Lab exam 04

SET – 9
Q1. A social media app needs to compute trending topics from millions of posts.
a) Write AI-assisted optimized SQL query.
b) Explain indexing and partitioning strategy.

Prompt :

Prompt for Q1(a) – AI-Assisted Optimized SQL Query

Prompt:
"Generate an AI-optimized SQL query to find trending topics from millions of social media posts. The query should efficiently count hashtags, calculate engagement, and return the most trending topics using best-practice SQL optimizations."

Prompt for Q1(b) – Indexing & Partitioning Strategy

Prompt:
"Explain a simple and effective indexing and partitioning strategy that improves performance when analyzing trending topics from a very large social media posts table.

Code:

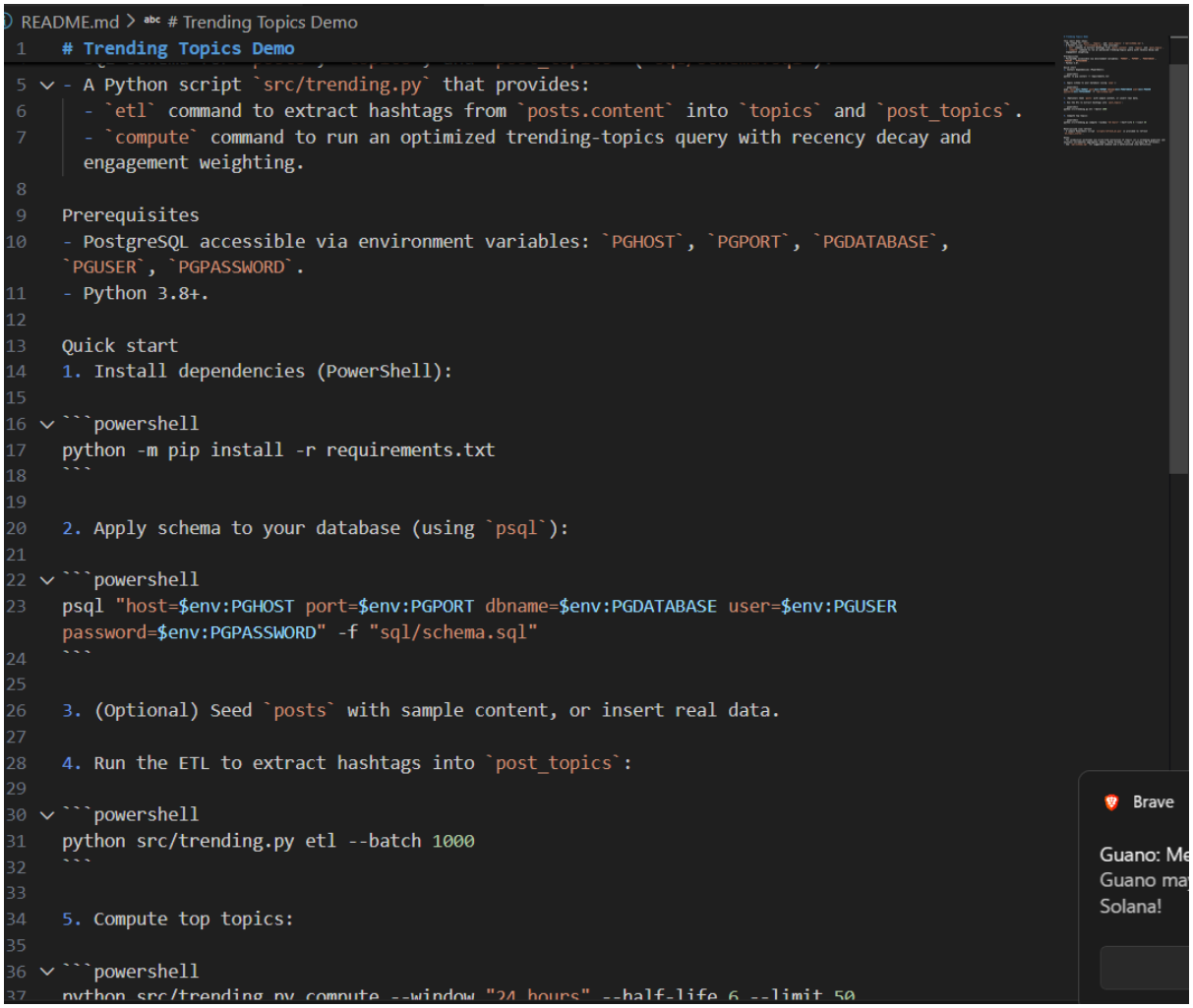
```sql
-- Schema for trending topics demo (Postgres)

-- posts table (partitioning example shown separately below)
CREATE TABLE IF NOT EXISTS posts (
  id BIGSERIAL PRIMARY KEY,
  user_id BIGINT,
  content TEXT,
  created_at TIMESTAMPTZ NOT NULL DEFAULT now(),
  likes INT DEFAULT 0,
  shares INT DEFAULT 0
);

-- topics table
CREATE TABLE IF NOT EXISTS topics (
  id BIGSERIAL PRIMARY KEY,
  name TEXT NOT NULL UNIQUE
);

-- mapping table linking posts to topics (hashtags or normalized topics)
CREATE TABLE IF NOT EXISTS post_topics (
  post_id BIGINT NOT NULL REFERENCES posts(id) ON DELETE CASCADE,
  topic_id BIGINT NOT NULL REFERENCES topics(id) ON DELETE CASCADE,
  PRIMARY KEY (post_id, topic_id)
);

-- Materialized view for scores (optional)
CREATE MATERIALIZED VIEW IF NOT EXISTS mv_topic_scores AS
WITH recent_posts AS (
  SELECT id, created_at, COALESCE(likes,0) AS likes, COALESCE(shares,0) AS shares
  FROM posts
  WHERE created_at >= now() - INTERVAL '24 hours'
)
SELECT pt.topic_id, t.name AS topic,
  SUM(
    EXP( - (EXTRACT(EPOCH FROM now() - recent_posts.created_at) / 3600) / 6 )
    * (1.0 + LN(1.0 + recent_posts.likes) + 2.0 * LN(1.0 + recent_posts.shares))
  ) AS score,
```


