Database Engineering For Beginners

> ***Week 1:*** Data Extraction

Day 1: **Introduction to Data Extraction**

Focus : Understanding the process of extracting data from various sources, including databases, files, and **APIs**.

Activity : Data sources and formats (databases, CSV, JSON, XML) Data extraction techniques (**SQL queries**, APIs, ETL tools) Data cleaning and transformation

Practical Exercise : Write a simple Python script to extract data from a CSV file and load it into a PostgreSQL table.

Day 2: **SQL for Data Extraction**

Focus : Using SQL to extract data from PostgreSQL databases.

Activity : SELECT statement and its clauses (WHERE, FROM, JOIN, GROUP BY, HAVING, ORDER BY) Subqueries and common table expressions Window functions

Practical Exercise : Write SQL queries to extract specific data from the generated dataset based on different criteria.

Day 3: **Advanced SQL Techniques**

Focus : Mastering advanced SQL techniques for complex data extraction.

Activity : Recursive queries Full text search JSON functions

Practical Exercise : Solve a challenging data extraction problem using advanced SQL techniques.

Day 4: **ETL Tools**

Focus : Introduction to ETL (Extract, Transform, Load) tools and their role in data management.

Activity : Popular ETL tools (e.g., pg_dump, pg_restore, Talend, Informatica) Basic ETL process and its stages

Practical Exercise : Use a simple ETL tool to extract data from a source, transform it, and load it into a PostgreSQL table.

Day 5: **Data Extraction Best Practices**

Focus : Best practices for efficient and reliable data extraction.

Activity : Data quality and validation Data security and privacy Performance optimization

Practical Exercise : Review and optimize the data extraction scripts and ETL processes developed during the week.

WEEK 1

In the first week, many things about Database engineering will be discussed and some activities like extracting data and loading data are shared, you will learn how to transform a file from json to csv and vice versa, learn how to create random data, and also transfering a csv fie to a RDBMS (postgres). An ETL uisng an API will also be shared for a simple pipeline setup. You will also learn as a bonus. How to effectively hide your credentials

---

One major activity was creating a random data and uploading it into a postgres database.

```
import pandas as pd
import numpy as np
from sqlalchemy import create_engine

# Specify Data
data = {
    'Date': pd.date_range(start='2023-01-01', end='2023-12-31', freq='D'),
    'Stock_Price': np.random.uniform(low=100, high=200, size=365),
    'Volume': np.random.randint(low=1000, high=10000, size=365),
    'Company': np.random.choice(['Apple', 'Google', 'Microsoft', 'Amazon'],
size=365)
}

# ransform data to pandas dataframe
df = pd.DataFrame(data)

# Connect to PostgreSQL database
conn_string = "postgresql://postgres:Daniroyal@localhost:5432/db"
engine = create_engine(conn_string, echo= True)

# Write DataFrame to PostgreSQL table
df.to_sql('financial_data', con=engine, if_exists='replace', index=False)

engine.dispose().
```

Learnt : importing the single particular function that i will be using from the library sqlalchemy was probably an optimizer for me, but i am sure we will find out in the code optimization part. Also, the echo under create_engine Function was meant for registering the log of transactions, hope to see this for regular data streaming. The randomization of data also included some funny functions called variables. Dont know why vscode calls it variables, but they are the Uniform, randint and choice.

---

Another activity which was carried out was sending a file to the database.

```
from sqlalchemy import create_engine
import pandas as pd

# Transform path to df
df= pd.read_csv("pokemon.csv")

# Create connection to DB
conn_string = "postgresql://postgres:Daniroyal@localhost:5432/db"
engine = create_engine(conn_string, echo= True)

# Transfer file to DB
df.to_sql('pokemon', con=engine, schema='movie_db', if_exists='replace',
index=False)

engine.dispose()
```

Learnt : The algorithm was simple, (i) Transform the file path from your local pc to a variable (ii) Create connection to database and send transformed variable to database. The to_sql() function by sqlalchemy is able to update a table in a database which will be nice for updating a database table. which will be discussed futher. to_sql variables are [retty straight forward. They ask for conditiona that complete access information like, database connection details, schema and table name from database, conditions to rollback, update or replace the table it was directed to. Do well to check for other connection string methods

----

Retrieving a section/table from the database using a SQL query was also carried out.

```
import psycopg2

host = "localhost"
port = "5432"
user = "postgres"
password = "Ganjo"
database = "db"
output_file = "CTE_data.csv"
query = ''' WITH
                AvgStockPrice AS (
                    SELECT "Company", AVG("Stock_Price") AS Avg_Price
                    FROM financial_data
                    GROUP BY "Company"
                )

                SELECT
                    fd."Date",
                    fd."Stock_Price",
                    fd."Company",
                    asp.Avg_Price
                FROM
                    financial_data fd
                JOIN
```

```
                        AvgStockPrice asp ON fd."Company" = asp."Company"
                    WHERE
                        fd."Stock_Price" > asp.Avg_Price
                    '''

try:
    with psycopg2.connect (host=host, port=port, user=user, password=password,
database=database) as conn:
            cur = conn.cursor()
        # open writable file
    with open(output_file, 'w') as fp:
        # Copy from DB to file
        cur.copy_expert(f"COPY ({query}) TO STDOUT CSV HEADER", fp)
except (Exception, psycopg2.Error) as error:
    print("error fetching data", error)
finally:
    if conn:
        cur.close()
        conn.close()
```

