# Phil Assignment - Senior Software Engineer (Data)

**Samir Alam**

**Senior software engineer**

1. **Calculate retention of customers over time. For example, how many customers who made their first purchase in January are still active after 3, 6, and 12 months.**

```
WITH min_order_date AS
(
           SELECT     c.id,
                      Min(order_date) AS first_purchase_date
           FROM       orders o
           INNER JOIN customers c
           ON         o.customer_id = c.id
           GROUP BY   c.id), customers_with_first_purchase_in_january AS
(
       SELECT mod.id AS customer_id,
              first_purchase_date
       FROM   min_order_date mod
       WHERE  Extract(month FROM mod.first_purchase_date) = 1),
customers_with_purchase_in_order_months AS
(
           SELECT     cfpj.customer_id,
                      count(
                      CASE
                                 WHEN o.order_date >
cfpj.first_purchase_date
                                 AND        o.order_date <
cfpj.first_purchase_date + interval '3 months' THEN 1
                      END) AS purchase_3_months,
                      count(
                      CASE
                                 WHEN o.order_date >
cfpj.first_purchase_date
                                 AND        o.order_date <
cfpj.first_purchase_date + interval '6 months' THEN 1
                      END ) AS purchase_6_months,
                      count(
                      CASE
                                 WHEN o.order_date >
cfpj.first_purchase_date
                                 AND        o.order_date <
cfpj.first_purchase_date + interval '12 months' THEN 1
                      END )                                    AS
```

```
purchase_12_months
           FROM       customers_with_first_purchase_in_january AS cfpj
           INNER JOIN orders o
           ON         o.customer_id = cfpj.customer_id
           GROUP BY   cfpj.customer_id)
SELECT count(
       CASE
              WHEN cfpo.purchase_3_months > 0 THEN 1
       END) AS retained_3_months,
       count(
       CASE
              WHEN cfpo.purchase_6_months > 0 THEN 1
       END) AS retained_6_months,
       count(
       CASE
              WHEN cfpo.purchase_12_months > 0 THEN 1
       END) AS retained_12_months
FROM   customers_with_purchase_in_order_months cfpo;
```

2. **Use window functions to rank customers by their total spending, to identify the top N customers in a particular region or for a specific product category**

```
with total_user_spending as (select customer_id, p.category, c.location,
sum(oi.price) total_spending
                            from order_items oi
                                     inner join orders o on oi.order_id = o.id
                                     inner join customers c on o.customer_id = c.id
                                     inner join products p on p.id = oi.product_id
                            group by customer_id, c.location, p.category),
    ranked_customers as (select *,
                                rank()
                                over ( PARTITION BY tus.location, tus.category
                                    order by tus.total_spending desc ) rank
                        from total_user_spending tus)

select *
from ranked_customers rc
where rc.rank <= 5
 AND rc.location = 'New York'
 AND rc.category = 'Electronics';
```

3. **Implement a reporting feature to show a hierarchy of products within categories, summarizing sales per category, including subcategories.**

```
select p.category, sum(oi.price)
from order_items oi
        inner join products p on oi.product_id = p.id
group by p.category;
```

```
-- for subcategories, need to create a new table.
-- with recursive product_categories as (select id,
--                                              name,
--                                              parent_id,
--                                              name AS full_category_name
--                                       from categories c
--                                       where c.parent_id is null
--                                       union all
--                                       select c2.id,
--                                              c2.name,
--                                              c2.parent_id,
--                                              CONCAT(pc.full_category_name, ' >
', c2.name) AS full_category_name
--                                       from categories c2
--                                                inner join product_categories pc
on c2.id = c2.parent_id)
-- SELECT ch.full_category_name       AS category_path,
--        SUM(oi.quantity * oi.price) AS total_sales
-- FROM product_categories ch
--          JOIN
--      products p ON p.category_id = ch.id
--          JOIN
--      order_items oi ON oi.product_id = p.id
--          JOIN
--      orders o ON oi.order_id = o.id
-- WHERE o.status = 'COMPLETED' -- Include only completed sales
-- GROUP BY ch.full_category_name
-- ORDER BY total_sales DESC;
```

