Task 1: Calculate a moving average of sales over time per product.

```
SELECT product_id, order_date, sales,
AVG(sales) OVER(partition by product_id order by order_date rows between 3 preceding and current row) as mvg_average
FROM orders;
```

Task 2: Find the 3rd highest salary:

```
WITH cte AS (
    SELECT salary ,
        DENSE_RANK() OVER(ORDER BY salary DESC) AS rnk
    FROM emp
    )
    SELECT salary FROM cte WHERE rnk=3;
```

Task 3: SQL Query Optimization:

Interviewer: *How do you optimize SQL queries for performance?*

Answer:

To optimize SQL queries, I follow a structured approach focused on analyzing the execution plan, using proper indexing, and writing efficient, sargable queries. Here's how I approach it:

1. 🔍 **Check the Execution Plan**

- I use EXPLAIN or EXPLAIN ANALYZE (in PostgreSQL) to understand how the query is executed.
- It helps identify:
    - Whether a **sequential scan** or **index scan** is being used
    - Join types like **Nested Loop** or **Hash Join**
    - Cost, estimated rows vs. actual rows
- Based on this, I locate bottlenecks such as full table scans, missing indexes, or inefficient joins.

2. ⚡ **Indexing Strategy**

- I add indexes to columns involved in WHERE, JOIN, ORDER BY, or GROUP BY.

- For queries filtering on multiple columns, I use **composite indexes**, keeping in mind that the **leading column** must match the query's filter.
- Example:
- CREATE INDEX idx_country_city ON customers(country, city);
- I avoid functions on indexed columns, like:
- WHERE UPPER(name) = 'JOHN'; -- Index ignored
- Instead, I either store a lowercase version or use a functional index.

3. ✂️ **Avoid SELECT ***

- I always select only the columns I need.
- SELECT id, name FROM employees WHERE department_id = 10;
- This reduces I/O and memory usage, and can enable **covering indexes** where queries are resolved using only the index.

4. ⚠️ **Write Sargable Conditions**

- I avoid wrapping columns in functions in the WHERE clause because it prevents index usage.
- For example:
  ❌ Bad:
- WHERE YEAR(order_date) = 2024;
- ✅ Better:
- WHERE order_date BETWEEN '2024-01-01' AND '2024-12-31';

5. 📄 **Use LIMIT and OFFSET for Pagination**

- For paginated results, I always apply:
- SELECT * FROM orders ORDER BY order_date DESC LIMIT 10 OFFSET 20;
- This improves response time and avoids unnecessary data transfer.

6. 🔄 **Avoid Redundant Subqueries**

- I eliminate repeated subqueries by using CTEs or joining pre-aggregated results.
- Example:
- WITH max_salary AS (
-   SELECT MAX(salary) AS ms FROM emp
-)
- SELECT e.name, m.ms
- FROM emp e
- JOIN max_salary m ON TRUE;

✅ Summary

Technique	Benefit
EXPLAIN / ANALYZE	Identifies performance bottlenecks
Indexing	Speeds up filtering, joins, sorting
Avoid SELECT *	Reduces memory and I/O usage
Sargable Conditions	Enables index usage
LIMIT/OFFSET	Efficient pagination
Avoid Redundant Logic	Reduces unnecessary computation

Task 4: View vs Materialized View + Materialized View Usage

1. **Difference Between View and Materialized View**

Feature	View	Materialized View
Stores Data?	❌ No (virtual, runs query live)	✅ Yes (stores query result)
Auto-Updates on base tables?	✅ Always fresh	❌ Requires manual refresh
Performance	Slower on complex queries	Faster, uses precomputed data
Storage	No extra storage	Uses disk space
Use Case	Simple queries, live data needed	Heavy/repeated queries, caching

2. **Create and Use a View**

CREATE VIEW top_customers AS

SELECT customer_id, SUM(amount) AS total

FROM orders

GROUP BY customer_id

ORDER BY total DESC

LIMIT 10;

Query the view:

SELECT * FROM top_customers;

3. **Create and Use a Materialized View (PostgreSQL example)**

Create:

CREATE MATERIALIZED VIEW top_customers_mv AS

SELECT customer_id, SUM(amount) AS total

FROM orders

GROUP BY customer_id

ORDER BY total DESC

LIMIT 10;

Query:

SELECT * FROM top_customers_mv;

Refresh:

REFRESH MATERIALIZED VIEW top_customers_mv;

Optionally, refresh concurrently (requires unique index):

REFRESH MATERIALIZED VIEW CONCURRENTLY top_customers_mv;

4. **Example: Cache Latest 10 Orders with Materialized View**

Create:

CREATE MATERIALIZED VIEW latest_orders_mv AS

SELECT order_id, customer_id, order_date, amount

FROM orders

ORDER BY order_date DESC

LIMIT 10;

Query:

SELECT * FROM latest_orders_mv;

Refresh periodically:

REFRESH MATERIALIZED VIEW latest_orders_mv;

5. **Example: Cache Latest 10 Orders with a Cache Table**

Create cache table:

CREATE TABLE latest_orders_cache (

  order_id INT,

  customer_id INT,

  order_date DATE,

  amount NUMERIC

);

Populate:

INSERT INTO latest_orders_cache

SELECT order_id, customer_id, order_date, amount

FROM orders

ORDER BY order_date DESC

LIMIT 10;

Refresh (e.g., via scheduled job):

TRUNCATE latest_orders_cache;

INSERT INTO latest_orders_cache

SELECT order_id, customer_id, order_date, amount

FROM orders

ORDER BY order_date DESC

LIMIT 10;

6. **How to Schedule Refresh (Brief)**

- Use **cron jobs** or **database scheduler** (like pgAgent, pg_cron) to run the refresh SQL at intervals.
- For example, a cron job running every 5 minutes could run:

psql -d your_database -c "REFRESH MATERIALIZED VIEW latest_orders_mv;"

Task 4: ACID Properties

ACID ensures **reliable and consistent transactions** in a database.

1. **A – Atomicity**

All or nothing.
A transaction must **completely succeed or completely fail**.
No partial updates.

✅ Example:
Money transfer → Deduct from A **and** add to B — both must happen, or none.

2. **C – Consistency**

Data integrity is maintained.
A transaction brings the database from one valid state to another, following rules (constraints, relationships).

✅ Example:
If balance must ≥ 0, transaction violating this is rolled back.

3. **I – Isolation**

Transactions don't interfere with each other.
Concurrent transactions must **not see partial results** of others.

✅ Example:
Two people booking last ticket — system ensures one gets it, no duplicates.

4. **D – Durability**

Once committed, it stays saved — even if system crashes.
Committed data is permanently stored.

✅ Example:
After successful transaction, power failure won’t lose the data.

💡 You can say:
“ACID properties ensure correctness, safety, and reliability of transactions in relational databases.”