Learnt : Retrieving a table without a query using the sql copy function was compatible when the table name is inserted into the curly brackets, on the other hand, brackets were needed when SQL queries were called from the copy_expert function. The psycopg2 is quite handy, even if it doesnt look like a link (setting a good lock is just what u need). Notice i used SQlAlchemy for sending to DB and psycopg2 for retrieving. you can check the vice versa on your free time, depending on the style of connection you would prefer working with. The main power here is the cur.copy_expert function which transfers the copy of the query into a specified file.

----

also transforming a json file to a csv file and vice versa.

```
import json
import csv

def json_to_csv(json_file, csv_file):
  """
  Converts a JSON file to a CSV file.

  Args:
    json_file: Path to the input JSON file.
    csv_file: Path to the output CSV file.
  """
  with open(json_file, 'r') as f:
    data = json.load(f)

  # Get field names from the first data object
  fields = data[0].keys()

  with open(csv_file, 'w', newline='') as f:
    writer = csv.DictWriter(f, fieldnames=fields)
    writer.writeheader()
    writer.writerows(data)

# Example usage
json_to_csv('rand.json', 'randtocsv.csv')
"""
  Transforming this file was directly administered
  using a single function by reading the specified json
  format and write out that format into csv format,
  identifying the headers and parsing identified headers.
  """


CSV >>> JSON

Pyhon = import csv
import json

def csv_to_json(csv_file, json_file):
  """
  Converts a CSV file to a JSON file.

  Args:
    csv_file: Path to the input CSV file.
    json_file: Path to the output JSON file.
  """
  # create a list
  data = []
  # read then write files
  with open(csv_file, 'r') as f:
    # send the files to variable
    reader = csv.DictReader(f)
    for row in reader:
      # update the created list step by step, as the data comes in
      data.append(row)

  with open(json_file, 'w') as f:
    json.dump(data, f, indent=4)

csv_to_json("rand.csv", "rand.json")
""" Transforming this file was  directly administered using a single function by
reading the specified csv format and write out that format into json key value
pair. """
```

Learnt : The code was made as a function, considering it would definitely be used some day. Anytime soon. csv_to_json; The two very simple steps for doing this was updating an empty list and writing it to a json file using the json.dump() json_to_csv; This step added one newer step and this was identifying the column names, which are the keys in a json format. in order for the csv writer to capture and display appropriately. Pretty sure some other libraries can carry this kind of function for several kinds of files, rater than installing particular file libraries.

Lastly connecting an API to my local postgres which will be similar for many other RDBMS's

```python
import requests
import pandas as pd

# set url
base_url = "https://pokeapi.co/api/v2/pokemon/"

# create function for extracting json data by id
def main_request(base_url, x):
    response = requests.get(base_url + f'{x}')
    return response.json()

# function to iterate across json data for required dataset and store required
data in a list
def parse_json(response):
    character_list = []
    char = {
    'name': response["name"],
    'height': response["height"],
    'weight': response["weight"],
    'base_experience': response["base_experience"]
    }
    character_list.append(char)
    return character_list

# create list to store required data of iterated id's
pokemon_info = []
for y in range(1, 20):
    data = (main_request(base_url, y))
    pokemon_info.extend(parse_json(data))

# transform list of json object to csv and save file
df = pd.DataFrame(pokemon_info)
df.to_csv('pokemon.csv', index=False)
```

Learnt : This code has several functions which are used to easily call the processes required for retrieving and transforming several json for further transformations. After this, the json was easily transformed into a csv file using pandas under 2 lines of short code. Optimization of this code will be done next and the week will be completed. Hope you are familiar with loading..? Its pretty easy 😉

Optimizing code for more frequent and bogous transactions.

```python
import requests
import pandas as pd
import logging

# Configure logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %
(message)s')
```

```
# Set URL
base_url = "https://pokeapi.co/api/v2/pokemon/"

# List of Pokemon IDs
pokemon_ids = list(range(1, 1000))  # Increased range to show more data

# Use a session for efficiency
with requests.Session() as session:
    def get_pokemon_data(ids):
        pokemon_info = []
        for id in ids:
            url = base_url + str(id)
            try:
                response = session.get(url)
                response.raise_for_status()  # Raise an exception for bad status
codes (4xx or 5xx)
                data = response.json()
                pokemon_info.append({
                    "name": data["name"],
                    "height": data["height"],
                    "weight": data["weight"],
                    "base_experience": data["base_experience"],
                })
            except requests.exceptions.RequestException as e:
                logging.error(f"Error fetching data for ID {id}: {e}")
                continue # Skip to the next ID if there is an error
            except (KeyError, TypeError) as e: # Catch errors if the JSON response
is not in the expected format
                logging.error(f"Error parsing JSON for ID {id}: {e}")
                continue
        return pokemon_info

    pokemon_info = get_pokemon_data(pokemon_ids)

# Create DataFrame and save as CSV
df = pd.DataFrame(pokemon_info)
df.to_csv("pokemon.csv", index=False)

logging.info("Pokemon data saved to pokemon.csv")
```

Learnt : This was amazing. when i got the code, it was slow but i got what i wanted. instead of using the normal requests.get, it was necessary to invoke the requests.Session() as this would be the way to extract iterated links using a single call. Sounds amazing right. Well that is the power of optimization. Hope you learn how to use this codes and dont be overwhelmed by the many mix of words, as it simply takes some time to get used to.

