D1HWf6isk>

How to Effectively Store & Index JSON Data in PostgreSQL

<https://scalegrid.io/blog/using-jsonb-in-postgresql-how-to-effectively-store-index-json-data-in-postgresql/>

Exporting table/query to XML with query_to_xml()

<https://www.youtube.com/watch?v=XWONag61elo>



postgres read/write binary data (bytea)

<https://jdbc.postgresql.org/documentation/binary-data/>

BINARY DATA PERFORMANCE IN POSTGRESQL

<https://www.cybertec-postgresql.com/en/binary-data-performance-in-postgresql/>

Upsert Using INSERT ON CONFLICT

<https://www.postgresqltutorial.com/postgresql-tutorial/postgresql-upsert/>

<https://www.prisma.io/dataguide/postgresql/inserting-and-modifying-data/insert-on-conflict>

PostgreSQL JSON cheatsheet

<https://devhints.io/postgresql-json>

JSON Types

<https://www.postgresql.org/docs/15/datatype-json.html>

Extracting JSON data into Excel

<https://www.youtube.com/watch?v=9EmSH9eBuMM>

Sample Database

<https://www.postgresqltutorial.com/postgresql-getting-started/postgresql-sample-database/>