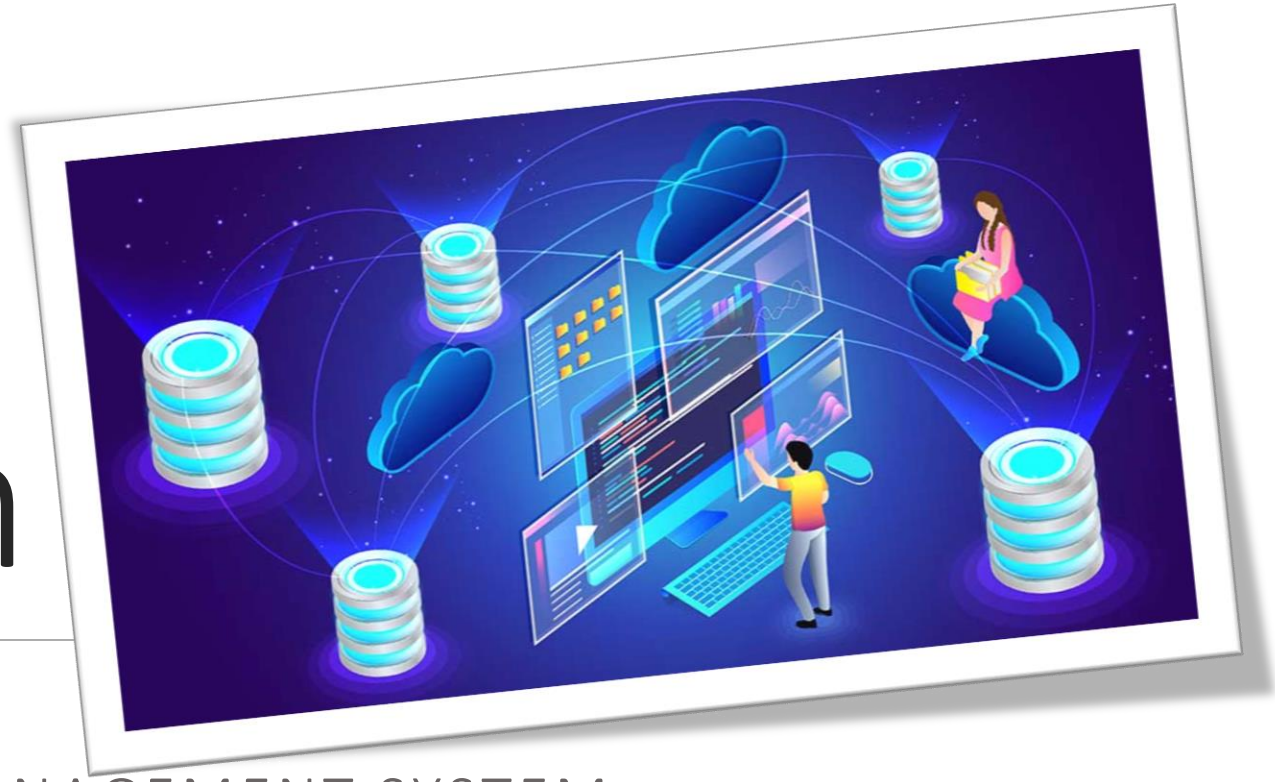
Advanced Database

COMP412

CHAPTER:01

Introduction



RDMS: RELATIONAL DATABASE MANAGEMENT SYSTEM

What are Database Management Systems

DBMS is a system for providing:

- EFFICIENT,
- CONVENIENT,
- SAFEMULTI-USER storage of and access to MASSIVE amounts of PERSISTENT data

Example: Banking System

- Data
  - Information on accounts, customers, balances, current interest rates, transaction histories, etc.
- MASSIVE
  - Many gigabytes at a minimum for big banks, more if keep history of all transactions, even more if keep images of checks -> Far too big to fit in main memory
- PERSISTENT
  - Data outlives programs that operate on it

Example: Banking System

- SAFE:
  - from system failures
  - from bad users
- CONVENIENT:
  - simple commands to debit account, get balance, write statement, transfer funds, etc.
  - also unpredicted queries should be easy
- EFFICIENT:
  - don't search all files in order to get balance of one account, get all accounts with low balances, get large transactions, etc.
  - massive data! -> DBMS's carefully tuned for performance

Multi-user Access

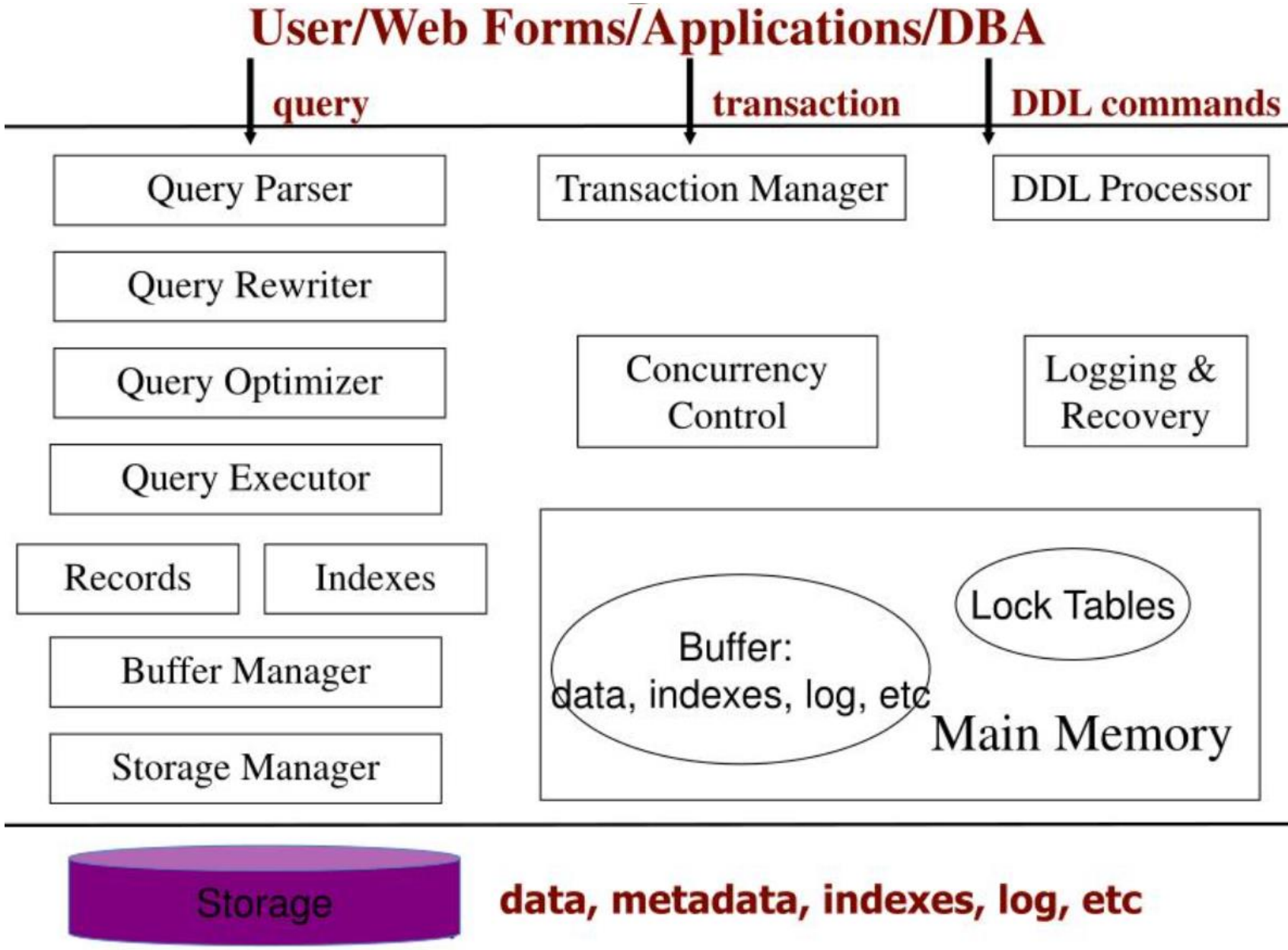
- Many people/programs accessing same database, or even same data, simultaneously -> Need careful controls
  - Alex @ ATM1: withdraw $100 from account #007
    - get balance from database;
    - if balance >= 100 then balance := balance - 100;
    - dispense cash;
    - put new balance into database;
  - Bob @ ATM2: withdraw $50 from account #007
    - get balance from database;
    - if balance >= 50 then balance := balance - 50;
    - dispense cash;
    - put new balance into database;
  - Initial balance = 120. Final balance = ??

Why File Systems Won't Work

- Storing data: file system is limited
  - size limit by disk or address space
  - when system crashes we may loose data
  - Password/file-based authorization insufficient
- Query/update:
  - need to write a new C++/Java program for every new query
  - need to worry about performance
- Concurrency: limited protection
  - need to worry about interfering with other users
  - need to offer different views to different users (e.g. registrar, students, professors)
- Schema change:
  - entails changing file formats
  - need to rewrite virtually all applications

That's why the notion of DBMS was motivated!

DMS Architecture



Data Structuring: Model, Schema, Data

- Data model
  - conceptual structuring of data stored in database
  - ex: data is set of records, each with student-ID, name, address, courses, photo
  - ex: data is graph where nodes represent cities, edges represent airline routes
- Schema versus data
  - schema: describes how data is to be structured, defined at set-up time, rarely changes (also called "metadata")
  - data: actual "instance" of database, changes rapidly
  - vs. types and variables in programming languages

Schema vs. Data

- Schema: name, name of each field, the type of each field
  - Students (Sid:string, Name:string, Age: integer, GPA: real)
  - A template for describing a student
- Data: an example instance of the relation

Sid	Name	Age	GPA
0001	Alex	19	3.55
0002	Bob	22	3.10
0003	Chris	20	3.80
0004	David	20	3.95
0005	Eugene	21	3.30

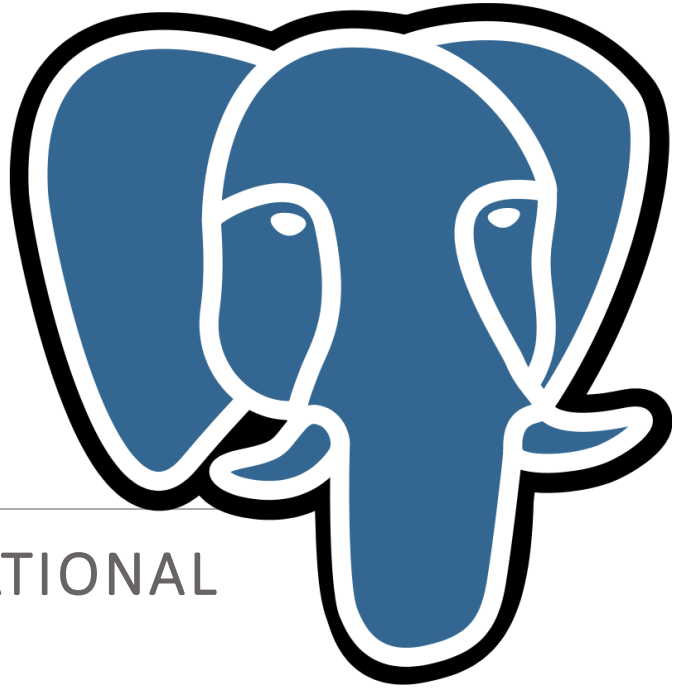
Data Structuring: Model, Schema, Data

- Data definition language (DDL)
  - commands for setting up schema of database

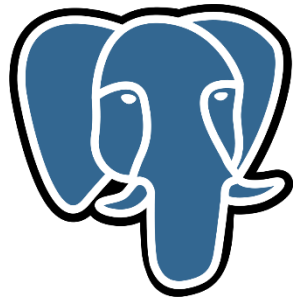
- Data Manipulation Language (DML)
  - Commands to manipulate data in database:
    - RETRIEVE, INSERT, DELETE, MODIFY
  - Also called "query language"

PostgreSQL

THE WORLD'S MOST ADVANCED OPEN SOURCE RELATIONAL DATABASE



PostgreSQL



- PostgreSQL is the world's most advanced open source database and the fourth most popular database. In development for more than 30 years, PostgreSQL is managed by a well-organized and highly principled and experienced open source community.

- PostgreSQL databases provide enterprise-class database solutions and are used by a wide variety of enterprises across many industries, including financial services, information technology, government and media and communications.

Administration Tools

Administration Tools

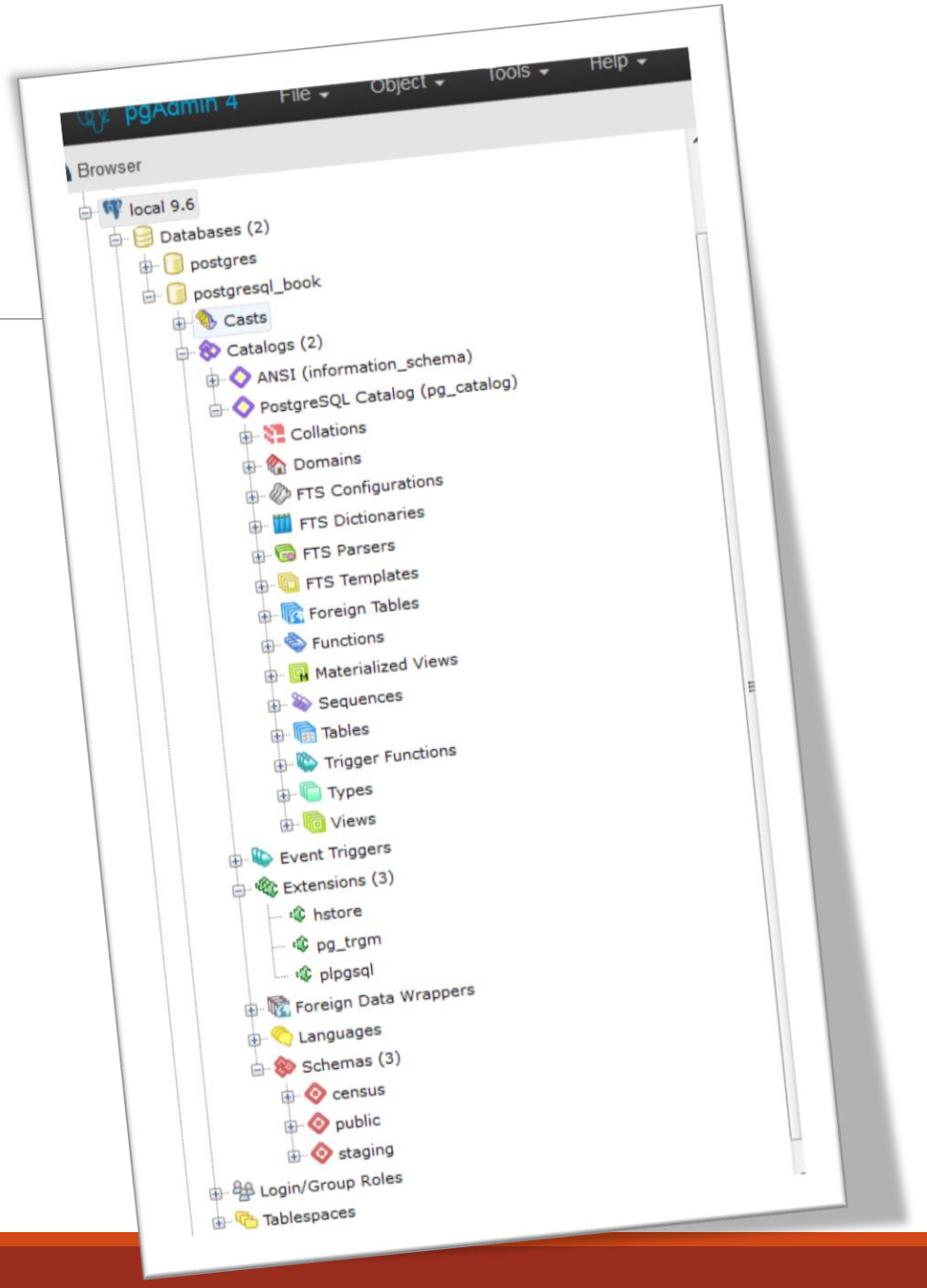
- Four tools widely used with PostgreSQL are:
  - psql,
  - pgAdmin,
  - phpPgAdmin,
  - Adminer.

psql

- psql is a command-line interface for running queries and is included in all distributions of PostgreSQL
- Is the tool of choice for many expert users, for people working in consoles without a GUI, or for running common tasks in shell scripts.
- psql has some unusual features, such as an import and export command for delimited files (CSV or tab), and a minimalistic report writer that can generate HTML output

pgAdmin

- pgAdmin is a popular, free GUI tool for PostgreSQL.
- Download it separately from PostgreSQL if it isn't already packaged with your installer
- pgAdmin recently entered its fourth release, dubbed pgAdmin4.
  - pgAdmin4 is a complete rewrite of pgAdmin3 that supports a desktop as well as a web server application version utilizing Python.