Bonus : Relevant way to hide Credentials

```
# Open .env file

host = "<ur_host>", password= "<ur_password>", database = "<ur_dbname>", user = "
<ur_username>", port =  1234

usually allowed to be an integer.
```

```python
# Access Method

import os
from dotenv import load_dotenv
import psycopg2

# Load environment variables from .env
load_dotenv()


# Connect to the database
def get_db():
    conn=psycopg2.connect(
    user=os.getenv("user"),
    password=os.getenv("password"),
    host=os.getenv("host"),
    port=os.getenv("port"),
    dbname=os.getenv("dbname")
    )
    return conn
```

Learnt : Create a configuration file and store connection details in a dedicated file. Import and use In your main script and access the values. For security, add the configuration file to your .gitignore file to prevent it from being committed to version control. And also, create separate configuration files for different environments (development, testing, production) with appropriate credentials.

> ***Week 2:*** User Management in PostgreSQL

Day 1: **Groups and Roles**

Focus : Basic Role Creation and Permissions.

Activity : Create a new user with limited permissions (e.g., only SELECT privileges on a specific table). Grant and revoke specific permissions to the newly created user. Experiment with different permission levels (e.g., SELECT, INSERT, UPDATE, DELETE).

Resources : PostgreSQL documentation on CREATE ROLE, GRANT, and REVOKE commands.

Day 2: **Role Hierarchies and Inheritance**

Focus : Role Hierarchies and Inheritance

Activity : Create a parent role with a set of permissions. Create child roles that inherit permissions from the parent role. Explore how to modify permissions for child roles without affecting the parent role.

Resources : PostgreSQL documentation on role inheritance and hierarchies.

Day 3: **Authentication Methods**

Focus : Authentication Methods

Activity : Research and compare different authentication methods (Peer, MD5, SCRAM-SHA-256). Configure PostgreSQL to use a specific authentication method. Understand the security implications of each authentication method.

Resources : PostgreSQL documentation on authentication methods.

Day 4: **Advanced User Management Scenarios**

Focus : Advanced User Management Scenarios

Activity : Create a complex user management scenario (e.g., multiple teams with different access needs). Design and implement a role hierarchy to manage permissions effectively. Test the user management system to ensure it meets the requirements.

Resources : PostgreSQL documentation on advanced user management concepts.

Day 5: **Security Best Practices**

Focus : Security Best Practices

Activity : Review security best practices for user management in PostgreSQL. Implement security measures to protect user accounts (e.g., strong passwords, password policies).

Learn about security audits and how to monitor user activity.

Resources : PostgreSQL documentation on security, OWASP guidelines.

WEEK 2

This is the second week and it is focused on Database Roles, Hierarchies and Inheritance. This path of learning is very crucial for a Database Manager who is concerned about the security of the database and the kinds of people allowed to come into the database. The assigning of roles is peculiar to the owner of the database and every role is giving an activity it can work with on the database. Undersanding the capabilities carried out by each user/role is very crucial for every Database Manager to understand.

One of the basic things to do when assigning a role is to create it. First things first, roles are made up of two categories, namely: Group and Users. These categories are professionally differentiated by their privilege to login. It is of professional practice to give a group role the NOLOGIN privilege, while a User role can have the LOGIN privilege. e.g

```
CREATE ROLE group_role --automatically holds no-login
CREATE USER user1 -- automatically holds login privilege
```

After a role has been created, it is expected to assign privilleges to those roles. Note: all roles created will automatically have the privilege of the public schema to operate with several privileges which are not allowed for certain users/roles. The two commands when working with privileges are REVOKE and GRANT.

steps in creating a new role:

```
-- step 1 (create role)
CREATE ROLE parent_role
-- step 2 (create user)
CREATE USER user1 WITH PASSWORD 'secrets'
CREATE USER user2 WITH PASSWORD 'secrets'
CREATE .....
-- step 3 (revoke all privileges)
REVOKE ALL ON SCHEMA <schema_name> FROM PUBLIC
-- step 4 (make grants)
GRANT USAGE ON SCHEMA <shema_name> TO parent_role
GRANT SELECT ON ALL TABLES IN SCHEMA <shema_name> TO parent_role
-- step 5 (connect all inheritance to particular users)
GRANT parent_role TO user1, user2, ....

-- NOTE: these privileges are simply meant for users to read or select from tables
only. You can also signify the particular tables, views or functions in a schema
that you want to be accessed by the user.
```

This module talks about authentication methods. Some authentication methods require external servers, certificates or simply password authentication. Some authentication methods to read on include SCRAM SHA256, SSL, GSSAPI, and Trust.

