**1.Optimizing Performance in Large Data Pipelines**

**- Partitioning & Bucketing**

Partitioning: Splitting data into logical subdirectories based on a column . This reduces the amount of data scanned during queries.

Bucketing: A technique to group similar values into the same files, reducing shuffle operations during joins.

**- Optimized File Formats**

Use Parquet or Delta Lake instead of CSV or JSON.

Parquet provides columnar storage, which speeds up read queries.

Delta Lake adds versioning, schema enforcement, and ACID transactions.

**-Z-Ordering & Data Skipping (Databricks-Specific)**

Z-Ordering: Reorders data within partitions based on frequently filtered columns.

Data Skipping: Delta tables store min/max statistics, helping Spark skip irrelevant files.

**-Indexing & Caching**

**-Scaling & Parallelism**

Increase Cluster Size: Add worker nodes in Spark to parallelize tasks.

Optimize Shuffle Operations: Reduce unnecessary repartitioning.

**2.Convert parquet to delta**

A screenshot of a computer program

AI-generated content may be incorrect.

**3**. **Ensuring Data Quality**

**-Schema Validation**

Enforcing data types and constraints (e.g., NOT NULL, UNIQUE, PRIMARY KEY).

Preventing ingestion of incorrect or malformed data.

**-Data Profiling**

Analyzing data for missing values, anomalies, or incorrect data types.

Identifying outliers using statistical methods.

**-Data Cleaning & Deduplication**

Removing duplicate records using window functions in SQL or dropDuplicates() in PySpark.

Standardizing formats (e.g., date formats, phone numbers).

**4. Data Stored in the Silver Layer**

* Cleaned and Processed Data – Duplicates removed, invalid records filtered out.
* Transformed data
* Incremental Updates – Only new or updated records are stored using merge and upsert strategies.

**5. Unity Catalog:**

Unity Catalog - is a tool in Databricks that helps manage data security, access, and organization in one place. It stores metadata (information about data) and makes it easier to control who can access or change data.

**Before Unity Catalog (Using Hive Metastore):**

* No central control – Each workspace had its own rules, making management hard.
* Difficult access control– Setting permissions for multiple users was complicated.
* No automatic tracking – Couldn’t easily see who used or changed the data.

**With Unity Catalog:**

* Manages all workspaces together – No need for separate setups.
* Better security – Controls access at the table, row, and column levels.
* Tracks data usage automatically – Shows who accessed and modified the data.
* Shares data across workspaces easily – No need to copy data.
* Works with all file types – Supports Parquet, Delta, CSV, JSON, etc.

**6**. **Writing Reusable Code**

Reusable code saves time and ensures consistency:

* Modular Functions – Generic functions with parameters.
* Configuration-Driven Approach – Store settings in YAML/JSON instead of hardcoding values.
* GitHub Version Control – Maintain reusable scripts in a shared repository.

**7.Difference between Coalesce and repartition:**

Coalesce:

* Used to reduce the number of partitions.
* It avoids full data shuffle, making it faster than repartition().
* use when you want to merge smaller partitions without incurring high computational costs.

Repartition:

* Used to increase or decrease the number of partitions.
* Performs a full shuffle, redistributing data evenly.
* Useful when you need to balance data across partitions.

**8. Data Modelling techniques:**

Data modeling - is the process of structuring and organizing data to ensure it is stored efficiently and can be retrieved easily.

**Main data modeling techniques:**

**1. Conceptual, Logical, and Physical Data Modeling**

Data modeling is typically done in three stages:

**a) Conceptual Data Model**

High-level representation of the data structure, focusing on business concepts.

Defines key entities (e.g., Customers, Orders, Products) and their relationships.

Does not include details like attributes or technical constraints.

Example:

A retail business might have Customers, Orders, and Products as key entities. The conceptual model just defines their relationships (e.g., "A customer places an order").

**b) Logical Data Model**

Adds attributes, primary keys, and foreign keys to the conceptual model.

Focuses on data organization but is still independent of any specific database system.

Example:

Customers (Customer\_ID, Name, Email)

Orders (Order\_ID, Customer\_ID, Order\_Date)

Products (Product\_ID, Product\_Name, Price)

Order\_Items (Order\_ID, Product\_ID, Quantity)

**c) Physical Data Model**

Defines how data is stored in a specific database (e.g., MySQL, SQL Server, PostgreSQL).

Includes data types, indexes, constraints, partitions, and performance optimizations.

Example:

A Customer\_ID might be defined as INT PRIMARY KEY AUTO\_INCREMENT in MySQL.

**2. Relational Data Modeling (Normalization & Denormalization)**

**a) Normalization (1NF, 2NF, 3NF, BCNF)**

Eliminates redundancy and ensures data integrity by splitting data into multiple tables.

1NF: Remove duplicate columns and ensure atomic values.

2NF: Remove partial dependencies (a non-key attribute should depend on the whole primary key).

3NF: Remove transitive dependencies (attributes should only depend on the primary key).

BCNF: Advanced version of 3NF to ensure higher normalization.

**Example:**

Instead of storing customer details repeatedly in an orders table, we separate them into a Customers table and reference them using Customer\_ID.

