**Shruti - UST GLOBAL -L1**

**1.optimization tech in spark**

**2.what is dag**

**3.optimize technical in sql**

**4.how you analyze sql query**

**5.how to your Design schema**

**6.implemeting and optimization in etl airflow**

**7. How do you data partition what is important of partition**

**8.how do you debug in your project**

**9.can u in detail matrices is**

**10.diff between Hadoop and hive**

**11.what is data modeling**

**12.what is cdc**

**13.how do you handle incremental load and implement CDC**

**14.u r data pipeline in airflow begins to fail in intermediate how to troubleshoot it.**

**15 .how to handle large data and delay in data**

**16.air flow dag has multiple depends and wrong running job how to handle it.**

**1. Optimization Techniques in Sparks**

* **Caching & Persistence**: Cache frequent intermediate results using .cache() or .persist().
* **Broadcast Variables**: Broadcast small dimension tables to avoid shuffling during joins.
* **Predicate Pushdown**: Use filters early in the data pipeline to minimize data load.
* **Partition Tuning**: Repartition (repartition(), coalesce()) for parallelism and performance.
* **Avoid Wide Transformations**: Use narrow transformations like map, filter where possible.
* **Enable Tungsten & Catalyst**: Default in Spark for code generation and query plan optimization.

**2. What is DAG (Directed Acyclic Graph)?**

* A **DAG** in Spark represents the **logical execution plan** of transformations.
* It’s a graph of stages with **no cycles**, ensuring no task repeats infinitely.
* Spark builds DAGs from your code and optimizes before execution.

**3. Optimization Techniques in SQL**

* **Use Proper Indexing**
* \*\*Avoid SELECT \*\*\* and fetch only required columns
* **Use Joins Wisely**: Prefer INNER JOIN over LEFT JOIN when possible
* **Use WHERE Clauses early**
* **Use EXISTS/IN** appropriately depending on data volume
* **Avoid functions on indexed columns** in WHERE clause (e.g., WHERE YEAR(date\_col) = 2024)

**4. How Do You Analyze a SQL Query?**

* Use EXPLAIN PLAN to see execution strategy.
* Check for:
  + Full table scans
  + Index usage
  + Join order and type
  + Estimated rows
* Optimize based on bottlenecks seen in explain plan.

**5. How Do You Design Schema?**

* Understand data types and relationships (1:1, 1:N, M:N).
* Normalize for OLTP, denormalize for OLAP.
* Choose appropriate keys (Primary/Foreign).
* Consider partitioning and clustering strategy (for large datasets).
* Document schema for maintainability.

**6. Implementing and Optimizing ETL in Airflow**

* Use **Task retries**, **timeouts**, and **failure notifications**.
* Split ETL into **atomic tasks** with proper dependencies.
* Use **XComs** or **Variables** for sharing dynamic data.
* Monitor DAGs with Airflow UI, logs, and alerting.
* Apply **modular code with custom operators** and **templating (Jinja2)**.

**7. How Do You Partition Data & Why Is It Important?**

* Partition by **frequently filtered columns** (e.g., date, region).
* Reduces scan time by **pruning unnecessary partitions**.
* Improves parallelism and query performance.
* In Spark: df.write.partitionBy("country").parquet(path)
* In Hive: Use PARTITIONED BY (col\_name).

**8. How Do You Debug in Your Project?**

* Use:
  + **Spark logs** for transformation issues.
  + **Airflow logs** per task instance.
  + **Try/Except blocks** in Python tasks.
  + **Unit and integration tests**.
  + **Data validation checks** post transformation.
  + Slack/email alerts in production.

**9. Can You Explain Matrices in Detail?**

You may want to clarify what "matrices" means in this context. But if it refers to **performance or project metrics**, answer:

* **ETL Performance Metrics**: Throughput, latency, success/failure rates.
* **Data Quality Metrics**: Null checks, duplicates, range checks.
* **Code Metrics**: Test coverage, code quality.
* **Project Metrics**: Sprint velocity, defect density, SLA adherence.

**10. Difference Between Hadoop and Hive**

| **Feature** | **Hadoop (HDFS + MapReduce)** | **Hive** |
| --- | --- | --- |
| Purpose | Distributed storage & processing | Data warehousing on Hadoop |
| Language | Java (MapReduce) | SQL-like (HiveQL) |
| Execution | Low-level, manual jobs | Automatically converted to MapReduce/Spark jobs |
| Use Case | Batch processing | Querying large datasets using SQL |

**11. What is Data Modeling?**

* Designing a **blueprint of how data is stored and accessed**.
* Includes **conceptual**, **logical**, and **physical** models.
* Focuses on relationships, constraints, keys, and normalization.

**12. What is CDC (Change Data Capture)?**

* Technique to **identify and capture only changed data** since the last load.
* Reduces ETL processing time.
* Can be implemented using:
  + Timestamps
  + Versioning
  + Triggers or logs (e.g., Debezium, log-based CDC)

**13. Handling Incremental Load & Implementing CDC**

* Track changes using:
  + Modified timestamp (last\_updated)
  + Primary key comparison
* Tools:
  + **Airflow for orchestration**
  + **SCD (Slowly Changing Dimensions)** logic
  + **Delta Lake** or **Apache Hudi** for merge-based updates

**14. Airflow DAG Fails Midway – Troubleshooting**

* **Check logs** of the failed task in UI.
* Use depends\_on\_past, trigger\_rule settings.
* Retry failed tasks manually or via CLI: airflow tasks retry dag\_id task\_id
* Use task\_instance.xcom\_pull() to debug data passing.
* Add Slack/email alerts.

**15. Handling Large Data & Delay**

* Use **partitioning**, **parallel processing**, and **data chunking**.
* Optimize network IO by **columnar formats (Parquet/ORC)**.
* Use **back-pressure controls** (e.g., batching, retry logic).
* Ensure **scalable infrastructure**: autoscaling clusters, distributed storage.

**16. DAG with Dependencies & Wrong Running Job – Handling**

* Use **TriggerRule**: all\_done, one\_success, etc.
* Set **depends\_on\_past=False** to avoid being blocked by prior runs.
* Add **sensors** or **pre-checks** before critical tasks.
* Use **BranchPythonOperator** to conditionally route execution.
* Implement **idempotent tasks** that can rerun safely.