Big Data SQL Interview Questions

In this lecture, we will cover a few big data SQL-related questions commonly asked in interviews. Data Engineering interviews comprise both theoretical and SQL related questions. Theoretical questions are already covered in your assignments while SQL-related questions and how to approach them will be discussed in this lecture. So we are going to discuss 8 case study questions

Case Study 1: Optimizing E-Commerce Analytics Queries

Scenario:

You are a data engineer working for an e-commerce platform. You have a large `orders` table (hundreds of millions of rows) that stores information about customer orders. The table is partitioned by `order_date`, and you need to generate daily sales reports for each product and region. The reports include the total sales amount, the number of orders, and the average order amount per region.

Problem:

Write a HiveQL query to generate this daily report,

Data:

`orders`: `order_id INT`, `product_id INT`, `customer_id INT`, `amount DOUBLE`, `region STRING`, `order_date STRING` (partitioned)

Solution:
Query:

```
SELECT product_id, region, COUNT(order_id) AS num_orders,
       SUM(amount) AS total_sales, AVG(amount) AS avg_order_amount
FROM orders
WHERE order_date = '2024-05-01'  -- Ensure partition pruning is used
GROUP BY product_id, region;
```

- **Optimization Steps**:
    - **Partition Pruning**: By using `order_date` in the `WHERE` clause, Hive will only scan the partition for `2024-05-01`, skipping other partitions.

- **Bucketing**: To improve the performance of future queries that involve frequent joins, you could bucket the `orders` table by `region` and `product_id`.
- **Vectorized Execution**: Enable vectorized execution to process rows in batches and improve memory and CPU usage.

Configurations:

```
SET hive.vectorized.execution.enabled=true;
SET hive.vectorized.execution.reduce.enabled=true;

-- If bucketing is applied:
CREATE TABLE orders_bucketed (
  order_id INT, product_id INT, customer_id INT, amount DOUBLE, region
STRING
)
PARTITIONED BY (order_date STRING)

CLUSTERED BY (region, product_id) INTO 32 BUCKETS;
```

Case Study 2: Handling Skewed Data in Social Media Analytics

Scenario:

You are responsible for processing social media analytics data in Hive. You have a table `user_activity` that stores user interactions (likes, comments, and shares) on posts. The table is partitioned by `event_date` and contains skewed data, as a few celebrity users generate 90% of the interactions. You need to generate a report showing the number of interactions per user, but the skew causes some reducers to be overloaded and the job takes too long to complete.

Problem:

Write a HiveQL query to calculate the total number of interactions (likes, comments, shares) per user.

Data:

`user_activity`: `user_id STRING`, `post_id STRING`, `interaction_type STRING`, `event_date STRING` (partitioned), `interaction_count INT`

Solution:

Query:

```
SELECT user_id, SUM(interaction_count) AS total_interactions

FROM user_activity

WHERE event_date = '2024-06-15'

GROUP BY user_id;
```

- **Handling Skew**:

In Hive, skew handling addresses the issue of data skew, where certain keys (like `user_id` for popular or "celebrity" users) appear with high frequency, leading some reducers to handle a disproportionate amount of data. When skew handling is enabled, Hive identifies these skewed keys and splits them into smaller, parallel tasks across multiple reducers, rather than sending all data for a skewed key to a single reducer. This reduces the load on individual reducers, distributes work more evenly, and speeds up processing by preventing anyone reducer from becoming a bottleneck. The results from these smaller tasks are then merged to form the final output, ensuring efficient and balanced execution for datasets with natural data skew.

Configurations:

```
SET hive.groupby.skewindata=true;
```

Explanation: The `hive.groupby.skewindata=true` setting allows Hive to detect the skewed `user_id`s and process them in smaller batches, distributing the load more evenly across reducers.

Case Study 3: Real-Time Clickstream Data Ingestion

Scenario:

You are managing a clickstream data pipeline where user click events are stored in Hive. The data comes from various sources every second, leading to the generation of a large number of small files. The small files are causing performance degradation in downstream processing. Your task is to optimize the Hive queries that run on this clickstream data, ensuring minimal I/O overhead while querying large amounts of data.

Problem:

Write a query to aggregate the number of clicks per `url` from the `clickstream` table.

Data:

`clickstream`: `event_id STRING`, `user_id STRING`, `url STRING`, `timestamp STRING`, `click_count INT`

Solution:

Query:

```
SELECT url, SUM(click_count) AS total_clicks

FROM clickstream

GROUP BY url;
```

Optimizing for Small Files:

The small files problem can be mitigated by merging these small files during the MapReduce stages using Hive's built-in file merging settings.