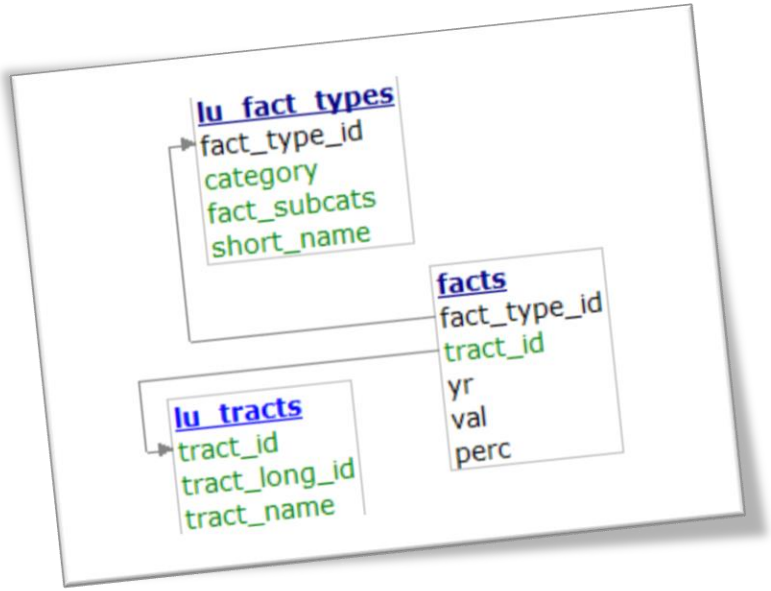


phpPgAdmin

- phpPgAdmin is a free, web-based administration tool patterned after the popular phpMyAdmin.
- phpPgAdmin differs from phpMyAdmin by including ways to manage PostgreSQL objects such as schemas, procedural languages, casts, operators, and so on.

Adminer

- If you manage other databases besides PostgreSQL and are looking for a unified tool, Adminer might fit the bill.
- Adminer is a lightweight, open source PHP application with options for PostgreSQL, MySQL, SQLite, SQL Server, and Oracle, all delivered through a single interface.
- One unique feature of Adminer we're impressed with is the relational diagramme that can produce a schematic layout of your database schema.



PostgreSQL Database Objects

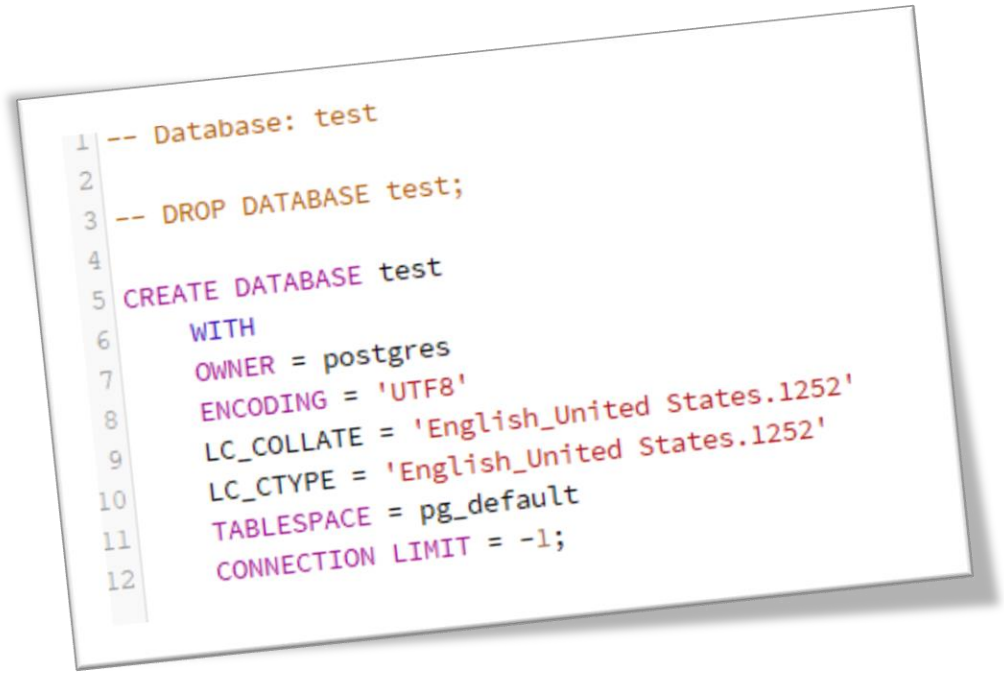
PostgreSQL Database Objects

- PostgreSQL has more database objects than most other relational database products

- We limit our quick overview to the most important objects that you should be familiar with.

Databases

- Each PostgreSQL service houses many individual databases.



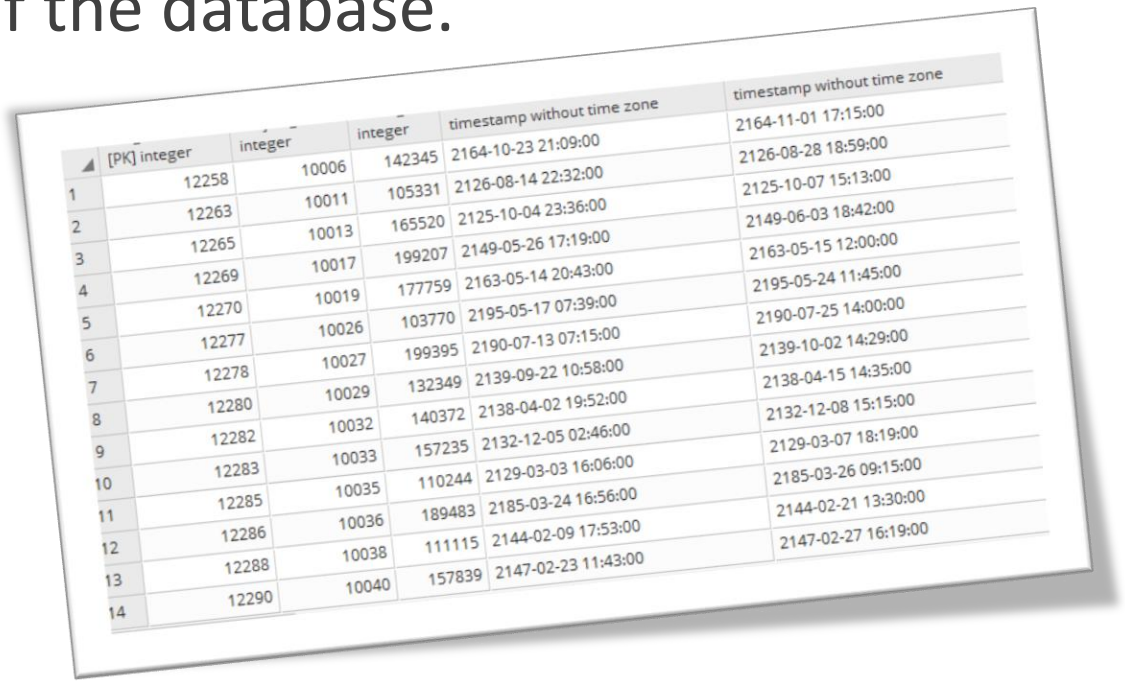
```
-- Database: test

-- DROP DATABASE test;

CREATE DATABASE test
    WITH
    OWNER = postgres
    ENCODING = 'UTF8'
    LC_COLLATE = 'English_United States.1252'
    LC_CTYPE = 'English_United States.1252'
    TABLESPACE = pg_default
    CONNECTION LIMIT = -1;
```

Schemas

- Schemas are part of the ANSI SQL standard.
- They are the immediate next level of organization within each database.
  - If you think of the database as a country, schemas would be the individual states.
- Most database objects first belong to a schema, which belongs to a database.
- When you create a new database, PostgreSQL automatically creates a schema named public to store objects that you create.
- If you have few tables, using public would be fine.
  - But if you have thousands of tables, you should organize them into different schemas.

Tables

- Tables are the workhorses of any database.
- In PostgreSQL, tables are first citizens of their respective schemas, which in turn are citizens of the database.



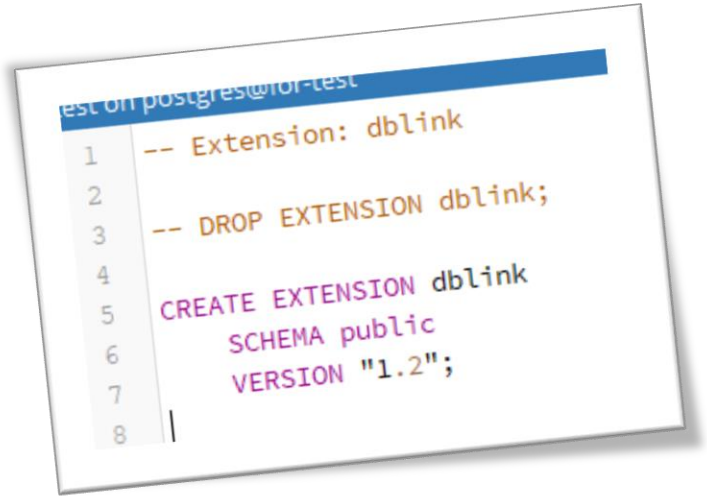
	[PK] integer	integer	integer	timestamp without time zone	timestamp without time zone
1	12258	10006	142345	2164-10-23 21:09:00	2164-11-01 17:15:00
2	12263	10011	105331	2126-08-14 22:32:00	2126-08-28 18:59:00
3	12265	10013	165520	2125-10-04 23:36:00	2125-10-07 15:13:00
4	12269	10017	199207	2149-05-26 17:19:00	2149-06-03 18:42:00
5	12270	10019	177759	2163-05-14 20:43:00	2163-05-15 12:00:00
6	12277	10026	103770	2195-05-17 07:39:00	2195-05-24 11:45:00
7	12278	10027	199395	2190-07-13 07:15:00	2190-07-25 14:00:00
8	12280	10029	132349	2139-09-22 10:58:00	2139-10-02 14:29:00
9	12282	10032	140372	2138-04-02 19:52:00	2138-04-15 14:35:00
10	12283	10033	157235	2132-12-05 02:46:00	2132-12-08 15:15:00
11	12285	10035	110244	2129-03-03 16:06:00	2129-03-07 18:19:00
12	12286	10036	189483	2185-03-24 16:56:00	2185-03-26 09:15:00
13	12288	10038	111115	2144-02-09 17:53:00	2144-02-21 13:30:00
14	12290	10040	157839	2147-02-23 11:43:00	2147-02-27 16:19:00

Views

- Relational database products offer views as a level of abstraction from tables.
- In a view, you can query multiple tables and present additional derived columns based on complex calculations.
- Views are generally read-only, but PostgreSQL allows you to update the underlying data by updating the view, provided that the view draws from a single table.
- Version 9.3 introduced materialized views, which cache data to speed up commonly used queries at the sacrifice of having the most up-to-date data.

Extension

- Extensions allow developers to package functions, data types, casts, custom index types, tables, attribute variables, etc., for installation or removal as a unit.



Functions

- You can program your own custom functions to handle data manipulation, perform complex calculations, or wrap similar functionality.
  - Create functions using PLs.
- PostgreSQL comes stocked with thousands of functions, which you can view in the postgres database that is part of every install.
- PostgreSQL functions can return scalar values, arrays, single records, or sets of records.

Foreign tables

- Foreign tables are virtual tables linked to data outside a PostgreSQL database.
- Once you've configured the link, you can query them like any other tables.
- Foreign tables can link to CSV files, a PostgreSQL table on another server, a table in a different product such as:
  - SQL Server or Oracle, a NoSQL database such as Redis, or even a web service such as Twitter or Salesforce.

Triggers and trigger functions

- Triggers detect data-change events.
- When PostgreSQL fires a trigger, you have the opportunity to execute trigger functions in response.
- A trigger can run in response to particular types of statements or in response to changes to particular rows, and can fire before or after a data-change event.
- Trigger functions are often used to write complex validation routines that are beyond what can be implemented using check constraints.

Catalogs

- Catalogs are system schemas that store PostgreSQL builtin functions and metadata.
- Every database contains two catalogs:
  - pg_catalog, which holds all functions, tables, system views, casts, and types packaged with PostgreSQL;
  - information_schema, which offers views exposing metadata in a format dictated by the ANSI SQL standard.

Types

- Data type such as: integers, characters, arrays, blobs, etc.
- PostgreSQL has composite types, which are made up of other types. Think of complex numbers, polar coordinates, or vectors as examples.

Full text search

- Full text search (FTS) is a natural language–based search.
- This kind of search has some “intelligence” built in.

- Unlike regular expression search, FTS can match based on the semantics of an expression, not just its syntactical makeup.
  - For example, if you’re searching for the word running in a long piece of text, you may end up with run, running, ran, runner, jog, sprint, dash, and so on.