Read about their use case and importance, as the functionality of these authentication methods cannot be completely covered in this book.

NOTE:

SCRAM SHA256: This type of authentication is made to manage passwords set in the database through the pg_hba.conf file to avoid password sniffing by third parties and middle-men

SSL: This is a way that communication passing through the database is protected and properly secured.

SSH: This is an encryption certificate that confirms the connection between two different clusters.

Trust Auth: This simply explains how a user can access the database from anywhere without a password. (unhealthy).

> ***Week 3*:** Data Backup and Recovery in PostgreSQL

Day 1: **Backup Strategies and pg_dump**

Focus : Backup Strategies

Activity : Learn about different backup strategies (full, incremental, logical, physical). Use pg_dump to create a full logical backup of a sample database. Experiment with different pg_dump options (e.g., -f for output file, -t for specific tables).

Resources : PostgreSQL documentation on pg_dump.

Day 2: **Incremental Backups**

Focus : Incremental Backups

Activity : Create an incremental backup using pg_basebackup. Compare the size of full and incremental backups. Understand the benefits of incremental backups in terms of time and storage.

Resources : PostgreSQL documentation on incremental backups.

Day 3: **Physical Backups with pg_basebackup**

Focus : Physical Backups

Activity : Use pg_basebackup to create a physical backup of a database. Understand the differences between logical and physical backups. Explore the options available with pg_basebackup (e.g., -x for excluding certain files).

Resources : PostgreSQL documentation on pg_basebackup.

Day 4: **Database Recovery**

Focus : Database Recovery

Activity : Restore a database from a full backup using pg_restore. Restore a database from a series of incremental backups. Experiment with point-in-time recovery (if applicable).

Resources : PostgreSQL documentation on pg_restore, point-in-time recovery.

Day 5: **Disaster Recovery Planning**

Focus : Disaster Recovery Planning

Activity : Develop a basic disaster recovery plan for a hypothetical PostgreSQL server. Identify potential risks and mitigation strategies. Learn about high availability options (e.g., replication) for improved disaster recovery.

Resources : PostgreSQL documentation on high availability, industry best practices for disaster recovery.

WEEK 3

Backing up and recovering a database can be very complex in many ways, as it is essential for the databases to be monitored properly and maintained accurately for its content not to be corrupted or mismanaged. It is advised not to write the complete script of backing up your server but to use an extension to carry out all the complex activities of a backup and recovery process.

Before accessing the pg_dump from your systems command prompt or terminal, it is expected to give access to the path of postgres' bin to the environmental variable e.g "C:\Program Files\PostgreSQL\17\bin" to a new space created in the system variable in the settings tab. Step by step process for using pg_dump and dump_all for capturing users:

```
pg_dump -U  <superuser>  -d  <db_name>  -Ft  -f  <location_of_file>
```

using the pg_dump utility in this way will simply dump the logical aspect and structure of the database into a specified file in your server. (type "pg_dump -- help" from root terminal to access info on how to use the parameters). You can research on more better ways to compress your file through his utility for larger databases.

```
pg_dumpall -U postgres -Ft -f <location_of_file> -g
```

Because all users are located on the cluster level, when using the pg_dumpall utility, it documents everything on the cluster level, thereby outlining the users on the global scale of the outline. using the parameter -g will extract only the global scope of the logical structure and save. When recovering this file, it is expected to call the users file first before the dumped database file for an optimized experience.

NOTE; pg_dumpall is used to access the cluster wide part of the database, while pg_dump will specify database and probably table where needed.

Step by step process for using pg_basebackup for incremental processes: Before accessing the incremental capabilities of postgres, the configuration files namely pg_hba.conf and postgresql.conf must be configured to allow this process. The pg_hba.conf must be configured for allowing communication between 2 clusters for streaming and incremental activities, and the postgresql.conf file must be edited to turn on the archiving of WAL (write ahead log) files for adequate documentation of transactional activities performed on or between clusters.

NOTE; do not forget to restart your cluster after turning the archive wal on.

```
pg_basebackup -D | gzip > path_to/sunday.gz
--This is a Full Backup
```

This will start a full backup of the files found in the whole cluster and will save it in the specified location of the file.

```
pg_basebackup  -D  | gzip >  path_to/monday.gz
-- incremental path_to/sunday.gz/backup_manifest
```

```
pg_basebackup   -D  | gzip >  tuesday.gz  -- incremental
path_to/monday.gz/backup_manifest
```

After running the incremental codes, it will be considered an incremental file based on its previous file as specified.

Learnt : In the process of starting a backup and recovery, there must be a backup first. There are several types of backups, which include: Full, Incremental and differential. Also, in the process of carrying out these different types of backup, it can either be a logical or physical backup. NOTE; Every type of backup must start with a full backup.

A logical backup involves the over-all structure of the database/cluster, including every object found in the database/cluster. There are 2 utilities used for carrying out a logical backup, which include; pg_dump and pg_dumpall.They are used for specifying database, table or cluster to be backed up respectively. These utilities in addition with various parameters determine the outcome of the backed up dataset/cluster. While the physical backup is carried out using the pg_basebackup utility, which includes every file that is associated with that database cluster. Carrying out an incremental backup can be a very complex process, as it involves manipulating the details of some configuration files located in the server/cluster. It is advisable to use pg_basebackup for incremental backups and PITR or a third party system like Barman (preferable).