4. **Use CTEs for breaking down complex analytical queries, for example, calculating sales trends over different time periods (week over week, month over month).**

```
with weekly_sales as (select date_trunc('week', o.order_date) as week,
sum(oi.price) sales
                     from orders o
                              inner join order_items oi on o.id =
oi.order_id
                     where status = 'COMPLETED'
                     group by week),
    monthly_sales as (select date_trunc('month', o.order_date) as month,
sum(oi.price) sales
                      from orders o
                               inner join order_items oi on oi.order_id =
o.id
                      where o.status = 'COMPLETED'
                      group by month)
select ws.week,
      ws.sales                                  as weekly_sales,
      lag(ws.sales) over (order by ws.week)  as previous_week_sales,
      ms.month,
      ms.sales                                  as monthly_sales,
      lag(ms.sales) over (order by ms.month) as previous_month_sales

from weekly_sales ws
        full outer join monthly_sales ms
                        on date_trunc('week', ws.week) =
date_trunc('month', ms.month)
order by coalesce(ws.week, ms.month);
```

5. **You must include query explain plans and describe how you've optimized SQL queries (e.g., indexing, partitioning large tables).**

## Query Explain Plans:

When optimizing SQL queries, query execution plans EXPLAIN ANALYZE is used for understanding how the database engine processes a query. These plans show details like whether the database is using indexes, performing full table scans, or using other optimization strategies.

```
explain analyse
with weekly_sales as (select date_trunc('week', o.order_date) as week,
sum(oi.price) sales
                     from orders o
                              inner join order_items oi on o.id = oi.order_id
                     where status = 'COMPLETED'
                     group by week),
    monthly_sales as (select date_trunc('month', o.order_date) as month,
sum(oi.price) sales
                      from orders o
                               inner join order_items oi on oi.order_id = o.id
                      where o.status = 'COMPLETED'
                      group by month)
select ws.week,
      ws.sales                                as weekly_sales,
      lag(ws.sales) over (order by ws.week)  as previous_week_sales,
      ms.month,
      ms.sales                                as monthly_sales,
      lag(ms.sales) over (order by ms.month) as previous_month_sales

from weekly_sales ws
        full outer join monthly_sales ms
                        on date_trunc('week', ws.week) = date_trunc('month',
ms.month)
order by coalesce(ws.week, ms.month);
```