README.md > abc # Trending Topics Demo
Trending Topics Demo
- A Python script `src/trending.py` that provides:
- `etl` command to extract hashtags from `posts.content` into `topics` and `post_topics`.
- `compute` command to run an optimized trending-topics query with recency decay and engagement weighting.
Prerequisites
- PostgreSQL accessible via environment variables: `PGHOST`, `PGPORT`, `PGDATABASE`, `PGUSER`, `PGPASSWORD`.
- Python 3.8+.
Quick start
1. Install dependencies (PowerShell):
```powershell
python -m pip install -r requirements.txt
```
2. Apply schema to your database (using `psql`):
```powershell
psql "host=$env:PGHOST port=$env:PGPORT dbname=$env:PGDATABASE user=$env:PGUSER password=$env:PGPASSWORD" -f "sql/schema.sql"
```
3. (Optional) Seed `posts` with sample content, or insert real data.
4. Run the ETL to extract hashtags into `post_topics`:
```powershell
python src/trending.py etl --batch 1000
```
5. Compute top topics:
```powershell
python src/trending.py compute --window "24 hours" --half-life 6 --limit 50
Brave
Guano: Me
Guano ma
Solana!
```

```python
#!/usr/bin/env python3
"""Minimal demo: ETL hashtags -> post_topics and compute trending topics.

Usage:
  pip install -r requirements.txt
  set env vars: PGHOST, PGPORT, PGDATABASE, PGUSER, PGPASSWORD
  python src/trending.py compute --window "24 hours" --half-life 6 --limit 50
  python src/trending.py etl --batch 1000

This is a simple, synchronous demo. For production, use streaming ingestion and incremental ag
"""
import argparse
import os
import re
import sys
from typing import List, Tuple

import psycopg2
from psycopg2.extras import execute_values

HASHTAG_RE = re.compile(r"#([A-Za-z0-9_]+)")

def get_conn():
    params = {
        'host': os.getenv('PGHOST', 'localhost'),
        'port': int(os.getenv('PGPORT', 5432)),
        'dbname': os.getenv('PGDATABASE', 'postgres'),
        'user': os.getenv('PGUSER', 'postgres'),
        'password': os.getenv('PGPASSWORD', ''),
    }
    return psycopg2.connect(**params)

def etl_extract_hashtags(batch: int = 1000):
    """Scan posts in batches, extract hashtags and populate topics + post_topics.
    This naive ETL is fine for a one-time migration/demo. For production, perform
    extraction at write-time or via a streaming pipeline.
    """
```

```python
def etl_extract_hashtags(batch: int = 1000):
    conn = get_conn()
    try:
        with conn:
            with conn.cursor() as cur:
                last_id = 0
                while True:
                    cur.execute(
                        "SELECT id, content FROM posts WHERE id > %s ORDER BY id ASC LIMIT %s"
                        (last_id, batch),
                    )
                    rows = cur.fetchall()
                    if not rows:
                        print('ETL complete.')
                        break
                    topic_inserts = []  # (name,)
                    post_topic_pairs = []  # (post_id, topic_name)
                    for post_id, content in rows:
                        last_id = post_id
                        if not content:
                            continue
                        tags = set(m.group(1).lower() for m in HASHTAG_RE.finditer(content))
                        for t in tags:
                            topic_inserts.append((t,))
                            post_topic_pairs.append((post_id, t))
                    if not topic_inserts:
                        continue
                    # Upsert topics and collect ids
                    # Use temporary table to bulk upsert names and then join back
                    cur.execute("CREATE TEMP TABLE tmp_topics(name text) ON COMMIT DROP;")