Recovering database to full backup from incremental backup:

```
pg_combinebackup sunday.gz monday.gz tuesday.gz -o  tuesday_full.gz
```

This is a very straight forward code and it should perform the combination of all the available metadata that relate across the several files meant to be restored together to become a Full Backup. PITR would require a recovery.conf file in the "path_to/postgres/main" path whereby a restore command and recovery target time will be specified. After the backup file has been specified and restored using the configuration settings of postgresql.conf file under the archive command. Database will return to specified time after a restart of the cluster. Restoring streamed data from different cluster after shutdown: Firstly configure the pg_hba.conf file of primary cluster to replicate its tasks to a particular IP address that has been created (serving as secondary cluster). Secondly, configure postgresql.conf file on the secondary under listen_addresses to accept * instead of localhost in order to accept communication from every IP coming from its server and vice versa for the primary in order to recieve from secondary after temporary shutdown. Now remove all the files available in secondary for new and unique files to come in from the primary cluster. Run this command:

```
psql -u postgres pg_basebackup -h <host/ip> -w -U postgres -Ft -X stream -R -S
my_slot -C -D  <path_to/postgresql/main>
--This step will stream primary cluster into secondary cluster.
```

In order to activate the secondary server as the primary server and carry out the activities of the primary server, considering it is currently in read only mode, based on the code passed before. There are many factors to be considered, especially on a production level mode. But running this command on your sql terminal will transform the secondary cluster into a primary cluster;

```
SELECT pg_promote();
```

Learnt : The process of recovering a database is the most important part of a backup and restore process. In recent times, it has been suggesed by best practices to stream your datasets rather than using incremental processes and this requires a database cluster to have other clusters which will serve as secondary/slave cluster that will hold identical data as the primary database cluster. The use-case of a streaming service is not more complex than having a secondary cluster take over the I/O tasks carried by the once primary database cluster whereby all data will remain the same and when the very first primary database comes back online, all the new datasets that were transacted during its shutdown would be restored. All in all, it is advisable to use third party systems like Barman to carry out your backup and recovery tasks for easier and concise ways of handling your system.

NOTE; pg_restore will not restore plain text and sql format kind of files, but will support directory, tar, dmp etc.

---

Disaster Recovery Plan using Barman.

Create users for connection and replication access on primary server. Set connection IPs on barman to allow primary database connect through barman server configuration directory called barman.d as well as turn on all streaming configuration settings including slot, archiver etc. and name the primary database from barman.d settings. Set listen address and archive configurations on postgreql.conf, Set replication settings and connection settings on pg_hba.conf from primary server for accepting database after temporary crash. Set up ssh keys between clusters to provide a secured connection

RESTART YOUR DB INSTANCE!!

```
barman check pg
--for cofirming all status of postgres utility and archiving configuration setups.
```

Learnt : A disaster recovery system or HA (High Availability) is very essential for every standard production database, and as previously advised, it is required for a database to employ a streaming style of backup in order to achieve next to zero downtime of server I/O tasks and another option to be cosidered especially for higly transactional databases is concurrency control. The use of barman to administer these processes will allow the database to be under adequate control. locking the communication path of your clusters with ssh keys is also essential for full security. An important method for maintaining a proper disaster recovery system is to allow your secondary or standby clusters to be located in a very distant location in order to mitigate physical hazards that could affect the entire region e.g earthquake, flood, fire etc. Maintaining a standard production database replication system requires proper editing of the necessary configuration files available after installing postgres and barman. Identifying and documenting these standards is crucial for maintaining a prosperous database for scalability, adaptability and sustainability.

Some container tools capable of giving you peace of mind are Cloudnativepg, Kubernetes, patroni, Datagrip, Tableplus, Amazon ECS, Docker swarm etc.

> ***Week 4***: Database Maintenance and Optimization

Day 1: **Query Optimization and Indexes**

Focus: Understanding slow queries and the basics of indexing.

Activity: Learn how to identify slow queries using EXPLAIN ANALYZE. Practice creating and dropping indexes on sample tables. Analyze query plans to see how indexes affect performance. Understand different index types (B-tree, hash, etc.).

Resources: PostgreSQL documentation on EXPLAIN, indexes.

Day 2: **Materialized Views and Vacuuming/Analyzing**

Focus : Using materialized views and maintaining table health.

Activity : Create and refresh materialized views for frequently used aggregated data. Learn the purpose and process of VACUUM and ANALYZE. Practice running VACUUM and ANALYZE on sample tables. Understand how VACUUM and ANALYZE impact query performance.

Resources : PostgreSQL documentation on materialized views, VACUUM, ANALYZE.

Day 3: **Resource Usage & Performance Metrics**

Focus : Monitoring system resource usage and key performance indicators.