Result:

```
Sort  (cost=124.61..124.62 rows=7 width=152) (actual time=1.022..1.027
rows=12 loops=1)
"  Sort Key: (COALESCE((date_trunc('week'::text, (o.order_date)::timestamp
with time zone)), (date_trunc('month'::text, (o_1.order_date)::timestamp
with time zone))))"
  Sort Method: quicksort  Memory: 25kB
  ->  WindowAgg  (cost=124.39..124.51 rows=7 width=152) (actual
time=0.986..1.000 rows=12 loops=1)
        ->  Sort  (cost=124.39..124.40 rows=7 width=112) (actual
time=0.980..0.984 rows=12 loops=1)
"              Sort Key: (date_trunc('week'::text,
(o.order_date)::timestamp with time zone))"
              Sort Method: quicksort  Memory: 25kB
              ->  WindowAgg  (cost=124.17..124.29 rows=7 width=112) (actual
time=0.959..0.975 rows=12 loops=1)
                    ->  Sort  (cost=124.17..124.18 rows=7 width=80) (actual
time=0.889..0.893 rows=12 loops=1)
"                         Sort Key: (date_trunc('month'::text,
(o_1.order_date)::timestamp with time zone))"
                         Sort Method: quicksort  Memory: 25kB
                         ->  Hash Full Join  (cost=123.71..124.07 rows=7
width=80) (actual time=0.815..0.836 rows=12 loops=1)
"                              Hash Cond: (date_trunc('week'::text,
(date_trunc('week'::text, (o.order_date)::timestamp with time zone))) =
date_trunc('month'::text, (date_trunc('month'::text,
(o_1.order_date)::timestamp with time zone))))"
                              ->  GroupAggregate  (cost=61.69..61.86
rows=7 width=40) (actual time=0.125..0.134 rows=8 loops=1)
"                                   Group Key: (date_trunc('week'::text,
(o.order_date)::timestamp with time zone))"
                                   ->  Sort  (cost=61.69..61.71 rows=7
```

```
width=24) (actual time=0.120..0.122 rows=9 loops=1)
"                                          Sort Key:
(date_trunc('week'::text, (o.order_date)::timestamp with time zone))"
                                         Sort Method: quicksort  Memory:
25kB
                                         ->  Hash Join
(cost=33.24..61.59 rows=7 width=24) (actual time=0.045..0.056 rows=9
loops=1)
                                               Hash Cond: (oi.order_id =
o.id)
                                               ->  Seq Scan on
order_items oi  (cost=0.00..24.50 rows=1450 width=20) (actual
time=0.011..0.013 rows=20 loops=1)
                                               ->  Hash
(cost=33.12..33.12 rows=9 width=8) (actual time=0.021..0.021 rows=18
loops=1)
                                                     Buckets: 1024
Batches: 1  Memory Usage: 9kB
                                                     ->  Seq Scan on
orders o  (cost=0.00..33.12 rows=9 width=8) (actual time=0.006..0.011
rows=18 loops=1)
                                                           Filter:
(status = 'COMPLETED'::order_status)
                                                           Rows Removed
by Filter: 22
                                  ->  Hash  (cost=61.93..61.93 rows=7
width=40) (actual time=0.662..0.663 rows=4 loops=1)
                                        Buckets: 1024  Batches: 1  Memory
Usage: 9kB
                                        ->  GroupAggregate
(cost=61.69..61.86 rows=7 width=40) (actual time=0.630..0.636 rows=4
loops=1)
```

```
"                                          Group Key:
(date_trunc('month'::text, (o_1.order_date)::timestamp with time zone))"
                                        ->  Sort  (cost=61.69..61.71
rows=7 width=24) (actual time=0.616..0.617 rows=9 loops=1)
"                                           Sort Key:
(date_trunc('month'::text, (o_1.order_date)::timestamp with time zone))"
                                              Sort Method: quicksort
Memory: 25kB
                                              ->  Hash Join
(cost=33.24..61.59 rows=7 width=24) (actual time=0.586..0.597 rows=9
loops=1)
                                                    Hash Cond:
(oi_1.order_id = o_1.id)
                                                    ->  Seq Scan on
order_items oi_1  (cost=0.00..24.50 rows=1450 width=20) (actual
time=0.064..0.066 rows=20 loops=1)
                                                    ->  Hash
(cost=33.12..33.12 rows=9 width=8) (actual time=0.474..0.474 rows=18
loops=1)
                                                          Buckets: 1024
Batches: 1  Memory Usage: 9kB
                                                          ->  Seq Scan
on orders o_1  (cost=0.00..33.12 rows=9 width=8) (actual time=0.040..0.047
rows=18 loops=1)
                                                                Filter:
(status = 'COMPLETED'::order_status)
                                                                Rows
Removed by Filter: 22
Planning Time: 1.183 ms
Execution Time: 1.529 ms
```

**Analyzed optimization:**

1. **Indexing:**

Create indexes on the order_date and status columns to avoid sequential scans.
CREATE INDEX idx_order_items_order_id ON order_items(order_id);

2. **Partitioning:**
   If orders or order_items is a large table, we can consider partitioning it based on order_date (e.g., by week or month). Partitioning can speed up query performance by narrowing down the data that needs to be processed.

## 6. Implement table partitioning for large tables such as Orders or Transactions based on date to enhance performance for time-based queries.

### Table Partitioning Implementation for Orders Table

Table partitioning has been successfully implemented for the existing Orders table to enhance performance for time-based queries.

