Mastercard Data Engineer Interview Questions
(4–5 YOE)
20-25 LPA

SQL

1. HOW WOULD YOU OPTIMIZE A SLOW-RUNNING SQL QUERY?

Query optimization is a critical skill for a Data Engineer. Here are practical strategies:

Steps to Optimize a SQL Query:

1. Check Execution Plan:
   - o Use EXPLAIN (MySQL/PostgreSQL) or SET SHOWPLAN_ALL ON (SQL Server) to analyze how SQL is executed.
   - o Identify costly operations like full table scans, nested loops, or missing indexes.
2. Use Proper Indexing:
   - o Create indexes on frequently filtered/joined columns.
   - o Use composite indexes when filtering on multiple columns together.
   - o Avoid indexes on columns with high cardinality or frequent updates.
3. Avoid SELECT *:
   - o Only select required columns to reduce I/O load.
4. Use Joins Efficiently:
   - o Prefer INNER JOIN over OUTER JOIN if NULLs are not needed.
   - o Ensure joined fields are indexed.
5. Filter Early:

Harshavardhana I
Data Engineer

o Apply WHERE clauses early to limit the data set before joins and aggregations.

6. Avoid Subqueries When Possible:

   o Use JOINs or CTEs (Common Table Expressions) for better performance and readability.

7. Limit Use of Functions in WHERE Clauses:

```
-- Avoid this:
WHERE YEAR(order_date) = 2024


-- Prefer this:
WHERE order_date >= '2024-01-01' AND order_date < '2025-01-01'
```

8. Partitioning and Sharding (for big data):

   o Use table partitioning to divide large tables logically for faster access.

   o Consider sharding for distributed systems.

2. Write a SQL query to find the second highest salary in a table.

Let's say we have a table called employees(salary).

Query Using Subquery:

```
SELECT MAX(salary) AS second_highest_salary FROM
employees WHERE salary < (SELECT MAX(salary) FROM
employees);
```

Alternative Using DENSE_RANK() (SQL Server, PostgreSQL, etc.):

```
SELECT salary
FROM (
 SELECT salary, DENSE_RANK() OVER (ORDER BY salary DESC) AS rnk
 FROM employees
```

Harshavardhana I
Data Engineer

```
) ranked
WHERE rnk = 2;
```

Note: Use RANK() if you want to skip duplicates, DENSE_RANK() if not.

3. What’s the difference between the WHERE and HAVING clauses in SQL?

Feature	WHERE	HAVING Filters groups
Purpose	Filters rows before aggregation	after aggregation
Usage	Can be used with SELECT, UPDATE, DELETE	Typically used with GROUP BY
Aggregate Functions	Cannot use (SUM, AVG, etc.)	Can use
Example	WHERE salary > 50000	HAVING COUNT(*) > 3

Example:

```
-- Using WHERE
SELECT * FROM employees
WHERE department = 'IT';

-- Using HAVING
SELECT department, COUNT(*) AS total_employees
FROM employees
GROUP BY department
HAVING COUNT(*) > 5;
```

4. How do you typically handle NULL values in your SQL queries?

Harshavardhana I
Data Engineer

Key Strategies:

1. Use IS NULL or IS NOT NULL:

```
SELECT * FROM employees WHERE manager_id IS NULL;
```

2. Use COALESCE() or IFNULL():
   - Replace NULLs with default values.

```
SELECT name, COALESCE(department, 'Not Assigned') AS dept FROM employees;
```

3. Use CASE statements:

```
SELECT
 name,
 CASE
  WHEN salary IS NULL THEN 'Not
  Disclosed' ELSE salary
 END AS salary_status
FROM employees;
```

4. Avoid NULLs in joins:
   - Use INNER JOIN when NULLs are not needed.
   - Use LEFT JOIN + COALESCE if necessary.

5. NULL-safe comparison in MySQL:

```
SELECT * FROM table WHERE column <=> NULL; -- Only TRUE if column is NULL
```

Data Normalization

1. What is normalization, and why is it important in data modeling?

Normalization is the process of structuring a relational database to:

Harshavardhana I
Data Engineer

- Eliminate data redundancy (duplicate data) Ensure data
- integrity Make the database more efficient and easier to
- maintain

Key Normal Forms:

Normal Form		Example
1NF (First Normal Form)	No repeating groups; atomic columns only	Avoid arrays or multiple values in a single column
2NF (Second Normal Form)	1NF + No partial dependency on a primary key	Every non-key column depends on the whole key
3NF (Third Normal Form)	2NF + No transitive dependencies	No non-key column depends on another non-key column

Why Normalization is Important:

- Reduces data redundancy (e.g., no repeated customer info in each order row)
- Improves data consistency (update in one place only)
- Makes updates, deletions, and insertions safer
- Minimizes storage costs (by avoiding repetition)

However, in OLAP/data warehouses, denormalization (opposite of normalization) is preferred to optimize for query speed.