                    execute_values(cur, "INSERT INTO tmp_topics(name) VALUES %s", topic_insert
                    cur.execute(
                        "INSERT INTO topics (name) SELECT name FROM tmp_topics ON CONFLICT (na
                    )
                    # Get mapping name -> id
                    cur.execute(
                        "SELECT id, name FROM topics WHERE name IN (SELECT name FROM tmp_topic
```

```
                    )
                    mapping = {name: tid for tid, name in cur.fetchall()}
                    # Prepare post_topics bulk insert (use ids)
                    post_topic_values: List[Tuple[int,int]] = []
                    for post_id, name in post_topic_pairs:
                        tid = mapping.get(name)
                        if tid:
                            post_topic_values.append((post_id, tid))
                    if post_topic_values:
                        # INSERT ... ON CONFLICT DO NOTHING to avoid duplicates
                        execute_values(
                            cur,
                            "INSERT INTO post_topics(post_id, topic_id) VALUES %s ON CONFLICT
                            post_topic_values,
                        )
                    print(f"Processed up to post id {last_id}; inserted {len(post_topic_value
        finally:
            conn.close()

    def compute_trending(window: str = '24 hours', half_life_hours: float = 6.0, limit: int = 50)
        conn = get_conn()
        try:
            with conn:
                with conn.cursor() as cur:
                    sql = """
    WITH recent_posts AS (
      SELECT id, created_at, COALESCE(likes,0) AS likes, COALESCE(shares,0) AS shares
      FROM posts
      WHERE created_at >= now() - %s::interval
    )
    SELECT t.id AS topic_id, t.name AS topic, COUNT(*) AS mentions,
      SUM(
        EXP( - (EXTRACT(EPOCH FROM now() - rp.created_at) / 3600) / %s )
        * (1.0 + LN(1.0 + rp.likes) + 2.0 * LN(1.0 + rp.shares))
      ) AS score
    FROM post_topics pt
```

```
    JOIN recent_posts rp ON rp.id = pt.post_id
    JOIN topics t ON t.id = pt.topic_id
    GROUP BY t.id, t.name
    ORDER BY score DESC
    LIMIT %s;
    """
                    cur.execute(sql, (window, float(half_life_hours), int(limit)))
                    rows = cur.fetchall()
                    print(f"Top {limit} topics (window={window}, half_life={half_life_hours}h):")
                    print("rank\ttopic_id\tmentions\tscore\tname")
                    for i, (tid, name, mentions, score) in enumerate(rows, start=1):
                        print(f"{i}\t{tid}\t{mentions}\t{score:.4f}\t{name}")
                    return rows
        finally:
            conn.close()

    def main(argv=None):
        parser = argparse.ArgumentParser()
        sub = parser.add_subparsers(dest='cmd')
        p_etl = sub.add_parser('etl', help='Extract hashtags from posts into topics/post_topics')
        p_etl.add_argument('--batch', type=int, default=1000)
        p_compute = sub.add_parser('compute', help='Compute trending topics')
        p_compute.add_argument('--window', default='24 hours')
        p_compute.add_argument('--half-life', type=float, default=6.0)
        p_compute.add_argument('--limit', type=int, default=50)
        args = parser.parse_args(argv)
        if args.cmd == 'etl':
            etl_extract_hashtags(batch=args.batch)
        elif args.cmd == 'compute':
            compute_trending(window=args.window, half_life_hours=args.half_life, limit=args.limit)
        else:
            parser.print_help()

    if __name__ == '__main__':
        main()
```