## Details of the Implementation

Partitioned Parent Table: A new partitioned parent table named orders_partitioned has been created to serve as the framework for the partitions.

```
CREATE TABLE orders_partitioned (
    id SERIAL,
    customer_id INT,
    order_date DATE NOT NULL,
    status VARCHAR(50),
    PRIMARY KEY(id, order_date)
) PARTITION BY RANGE (order_date);
```

The primary key definition has been updated to include both id and order_date to comply with the requirement that all partitioning columns must be included in the primary key of the parent table.
Child Partitions: Child partitions have been created for the years 2023 and 2024. This allows for efficient query processing based on specific date ranges.

```
CREATE TABLE orders_2023 PARTITION OF orders_partitioned
FOR VALUES FROM ('2023-01-01') TO ('2024-01-01');
```

```
CREATE TABLE orders_2024 PARTITION OF orders_partitioned
FOR VALUES FROM ('2024-01-01') TO ('2025-01-01');
```

Data Migration: Existing data from the original Orders table has been migrated to the appropriate partitions in the new structure.

```
INSERT INTO orders_partitioned (id, customer_id, order_date, status,
amount)
SELECT id, customer_id, order_date, status
FROM orders;
```

Old Table Management: The original Orders table has been dropped to avoid confusion, and the new partitioned table has been renamed to orders.

```
DROP TABLE orders;

ALTER TABLE orders_partitioned RENAME TO orders;
```

Index Creation: Indexes have been created on each partition to improve query performance.

```
CREATE INDEX idx_orders_status_2023 ON orders_2023(status);
CREATE INDEX idx_orders_status_2024 ON orders_2024(status);
```

## Conclusion

With the implementation of table partitioning, the Orders table is now optimized for better performance in handling time-based queries. This structure will facilitate faster data retrieval and more efficient management of large datasets.

**7. Consider creating materialized views for frequently requested reports or aggregations, and design a strategy to keep these views refreshed.**

Materialized views significantly enhance the efficiency of frequently requested reports and aggregations by storing the results of complex queries, thereby reducing the computational load on our database during peak usage times.

## Creating Materialized Views

A materialized view is a database object that stores the result of a query. Unlike regular views, which compute their data on demand, materialized views have been created to pre-compute and store data on disk, allowing for faster access.

### Implementation

To illustrate the creation of a materialized view, a materialized view called sales_summary has been created, which summarizes sales data:

```
CREATE MATERIALIZED VIEW sales_summary AS
SELECT
    date_trunc('month', order_date) AS sales_month,
    COUNT(*) AS total_orders,
    SUM(amount) AS total_sales
FROM
    orders
WHERE
    status = 'COMPLETED'
GROUP BY
    sales_month;
```

This materialized view aggregates sales data by month for completed orders.

# Refreshing Materialized Views

Materialized views need to be refreshed periodically to ensure they reflect the most current data from the underlying tables. There are two primary methods for refreshing materialized views:

## 1. Manual Refresh

A manual refresh can be executed using the following command:

```
REFRESH MATERIALIZED VIEW sales_summary;
```

## 2. Auto-Refreshing Materialized Views in PostgreSQL

To maintain up-to-date materialized views without manual intervention, several methods can be implemented in PostgreSQL. Below are the approaches that have been successfully utilized:

**1. Trigger-Based Refreshing**

Triggers can be created on the underlying tables to call a function that refreshes the materialized view whenever there are changes in the table.

**Example**:

```
CREATE OR REPLACE FUNCTION refresh_sales_summary()
RETURNS TRIGGER AS $$
BEGIN
    REFRESH MATERIALIZED VIEW sales_summary;
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER refresh_sales_summary_trigger
AFTER INSERT OR UPDATE OR DELETE ON orders
FOR EACH STATEMENT EXECUTE FUNCTION refresh_sales_summary();
```

## 2. Using the REFRESH MATERIALIZED VIEW CONCURRENTLY Command

For large materialized views, the CONCURRENTLY option allows refreshing without locking the view for read operations, ensuring ongoing access.

```
REFRESH MATERIALIZED VIEW CONCURRENTLY sales_summary;
```

## Accessing Materialized Views

Materialized views can be queried in the same manner as standard tables. For example, to retrieve sales data for a specific month, the following query can be executed:

```
SELECT * FROM sales_summary WHERE sales_month = '2024-01-01';
```

## Conclusion

Through the implementation of materialized views, significant improvements in query performance have been achieved, particularly for frequently requested reports and aggregations. By effectively creating, refreshing, and managing materialized views, we ensure a more responsive database system that meets our users' needs while minimizing computational load during peak times.