2. Explain the difference between a star schema and a snowflake schema.

Star Schema vs Snowflake Schema:

Feature	Star Schema	Snowflake Schema
Structure	Central fact table linked to dimension tables	Central fact table linked to normalized dimension tables
Normalization	Denormalized	Normalized
Query Performance	Faster (fewer joins)	Slightly slower (more joins)
Storage	Uses more space	Uses less space
Simplicity	Easier to understand and query	More complex

Star Schema Example: Fact Table:

Fact_Transactions (transaction_id, customer_id, product_id, amount, date_id)

Dimension Tables:

- Dim_Customer (customer_id, name, gender, age)
- Dim_Product (product_id, name, category)
- Dim_Date (date_id, full_date, month, year)

Snowflake Schema Example:

- Same as Star Schema but dimension tables are normalized:
    - o Dim_Product is split into Product, Category o
    - Dim_Date might be split into Day, Month, Year tables

3. Designing a data warehouse for a banking system — How would you approach it?

This is an open-ended system design question. The interviewer is looking for your ability to think at the architectural level.

Harshavardhana I
Data Engineer

Approach:

Step 1: Requirement Gathering

- Understand business KPIs: e.g., number of transactions, loan approvals, daily balances
- Identify stakeholders: finance, fraud detection, compliance, marketing

Step 2: Identify Key Subject Areas (Data Marts)

- Accounts (savings, current, loans) Transactions
- (deposits, withdrawals, transfers)
- Customers
- Cards (credit, debit)
- Loans

Step 3: Design the Schema

- Choose Star Schema for better reporting performance
- Example:

Fact Table:

- Fact_Transactions(transaction_id, customer_id, account_id, amount, date_id, branch_id)

Dimension Tables:

- Dim_Customer(customer_id, name, address, dob, kyc_status)
- Dim_Account(account_id, account_type, open_date)
- Dim_Date(date_id, date, month, quarter, year)
- Dim_Branch(branch_id, branch_name, region)

Step 4: ETL/ELT Design

Harshavardhana I
Data Engineer

- Source systems: Core banking systems, customer CRM, external KYC
- APIs Use tools like Apache NiFi, Airflow, or Informatica Implement:
  - Data cleaning (handle NULLs, outliers)
  - Deduplication
  - Historical tracking using SCD (Slowly Changing Dimensions)

Step 5: Data Warehouse Layer

- Use cloud DWs like Snowflake, Amazon Redshift, Google BigQuery, or on-premise like Teradata
- Partition large fact tables
- Use Materialized Views for reporting

Step 6: Reporting Layer

- Build dashboards using Power BI, Tableau, or Looker
- Serve to teams: operations, fraud analytics, compliance

Step 7: Security & Compliance

- Encrypt PII data Mask sensitive info
- (like PAN, Aadhar) Role-based access
- (RLS) Retain logs for audit

Big Data Tools

1. Compare Hadoop and Spark in terms of architecture and use cases.

1.1 Hadoop Architecture:

- Core Components:
    - HDFS (Hadoop Distributed File System): Stores massive data across clusters.
    - YARN (Yet Another Resource Negotiator): Manages cluster resources.
    - MapReduce: Batch processing framework using map → shuffle → reduce.
- Workflow:
    - Data is stored in HDFS → processed using MapReduce → output written back to HDFS.
    - Disk I/O intensive (writes intermediate data to disk).

1.2 Spark Architecture:

- Core Components:
    - Spark Core: Handles distributed task scheduling.
    - RDD (Resilient Distributed Dataset): Immutable, distributed data.
    - Catalyst Engine: For SQL optimization.
    - DAG Scheduler: Executes jobs in memory using Directed Acyclic Graphs.
- Spark Ecosystem:
    - Spark SQL – Structured data
    - Spark Streaming – Real-time data
    - MLlib – Machine learning
    - GraphX – Graph processing
- In-Memory Processing: Stores intermediate data in memory (RAM), making it much faster than MapReduce.

Harshavardhana I
Data Engineer

1.3 Comparison Table:

Feature	Hadoop (MapReduce)	Spark
Processing Type	Batch only	Batch + Real-time
Speed	Slower (due to disk I/O)	10–100x faster (in-memory)
Ease of Use	Java-based, verbose	Supports Scala, Python, SQL
Fault Tolerance	Yes (via HDFS replication)	Yes (via lineage of RDDs)
Use Cases	Legacy batch ETL	Real-time processing, ML, ETL

When to Use:

- Hadoop: Archival, cold data storage, traditional batch jobs
- Spark: Real-time analytics, machine learning pipelines, interactive querying

2. Explain how partitioning works in Apache Spark.

Partitioning in Spark:

- Partitioning is how Spark logically divides data across multiple executors or nodes for parallel processing.
- Spark processes each partition in parallel, leading to high performance in distributed environments.

Types of Partitioning:

1. Default Partitioning:
   - o Automatically based on cluster configuration and number of cores.
   - o Controlled using spark.default.parallelism.
2. Hash Partitioning (via transformations):

rdd.partitionBy(4)

3. Range Partitioning:
    - Used in sorted or range-based data.

Repartition vs Coalesce:

Operation	Description	Use Case
repartition(n)	Increases/decreases partitions (full shuffle)	When increasing partitions
coalesce(n)	Reduces partitions (no full shuffle)	When reducing partitions

Why Partitioning Matters:

- Optimizes parallelism
- Reduces data shuffling in joins
- Improves cache efficiency
- Controls skewed data issues

Example: Partitioning in Spark SQL df.write.partitionBy("country", "year").parquet("output_path") This creates folders by country/year, making queries faster on those filters.