Activity : Learn how to monitor CPU, memory and disk usage using system tools like (top, vmstat, iostat). Explore PostgreSQL monitoring tools (e.g., pg_stat_statements, `pgAdmin monitoring). Identify key performance indicators (KPIs) like query response times, transaction rates, and database size. Learn how to read and interpret postgres logs.

Resources : PostgreSQL documentation on monitoring tools, system monitoring tools.

Day 4: **Security Audits & Vulnerability Management**

Focus : Implementing security audits and staying updated on vulnerabilities.

Activity : Learn how to audit user activity and database access using PostgreSQL logging. Research common PostgreSQL vulnerabilities and security best practices. Explore tools for vulnerability scanning and management. Understand how to apply security patches and updates.

Resources : : PostgreSQL documentation on security, security advisories, vulnerability databases.

Day 5: **Encryption & Overall Review**

Focus : Implementing encryption and reviewing all learned concepts.

Activity : Learn about data encryption at rest and in transit. Explore PostgreSQL extensions for encryption (e.g., pgcrypto). Review all the topics covered during the week (query optimization, monitoring, security). Create a checklist of best practices for PostgreSQL performance and security.

Resources : PostgreSQL documentation on encryption, pgcrypto extension.

WEEK 4

This week is comprised of all things optimization and security. This is the fun part in Database MANAGEMENT, because this is Database MANAGEMENT. Properly optimizing the queries a database works with on a daily basis will assist that system to MANAGE its resources as well as return faster results for better performance and suitability by all and sundry. This weeks study does not only involve optimization techniques but alsocarries areas associated with securing your database, another key area in MANAGEMENT of databases. Therefore, the study of database maintainability will be the focus of this weeks study.

Creating and Dropping an index:

```
CREATE INDEX idx_poke_hei ON movie_db.pokemon (height);
```

- idx_poke_hei > name of index created
- movie_db > name of schema
- pokemon > name of table
- height > name of column to be indexed.

```
DROP INDEX movie_db.idx_poke_hei;
```

Notice the name of the index matches the name of the column and table.

Learnt : Creating an index is pretty simple on a database, and its essence can very much impact the performance of the database to a great extent, as i have just examined a 1000 row dataset I assigned an index to. The dataset will not return the exact same capacity it uses to retrieve data everytime it requests a query, even when it is the same dataset. Its retrieval capacity usually flunctuates. It is also a standard practice to understand the data you are assigning an index to. An index on a column that is more frequently UPDATED, INSERTED or DELETED on the Database is most likely to slow down that database and affect the overall performance of the database. Therefore, creating an index is an important job for every Database Manager who will be avaialable to analyze the tasks carried out by the system. Also, before creating an index, there must be a confirmation of possible columns that will be regularly found in the WHERE, ORDER BY and JOIN conditions of the database, as these criteria must be considered before creating an index for a faster peerfomance of the database.

There are different types of indexes and the one i just explained about is the B-tree index. Still on the matter, indexes carry a single functionality and that is to make queries faster. The B-tree index is considered the powerhouse of relational databases as it is considered for range based queries, other than this, the database will be used for direct queries whereby the Hash index is a best practice for. Some othe indexes are GIN, GiST etc.

The EXPLAIN command of postgres is used to provide the details of the plan with which the cluster has used to query its statements, showing how it started and where it ended. While adding more commands like ANALYZE and BUFFER will show even more details of the transaction going on in the cluster which will evaluate the costs and capability of the query.

```
EXPLAIN ANALYZE SELECT * FROM pokemon;
```

```
CREATE MATERIALIZED VIEW <poke_view> AS
SELECT height, weight
FROM pokemon
WHERE not null;
```

Learnt : Materialized views are a part of a whole database that extracts specific columns which are usually needed for frequent operations by the users of the database. there are basically two types of views in a database cluster. The first is Views. These are logical queries expected to be run everytime those columns are needed. These queries would be considered redundant whenever it has to be called everytime it is needed. While the second is called Materialized Views. This type of view is a physical storage which holds an instance of a query, whereby, it is always available for easy reads, like a cached representation of Views. It is only refreshed whenever there are updates on the table where the materialized views were querried.

```
VACUUM FULL pokemon;
```

This code will clean all dead tuples and create a new database without any dead tuples, thereby reducing the overall size of the database.

```
ALTER TABLE <table_name> SET (autovacuum_enabled = true/false);
```

This wiil autovacuum only the specified table.

```
ALTER DATABASE <db_name> SET autovacuum = on/off;
```

This wiil autovacuum only the specified database.

```
ALTER SYSTEM SET autovacuum = on/off;
```

This will autovacuum the whole cluster after which the below code must be queried to return the configuration settings (not widely recommended)

```
SELECT pg_reload_conf();
---Monitor your autovacuum statistics using pg_stat_activity and
pg_stat_all_tables.
```

Learnt : A vacuum is a process in a database where a single table or database can be cleaned up of empty spaces, otherwise known as dead tuples. These dead tuples pile up and make your table or database bloated, thereby hindering a smooth operation in your I/O tasks, therefore making your disk space too bulky to work with. The two categories of vacuums inlude; FULL VACUUM and STANDARD VACUUM. the Full Vacuum will simply stop your database from performing any other activity and focus solely on making your disk space free of dead tuples. NOTE: This process requires double the normal disk space for complete read/write capacity and will also activate an ACCESS EXCLUSIVE LOCK. The standard vacuum has a less stiff approach which will vacuum the database while it is still sending and accepting queries. The automatic process for cleaning up your database is called autovacuum which is considered a standard vacuum. This autovacuum has to be configured from postgresql.conf for an optimal performance because, a database would require a more aggressive autovacuum with workers when its tables are large and also administer a lot of transactions rather than a smaller database. NOTE: This system of vacuum will not entirely free up your disk space, but will actually show the database that there are leftover tuples that need to be filled.