Sequences

- A sequence controls the auto incrementation of a serial data type.
- You can easily change the initial value, step, and next available value.
- More than one table can share the same sequence object.
  - This allows you to create a unique key value that can span tables.

Versions of PostgreSQL

Versions of PostgreSQL

- Every September a new PostgreSQL is released.
- With each new release comes greater stability, heightened security, better performance and grade features.
- The upgrade process itself gets easier with each new version.

Database Drivers

Database Drivers

- PostgreSQL works with free drivers for many programming languages and tools:
  - For **PHP**: most PHP distributions include at least one PostgreSQL driver: the old pgsql driver or the newer pdo_pgsql
  - For **Java** developers, the JDBC driver keeps up with latest PostgreSQL versions.
  - For .**NET** (both Microsoft or Mono), you can use the Npgsql driver. Both the source code and the binary are available for .NET Framework, Microsoft Entity Framework, and Mono.NET.

Database Drivers

- If you need to connect from Microsoft **Access**, **Excel**, or any other products that support Open Database Connectivity (ODBC), download drivers from the PostgreSQL ODBC drivers site.
- **Python** has support for PostgreSQL via many database drivers. At the moment, psycopg2 is the most popular.
- **Node.js** is a JavaScript framework for running scalable network programs. There are two PostgreSQL drivers currently: Node Postgres with optional native libpq bindings and pure JS (no compilation required) and Node-DBI.

Advanced Database

COMP412

CHAPTER:02 DATABASE ADMINISTRATION

Introduction

- This chapter covers what we consider basic administration of a PostgreSQL server:
  - Configurations
  - Managing roles and permissions,
  - Creating databases,
  - Backing up and restoring data.

Configuration Files

CONTROL OPERATIONS OF A POSTGRESQL SERVER

postgresql.conf

- Controls general settings, such as:
- memory allocation,
- default storage location for new databases,
- the IP addresses that PostgreSQL listens on,
- location of logs,
- ...

pg_hba.conf

- Controls access to the server
- dictating which users can log in to which databases
- which IP addresses can connect,
- which authentication scheme to accept
- …

pg_ident.conf

- If present, this file maps an authenticated OS login to a PostgreSQL user.
- People sometimes map the OS root account to the PostgresSQL superuser account, postgres.

Location of configuration files

- Query as a superuser while connected to any database.
  - SELECT name, setting FROM pg_settings WHERE category = 'File Locations';

```
      name         |                  setting
-------------------+-----------------------------------
 config_file       | /etc/postgresql/9.6/main/postgresql.conf
 data_directory    | /var/lib/postgresql/9.6/main
 external_pid_file | /var/run/postgresql/9.6-main.pid
 hba_file          | /etc/postgresql/9.6/main/pg_hba.conf
 ident_file        | /etc/postgresql/9.6/main/pg_ident.conf
(5 rows)
```

Making Configurations Take Effect

- Reloading
  - pg_ctl reload -D your_data_directory_here
  - SELECT pg_reload_conf();

- Restarting
  - service postgresql-9.6 restart
  - pg_ctl restart -D your_data_directory_here (For any PostgreSQL instance not installed as a service)

  - Rem: use “show data_directory” to get data directory of current instance

The postgresql.conf File

- Changig the postgresql.conf settings
  - using the ALTER SYSTEM SQL command.

```
ALTER SYSTEM SET work_mem = '500MB';
```

"I edited my postgresql.conf and now my server won't start."

- The easiest way to figure out what you screwed up is to look at the logfile, open the latest file and read what the last line says. The error raised is usually self-explanatory.

The postgresql.conf File

- Instead of editing **postgresql.conf** directly, you should override settings using an additional file called **post-gresql.auto.conf**.
- Checking postgresql.conf settings without opening the configuration files is to query the view named pg_settings.

```
SELECT  name,  context ,  unit , setting,  boot_val,  reset_val
FROM pg_settings
WHERE name IN ('listen_addresses','deadlock_timeout','shared_buffers',
'effective_cache_size','work_mem','maintenance_work_mem')
ORDER BY context, name;
```

The postgresql.conf File

```
name                  | context    | unit | setting | boot_val  | reset_val
----------------------+------------+------+-------- +-----------+----------
listen_addresses      | postmaster |      | *       | localhost | *
deadlock_timeout      | superuser  | ms   | 1000    | 1000      | 1000
effective_cache_size | user       | 8kB  | 16384   | 16384     | 16384
```

- The context is the scope of the setting. Some settings have a wider effect than others, depending on their context.
- Postmaster settings affect the entire server (postmaster represents the PostgreSQL service) and take effect only after a restart.
- If set by the superuser, the setting becomes a default for all users

The postgresql.conf File

- The pg_file_settings can be used also to query settings.

```
SELECT name, sourcefile, sourceline, setting, applied
FROM pg_file_settings
WHERE name IN ('listen_addresses','deadlock_timeout','shared_buffers',
'effective_cache_size','work_mem','maintenance_work_mem')
ORDER BY name;
```

```
name                | sourcefile                    | sourceline | setting | applied
--------------------+-------------------------------+------------+---------+--------
effective_cache_size | E:/data96/postgresql.auto.conf|         11 | 8GB     | t
listen_addresses     | E:/data96/postgresql.conf     |         59 | *       | t
```

- incorrect changing in postgresql.conf will prevent clients from connecting.

The postgresql.conf File

- Changing values requires a service restart:
  - **listen_addresses**: Informs PostgreSQL which IP addresses to listen on.
  - **Port**: Defaults to 5432. You may wish to change for security or if you are running multiple PostgreSQL services on the same server.
  - **max_connections**: The maximum number of concurrent connections allowed.
  - **log_destination**: specifies the format of the logfiles rather than their physical location.

The postgresql.conf File

- The following settings affect performance.
  - **shared_buffers**: Allocated amount of memory shared among all connections to store recently accessed pages.
  - **effective_cache_size**: An estimate of how much memory PostgreSQL expects the operating system to devote to it. This setting has no effect on actual allocation, but the query planner figures in this setting to guess whether intermediate steps and query output would fit in RAM. If you set this much lower than available RAM, the planner may forgo using indexes.

The postgresql.conf File

- **work_mem**: Controls the maximum amount of memory allocated for each operation such as sorting, hash join, and table scans. If you have many users running simple queries, you want this setting to be relatively low to be democratic; otherwise, the first user may hog all the memory.
- **maintenance_work_mem:** The total memory allocated for housekeeping activities such as vacuuming (pruning records marked for deletion).
- **max_parallel_workers_per_gather:** This is a new setting introduced in 9.6 for parallelism. The setting determines the maximum parallel worker threads that can be spawned for each gather operation. The default setting is 0, which means parallelism is completely turned off. If you have more than one CPU core, you will want to elevate this.

The pg_hba.conf File

- The pg_hba.conf file controls which IP addresses and users can connect to the database.
- Furthermore, it dictates the authentication protocol that the client must follow

```
# TYPE  DATABASE USER ADDRESS          METHOD
host    all      all  127.0.0.1/32     ident ❶
host    all      all  ::1/128          trust ❷
host    all      all  192.168.54.0/24 md5 ❸
hostssl all      all  0.0.0.0/0       md5 ❹
```

1) Authentication method. The usual choices are ident, trust, md5, peer, and password.
2) IPv6 syntax for defining network range.
3) IPv4 syntax for defining network range.
4) SSL connection rule. In our example, we allow anyone to connect to our server outside of the allowed IP range as long as they can connect using SSL.

The pg_hba.conf File

- This entry allows any IP address to connect to any database with any username using the "md5" authentication method.
- Note that using "0.0.0.0/0" as the address allows connections from any IP address, which can be a potential security risk.
  - # TYPE DATABASE USER ADDRESS METHOD
  - host all all 0.0.0.0/0 md5
- It's generally recommended to restrict access to specific IP addresses or ranges whenever possible.

Managing Connections

Managing Connections

- Retrieve a listing of recent connections and process IDs (PIDs)

```
SELECT * FROM pg_stat_activity;
```

- Cancel active queries on a connection with PID 1234:

```
SELECT pg_cancel_backend(1234);
```

  - This does not terminate the connection itself, though.

- Terminate the connection:

```
SELECT pg_terminate_backend(1234);
```

- Kill all connections belonging to a role with a single blow

```
SELECT pg_terminate_backend(pid) FROM pg_stat_activity
WHERE usename = 'some_role'
```

Managing Connections

- You can set certain operational parameters at the server, database, user, session, or function level. Any queries that exceed the parameter will automatically be cancelled by the server. Setting a parameter to 0 disables the parameter:
  - **deadlock_timeout**: This is the amount of time a deadlocked query should wait before giving up. This defaults to 1000 ms.
  - **statement_timeout**: This is the amount of time a query can run before it is forced to cancel. This defaults to 0, meaning no time limit.
  - **lock_timeout**: This is the amount of time a query should wait for a lock before giving up, and is most applicable to update queries. The default is 0, meaning that the query will wait infinitely

Roles

Roles

- PostgreSQL handles credentialing using roles.
- Roles that can log in are called login roles.
- Roles can also be members of other roles; the roles that contain other roles are called group roles.
  - for security, group roles generally cannot log in
- CREATE USER and CREATE GROUP still work in current versions, but shun them and use CREATE ROLE instead.

Creating Login Roles

- pgAdmin has a graphical section for creating user roles, but if you want to create one using SQL, execute an SQL command.
  - CREATE ROLE leo LOGIN PASSWORD 'king' VALID UNTIL 'infinity' CREATEDB;
    - Specifying VALID UNTIL is optional. If omitted, the role remains active indefinitely.
    - CREATEDB grants database creation privilege to the new role.
  - CREATE ROLE regina LOGIN PASSWORD 'queen' VALID UNTIL '2020-1-1 00:00' SUPERUSER;
    - CREATE ROLE regina LOGIN PASSWORD 'queen' VALID UNTIL '2020-1-1 00:00' SUPERUSER;
      - To create a user with superuser privileges.

Creating Group Roles

- Group roles generally cannot log in
  - CREATE ROLE royalty INHERIT;
    - INHERIT means that any member of royalty will automatically inherit privileges of the royalty role.
    - except for the superuser privilege
- To add members to a group role, you would do:
  - GRANT royalty TO leo;
  - GRANT royalty TO regina;
- Let's give the royalty role superuser rights with the command:
  - ALTER ROLE royalty SUPERUSER;
  - leo can gain superuser rights by doing:
    - SET ROLE royalty;

Creating Group Roles

- Group roles generally cannot log in
  - CREATE ROLE royalty INHERIT;
    - INHERIT means that any member of royalty will automatically inherit privileges of the royalty role.
- To add members to a group role, you would do:
  - GRANT royalty TO leo;
  - GRANT royalty TO regina;
- Let's give the royalty role superuser rights with the command:
  - ALTER ROLE royalty SUPERUSER;
- leo istill will not have superuser rights. He can gain superuser rights by doing:
  - SET ROLE royalty;

Creating Group Roles

- Example 2-7. SET ROLE and SET AUTHORIZATION

Database Creation

Database Creation

- The minimum SQL command to create a database is:

  CREATE DATABASE mydb;

- This creates a copy of the template1 database. Any role with CREATEDB privilege can create new databases.

- The basic syntax to create a database modeled after a specific template is:
  - CREATE DATABASE my_db TEMPLATE my_template_db;

Template Databases

- A template database serves as a skeleton for new databases.
- PostgreSQL copies all the database settings and data from the template database to the new database.
- The default PostgreSQL installation comes with two template databases: template0 and template1.
- template1 is used as default

Using Schemas

- Schemas organize your database into logical groups.
- CREATE SCHEMA my_schema;
- Take advantage of the default search path set in postgresql.conf to search in same schema as login roles

```
set search_path = "$user", public;
```

Privileges

Privileges

- A privilege (often called permissions) is **a right to execute a particular type of SQL statement or to access another user's object**.
- Privileges can bore down to the column and row level.
- PostgreSQL has a few dozen privileges
  - The more mundane privileges are SELECT, INSERT, UPDATE, ALTER, EXECUTE, DELETE, and TRUNCATE
- Most privileges must have a context.
  - A role having an ALTER privilege is meaningless unless qualified with a database object such as ALTER privilege on tables1, SELECT privilege on table2, EXECUTE privilege on function1, and so on.
  - Not all privileges apply to all objects: an EXECUTE privilege for a table is nonsense.
- Some privileges make sense without a context.
  - CREATEDB and CREATE ROLE are two privileges where context is irrelevant

Getting Started

1. PostgreSQL creates one superuser and one database for you at installation, both named postgres. Log in to your server as postgres.
2. Before creating your first database, create a role that will own the database and can log in, such as:

```
CREATE ROLE mydb_admin LOGIN PASSWORD 'something';
```

3. Create the database and set the owner:

```
CREATE DATABASE mydb WITH owner = mydb_admin;
```

4. Now log in as the mydb_admin user and start setting up additional schemas and tables.

GRANT

- The GRANT command is the primary means to assign privileges. Basic usage is: *GRANT some_privilege TO some_role*;
- A few things to keep in mind when it comes to GRANT:
  - You need to have the privilege you're granting. And, you must have the GRANT privilege yourself.
  - Some privileges always remain with the owner of an object and can never be granted away. These include DROP and ALTER.
  - The owner of an object retains all privileges. That ownership does not drill down to child objects.

GRANT

- A few things to keep in mind when it comes to GRANT:
  - When granting privileges, you can add WITH GRANT OPTION. This means that the grantee can grant her own privileges to others, passing them on:
    - GRANT ALL ON ALL TABLES IN SCHEMA public TO mydb_admin WITH GRANT OPTION;
    - ALTER role mydb_admin WITH CREATEROLE
  - To grant specific privileges on ALL objects of a specific type use ALL instead of the specific object name, as in:
    - GRANT SELECT, REFERENCES, TRIGGER ON ALL TABLES IN SCHEMA my_schema TO PUBLIC;
  - To grant privileges to all roles, you can use the PUBLIC alias, as in:
    - GRANT USAGE ON SCHEMA my_schema TO PUBLIC;

REVOKE

- In many cases you might consider revoking some of the defaults with the REVOKE command, as in:

```
REVOKE EXECUTE ON ALL FUNCTIONS IN SCHEMA my_schema FROM PUBLIC;
```

```
REVOKE privilege | ALL
ON TABLE table_name |  ALL TABLES IN SCHEMA schema_name
FROM role_name;
```

Backup and Restore

RECOVER YOUR DATA FROM MANY FAILURES

Selective Backup Using pg_dump

- **pg_dump** can selectively back up tables, schemas, and databases.
- **pg_dump** can backup to plain SQL, as well as compressed, TAR, and directory formats.
  - **Compressed**, **TAR**, and **directory** format backups can take advantage of the parallel restore feature of **pg_restore**.
  - **Directory** backups allow parallel **pg_dump** of a large database.