3. Why might you choose Parquet over CSV for storing large datasets?

Parquet vs CSV Comparison:

Feature	CSV Text-based, row-	Parquet Columnar
Format Type	oriented	binary format

Feature	CSV	Parquet
Compression	Poor (large file size)	Highly compressed (Snappy, GZIP)
Performance	Reads entire file	Reads only required columns
Schema Support	None	Yes (self-describing metadata)
Data Types	Strings (needs manual parsing)	Strongly typed (ints, floats, etc.)
Splittable for HDFS	Yes	Yes

Why Parquet is Preferred:

1. Columnar Storage:
   - Efficient for analytical queries (OLAP).
   - Only loads relevant columns into memory.
2. Compression:
   - Up to 75% smaller than CSV.
   - Reduces I/O and storage costs.
3. Schema Enforcement:
   - Helps validate and track schema evolution.
4. Integration:
   - Well supported in Spark, Hive, AWS Athena, BigQuery.

Use Case Example:

For a banking analytics pipeline, where analysts want to aggregate transactions by account or region:

- CSV would scan the full dataset, including unused columns.
- Parquet would only load account_id, region, amount columns → faster and cheaper.

Coding

1. Python script to read a large CSV file and apply transformations

Reading Large CSV Files: When working with large datasets (e.g., millions of rows), it's efficient to:

- Read data in chunks using pandas.read_csv() with chunksize
- Apply transformations chunk by chunk to avoid memory overflow

Example Code:

```
import pandas as pd


# Define chunk size
chunk_size = 100000
result = []


# Read CSV in chunks
for chunk in pd.read_csv("large_file.csv", chunksize=chunk_size):
  # Transformation: Drop nulls and add a new column
  chunk = chunk.dropna()
  chunk['Total'] = chunk['Price'] * chunk['Quantity']


  result.append(chunk)


# Combine all processed chunks
final_df = pd.concat(result)
```

Harshavardhana I
Data Engineer

```
# SAVE TO NEW FILE
FINAL_DF.TO_CSV("TRANSFORMED_FILE.CSV", INDEX=FALSE)
```

BEST PRACTICES:

- USE DTYPES ARGUMENT TO OPTIMIZE MEMORY USAGE
- AVOID LOADING FULL DATA IN RAM IF NOT NECESSARY

2. Handling missing data in Python (Pandas)

COMMON MISSING DATA TECHNIQUES:

TECHNIQUE	CODE EXAMPLE	USE CASE
DROP MISSING VALUES	DF.DROPNA()	REMOVE ROWS/COLUMNS WITH NULLS
FILL WITH CONSTANT	DF.FILLNA(0)	DEFAULT VALUE LIKE 0 OR "UNKNOWN"
FORWARD FILL (FFILL)	DF.FILLNA(METHOD='FFILL')	TIME SERIES DATA
BACKWARD FILL (BFILL)	DF.FILLNA(METHOD='BFILL')	ALTERNATIVE TO FFILL
FILL WITH MEAN/MEDIAN/MODE	DF['COL'].FILLNA(DF['COL'].MEAN())	NUMERICAL COLUMNS
CHECK % OF MISSING DATA	DF.ISNULL().MEAN() * 100	DATA QUALITY CHECK

EXAMPLE:

```
# FILL MISSING AGE WITH MEAN
DF['AGE'] = DF['AGE'].FILLNA(DF['AGE'].MEAN())


# DROP ROWS WHERE 'SALARY' IS MISSING
DF = DF.DROPNA(SUBSET=['SALARY'])
```

Harshavardhana I
Data Engineer

```
# Fill missing city names with "Unknown"
df['City'] = df['City'].fillna("Unknown")
```

3. PYTHON DECORATORS – EXPLANATION & USE CASE

What is a Decorator?

- A decorator is a function that modifies another function’s behavior without changing its code.
- It is widely used in logging, timing, authentication, and caching.

```
 Simple Decorator Example: def
my_decorator(func):
  def wrapper():
    print("Before function
    runs") func() print("After
    function runs")
  return wrapper

@my_decorator
def say_hello():
  print("Hello!")

say_hello() Output:
Before function runs
Hello!
```

Harshavardhana I
Data Engineer

After function runs

Real Use Case – Logging Execution Time:

```
import time


def timer_decorator(func):
  def wrapper(*args, **kwargs):
    start = time.time() result = func(*args, **kwargs) end =
    time.time() print(f"{func.__name__} took {end -
    start:.2f} seconds") return result



  return wrapper


@timer_decorator
def process_data():
  time.sleep(2)
  print("Data processed")


process_data() Output: Data
processed process_data took
2.00 seconds
```

Harshavardhana I
Data Engineer