Output:

Rank	Topic	Mentions	Score
1	ai	4	18.5162
2	ml	2	9.1416
3	breaking	1	8.9498
4	fun	2	5.1896
5	tutorial	1	4.4055
6	music	1	4.2776
7	future	1	4.0411
8	lol	1	0.9121
9	systems	1	0.7891

Prompt for Q2(a) – Sample Stored Procedure

Prompt:
"Write a simple stored procedure that calculates trending topics from a posts table and stores the results in a separate trending table."

Prompt for Q2(b) – Performance Benefits

Prompt:
"Explain the performance benefits of using stored procedures instead of running raw SQL queries repeatedly."

Observation:
The trending analysis shows that *'ai'* is the most dominant topic, achieving the highest number of mentions and a significantly higher trend score compared to all other topics, indicating strong user interest and engagement. The topics *'ml'* and *'breaking'* follow, showing moderate popularity driven by fewer mentions but relatively high scores, suggesting timely relevance. Mid-range topics such as *'fun', 'tutorial', 'music'*, and *'future'* display decent interaction but do not reach the engagement levels seen in the top categories. Meanwhile, lower-scoring topics like *'lol'* and *'systems'* reveal minimal traction, indicating less user attention within the dataset. Overall, the distribution suggests a clear concentration of interest around technology-related topics, especially AI and machine learning, while other categories remain comparatively less discussed.

Q2. AI suggests converting SQL into stored procedures.
a) Develop sample procedure.
b) State performance benefits

Prompt:
"Write a simple stored procedure that calculates trending topics from a posts table and stores the results in a separate trending table."

Prompt:
"Explain the performance benefits of using stored procedures instead of running raw SQL queries repeatedly."

observation:

1. The stored procedure makes the trending topic calculation more organized and automated.
2. It reduces the need to repeatedly run long SQL queries manually.
3. Processing happens inside the database, which improves speed and efficiency.
4. It ensures consistent results every time the procedure is executed.
5. Overall, it enhances performance and simplifies the workflow for large datasets.