Types of Transaction Lock :

Table level lock, Row level lock, Page level lock, deadlock and advisory locks. There are several locks that represent these locks and they will be explained below.

ACCESS EXCLUSIVE - This is the most stiff lock among all table locks, and it requires every transaction to be held back for its own transaction statement to be completed. Even when another ACCESS EXCLUSIVE tries to access that table, it will not allow.

ACCESS SHARE - This is the least stiff lock among the table locks, and it only conflicts with the ACCESS EXCLUSVE lock. This lock is acquired when the table is being read (parsing SELECT) concurrently.

SHARE UPDATE EXCLUSIVE - This is the most recommended lock system for major transactions on a large database with high I/O activities. This lock will allow the table to perform other activities concurrently like vacuum, DML etc

FOR KEY SHARE - This lock will accept every other lock in a transaction except FOR UPDATE. It accepts DML on rows that are not indexed or keyed.

FOR UPDATE - This is the strictest row level lock as it will conflict every other lock, even itself. It will be activated immediately after an UPDATE or DELETE occurs on a row with a unique index rather than partial or expressional indexes.

DEADLOCK - This is a system set-up by postgres to invalidate a transaction that requires a type of lock that conflcts another lock on a table or row.

ADVISORY LOCK - This can also be called a custom lock admitted by a developer for an application whereby it will perform its custom use for that application.

Learnt : Locks are a major essence for all the processes occuring in a database ranging from DML, vacuum, reindexing, triggers, materialized views etc. Conflicting these locks will lead to a deadlock which will set off all existing locks. When a transaction is in process, its specific locks are active and this will affect every other statement that contradicts what is currently on the transaction script. Locks are usually associated with MVCC.

Some Key Performance indicators associated with maintaining a database include: CPU Usage, Memory (Total, Shared, Buffer), Disk I/O metrics, Query Performance, Lock Wait Time, Transaction Per Second (TPS), Sandard Vacuum Activity, Replication Lag etc. Some monitoring tools available have successfully classed this performance indicators in charts for easier identification and alerts to notify the Database Admin for proper Administration of management on the database.

Learnt : There are a lot of ways whereby a database can be monitored, and it is usually advised to make it a practice, most especially when your database accumulates a whole bunch of I/O transactions in a day. These transactions will cause a lot of hiccups to your database and without a proper system setup, that database will over bloat, be slow, and probably crash at the end of an un-monitored session. The tools used to monitor a database are usually connected to the database either from a cloud system or on premise. Databases are expected to be monitored in order for them to be scaled when needed, reoptimized when expected and shutdown when approved, all for the database to not be damaged and lost forever. This is why a good practice of setting up the database from start is advised by an experienced Database Administrator. The logs in a database differ. One is set to monitor all the transactions that actually occur in the database for Point In Time Recovery (PITR). This type of log is called Write Ahead Log (WAL), while the second is made to monitor the capacity of the system cluster per head, which overall is meant to keep the database active and stable.

Some Vulnerabilities:

- Dont use root user
- Set a password for all users
- Only grant superuser so it can be revoked
- Always re-configure the configuration settings
- The world should not know your Database exists
- Use scram or md5 instead of trust, peer and ident for authenticaion
- Set TLS/SSL on with consequent certificate renewal/modifications for encryption
- Grant users access to specific tables, databases or even columns only

Learnt : Identifying vulnerabilities in a database cluster can be realised through proper log monitoring, and these monitoring activities can be performed using extensions of different kinds e.g pgaudit, pg_stat_statements etc. Some monitoring tools can be accessed using third party vendors, and cloud solutions. All in all, it is always necessary to track your user and database activities, as often as you can, or simply set automated alerts for expected errors.

Configure postgresql.conf for SSL and TSL usage then request for certificates and save them on specified path through the postgresql.conf file. Install the pgcrypo extension using:

```
CREATE EXTENSION pgcrypto IF NOT EXISTS;
```

Use the pgcrypto to encrypt columns using functions like armor, digest, encrypt and decrypt etc.

Learnt : When securing your database, your major concern will be securing the data in the database. Data is being transferred from client to server and server to client everytime which is called input/output (I/O) and these transactions are carried out by codes, written in both the server and client side called frontend/backend. When data is sent to the server, the server will have to confirm its eligibility as per where it is coming from, for the database not to be compromised. Using things like SSL and TLS are very essential for encrypted communication between server and client, whereas pgcrypto is appropriate in encrypting columns in the cluster, especially for columns holding sensitive info like password of application users.