**b) Denormalization**

Used in analytical systems (OLAP) to improve query performance by reducing the number of table joins.

Some redundancy is introduced for faster data retrieval.

**Example:**

A Sales\_Report table may store pre-aggregated sales data instead of calculating it dynamically from multiple tables.

**3. Dimensional Data Modeling (Star Schema & Snowflake Schema)**

Used in data warehousing for reporting and analytics.

**a) Star Schema**

A central fact table (contains transactional data) connected to multiple dimension tables.

Simplifies queries and improves performance.

**Example:**

A Sales\_Fact table stores Sales\_Amount, Date\_ID, Product\_ID, and Store\_ID, with Date\_Dim, Product\_Dim, and Store\_Dim as dimension tables.

**b) Snowflake Schema**

Similar to a star schema, but dimensions are normalized to remove redundancy.

Saves storage but can lead to slower queries due to more joins.

**Example:**

Instead of storing all product details in Product\_Dim, separate tables like Product\_Category and Product\_Brand are created to avoid repetition.

**4. Data Vault Modeling**

Used for scalable and historical tracking in data warehouses.

Splits data into Hub (business keys), Link (relationships), and Satellite (attributes & history) tables.

**Example:**

For a banking system, Customer\_Hub stores customer IDs, Account\_Hub stores account numbers, and Transaction\_Satellite stores changing transaction details.

**5. NoSQL Data Modeling**

For big data, real-time, and non-relational databases like MongoDB, Cassandra, Redis, Neo4j.

**a) Document Model (MongoDB)**

Stores data as JSON-like documents, flexible schema.

**Example:** A user profile in MongoDB:

{

"user\_id": 123,

"name": "John Doe",

"email": "john@example.com",

"addresses": [

{"type": "home", "city": "New York"},

{"type": "work", "city": "San Francisco"}

]

}

**b) Key-Value Model (Redis, DynamoDB)**

Simple key-value pairs for fast lookups.

**Example**: user\_123 → {name: "John", age: 30}

**c) Column-Family Model (Cassandra, HBase)**

Stores data in wide-column format for fast retrieval.

**Example:** A table with multiple columns grouped into families:

| User\_ID | Name | Age | Address\_City | Address\_Zip |

|---------|------|----|--------------|-------------|

| 123 | John | 30 | New York | 10001 |

**d) Graph Model (Neo4j)**

Stores nodes (entities) and edges (relationships).

**Example:** Used in social networks (e.g., "User A follows User B").

Airflow:

Airflow UI provides multiple views and tools to monitor and troubleshoot DAGs and tasks.

1. DAGs View (Home Page)

Airflow UI → DAGs Tab

-Displays a list of DAGs in your Airflow environment.

- Shows status, last run, schedule, and owner of each DAG.

-Allows triggering, pausing, refreshing, or deleting DAGs.

Key Features:

Last Run Status → Helps identify if a DAG has failed.

Next Run Countdown → Shows when the DAG will run next.

Quick Actions → Enable, pause, or delete a DAG.

2. Graph View (Visualizing DAG Execution)

Click on a DAG → Graph View

Displays the flow of tasks in a DAG using a graph structure.

Each task is color-coded to indicate status:

🟢 Success

🔴 Failed

🔵 Running

🟡 Queued

Easily identify failures in DAG execution.

Check task dependencies and flow between tasks.

Click on a task to view logs, retries, and execution details.

3. Tree View (Task Execution Status Over Time)

Click on a DAG → Tree View

Provides a historical view of DAG runs over time.

Shows each task instance's status for multiple DAG runs.

Track long-running or failing tasks over multiple executions.

Identify patterns in task failures.

Restart or clear failed tasks directly from this view.

4. Gantt Chart (Task Execution Timeline)

Click on a DAG → Gantt Chart

Displays the start time and duration of each task in a DAG run.

Helps identify bottlenecks and tasks that take longer than expected.

Detect performance issues (long-running tasks).

Optimize task dependencies to reduce execution time.

5. Task Instance Logs (For Debugging Failures)

Location: Click on a Task → Log Tab

Provides detailed execution logs of a task instance.

Used for debugging failures, API calls, and error messages.

Find Python exceptions or errors that caused a failure.

Verify if a task executed successfully.

Check logs for retries or timeouts.

6. Code View (Checking DAG Source Code)

Click on a DAG → Code Tab

Shows the Python source code of the DAG.

Useful for debugging and modifying DAG logic.

Verify if the DAG definition is correct.

Check schedule interval, task dependencies, and parameters.

7. DAG Runs (Execution History)

Click on a DAG → DAG Runs Tab

Displays all past DAG runs with status information.

View execution times, trigger type, and duration of past runs.

Identify frequently failing DAG runs.

8. Monitoring with Alerts & Notifications

Airflow UI → Admin → Connections & Variables

Configure email, Slack, or PagerDuty alerts for failures.

Receive real-time notifications when a DAG fails.

Example – Send Email on Failure

Hackerrank problems:

1.

A screenshot of a computer

Description automatically generated

2.

A screenshot of a computer

Description automatically generated

3. 