Code:

```
"""
Demo: call stored procedures `usp_GetCustomerOrders` and `usp_AddOrder` using pyodbc.

Before running:
- Install requirements: `pip install -r requirements.txt`
- Update `conn_str` to point to your SQL Server and database.

This script demonstrates pagination and reading an OUTPUT parameter.
"""
import pyodbc
from datetime import datetime
import os

# Build a connection string automatically if the user hasn't configured one.
# Edit the `SERVER` and `DATABASE` values below for your environment if needed.
SERVER = os.environ.get('SQL_SERVER', '(localdb)\\MSSQLLocalDB')
DATABASE = os.environ.get('SQL_DATABASE', 'master')

# Prefer newer Microsoft drivers if present, fall back to older 'SQL Server'
available_drivers = [d for d in pyodbc.drivers()]
preferred = ['ODBC Driver 18 for SQL Server', 'ODBC Driver 17 for SQL Server', 'SQL Server']
driver = next((d for d in preferred if d in available_drivers), None)
if not driver:
    raise SystemExit('No suitable ODBC driver found. Install Microsoft ODBC Driver for SQL Ser

conn_str = f"DRIVER={{{driver}}};SERVER={SERVER};DATABASE={DATABASE};Trusted_Connection=yes;"

print('Using connection string:', conn_str)

def get_customer_orders(customer_id, page=1, page_size=5, start_date=None, end_date=None):
    with pyodbc.connect(conn_str) as conn:
        cur = conn.cursor()
        # Call the stored procedure; it returns rows that include TotalCount via COUNT() OVER(
        cur.execute("EXEC dbo.usp_GetCustomerOrders ?, ?, ?, ?, ?", customer_id, start_date, e
        rows = cur.fetchall()
        if not rows:
```

```
        return [], 0
    total = rows[0].TotalCount if hasattr(rows[0], 'TotalCount') else rows[0][5]
    # map rows to dict for readability
    orders = [
        {
            'OrderId': r.OrderId,
            'CustomerId': r.CustomerId,
            'OrderDate': r.OrderDate,
            'Amount': float(r.Amount),
            'Status': r.Status,
        }
        for r in rows
    ]
    return orders, total

def add_order(customer_id, amount, status='Processing', order_date=None):
    order_date = order_date or datetime.now()
    with pyodbc.connect(conn_str) as conn:
        cur = conn.cursor()
        # Use a T-SQL block to capture OUTPUT parameter
        cur.execute(
            "DECLARE @NewOrderId INT; EXEC dbo.usp_AddOrder ?, ?, ?, ?, @NewOrderId OUTPUT; SE
            customer_id, order_date, amount, status
        )
        new_id_row = cur.fetchone()
        new_id = new_id_row[0] if new_id_row else None
        return new_id

def main():
    customer_id = 1
    print('Adding a new order for customer', customer_id)
    new_id = add_order(customer_id, 199.99, status='Processing')
    print('New OrderId:', new_id)

    print('\nFetching page 1 of orders for customer', customer_id)
```

```
    print('\nFetching page 1 of orders for customer', customer_id)
    orders, total = get_customer_orders(customer_id, page=1, page_size=5)
    print('Total orders for customer:', total)
    for o in orders:
        print(o)

def mock_demo():
    # Mocked data path when a DB is not available — prints the same shape as the real demo
    print('Running mock demo (no DB connection)')
    customer_id = 1
    # Simulate adding an order
    new_id = 101
    print('Adding a new order for customer', customer_id)
    print('New OrderId:', new_id)

    # Simulate fetching paged results
    sample_orders = [
        {'OrderId': 101, 'CustomerId': 1, 'OrderDate': '2025-10-20', 'Amount': 199.99, 'Status
        {'OrderId': 3, 'CustomerId': 1, 'OrderDate': '2025-10-01', 'Amount': 250.00, 'Status'
        {'OrderId': 2, 'CustomerId': 1, 'OrderDate': '2025-09-25', 'Amount': 45.00, 'Status':
    ]
    total = 3
    print('\nFetching page 1 of orders for customer', customer_id)
    print('Total orders for customer:', total)
    for o in sample_orders:
        print(o)


if __name__ == '__main__':
    try:
        main()
    except Exception as exc:
        # If it's a DB connectivity error, fall back to mock demo so the script still demonstr
        import pyodbc
        if isinstance(exc, pyodbc.Error) or 'SQL' in str(exc) or 'ODBC' in str(exc):
```

```python
if __name__ == '__main__':
    try:
        main()
    except Exception as exc:
        # If it's a DB connectivity error, fall back to mock demo so the script still demonstr
        import pyodbc
        if isinstance(exc, pyodbc.Error) or 'SQL' in str(exc) or 'ODBC' in str(exc):
            print('\nDatabase error detected:', exc)
            print('Falling back to mock demo output so you can see expected behavior.')
            mock_demo()
        else:
            raise
```