Partitioning : One other way to maintain your database is by using partitioning. This form of maintenance is practiced in several ways under file management. Partitioning is simply seperating your table into different tables accordingly for easy identification, faster throughput and also quick dissemination. Some types of partition include RANGE, LIST and HASH. Working with partitions is a basic knowledge for a Database Administrator and it must be done before any data is inputted into the table. When a table exists without partitions, creating a new table and moving previous data into new partitioned table can be possible.

```
CREATE TABLE <table_name> (id INT, data TEXT, creation_date DATE)
PARTITION BY RANGE (creation_date);

CREATE TABLE <partition_table_name>
PARTITION OF <table_name> FOR VALUES FROM ('2023-01-01') TO ('2023-02-01');
```

Extensions : Extensions are a great addin to postgres as they add an extra layer of value to postgres in numerous areas that have been recorded in times past. These extensions ae usually embedded to every postgres package, but limited. In this scenario, it is expected to install other extensions from their websites or reliable platforms like Git. Before activating extensions in a postgres package:

```
CREATE EXTENSION <extension_name>;
```

INDEX :

DML STATEMENTS

SELECT : This statement retrieves data from a database table. It allows you to query specific columns or all columns from one or more tables, and you can filter the results using conditions.

INSERT : This statement adds new rows (records) to a table. You specify the values to be inserted into the table's columns.

UPDATE: This statement modifies existing records in a table. You specify which rows to update and the new values for the columns.

DELETE: This statement removes one or more rows from a table. You can use a WHERE clause to specify which rows to delete.

MERGE: This statement combines data from two or more tables, allowing you to perform insert, update, and delete operations in a single statement

DDL STATEMENTS

CREATE : This statement is used to create new database objects, such as tables, indexes, and databases. For example,

```
CREATE TABLE employees (id INT PRIMARY KEY, name VARCHAR(255));
```

will create a new table named "employees".

ALTER : This statement allows you to modify existing database objects, such as adding or removing columns from a table, or changing the data type of a column. For example,

```
ALTER TABLE employees ADD COLUMN salary INT;
```

would add a new column named "salary" to the "employees" table.

DROP : This statement is used to delete database objects, such as tables, indexes, and views. For example, DROP TABLE employees; would delete the "employees" table from the database.

TRUNCATE : This statement is used to remove all data from a table while keeping the table's structure intact. It's faster than deleting records one by one. For example,

```
TRUNCATE TABLE employees;
```

would remove all rows from the "employees" table.

RENAME : This statement is used to rename database objects, such as tables and indexes. For example,

```
RENAME TABLE old_table TO new_table;
```

would rename the table "old_table" to "new_table"

DCL STATEMENTS

GRANT : The GRANT command is used to grant specific privileges to users or roles. These privileges can include things like SELECT, INSERT, UPDATE, DELETE, and more.

```
GRANT USAGE ON ALL VIEWS IN SCHEMA <shema_name> TO <role_name>;
```

REVOKE : The REVOKE command is used to remove privileges from users or roles. This means that a user who previously had certain permissions on a database object will no longer have those permissions.

```
REVOKE <role_name> FROM <role_name>;
```

LOCK TABLE : The LOCK TABLE statement in SQL is used to explicitly acquire a lock on one or more tables. This mechanism helps manage concurrent access to data and prevent data corruption or inconsistencies when multiple users or transactions are trying to read or modify the same data simultaneously.

```
LOCK TABLE employees IN SHARE MODE;

LOCK TABLE inventory PARTITION (march_data) IN EXCLUSIVE MODE NOWAIT;
```

Procedural Language SQL

DECLARE : This is used to declare variables and other PL/SQL objects like cursors, constants, and record types.

BEGIN : This marks the beginning of the executable part of a PL/SQL block.

EXCEPTION : Allows you to handle errors and exceptions that may occur during the execution of the block.

END : Marks the end of the PL/SQL block.

IF-THEN-ELSE : A conditional statement that allows you to execute different blocks of code based on a condition.

CASE : Another conditional statement that provides a more structured way to handle multiple conditions.

LOOP : A loop statement that allows you to repeat a block of code.

FOR LOOP : A loop statement that iterates a specific number of times or over a range of values.

WHILE LOOP : A loop statement that repeats as long as a condition is true.

GOTO : A sequential control statement that transfers control to a labeled statement within the same PL/SQL block (use sparingly as it can reduce code readability).

NULL : A statement that does nothing (useful for cases where you need a placeholder).

EXIT : Transfers control to the end of the loop.

CONTINUE : Exits the current iteration of the loop and continues with the next iteration.

CURSOR : A PL/SQL object used to retrieve and process multiple rows from a SQL query.

FETCH : Used with cursors to retrieve the next row of data from the cursor.

PROCEDURE : A named PL/SQL block that can be called from other PL/SQL blocks or SQL statements.

FUNCTION : A named PL/SQL block that returns a value (like a stored function in SQL).

PACKAGE : A grouping of related procedures, functions, variables, and types, providing a way to organize and share code.

TRIGGER : A PL/SQL block that automatically executes in response to a database event (like an INSERT, UPDATE, or DELETE) etc.