  - https://www.postgresql.org/docs/9.6/app-pgdump.html

Selective Backup Using pg_dump

- To create a compressed, single database backup:
  - *pg_dump* -h localhost -p 5432 -U someuser -F c -v -f mydb.backup mydb
  - -v: This will cause pg_dump to output detailed object comments

- To create a plain-text single database backup:
  - *pg_dump* -h localhost -p 5432 -U someuser -F p -v -f mydb.backup mydb

- To create a compressed backup of tables whose names start with pay in any schema:
  - *pg_dump* -h localhost -p 5432 -U someuser -F c -v -t *.pay* -f pay.backup mydb

Selective Backup Using pg_dump

- To create a plain-text SQL backup of select tables, useful for porting structure and data to lower versions of PostgreSQL or non-PostgreSQL databases
  - plain text generates an SQL script that you can run on any system that speaks SQL:
  - pg_dump -h localhost -p 5432 -U someuser -F p --column-inserts -f select_tables.backup mydb

Restoring Data

- There are two ways to restore data in PostgreSQL from backups created with pg_dump
  - Use psql to restore plain-text backups generated with pg_dump.
  - Use pg_restore to restore compressed, TAR, and directory backups created with pg_dump.

Using psql to restore plain-text SQL backups

- To restore a backup and ignore errors:
  - psql -U postgres -f myglobals.sql
- To restore to a specific database:
  - psql -U postgres -d mydb -f select_objects.sql

Using pg_restore

- To perform a restore using pg_restore, first create the database using SQL:
  - CREATE DATABASE mydb;
- Then restore:
  - pg_restore --dbname=mydb --jobs=4 --verbose mydb.backup
- If the name of the database is the same as the one you backed up, you can create and restore the database in one step:
  - pg_restore --dbname=postgres --create --jobs=4 --verbose mydb.backup
- You can perform parallel restores using the -j (equivalent to --jobs=) option to indicate the number of threads to use

Tablespaces

Tablespaces

- PostgreSQL uses tablespaces to ascribe logical names to physical locations on disk.
- To create a new tablespace, specify a logical name and a physical folder
  - CREATE TABLESPACE secondary LOCATION 'C:/pgdata94_secondary';
- To move all objects in the database to your secondary tablespace, issue the following SQL command:
  - ALTER DATABASE mydb SET TABLESPACE secondary;

Advanced Database

COMP412

CHAPTER:03 PSQL

Introduction

- psql is the de rigueur command-line utility packaged with PostgreSQL.
  - you can use psql to execute scripts,
  - import and export data,
  - restore tables,
  - and do other database administration,
- If you have access only to a server's command line with no GUI, psql is your only choice to interact with PostgreSQL.

Environment Variables

- The file .pgpass in a user's home directory can contain passwords to be used if the connection requires a password.

- On Microsoft Windows the file named %APPDATA%\postgresql\pgpass.conf.

- This file should contain lines of the following format:
  - hostname:port:database:username:password

Interactive versus Noninteractive psql

CONTROL OPERATIONS OF A POSTGRESQL SERVER

Interactive versus Noninteractive psql

- Run psql interactively by typing psql from your OS command line.
- Begin typing in commands. For SQL statements, terminate with a semicolon. If you press Enter without a semicolon, psql will assume that your statement continues to the next line.
- Typing \?” while in the psql console brings up a list of available commands.

  - psql -d postgresql_book -c "DROP TABLE IF EXISTS dross; CREATE SCHEMA staging;"

Interactive versus Noninteractive psql

- To run commands repeatedly or in a sequence, you're better off creating a script first and then running it using psql noninteractively.
- At your OS prompt, type psql followed by the name of the script file.
  - psql -f some_script_file

Importing and Exporting Data

Importing and Exporting Data

- psql has a \copy command that lets you import data from and export data to a text file.
- The tab is the default delimiter, but you can specify others.
- Newline breaks must separate the rows.

psql Export

- You can export selected rows from a table.
- Use the psql \copy command to export.

```
\connect postgresql_book
\copy (SELECT * FROM staging.factfinder_import WHERE
TO '/test.tab'
WITH DELIMITER E'\t' CSV HEADER
```

psql Import

- Before bringing the data into PostgreSQL, you must first create a table to store the incoming data.
- The data must match the file both in the number of columns and in data types.
- psql processes the entire import as a single transaction; if it encounters any errors in the data, the entire import fails.

psql Import

- Importing data using the \copy command.
- If your file has nonstandard delimiters such as pipes, indicate the delimiter as follows:
  - \copy sometable FROM somefile.txt DELIMITER '|';
- During import, you can replace null values with something of your own choosing by adding a NULL AS, as in the following:
  - \copy sometable FROM somefile.txt NULL As '';

Advanced Database

COMP412

CHAPTER: 04 USING PGADMIN

Introduction

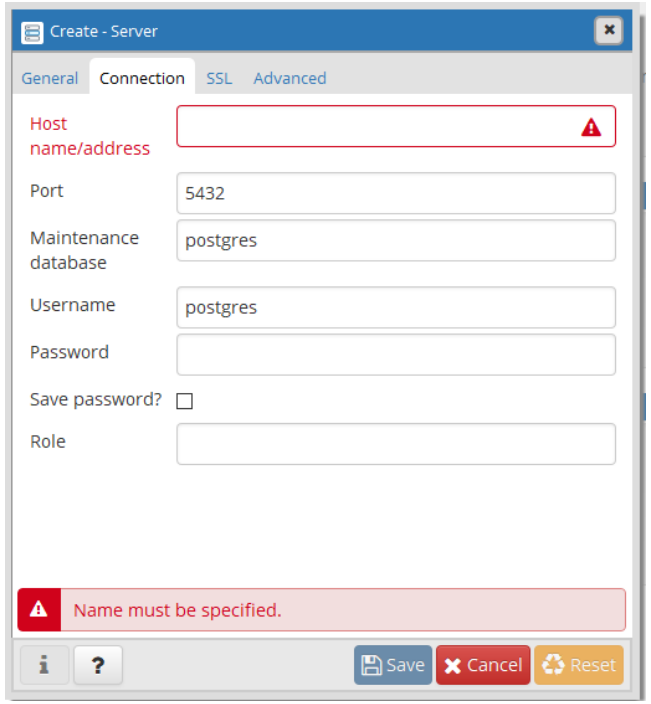
- Because the PostgreSQL developers position pgAdmin as the most commonly used graphical administration tool for PostgreSQL and it is packaged with many binary distributions of PostgreSQL, the developers have taken on the responsibility of keeping pgAdmin always in sync with the latest PostgreSQL releases.

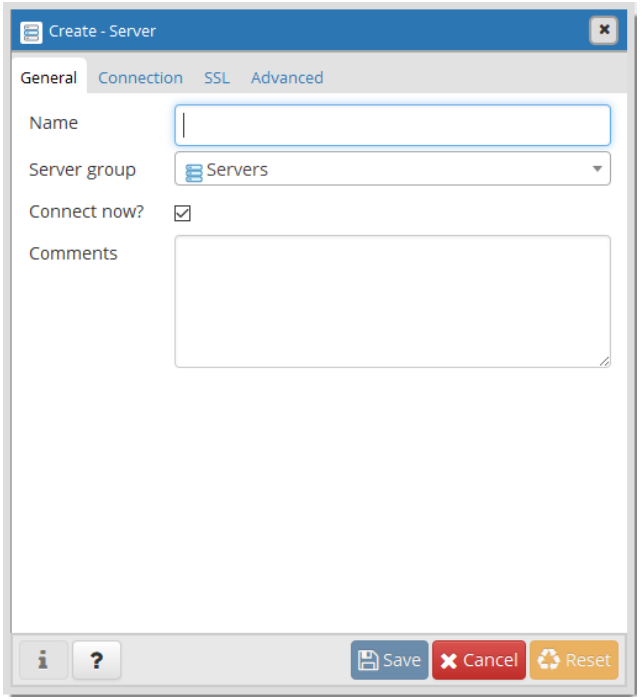
- If a new release of PostgreSQL introduces new features, you can count on the latest pgAdmin to let you manage it.

PgAdmin Features

Connecting to a PostgreSQL Server

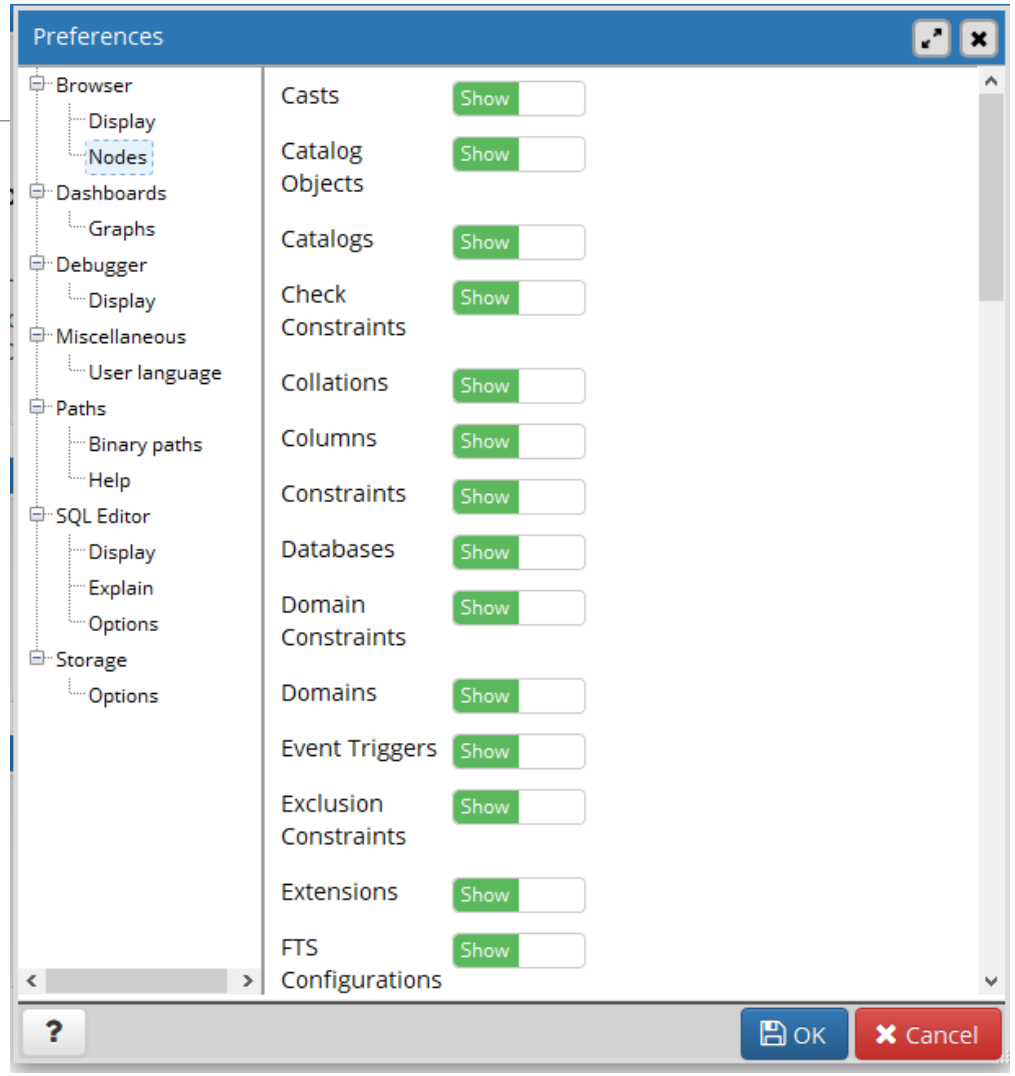
- Connecting to a PostgreSQL server with pgAdmin is straightforward.





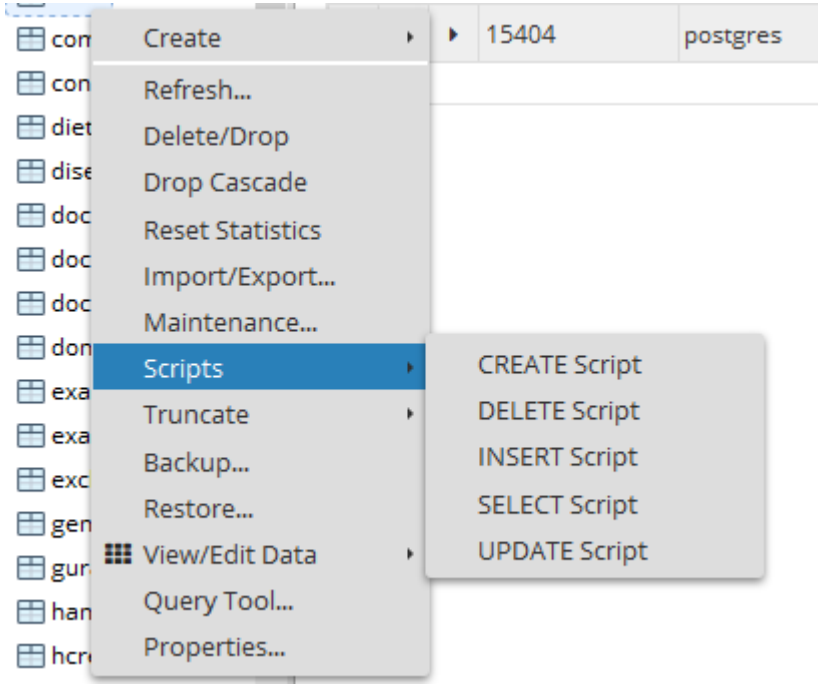
Navigating pgAdmin

- The tree layout of pgAdmin is intuitive to follow but does engender some possible anxiety.
- You can pare down the tree display by going into the Browser section of Preferences and deselecting objects that you would rather not have to stare at every time you use pgAdmin.
- To declutter the browse tree sections, go to Files→Preferen- ces→Browser→Nodes.