Configurations:

```
SET hive.merge.mapfiles=true;  -- Merge small files at the map stage

SET hive.merge.mapredfiles=true;  -- Merge small files at the reduce stage

SET hive.merge.size.per.task=256000000;  -- Set target file size to 256MB

SET hive.merge.smallfiles.avgsize=128000000;  -- Set threshold for small file merging
```

Explanation: By merging small files during the map and reduce stages, you reduce the number of files processed, thereby minimizing I/O and improving performance. Setting the target and average file size ensures that small files are merged into larger blocks, reducing NameNode overhead.

Case Study 4: Complex Joins and Aggregations in Retail Analytics

Scenario:

You are tasked with analyzing data from a retail system to generate insights on customer purchasing patterns. You have three tables: `customers`, `orders`, and `products`. You need to generate a report showing the total sales for each customer, categorized by product type, while avoiding data shuffling as much as possible.

Problem:

1. Write a query to calculate the total sales amount for each customer by product type.
2. Optimize this query using bucketing and map-side joins to minimize data shuffling.

Data:

- `customers`: `customer_id INT`, `customer_name STRING`
- `orders`: `order_id INT`, `customer_id INT`, `product_id INT`, `amount DOUBLE`
- `products`: `product_id INT`, `product_name STRING`, `product_type STRING`

Solution:

Query:

```
SELECT c.customer_name, p.product_type, SUM(o.amount) AS total_sales

FROM customers c

JOIN orders o ON c.customer_id = o.customer_id

JOIN products p ON o.product_id = p.product_id

GROUP BY c.customer_name, p.product_type;
```

Optimization Using Bucketing:

- Bucket the `orders` and `products` tables by `product_id` to optimize the join process and minimize data shuffling.

Configurations:

```
-- Bucketing the tables:

CREATE TABLE orders_bucketed (

  order_id INT, customer_id INT, product_id INT, amount DOUBLE
```

```
) CLUSTERED BY (product_id) INTO 32 BUCKETS;


CREATE TABLE products_bucketed (
  product_id INT, product_name STRING, product_type STRING
) CLUSTERED BY (product_id) INTO 32 BUCKETS;


SET hive.optimize.bucketmapjoin=true;
```

Explanation: By clustering both the `orders` and `products` tables into buckets based on `product_id`, Hive can use a bucket map-side join, reducing the amount of data that needs to be shuffled. This optimization is especially useful when joining large datasets and performing aggregations simultaneously.

Case Study 5: Using Compression and File Formats for Query Optimization

Scenario:

You are working with a large table `sales_data` that stores monthly sales records for millions of transactions. The table is queried frequently for reporting purposes, but the storage costs are increasing, and the query performance is degrading. You need to improve the performance and reduce storage costs by compressing the data and storing it in an efficient file format.

Problem:

1. Create the `sales_data` table using the ORC file format and enable compression to reduce the size of the stored data.

Data:

- `sales_data`: `sale_id INT`, `region STRING`, `product_category STRING`, `sale_amount DOUBLE`, `sale_date STRING`

Solution:

Table Creation:

```
CREATE TABLE sales_data (
```

```
  sale_id INT,
  region STRING,
  product_category STRING,
  sale_amount DOUBLE,
  sale_date STRING
)
STORED AS ORC
TBLPROPERTIES ("orc.compress"="ZLIB");
```

Query:

```
SELECT region, product_category, SUM(sale_amount) AS total_sales
FROM sales_data
WHERE sale_date BETWEEN '2024-01-01' AND '2024-01-31'
GROUP BY region, product_category;
```

Hive Configurations:

```
SET hive.exec.compress.output=true;
SET hive.exec.orc.default.compress=ZLIB;  -- Use ZLIB for ORC compression
SET hive.vectorized.execution.enabled=true;  -- Enable vectorized execution for ORC files
SET hive.vectorized.execution.reduce.enabled=true;
```

Explanation: ORC (Optimized Row Columnar) format is designed for efficient storage and reading of large datasets, especially for analytical queries. The ZLIB compression reduces storage space while maintaining query performance. Vectorized execution processes rows in batches, improving CPU efficiency and reducing query latency. Here parquet can also be used, but as Hive natively supports ORC and has a few other optimizations like merge file support, etc, we can prefer ORC here.

Case Study 6: Partitioning and Bucketing for Improved Query Performance

Scenario:

You are managing a Hive table `user_activity` that tracks user interactions on a website (logins, clicks, etc.). The data is queried based on the `event_date` and the `user_id`. The queries are running slowly due to the size of the table. You need to partition and bucket the table to improve query performance.