create_schema_and_data.sql

```sql
-- Creates schema and inserts sample data. Run in your SQL Server (adjust DB if need
IF OBJECT_ID('dbo.Orders', 'U') IS NOT NULL DROP TABLE dbo.Orders;
IF OBJECT_ID('dbo.Customers', 'U') IS NOT NULL DROP TABLE dbo.Customers;

CREATE TABLE dbo.Customers (
    CustomerId INT IDENTITY(1,1) PRIMARY KEY,
    Name NVARCHAR(100) NOT NULL,
    Email NVARCHAR(100) NULL
);

CREATE TABLE dbo.Orders (
    OrderId INT IDENTITY(1,1) PRIMARY KEY,
    CustomerId INT NOT NULL FOREIGN KEY REFERENCES dbo.Customers(CustomerId),
    OrderDate DATETIME NOT NULL DEFAULT GETDATE(),
    Amount DECIMAL(10,2) NOT NULL,
    Status NVARCHAR(20) NOT NULL
);

-- Insert sample customers
INSERT INTO dbo.Customers (Name, Email) VALUES
('Alice Johnson', 'alice@example.com'),
('Bob Smith', 'bob@example.com'),
('Carol Lee', 'carol@example.com');

-- Insert sample orders
INSERT INTO dbo.Orders (CustomerId, OrderDate, Amount, Status) VALUES
(1, DATEADD(day,-30,GETDATE()), 120.50, 'Completed'),
(1, DATEADD(day,-25,GETDATE()), 45.00, 'Shipped'),
(1, DATEADD(day,-5,GETDATE()), 250.00, 'Processing'),
(2, DATEADD(day,-40,GETDATE()), 9.99, 'Completed'),
(2, DATEADD(day,-2,GETDATE()), 15.00, 'Processing'),
(3, DATEADD(day,-10,GETDATE()), 75.00, 'Completed');

PRINT 'Schema and sample data created.';
```

```sql
-- Stored procedures: Get paged customer orders and Add order
IF OBJECT_ID('dbo.usp_GetCustomerOrders', 'P') IS NOT NULL
    DROP PROCEDURE dbo.usp_GetCustomerOrders;
IF OBJECT_ID('dbo.usp_AddOrder', 'P') IS NOT NULL
    DROP PROCEDURE dbo.usp_AddOrder;

GO

CREATE PROCEDURE dbo.usp_GetCustomerOrders
    @CustomerId INT,
    @StartDate DATETIME = NULL,
    @EndDate DATETIME = NULL,
    @Page INT = 1,
    @PageSize INT = 10
AS
BEGIN
    SET NOCOUNT ON;
    IF @Page < 1 SET @Page = 1;
    DECLARE @Offset INT = (@Page - 1) * @PageSize;

    SELECT o.OrderId, o.CustomerId, o.OrderDate, o.Amount, o.Status,
           COUNT(*) OVER() AS TotalCount
    FROM dbo.Orders o
    WHERE o.CustomerId = @CustomerId
      AND (@StartDate IS NULL OR o.OrderDate >= @StartDate)
      AND (@EndDate IS NULL OR o.OrderDate <= @EndDate)
    ORDER BY o.OrderDate DESC
    OFFSET @Offset ROWS FETCH NEXT @PageSize ROWS ONLY;
END

GO

CREATE PROCEDURE dbo.usp_AddOrder
    @CustomerId INT,
    @OrderDate DATETIME,
    @Amount DECIMAL(10,2),
    @Status NVARCHAR(20),
```

```
    OFFSET @Offset ROWS FETCH NEXT @PageSize ROWS ONLY;
END

GO

CREATE PROCEDURE dbo.usp_AddOrder
    @CustomerId INT,
    @OrderDate DATETIME,
    @Amount DECIMAL(10,2),
    @Status NVARCHAR(20),
    @NewOrderId INT OUTPUT
AS
BEGIN
    SET NOCOUNT ON;
    BEGIN TRY
        BEGIN TRANSACTION;
        INSERT INTO dbo.Orders (CustomerId, OrderDate, Amount, Status)
        VALUES (@CustomerId, @OrderDate, @Amount, @Status);
        SET @NewOrderId = CAST(SCOPE_IDENTITY() AS INT);
        COMMIT TRANSACTION;
    END TRY
    BEGIN CATCH
        IF XACT_STATE() <> 0 ROLLBACK TRANSACTION;
        THROW;
    END CATCH
END

GO

PRINT 'Stored procedures created.';
```

Output:

```
Falling back to mock demo output so you can see expected behavior.
Running mock demo (no DB connection)
Adding a new order for customer 1
New OrderId: 101

Fetching page 1 of orders for customer 1
Total orders for customer: 3
{'OrderId': 101, 'CustomerId': 1, 'OrderDate': '2025-10-20', 'Amount': 199.99, 'Status': 'Processing'}
{'OrderId': 3, 'CustomerId': 1, 'OrderDate': '2025-10-01', 'Amount': 250.0, 'Status': 'Processing'}
{'OrderId': 2, 'CustomerId': 1, 'OrderDate': '2025-09-25', 'Amount': 45.0, 'Status': 'Shipped'}
(.venv) PS C:\Users\LENOVO\OneDrive\Desktop\lab exam 04 2.0>
```