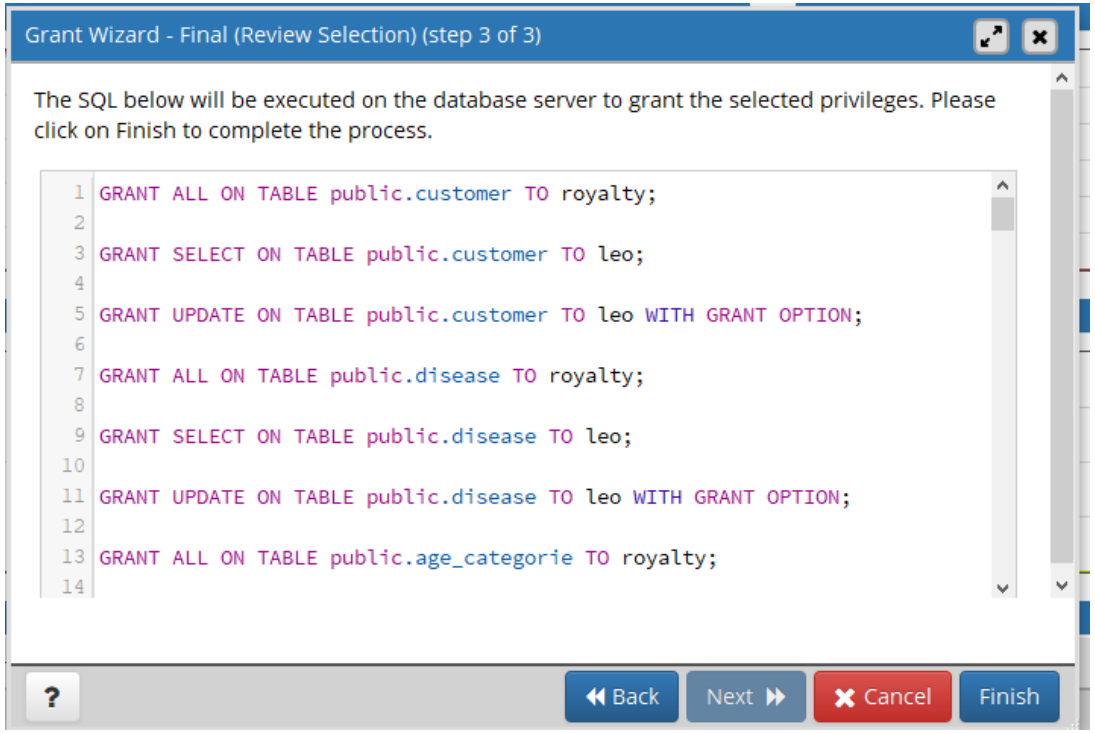
Autogenerating Queries from Table Definitions

- pgAdmin has this menu option that will autogenerate a template for SELECT, INSERT, and UPDATE statements from a table definition. You access this feature by right-clicking the table and accessing the SCRIPTS context menu option



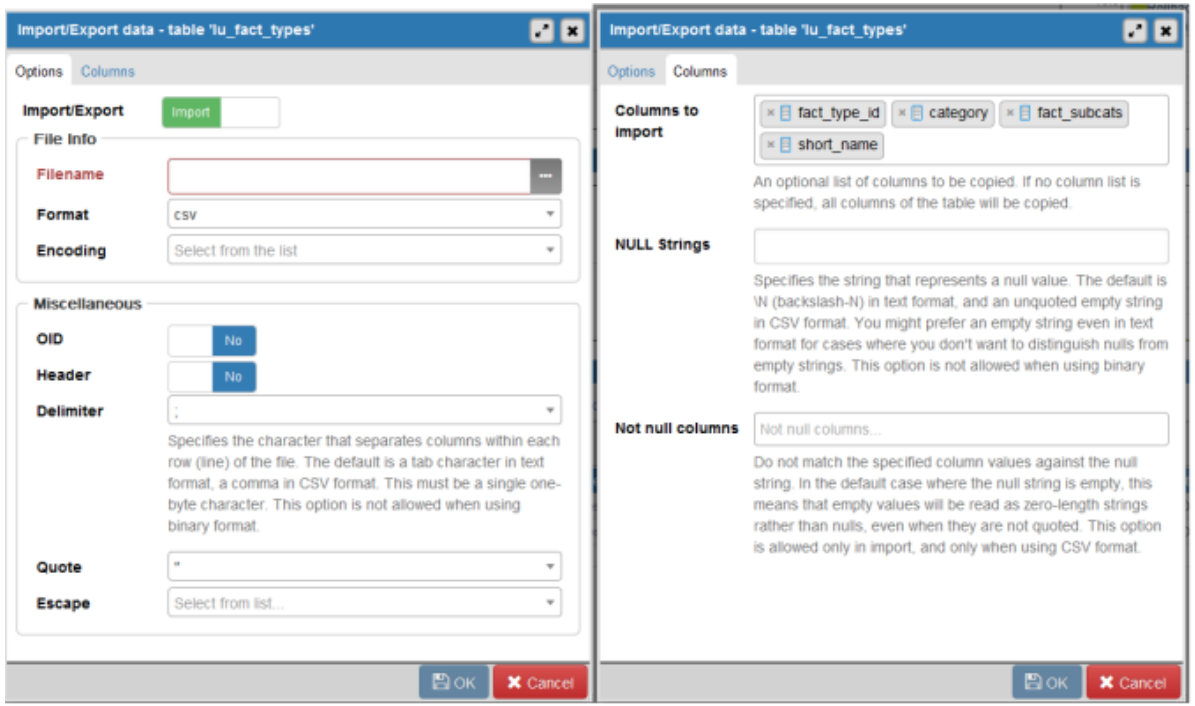
Creating Database Assets and Setting Privileges

- Creating databases and other database assets.
- Privilege management
  - access from the Tools→Grant Wizard menu
  - right-click the schema or database, select Properties, and then go to the Default Privileges tab



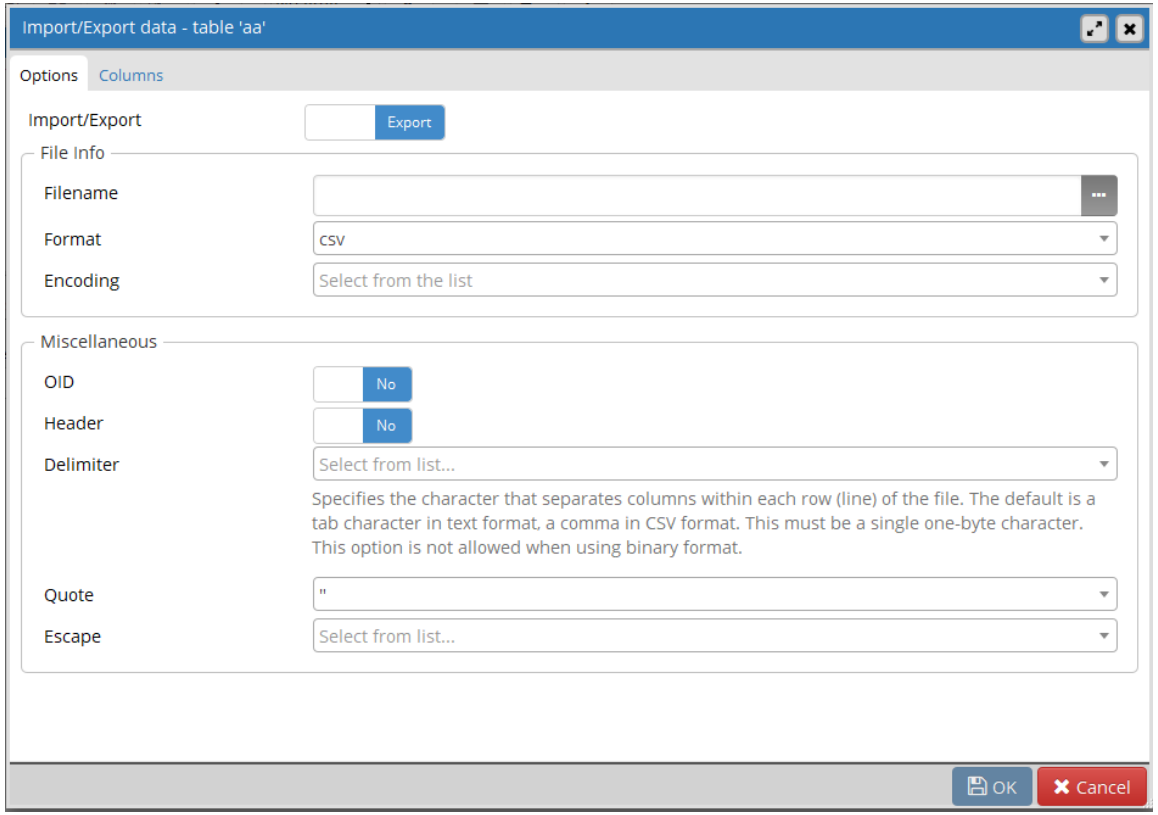
Import and Export

- Like psql, pgAdmin allows you to import and export text files.
  - right-click the table you want to import/export data to



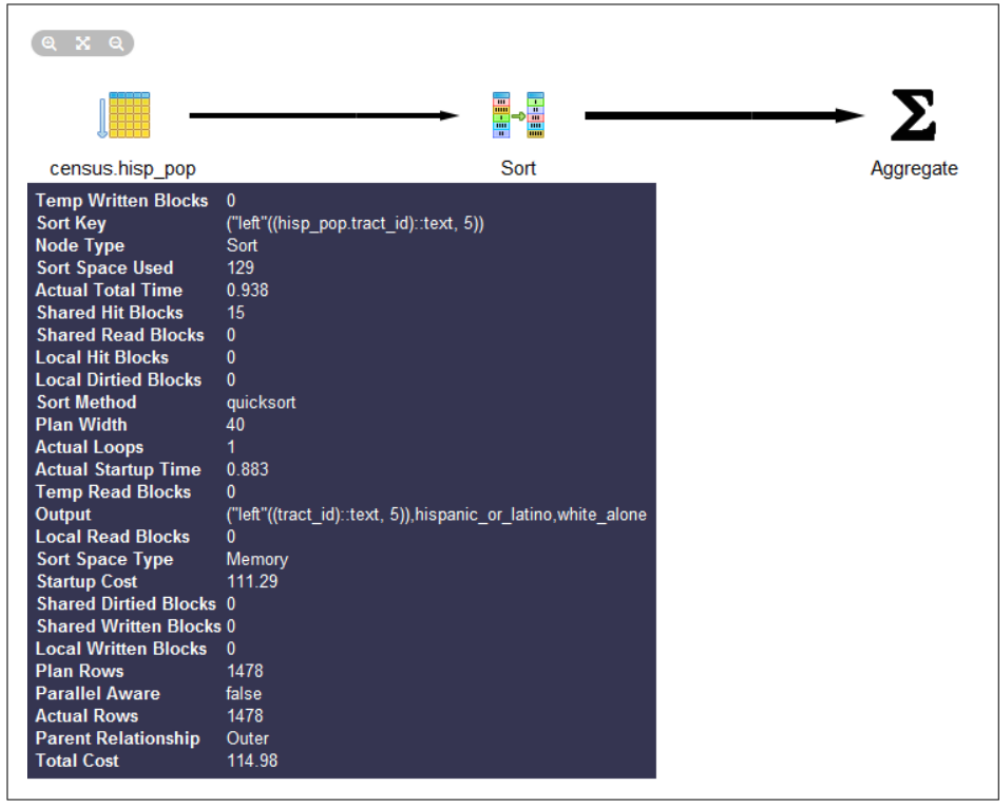
Backing up an entire database

- pgAdmin offers a graphical interface to pg_dump and pg_restore

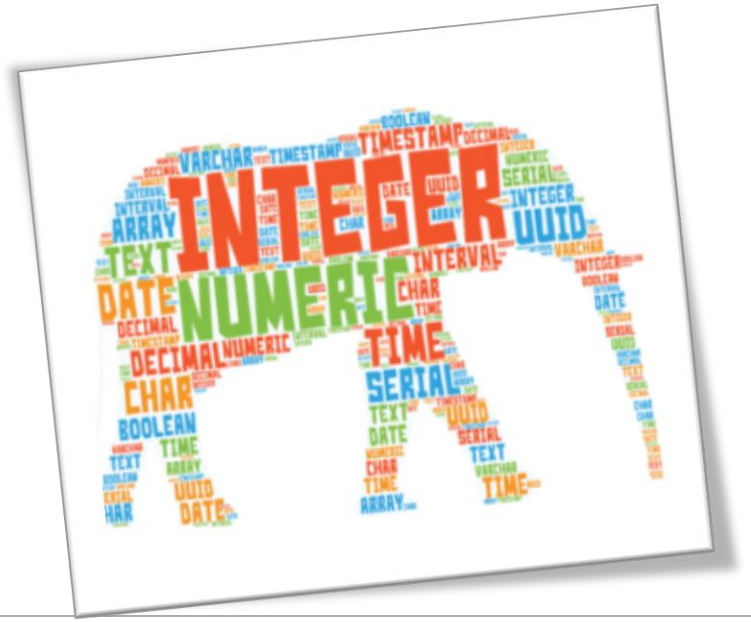


Graphical Explain

- One of the great gems in pgAdmin is its at-a-glance graphical explain of the query plan.



Advanced Database



COMP412

CHAPTER 05: DATA TYPES

Introduction

- PostgreSQL supports the workhorse data types of any database: numerics, strings, dates, times, and booleans.

- PostgreSQL adds support also for arrays, time zone, datetimes, time intervals, ranges, JSON, XML, and many more.

- If that's not enough, you can invent custom types.

Numerics

Numeric

- You will find your everyday integers, decimals, and floating-point numbers in PostgreSQL.
  - Ex: NUMERIC (precision, scale)

Parameter	Description
Numeric	It is a keyword, which is used to store the numeric numbers.
Precision	It is the total number of digits
Scale	It is several digits in terms of the fraction part.

  - Suppose we have the number 2356.78. In this number, the precision is 6, and the scale is 2.
  - Of the numeric types, we want to discuss serial data types.

Serials

- Serial and bigserial are auto-incrementing integers often used as primary keys of tables in which a natural key is not apparent.

- When you create a table and specify a column as serial, PostgreSQL first creates an integer column and then creates a sequence object named table_name_column_name_seq located in the same schema as the table.
  - It then sets the default of the new integer column to read its value from the sequence.

- If you drop the column, PostgreSQL also drops the companion sequence object.

Sequence type

- You can inspect and edit the sequences using SQL with the ALTER SEQUENCE command or using PGAdmin.
- You can set the current value, boundary values (both the upper and lower bounds), and even how many numbers to increment each time.
- Though decrementing is rare, you can do it by setting the increment value to a negative number.

Sequence type

- Because sequences are independent database assets, you can create them separately from a table using the CREATE SEQUENCE command, and you can use the same sequence across multiple tables.
- The cross-table sharing of the same sequence comes in handy when you're assigning a universal key in your database.

```
CREATE SEQUENCE s START 1;
CREATE TABLE stuff(id bigint DEFAULT nextval('s') PRIMARY KEY, name text);
```

Textuals

Textuals

- There are three primitive textual types in PostgreSQL:
  - character (abbreviable as char),
  - character varying (abbreviable as varchar),
  - text.

Textuals

- Use char only when the values stored are fixed length, such as
  - postal codes, phone numbers, and Social Security numbers in the US.
- If your value is under the length specified, PostgreSQL automatically adds spaces to the end.
- When compared with varchar or text, the right-padding takes up more superfluous storage, but you get the assurance of an invariable length.
- There is absolutely no speed performance benefit of using char over varchar or text and char will always take up more disk space.