Problem:

1. Create the `user_activity` table partitioned by `event_date` and bucketed by `user_id` into 32 buckets.
2. Write a query to retrieve the total number of interactions for each user on a specific day, taking advantage of partitioning and bucketing.
3. Suggest the Hive configurations required for optimized performance when using partitioning and bucketing.

Data:

`user_activity`: `user_id STRING`, `interaction_type STRING`, `interaction_count INT`, `event_date STRING` (partitioned)

Solution:

Table Creation:

```
CREATE TABLE user_activity (

  user_id STRING,

  interaction_type STRING,

  interaction_count INT

)

PARTITIONED BY (event_date STRING)

CLUSTERED BY (user_id) INTO 32 BUCKETS

STORED AS ORC;
```

Query:

```
SELECT user_id, SUM(interaction_count) AS total_interactions

FROM user_activity

WHERE event_date = '2024-06-01'

GROUP BY user_id;
```

Hive Configurations:

```
SET hive.optimize.bucketmapjoin=true;

SET hive.exec.dynamic.partition=true;

SET hive.exec.dynamic.partition.mode=nonstrict;

SET hive.vectorized.execution.enabled=true;
```

Explanation: By partitioning the table by `event_date`, you ensure that only the relevant partition is scanned during the query, reducing I/O. Bucketing by `user_id` enables efficient joins and aggregations on the `user_id` field. The Hive configurations further enhance query performance by using bucketed joins and vectorized execution.

Case Study 7: Windowing Functions and Data Analysis

Scenario:

You have a table `employee_salaries` that stores monthly salary information for employees. You need to analyze the salary trend over time for each employee and calculate the cumulative total salary and the rank of each employee based on their total salary each month.

Problem:

1. Create the `employee_salaries` table using the Parquet file format.
2. Write a query to calculate the cumulative salary of each employee over time and rank them by their total salary within each month using windowing functions.

Data:

- `employee_salaries`: `employee_id STRING`, `salary DOUBLE`, `salary_month STRING`

Solution:

Table Creation:

```
CREATE TABLE employee_salaries (

  employee_id STRING,

  salary DOUBLE,

  salary_month STRING

)

STORED AS PARQUET;
```

Query:

```
SELECT employee_id, salary_month, salary,

       SUM(salary) OVER (PARTITION BY employee_id ORDER BY
salary_month) AS cumulative_salary,

       RANK() OVER (PARTITION BY salary_month ORDER BY SUM(salary)
DESC) AS salary_rank

FROM employee_salaries;
```

Explanation: The Parquet format is a columnar storage format that offers efficient read performance, especially for analytical queries involving large datasets with many columns. Windowing functions allow you to perform complex analytics like cumulative sums and rankings without needing self-joins or multiple subqueries, making them ideal for time-based and ordered data analysis.

Case Study 8: Cost-Based Optimization (CBO) for Complex Joins

Scenario:

You are working on a project that involves joining three large tables: `orders`, `customers`, and `products`. The current query performance is slow due to inefficient join ordering. You need to enable Hive's Cost-Based Optimizer (CBO) to improve query execution by determining the most efficient join strategy based on table statistics.

Problem:

1. Write a query to join the three tables and calculate the total sales amount for each customer by product category.

Data:

- `orders`: `order_id INT`, `customer_id INT`, `product_id INT`, `amount DOUBLE`
- `customers`: `customer_id INT`, `customer_name STRING`
- `products`: `product_id INT`, `product_category STRING`

Solution:

Query:

```
SELECT c.customer_name, p.product_category, SUM(o.amount) AS
total_sales

FROM orders o

JOIN customers c ON o.customer_id = c.customer_id

JOIN products p ON o.product_id = p.product_id

GROUP BY c.customer_name, p.product_category;
```

Enabling CBO:

```
SET hive.cbo.enable=true;

SET hive.compute.query.using.stats=true;



ANALYZE TABLE orders COMPUTE STATISTICS;

ANALYZE TABLE customers COMPUTE STATISTICS;

ANALYZE TABLE products COMPUTE STATISTICS;
```

Explanation: The Cost-Based Optimizer uses statistics about the size and distribution of data in the tables to determine the best join order and query execution plan. Gathering statistics on the tables allows the optimizer to choose the most efficient join strategy, which can significantly improve query performance in complex multi-table joins. Here we could also use Rule based optimization, but Rule based optimization are static , so its better to go with Cost based optimization