Textuals

- Use character varying to store strings with varying length.
- When defining varchar columns, you should specify the maximum length of a varchar.
- Text is the most generic of the textual data types. With text, you cannot specify a maximum length.
- The max length modifier for varchar is optional. Without it, varchar behaves almost identically to text.
- Both varchar and text have a maximum storage of 1G for each value

String Functions

- Common string manipulations are
  - padding (lpad, rpad),
    - The PostgreSQL LPAD() function pads a string on the left to a specified length with a sequence of characters.

```
SELECT LPAD('PostgreSQL',15,'*');
*****PostgreSQL
```

String Functions

- Common string manipulations are
  - trimming whitespace (rtrim, ltrim, trim, btrim)
    - The PostgreSQL rtrim function is used to remove spaces(if no character(s) is provided as trimming_text)

```
SELECT LTRIM('enterprise', 'e');

SELECT RTRIM('enterprise', 'e');

SELECT BTRIM('enterprise', 'e');
```

ltrim
nterprise

rtrim
enterpris

btrim
nterpris

Temporals

Temporals

- In addition to the usual dates and times types, PostgreSQL supports time zones, enabling the automatic handling of daylight saving time (DST) conversions by region.
- At last count, PostgreSQL has nine temporal data types. If
- a type is time zone–aware, the time changes if you change your server’s time zone.

Temporals

- The main types are:
  - Date: Stores the month, day, and year, with no time zone
  - Time: Stores hours, minutes, and seconds with no awareness of time zone
  - Timestamp: Stores both calendar dates and time (hours, minutes, seconds) but does not care about the time zone.
  - Timestamptz: A time zone–aware date and time data type. Internally, timestamptz is stored in Coordinated Universal Time (UTC), but its display defaults to the time zone of the server, the service config, the database, the user, or the session.
    - If you input a timestamp with no time zone and cast it to one with the time zone, PostgreSQL assumes the default time zone in effect. If you don't set your time zone in postgresql.conf, the server's default takes effect. This means that if you change your server's time zone, you'll see all the displayed times change after the PostgreSQL server restarts.
  - Timetz: It is time zone–aware but does not store the date.

Time Zones: What They Are and Are Not

- If you save 2022-11-03 05:32:00+02 (+2 being the Beirut offset from UTC), PostgreSQL internally thinks like this:
- Calculate the UTC time for 2022-11-03 05:32:00+02. This is 2022-11-03 03:32:00-0.
- Store the value 2022-11-03 03:32:00.
- So PostgreSQL doesn’t store the time zone, but uses it only to convert the datetime to UTC before storage.
- select date with timezone
  - select some_date at time zone 'Europe/Berlin' from some_table
  - SELECT '2012-02-28 10:00 PM America/Los_Angeles'::timestamptz AT TIME ZONE 'Europe/Paris';

Datetime Operators and Functions

- With intervals, we can add and subtract timestamp data simply by using the arithmetic operators we're intimately familiar with.
  - The addition operator (+) adds an interval to a timestamp:
  - SELECT '2012-02-10 11:00 PM'::timestamp + interval '1 hour';

  - The subtraction operator (-) subtracts an interval from a temporal type:
  - SELECT '2012-02-10 11:00 PM'::timestamptz - interval '1 hour';

  - OVERLAPS, demonstrated in Example 5-12, returns true if two temporal ranges overlap.

```
SELECT
('2012-10-25 10:00 AM'::timestamp, '2012-10-25 2:00 PM'::timestamp)
OVERLAPS
('2012-10-25 11:00 AM'::timestamp,'2012-10-26 2:00 PM'::timestamp)
```

Arrays

Arrays

- PostgreSQL allows a table column to contain multi-dimensional arrays that can be of any built-in or user-defined data type.
- Arrays have an advantage over large plain text fields in that data remains in a discreet and addressable form.
- Every data type in Postgres has a companion array type.
- If you define your own data type, PostgreSQL creates a corresponding array type in the background for you.
  - For example, integer has an integer array type integer[], character has a character array type character[], …

Array Constructors

- The most rudimentary way to create an array is to type the elements:

```
SELECT ARRAY[2001, 2002, 2003] As yrs;
```

Create

- Assuming that you are going to create a table of people that contains a column called aliases that is a text array of various other names for a person.
- Creating an array column in PostgreSQL is as easy as appending the [ ] operator after your data type in a create statement such as this:

```
CREATE TABLE people
(   id serial,
   full_name text,
   aliases text[],
  CONSTRAINT people_id_pkey PRIMARY KEY (id) );
```

Insert

- Inserting data into a table with an array column can be as easy as:

  ```
  insert into people (full_name, aliases) values ('Abraham Lincoln', {"President Lincoln", "Honest Abe"}');
  ```

- Doing an insert into an array column requires using the **{ }** operator to specify the array and using double quotes (“) and a comma to delimit the separate values going into the array.

- We can see how the data went into our table by running:

  ```
  select * from people;
  ```

Access specific elements

- Our data are indeed separate inside of PostgreSQL looking at the *aliases* column.
  - You can access specific elements in the array by also using the **[ ]** notation in selects.

```
select full_name, aliases[1] from people;
update people set aliases[1] = 'President Abraham Lincoln' where id = 1;
```

- If you already know how many elements are in the array, you can add to it using standard syntax.

```
update people set aliases[3] = 'President Lincoln' where id = 1;
```

- Note that array notation in PostgreSQL starts at element one and not zero

Access specific elements

- Using the array functions that PostgreSQL provides with varying numbers of array sizes.
- For example, if we want to append a value to an array we would use a query that makes use of the **array_append** function.

```
update people set aliases = array_append(aliases, 'Not Abe Vigoda') where id = 1;
```

- Using the array_remove function to remove an element.

```
update people set aliases = array_remove(aliases, 'Not Abe Vigoda') where id = 1;
```

-

Searching

- PostgreSQL provides the contains operators (@>, <@) for this type of search.
  - SELECT '{1,2,3}'::int[] @> '{3,2}'::int[] AS contains;
  - SELECT '{1,2,3}'::int[] <@ '{3,2}'::int[] AS contained_by;
  - select * from people where aliases @> '{"Prince Harry"}';

JSON

JSON

- PostgreSQL provides JSON (JavaScript Object Notation) and many support functions. JSON has become the most popular data interchange format for web applications.
- Version 9.3 significantly beefed up JSON support with new functions for extracting, editing, and casting to other data types.
- Version 9.4 introduced the JSONB data type, a binary form of JSON that can also take advantage of indexes.
- Version 9.5 introduced more functions for jsonb, including functions for setting elements in a jsonb object.
- Version 9.6 introduced the jsonb_insert function for inserting elements into an existing jsonb array or adding a new key value.

Inserting JSON Data

- To create a table to store JSON, define a column as a json type:

  CREATE TABLE persons (id serial PRIMARY KEY, person json);

- PostgreSQL automatically validates the input to make sure what you are adding is valid JSON.

```
INSERT INTO persons (person)
VALUES (
    '{
        "name":"Sonia",
        "spouse":
        {
            "name":"Alex",
            "parents":
            {
                "father":"Rafael",
                "mother":"Ofelia"
```

Querying JSON

- The easiest way to traverse the hierarchy of a JSON object is by using pointer symbols.
- SELECT person->'name' FROM persons;
- SELECT person->'spouse'->'parents'->'father' FROM persons;

- You can also write the query using a path array as in the following example:
- SELECT person#>array['spouse','parents','father'] FROM persons;

Querying JSON

- To penetrate JSON arrays, specify the array index. JSON arrays is zero-indexed, unlike PostgreSQL arrays, whose indexes start at 1.
- SELECT person->'children'->0->'name' FROM persons;
- And the path array equivalent:
- SELECT person#>array['children','0','name'] FROM persons;

Querying JSON

- PostgreSQL provides two native operators -> and ->> to help you query JSON data. The operator -> returns JSON object field as JSON. The operator ->> **returns JSON object field as text**.
- All queries in the prior examples return the value as JSON primitives (numbers,strings, booleans).
- To return the text representation, add another greater-than sign as in the following examples:
  - SELECT person->'spouse'->'parents'->>'father' FROM persons;
  - SELECT person#>>array['children','0','name'] FROM persons;

Querying JSON

- SELECT id, person->'children'
- FROM public.persons
- where person->'children'->1->>'name' = 'Azaleah'

json_array_elements to expand JSON array

- The json_array_elements function takes a JSON array and returns each element of the array as a separate row.

- SELECT json_array_elements(person->'children')->>'name' As name FROM persons;

Binary JSON: jsonb

It is handled through the same operators as those for the json type, and similarly named functions, plus several additional ones.

jsonb performance is much better than json performance because jsonb doesn’t need to be reparsed during operations.

Basics of PostgreSQL's JSONB data type

- Creating a JSONB column
  - create table sales (
  - id serial not null primary key,
  - info jsonb not null
  -);
- Inserting a JSON document
  - insert into sales values (1, '{"name": "Alice", "paying": true, "tags": ["admin"]}');

Update jsonb

- Updating by inserting a whole document:

```
update sales set info = '{" name " : "Bob", "paying" : false, "tags" : []}';
```

- Updating by adding a key:
  - Use the || operator to concatenate existing data with new data. The operator will either update or insert the key to the existing document.

```
update sales set info = info || '{"country": "Canada"}';
```

- Update by removing a key:
  - Use the - operator to remove a key from the document.

```
update sales set info = info - 'country';
```

Jsonb operators

- In addition to the operators supported by json, jsonb has additional comparator operators such as equality (=), contains (@>), contained (<@), ...
- JSONB Containment with @>
- select info from sales where info::jsonb @> '{"name":"Bob"}'

Advanced Database

COMP412

CHAPTER 05: FULL TEXT SEARCH

Introduction

Real life scenario

- Let’s say you have an IT store and the user searched for ‘**desktop and laptop**’.
- No problem there. But do you have a product whose title says ‘**XXX Desktop and Laptop**’ exactly as the user searched for?
- Most probably no! The search would fail to show any relevant results.
- The user probably wanted to list all the computers in your store that he or she can use as a desktop and a laptop, most likely a **convertible tablet**.
- Since the search failed to show any result to the user, the user may think you’re out of stock or you don’t have it in your IT store.

Real life scenario

- But you do have many convertible tablets that can be used as a desktop and a laptop in your store's database.
- If the users can't find it, you won't get any sales.
- You do want to your website to list all the convertible computers you have in stock when users do a search query like that.

This is where Full Text Search comes into play.

Demo

- Create Store Database

```
createdb it_store
```

- Create products table

```
CREATE TABLE public.products
    (id serial primary key,
    title character varying(200) NOT NULL,
    description text NOT NULL )
```

Demo

- Insert some products into the **products** table.
  - INSERT INTO public.products(title, description) VALUES ('HP ProDesk 400 G3 4GB RAM 1TB HDD', 'This is great and cheap desktop computer');
  - INSERT INTO public.products(title, description) VALUES ('ASUS UX303UB 8GB RAM 256GB SSD', 'This is a great think and light laptop computer. it has good sound quality');
  - INSERT INTO public.products(title, description) VALUES ('Microsoft Surface Book 2 1TB SSD', 'this is a great convertible laptop computer, you can use it as a laptop and desktop as well if you want.');

Demo

- In PostgreSQL, you use two functions to perform Full Text Search. They are **to_tsvector()** and **to_tsquery()**.

- **to_tsvector()** function breaks up the input string and creates tokens out of it, which is then used to perform Full Text Search using the **to_tsquery()** function.

```
select to_tsvector('I love linux. Linux IS a great operating system.');
```

- You can use **to_tsquery()** function as follows:

```
SELECT fieldNames FROM tableName
    WHERE to_tsvector(fieldName) @@ to_tsquery(conditions)
```

Demo

- If you're looking for 'laptop and desktop', you should put **'laptop & desktop'** to **to_tsquery()** function.
  - For 'laptop or desktop', the condition should be **'laptop | desktop'**.

```
SELECT id, title, description FROM public.products where
to_tsvector(description) @@ to_tsquery('desktop & laptop')
```

- Let's take a look at another example. The user is looking for all the laptops in your store but not the convertible ones. The user query may be 'not convertible laptops'. The condition of to_tsquery() function may be '!convertible & laptops'

```
to_tsquery('!convertible & laptops')
```

FTS Configurations

- Most PostgreSQL distributions come packaged with over 10 FTS configurations. All these are installed in the pg_catalog schema.
- To see the listing of installed FTS configurations, run the query

  ```
  SELECT cfgname FROM pg_ts_config;
  ```

- You're not limited to built-in FTS configurations. You can create your own

FTS Configurations

- Install the popular hunspell configuration.
  - Start by downloading hunspell configurations from hunspell_dicts.
  - Copy **en_us.affix** and **en_us.dict** to your PostgreSQL installation directory **share/tsearch_data**.
  - Copy the **hunspell_en_us--*.sql** and **hunspell_en_us.control** files to your Post-greSQL installation directory **share/extension** folder.
  - Next, run: CREATE EXTENSION hunspell_en_us SCHEMA pg_catalog;

Default configuration

- Not sure which configuration is the default? Run:
  - SHOW default_text_search_config;

- To replace the default with another, run:
  - ALTER DATABASE postgresql_book SET default_text_search_config = 'pg_catalog.english';

TSVector

- To create a **tsvector** from text, you must specify the FTS configuration to use.
- The vectorization reduces the original text to a set of word skeletons, referred to as lexemes, by removing stop words.
- For each lexeme, the TSVector records where in the original text it appears.
- The more frequently a lexeme appears, the higher the weight. Each lexeme therefore is imbued with at least one position, much like a vector in the physical sense.
- Use the **to_tsvector** function to vectorize a text. This function will resort to the default FTS configuration unless you specify another.

Example: TSVector derived from different FTS configurations

- select to_tsvector('Just dancing in the rain. I like to dance.'::text) ;
- select to_tsvector('english','Just dancing in the rain. I like to dance.'::text) ;
- select to_tsvector('english_hunspell','Just dancing in the rain. I like to dance.'::text) ;
- select to_tsvector('simple','Just dancing in the rain. I like to dance.'::text)

TSVector

- English and Hunspell configurations remove all stop words, such as just and to.

- English and Hunspell also convert words to their normalized form as dictated by their dictionaries, so dancing becomes danc and dance, respectively.

- The to_tsvector function returns where each lexeme appears in the text. So, for example, 'danc':2,9 means that dancing and dance appear as the second and the ninth words.

Add FTS to the database

- To incorporate FTS into your database, add a tsvector column to your table.
  - You then either schedule the tsvector column to be updated regularly, or
  - add a trigger to the table so that whenever relevant fields update, the tsvector field recomputes.

Create table with tsvector column

```
CREATE TABLE film1 (
    film_id1 integer DEFAULT nextval('film_film_id_seq'::regclass) NOT NULL,
    title character varying(255) NOT NULL,
    description text,
    fulltext tsvector NOT NULL
);
```

```
UPDATE film1 SET fulltext = setweight(to_tsvector(COALESCE(title,'')),'A')
                                                ||
setweight(to_tsvector(COALESCE(description,'')),'B');
```

Setweight and (||) operator

- We've introduced two new constructs, the setweight function and the concatenation operator (||), to tsvector.
- To distinguish the relative importance of different lexemes, you could assign a weight to each. The weights must be A, B, C, or D, with A ranking highest in importance.
- TSVectors can be formed from other tsvectors using the concatenation (||) operator.
  - We used it here to combine the title and description into a single tsvector.
  - This way when we search, we have to contend with only a single column.

Trigger to automatically update tsvector

- Should data change in one of the basis columns forming the tsvector, you must revectorize.
- To avoid having to manually run to_tsvector every time data changes, create a trigger that responds to updates.

```
CREATE TRIGGER trig_tsv_film_iu
BEFORE INSERT OR UPDATE OF title, description ON film FOR EACH ROW
EXECUTE PROCEDURE tsvector_update_trigger(fts,'pg_catalog.english',
title,description);
```

Create index for tsvector

- To speed up searches, we add a GIN index on the tsvector column.
  - CREATE INDEX ix_film1_fts_gin ON film1 USING gin (fulltext);

TSQueries

- We have already seen how to vectorize the searched text to create tsvector columns.
  - We now show you how to vectorize the search terms.
- FTS refers to vectorized search terms as tsqueries,
- PostgreSQL offers several functions that will convert plain-text search terms to tsqueries: to_tsquery, plainto_tsquery, and phraseto_tsquery.
  - The latter takes the ordering of words in the search term into consideration.

to_tsquery

- Using the to_tsquery functions against two configurations: the default English configuration and the Hunspell configuration.
  - SELECT to_tsquery('business & analytics');
  - SELECT to_tsquery('english_hunspell','business & analytics');
  - The and operator (&) means that both words must appear in the searched text.
  - The or operator (|) means one or both of the words must appear in the searched text.

plainto_tsquery

- A slight variant of to_tsquery is plain_totsquery. This function automatically inserts the and operator between words for you, saving you a few key clicks.
  - SELECT plainto_tsquery('business analytics');

phraseto_tsquery

- to_tsquery and plainto_tsquery look only at words, not their sequence.
- The phraseto_tsquery vectorizes the words, inserting the distance operator between the words.
- This means that the searched text must contain the words business and analytics in that order, upgrading a word search to a phrase search.
  - SELECT phraseto_tsquery('business analytics');
  - SELECT phraseto_tsquery('english_hunspell','business analytics');

Combining tsqueries

- SELECT plainto_tsquery('business analyst') || phraseto_tsquery('data scientist');
- SELECT plainto_tsquery('business analyst') && phraseto_tsquery('data scientist');

Using Full Text Search

- We have created a tsvector from our text; we have created a tsquery from our search terms.
  - Now, we can perform an FTS. We do so by using the @@ operator.
  - finds all films with a title or description containing the word hunter and either the word scientist, or the word chef, or both.

```
SELECT title , description
FROM film
WHERE fulltext @@ to_tsquery('hunter  & (scientist | chef)') AND title > '';
```

Using Full Text Search

- you can specify the proximity and order of words.

```
SELECT title , description
FROM film
WHERE fulltext @@ to_tsquery('hunter <4> (scientist | chef)') AND title > '';
```

Ranking Results

Advanced Database

Sample Midterm Exam

Questions & Complete Solutions

Study Material for Open-Book Exam

Section 1: Managing Database Connections

Question 1a: List Connections and PIDs

How can you retrieve a list of recent connections and process IDs (PIDs) in PostgreSQL?

Answer:

Use the **pg_stat_activity** system view to retrieve active connections and their PIDs. This view shows all current connections, queries, and session information.

```
-- Basic query to see all connections
SELECT pid, usename, datname, state, query_start, query
FROM pg_stat_activity;

-- See only active connections (not idle)
SELECT pid, usename, datname, application_name,
       client_addr, state, query
FROM pg_stat_activity
WHERE state = 'active';

-- See connections for a specific database
SELECT pid, usename, state, query
FROM pg_stat_activity
WHERE datname = 'your_database_name';
```

Key columns: pid (process ID), usename (user), datname (database), state (active/idle), query (current/last query), query_start (when query began).

Question 1b: Cancel and Terminate Connections

Assume that you need to terminate a specific connection to the PostgreSQL database. How can you cancel active queries on a connection with a given PID, and how can you terminate the connection itself?

Answer:

PostgreSQL provides two functions: **pg_cancel_backend(pid)** for canceling queries and **pg_terminate_backend(pid)** for forcefully closing connections.

```
-- Step 1: Find the PID you want to target
SELECT pid, usename, datname, state, query
FROM pg_stat_activity
WHERE usename = 'problematic_user';

-- Step 2: Cancel the running query (gentle approach)
SELECT pg_cancel_backend(12345);  -- Replace 12345 with actual PID
-- This sends SIGINT, allowing graceful query cancellation
-- Returns TRUE if successful

-- Step 3: Terminate the connection (forceful approach)
SELECT pg_terminate_backend(12345);  -- Replace 12345 with actual PID
-- This sends SIGTERM, immediately closing the connection
-- Returns TRUE if successful

-- Check if the connection is gone
SELECT pid FROM pg_stat_activity WHERE pid = 12345;
-- Should return 0 rows if terminated
```

Key differences: pg_cancel_backend() lets the query stop gracefully (like Ctrl+C), while pg_terminate_backend() forcefully kills the entire connection. Try cancel first, then terminate if needed.

Question 1c: Kill All Connections for a Role

How can you kill all connections belonging to a specific role in PostgreSQL?

Answer:

Use a query that selects all PIDs for the role and terminates them in a loop or with a subquery. This is useful when removing a role or maintenance tasks.

```
-- Method 1: Using a subquery to terminate all connections for 'dev_user'
SELECT pg_terminate_backend(pid)
FROM pg_stat_activity
WHERE usename = 'dev_user'
  AND pid <> pg_backend_pid();  -- Don't kill your own connection!

-- Method 2: Terminate all connections to a specific database
SELECT pg_terminate_backend(pid)
FROM pg_stat_activity
WHERE datname = 'sample_db'
  AND pid <> pg_backend_pid();

-- Verify all connections are terminated
SELECT pid, usename, datname, state
FROM pg_stat_activity
WHERE usename = 'dev_user';
-- Should return 0 rows

-- Common use case: Before dropping a database
SELECT pg_terminate_backend(pid)
FROM pg_stat_activity
WHERE datname = 'old_database';

DROP DATABASE old_database;
```

Important: Always use 'pid <> pg_backend_pid()' to avoid terminating your own connection! The function pg_backend_pid() returns your current session's PID.

Section 2: Users, Roles & Permissions

Question 2a: Create User with Expiration

Create a new user in the database named 'dev_user' with a password of your choice, but ensure that the user is valid only until a specific date. Grant this user permission to read and write to a specific table named 'employee_data'.

Answer:

Use CREATE USER with VALID UNTIL clause for time-limited access, then grant specific table privileges using GRANT statement.

```
-- Step 1: Create user with password and expiration date
CREATE USER dev_user
WITH PASSWORD 'SecurePass123!'
VALID UNTIL '2025-12-31';

-- Alternative: Set expiration to exactly 6 months from now
CREATE USER dev_user
WITH PASSWORD 'SecurePass123!'
VALID UNTIL '2025-05-09';  -- Replace with desired date

-- Step 2: Grant read (SELECT) permission on employee_data table
GRANT SELECT ON employee_data TO dev_user;

-- Step 3: Grant write (INSERT, UPDATE, DELETE) permissions
GRANT INSERT, UPDATE, DELETE ON employee_data TO dev_user;

-- Or grant all table privileges at once:
GRANT SELECT, INSERT, UPDATE, DELETE ON employee_data TO dev_user;

-- Verify the user and permissions
SELECT usename, valuntil FROM pg_user WHERE usename = 'dev_user';

-- Verify table privileges
SELECT grantee, privilege_type
FROM information_schema.table_privileges
WHERE table_name = 'employee_data' AND grantee = 'dev_user';
```

Explanation: VALID UNTIL enforces automatic expiration - the user cannot login after that date. The GRANT statement gives specific privileges on the table. For read+write, you need SELECT (read), INSERT (add), UPDATE (modify), DELETE (remove).

Question 2b: Create Role with Inheritance

Create a new role in the database named 'dev_team' and assign the 'dev_user' to this role. Grant the 'dev_team' role permission to access all tables in the database, but ensure that this permission is inherited by all members of the 'dev_team' role.

Answer:

Create a role with INHERIT privilege (default), grant it database-wide permissions, then assign users to the role using GRANT role TO user.

```
-- Step 1: Create the dev_team role with inheritance enabled
CREATE ROLE dev_team WITH INHERIT;
-- INHERIT is default, meaning members automatically get role's privileges

-- Step 2: Grant access to all tables in a schema (e.g., public schema)
GRANT SELECT, INSERT, UPDATE, DELETE
ON ALL TABLES IN SCHEMA public
TO dev_team;

-- Step 3: Also grant privileges on future tables (important!)
ALTER DEFAULT PRIVILEGES IN SCHEMA public
GRANT SELECT, INSERT, UPDATE, DELETE ON TABLES TO dev_team;

-- Step 4: Assign dev_user to the dev_team role
GRANT dev_team TO dev_user;

-- Verify role membership
SELECT
    r.rolname as role_name,
    m.rolname as member_name
FROM pg_roles r
JOIN pg_auth_members ON r.oid = pg_auth_members.roleid
JOIN pg_roles m ON m.oid = pg_auth_members.member
WHERE r.rolname = 'dev_team';

-- Verify dev_user inherits permissions
SET ROLE dev_user;
SELECT current_user, session_user;
-- Should show dev_user has access to all tables
```

Key concepts: Roles can be groups. INHERIT (default) means members automatically get the role's privileges. ALTER DEFAULT PRIVILEGES ensures future tables also get the permissions. Use 'GRANT role TO user' for membership.

Question 2c: Add New Developer & Test Access

Assume that a new developer has joined the team and needs access to the database. Create a new user for this developer with a password of your choice, and add this user to the 'dev_team' role. Test the user's access to the 'employee_data'.

Answer:

Create the new user with LOGIN privilege, grant them the dev_team role membership, then test by connecting as that user and querying employee_data.

```
-- Step 1: Create new developer user with login capability
CREATE USER new_developer
WITH PASSWORD 'DevPass456!'
LOGIN;

-- Step 2: Add new developer to dev_team role
GRANT dev_team TO new_developer;

-- Step 3: Verify role membership
SELECT
    r.rolname as role_name,
    m.rolname as member_name
FROM pg_roles r
JOIN pg_auth_members ON r.oid = pg_auth_members.roleid
JOIN pg_roles m ON m.oid = pg_auth_members.member
WHERE r.rolname = 'dev_team';

-- Step 4: Test access by switching to new user
SET ROLE new_developer;

-- Test SELECT access
SELECT * FROM employee_data LIMIT 5;

-- Test INSERT access
INSERT INTO employee_data (name, email, department)
VALUES ('Test User', 'test@example.com', 'Engineering');

-- Test UPDATE access
UPDATE employee_data
SET department = 'DevOps'
WHERE name = 'Test User';

-- Test DELETE access
DELETE FROM employee_data WHERE name = 'Test User';

-- Reset to original role
RESET ROLE;

-- Alternatively, test by connecting from command line:
-- psql -U new_developer -d your_database -c "SELECT * FROM employee_data;
```

Testing note: Use SET ROLE to test within psql, or connect with a new psql session using 'psql -U new_developer'. The user should inherit all dev_team permissions automatically due to the INHERIT property.

Section 3: Tables, Data Types & Arrays

Question 3a: Create Database and Table

Create a database called 'sample_db' with a single table called 'sample_table'. The table should have: id (serial auto-increment), name (varchar 50), age (numeric 4,2), description (text).

Answer:

```
-- Step 1: Create the database
CREATE DATABASE sample_db;

-- Step 2: Connect to the new database
\c sample_db

-- Step 3: Create the table with specified columns
CREATE TABLE sample_table (
    id SERIAL PRIMARY KEY,              -- Auto-incrementing integer
    name VARCHAR(50),                    -- Variable character, max 50 chars
    age NUMERIC(4, 2),                  -- Precision 4, scale 2 (e.g., 99.99)
    description TEXT                     -- Unlimited text
);

-- Verify table structure
\d sample_table

-- Alternative: View table info
SELECT column_name, data_type, character_maximum_length,
       numeric_precision, numeric_scale
FROM information_schema.columns
WHERE table_name = 'sample_table';
```

Data type notes: SERIAL = auto-incrementing integer (shorthand for sequence). VARCHAR(50) = variable length, max 50. NUMERIC(4,2) = max 99.99 (4 digits total, 2 after decimal). TEXT = unlimited length text.

Question 3b: Load Data from CSV

Populate the 'sample_table' with data from CSV file 'sample_data.csv' containing: John Smith (25), Jane Doe (33), Bob Johnson (45) with descriptions.

Answer:

```
-- First, create the CSV file (sample_data.csv):
-- name,age,description
-- John Smith,25,"Lorem ipsum dolor sit amet, consectetur adipiscing elit."
-- Jane Doe,33,"Nullam imperdiet massa ac elementum laoreet."
-- Bob Johnson,45,"Pellentesque euismod quam non mi rutrum, non malesuada magna malesuada."

-- Method 1: Using COPY command (requires superuser or appropriate privileges)
COPY sample_table(name, age, description)
FROM '/path/to/sample_data.csv'
WITH (FORMAT csv, HEADER true);

-- Method 2: Using \copy in psql (runs as client, works without superuser)
\copy sample_table(name, age, description) FROM 'sample_data.csv' WITH CSV HEADER

-- Method 3: Manual INSERT if CSV not available
INSERT INTO sample_table (name, age, description) VALUES
('John Smith', 25, 'Lorem ipsum dolor sit amet, consectetur adipiscing elit.'),
('Jane Doe', 33, 'Nullam imperdiet massa ac elementum laoreet.'),
('Bob Johnson', 45, 'Pellentesque euismod quam non mi rutrum, non malesuada magna malesuada.');

-- Verify data loaded
SELECT * FROM sample_table;

-- Check row count
SELECT COUNT(*) FROM sample_table;  -- Should return 3
```

Important: Use \copy (with backslash) in psql - it runs client-side and doesn't require superuser. COPY (without backslash) runs server-side and requires file access. Note: id column auto-fills due to SERIAL.

Question 3c & 3d: String Functions

c) Show all data and use string functions to pad the 'name' column with spaces on the left so that all values are exactly 20 characters long. d) Show all data and use string functions to trim leading and trailing whitespace from the 'description' column.

Answer:

```
-- Question 3c: Left-pad name to 20 characters
SELECT
    id,
    LPAD(name, 20, ' ') AS padded_name,  -- Left pad with spaces
    age,
    description
FROM sample_table;

-- Output will look like:
-- "          John Smith"  (10 spaces + 10 chars = 20 total)
-- "            Jane Doe"  (12 spaces + 8 chars = 20 total)
-- "         Bob Johnson"  (9 spaces + 11 chars = 20 total)

-- Alternative: Right-pad instead (RPAD)
SELECT RPAD(name, 20, ' ') AS right_padded FROM sample_table;

-- Question 3d: Trim whitespace from description
SELECT
    id,
    name,
    age,
    TRIM(description) AS trimmed_description,     -- Remove both leading & trailing
    LTRIM(description) AS left_trimmed,           -- Remove only leading
    RTRIM(description) AS right_trimmed           -- Remove only trailing
FROM sample_table;

-- Combined: Both padding and trimming
SELECT
    id,
    LPAD(name, 20, ' ') AS padded_name,
    age,
    TRIM(description) AS clean_description
FROM sample_table;

-- Other useful string functions:
-- LENGTH(name) - get string length
-- UPPER(name), LOWER(name) - change case
-- SUBSTRING(name, 1, 4) - extract substring
```

Function summary: LPAD(string, length, fill) pads on left. RPAD pads on right. TRIM removes spaces from both ends. LTRIM removes left spaces, RTRIM removes right spaces.

Question 3e & 3f: Add Date/Timestamp Columns

e) Add a new column called 'birthdate' of type 'date'. f) Add a new column called 'last_login' of type 'timestamp with time zone'.

Answer:

```
-- Question 3e: Add birthdate column (DATE type)
ALTER TABLE sample_table
ADD COLUMN birthdate DATE;

-- Question 3f: Add last_login column (TIMESTAMP WITH TIME ZONE)
ALTER TABLE sample_table
ADD COLUMN last_login TIMESTAMP WITH TIME ZONE;

-- Verify new columns
\d sample_table

-- Populate with sample data
UPDATE sample_table
SET birthdate = CURRENT_DATE - (age * 365)::INTEGER,  -- Approximate birthdate
    last_login = CURRENT_TIMESTAMP
WHERE id IN (1, 2, 3);

-- More precise example: Set specific dates
UPDATE sample_table SET birthdate = '2000-01-15',
                        last_login = '2025-11-08 14:30:00+00'
WHERE name = 'John Smith';

-- View updated data
SELECT id, name, age, birthdate, last_login FROM sample_table;

-- Date/time functions you can use:
SELECT
    CURRENT_DATE,                          -- Today's date
    CURRENT_TIMESTAMP,                     -- Now with timezone
    NOW(),                                 -- Same as CURRENT_TIMESTAMP
    AGE(birthdate),                        -- Calculate age from birthdate
    EXTRACT(YEAR FROM birthdate) AS year   -- Extract components
FROM sample_table;
```

Data type notes: DATE stores only the date (no time). TIMESTAMP WITH TIME ZONE (or TIMESTAMPTZ) stores date, time, and timezone info - best practice for timestamps. Always use WITH TIME ZONE for timestamps!

Question 3g: Create Sequence

Create a sequence called 'sample_sequence' that starts at 100 and increments by 10.

Answer:

```
-- Create sequence starting at 100, incrementing by 10
CREATE SEQUENCE sample_sequence
    START WITH 100
    INCREMENT BY 10;

-- Get next value from sequence
SELECT nextval('sample_sequence');  -- Returns 100 (first call)
SELECT nextval('sample_sequence');  -- Returns 110 (second call)
SELECT nextval('sample_sequence');  -- Returns 120 (third call)

-- Get current value without incrementing
SELECT currval('sample_sequence');  -- Returns last value retrieved (120)

-- Set sequence to a specific value
SELECT setval('sample_sequence', 500);

-- View sequence information
SELECT * FROM sample_sequence;

-- Use sequence in a table
CREATE TABLE orders (
    order_id INTEGER DEFAULT nextval('sample_sequence'),
    order_date DATE,
    customer_name VARCHAR(100)
);

-- Insert will auto-use sequence
INSERT INTO orders (order_date, customer_name)
VALUES (CURRENT_DATE, 'Alice');
-- order_id will be next sequence value

-- View all sequences in database
SELECT sequence_name, start_value, increment_by, last_value
FROM information_schema.sequences;
```

Sequence functions: nextval() gets next value and increments. currval() returns last value without incrementing (must call nextval first in session). setval() manually sets the sequence value. Sequences are great for custom ID generation.

Question 3h & 3i: Array Columns

h) Create an array column called 'interests' that stores a list of integers. i) Update interests: John Smith = {1,3,5}, Jane Doe = {2,4}, Bob Johnson = {1,2,3,4,5}.

Answer:

```
-- Question 3h: Add array column for integer interests
ALTER TABLE sample_table
ADD COLUMN interests INTEGER[];

-- Verify column added
\d sample_table

-- Question 3i: Update interests for each person
UPDATE sample_table
SET interests = '{1,3,5}'::INTEGER[]
WHERE name = 'John Smith';

UPDATE sample_table
SET interests = '{2,4}'::INTEGER[]
WHERE name = 'Jane Doe';

UPDATE sample_table
SET interests = '{1,2,3,4,5}'::INTEGER[]
WHERE name = 'Bob Johnson';

-- Shorter syntax (without cast):
UPDATE sample_table SET interests = ARRAY[1,3,5] WHERE name = 'John Smith';

-- View all data with interests
SELECT id, name, interests FROM sample_table;

-- Array operations examples:
SELECT name,
       interests,                          -- Show array
       array_length(interests, 1) AS count, -- Count elements
       interests[1] AS first_interest,     -- Access first element (1-indexed!)
       interests[2:4] AS slice             -- Array slice
FROM sample_table;
```

Array syntax: Use '{1,2,3}'::INTEGER[] or ARRAY[1,2,3]. Arrays are 1-indexed (first element is interests[1]). Use INTEGER[] for variable-length array of integers.

Question 3j: Query Arrays - Contains 1 AND 5

Write a query that selects all data whose 'interests' column contains the values 1 and 5 (in any order).

Answer:

```
-- Method 1: Using @> operator (contains)
SELECT * FROM sample_table
WHERE interests @> ARRAY[1, 5];
-- @> means "left array contains right array"
-- Returns: John Smith {1,3,5} and Bob Johnson {1,2,3,4,5}

-- Method 2: Using && operator with subquery
SELECT * FROM sample_table
WHERE interests && ARRAY[1]        -- Contains 1
  AND interests && ARRAY[5];       -- AND contains 5

-- Method 3: Using ANY operator
SELECT * FROM sample_table
WHERE 1 = ANY(interests)
  AND 5 = ANY(interests);

-- Explanation of array operators:
-- @>  : left contains right (array contains elements)
-- <@  : left is contained by right
-- &&  : arrays overlap (have common elements)
-- =   : arrays are equal

-- Example: Test which rows contain ONLY 1 or 5
SELECT * FROM sample_table
WHERE interests <@ ARRAY[1, 5];
-- Returns: None, because all have other values too

-- Count results
SELECT COUNT(*) FROM sample_table
WHERE interests @> ARRAY[1, 5];
-- Returns: 2 (John Smith and Bob Johnson)
```

Best operator: Use @> (contains) for 'has all these elements' queries. It checks if left array contains ALL elements from right array, regardless of order. This is the most efficient and clearest solution.

Question 3k: Query Arrays - Has 2 but NOT 4

Write a query that selects all data whose 'interests' column contains the value 2 but not the value 4.

Answer:

```
-- Method 1: Using @> and NOT
SELECT * FROM sample_table
WHERE interests @> ARRAY[2]           -- Contains 2
  AND NOT (interests @> ARRAY[4]);    -- Does NOT contain 4
-- Returns: Bob Johnson {1,2,3,4,5}... wait, that has 4!
-- Actually returns: Only Bob Johnson? No - he has 4 too.
-- Correct result: None of our sample data matches!
-- (Jane has 2 AND 4, Bob has 2 AND 4, John has neither)

-- Method 2: Using ANY operator
SELECT * FROM sample_table
WHERE 2 = ANY(interests)              -- Contains 2
  AND NOT (4 = ANY(interests));       -- Does NOT contain 4

-- Let's add test data that matches:
INSERT INTO sample_table (name, age, interests)
VALUES ('Test User', 30, ARRAY[1, 2, 3, 5]);

-- Now the query returns Test User

-- Method 3: Using array_position (more verbose)
SELECT * FROM sample_table
WHERE array_position(interests, 2) IS NOT NULL    -- Has 2
  AND array_position(interests, 4) IS NULL;       -- Doesn't have 4

-- Verify with sample data:
-- John Smith: {1,3,5} - No 2 -> excluded
-- Jane Doe: {2,4} - Has 2 AND 4 -> excluded
-- Bob Johnson: {1,2,3,4,5} - Has 2 AND 4 -> excluded
-- Test User: {1,2,3,5} - Has 2, No 4 -> INCLUDED!

SELECT name, interests FROM sample_table
WHERE interests @> ARRAY[2] AND NOT (interests @> ARRAY[4]);
```

Logic: Use @> ARRAY[2] to check for 2, then negate the check for 4 with NOT. Remember: None of the original 3 records match this criteria! Jane has both 2 and 4, Bob has both, John has neither.

Question 3l: Update Array - Append Element

Write a query that updates the 'interests' column of the row with id=1 to add the value 2 at the end of the array.

Answer:

```
-- Method 1: Using array_append function
UPDATE sample_table
SET interests = array_append(interests, 2)
WHERE id = 1;
-- John Smith was {1,3,5}, now becomes {1,3,5,2}

-- Method 2: Using || operator (array concatenation)
UPDATE sample_table
SET interests = interests || 2
WHERE id = 1;
-- Same result: {1,3,5,2}

-- Method 3: Concatenate with array
UPDATE sample_table
SET interests = interests || ARRAY[2]
WHERE id = 1;

-- Verify the update
SELECT id, name, interests FROM sample_table WHERE id = 1;
-- Should show: 1 | John Smith | {1,3,5,2}

-- Other useful array modification functions:
-- array_prepend(2, interests) - Add to beginning
-- array_remove(interests, 3) - Remove all occurrences of 3
-- array_cat(interests, ARRAY[6,7]) - Concatenate arrays

-- Example: Add multiple values at once
UPDATE sample_table
SET interests = interests || ARRAY[6, 7, 8]
WHERE id = 1;
-- Now: {1,3,5,2,6,7,8}

-- Remove a value
UPDATE sample_table
SET interests = array_remove(interests, 3)
WHERE id = 1;
-- Removes all 3's from array
```

Best method: Use array_append(interests, value) for clarity, or use || operator for concatenation. Both work for adding elements. The || operator is more flexible - can append single values or entire arrays.

Section 4: Backup & Restore

Question 4: pg_dump and pg_restore Commands

Write command examples for: a) Backup entire database to plain SQL file 'mydb_backup.sql'. b) Restore entire database from plain SQL file 'mydb_backup.sql'. c) Restore entire database from directory 'mydb_backup_dir'.

Answer:

```
# ============================================================
# Question 4a: Backup to plain SQL file
# ============================================================

# Basic backup to plain SQL
pg_dump mydb > mydb_backup.sql

# With username
pg_dump -U postgres mydb > mydb_backup.sql

# With host and port
pg_dump -h localhost -p 5432 -U postgres mydb > mydb_backup.sql

# More complete backup with ownership and privileges
pg_dump -U postgres --no-owner --no-acl mydb > mydb_backup.sql

# Best practice: Include CREATE DATABASE statement
pg_dump -U postgres -C mydb > mydb_backup.sql
# -C or --create: Include CREATE DATABASE command

# With compression (gzip)
pg_dump -U postgres mydb | gzip > mydb_backup.sql.gz

# ============================================================
# Question 4b: Restore from plain SQL file
# ============================================================

# Method 1: Using psql (for plain SQL files)
psql -U postgres -d mydb < mydb_backup.sql

# If database doesn't exist, create it first:
createdb -U postgres mydb
psql -U postgres -d mydb < mydb_backup.sql

# Or if backup includes CREATE DATABASE (-C flag):
psql -U postgres < mydb_backup.sql

# From compressed backup:
gunzip -c mydb_backup.sql.gz | psql -U postgres -d mydb

# With verbose output:
psql -U postgres -d mydb -f mydb_backup.sql -v ON_ERROR_STOP=1

# Method 2: Drop and recreate (clean restore):
dropdb -U postgres mydb
createdb -U postgres mydb
psql -U postgres -d mydb < mydb_backup.sql

# ============================================================
# Question 4c: Restore from directory format backup
# ============================================================

# First, create directory format backup (for reference):
pg_dump -U postgres -F d -f mydb_backup_dir mydb
# -F d : directory format
# -f : output directory path

# Restore from directory format using pg_restore:
pg_restore -U postgres -d mydb mydb_backup_dir

# Clean restore (drop and recreate):
dropdb -U postgres mydb
```

```
createdb -U postgres mydb
pg_restore -U postgres -d mydb mydb_backup_dir

# With connection parameters:
pg_restore -h localhost -p 5432 -U postgres -d mydb mydb_backup_dir

# Restore with parallelism (faster for large databases):
pg_restore -U postgres -d mydb -j 4 mydb_backup_dir
# -j 4 : Use 4 parallel jobs

# Restore only schema (no data):
pg_restore -U postgres -d mydb --schema-only mydb_backup_dir

# Restore only data (no schema):
pg_restore -U postgres -d mydb --data-only mydb_backup_dir

# Restore specific table only:
pg_restore -U postgres -d mydb -t sample_table mydb_backup_dir

# ============================================================
# Summary of dump formats:
# ============================================================

# Plain SQL (-F p or default): Human-readable, use psql to restore
pg_dump mydb > backup.sql
psql mydb < backup.sql

# Custom format (-F c): Compressed, use pg_restore
pg_dump -F c mydb > backup.dump
pg_restore -d mydb backup.dump

# Directory format (-F d): Parallel dump/restore, use pg_restore
pg_dump -F d -f backup_dir mydb
pg_restore -d mydb backup_dir

# Tar format (-F t): Archived, use pg_restore
pg_dump -F t mydb > backup.tar
pg_restore -d mydb backup.tar
```

Key points: Plain SQL uses psql for restore. Custom/directory/tar formats use pg_restore. Directory format (-F d) supports parallel operations with -j flag. Always specify -U username and -d database. Use -C flag in pg_dump to include CREATE DATABASE.

Study Tips

Key Concepts to Remember:

- Remember: pg_stat_activity for connections, pg_terminate_backend() to kill them
- Use @> operator for 'array contains' queries
- SERIAL = auto-increment, don't insert values for it
- Always use TIMESTAMP WITH TIME ZONE (not without!)
- pg_dump outputs SQL, pg_restore reads binary formats
- Use \d tablename in psql to see table structure
- GRANT gives permissions, REVOKE removes them
- Role inheritance is automatic with INHERIT (default)
- Array indexes start at 1, not 0!
- Test your queries on sample data before the exam

Good luck on your exam! Remember: It's open book, so use these materials effectively.