If you have any questions or suggestions regarding the notes, please feel free to reach out to me on

Sanjay Yadav Phone: 8310206130

https://www.linkedin.com/in/yadav-sanjay/

https://www.youtube.com/@GeekySanjay

https://github.com/geeky-sanjay/

Data Processing With Spark: Dataframe 2 - Nov 20

In previous lectures, we explored the basics of DataFrames. Now, let's build a small Spark application using DataFrame APIs to solve a use case.

Objective

Develop a scalable Spark application for comprehensive data analysis and a recommendation system tailored to a large e-commerce platform. The application will:

- Use JSON files as the primary data source.
- Integrate multiple datasets.
- Perform complex transformations.
- Implement machine learning models for insights and personalized recommendations.

Background

The e-commerce platform has millions of users and thousands of products across various categories. The goal is to enhance customer experience and drive sales by:

1. Analyzing customer behavior.
2. Identifying popular and trending products.
3. Providing personalized product recommendations.

Data Sources (JSON Files):

1. **User Activity Log (`user_activity.json`)**: Contains user interactions like session data, search queries, and browsing patterns.
2. **Transaction Data (`transactions.json`)**: Includes details of purchases, product info, and timestamps.
3. **Product Catalog (`product_catalog.json`)**: Static details like category, brand, price, and ratings.
4. **User Profile (`user_profile.json`)**: Demographics like age, gender, location, and purchase history.

Key Requirements

1. Data Cleaning & Transformation

- Remove duplicates in user activity and transaction logs.
- Handle missing values in user profiles and product catalog data.
- Normalize categorical data (e.g., product category).

2. Data Aggregation and Feature Engineering

- Calculate key metrics:
  - Average Session Time.
  - Conversion Rate.
  - Average Order Value.
- Identify high-frequency users and top products based on user activity.
- Create recommendation features such as:
  - User preferences.
  - Product popularity.
  - Purchase recency.

3. Recommendation System

- Implement a collaborative filtering model using **ALS (Alternating Least Squares)** for recommendations based on user behavior.
- Generate and cache personalized product recommendations for efficient access.
- Create a function for real-time recommendations using recent activity data.

4. Analytics and Insights

- Provide insights into:
  - Trending products.
  - Customer segments.
  - Seasonal purchasing trends.
- Identify cross-sell and up-sell opportunities.

By the end of this project, you'll gain practical experience with Spark's data processing capabilities and machine learning integration to build a real-world scalable application.

```
from pyspark.sql import SparkSession

# Initialize Spark Session
spark = SparkSession.builder \
    .appName("E-commerce Data Analytics") \
    .getOrCreate()
```

Code Explanation

Importing SparkSession

```
from pyspark.sql import SparkSession
```

- **Purpose**: Imports the `SparkSession` class from the `pyspark.sql` module.
- **Role**: `SparkSession` is the entry point for programming with Spark using DataFrame and SQL APIs.

- **Unified Interface**: Since Spark 2.0, `SparkSession` combines previous contexts like `SQLContext` and `HiveContext` into a single object for simpler access to Spark's structured data processing features.

Building the SparkSession

```
spark = SparkSession.builder \
    .appName("E-commerce Data Analytics") \
    .getOrCreate()
```

- **Step-by-Step Explanation**:
    1. **`SparkSession.builder`**:
        - A builder pattern to configure and initialize a Spark session.
        - Allows you to set options like application name, memory allocation, or other configurations.
        - Creates or retrieves an active `SparkSession` instance.
    2. **`.appName("E-commerce Data Analytics")`**:
        - Assigns a name to the Spark application. In this case, it's set to `"E-commerce Data Analytics"`.
        - **Why it's important**:
            - Helps identify your application in the Spark UI when multiple jobs are running.
            - Aids in tracking and debugging.
    3. **`.getOrCreate()`**:
        - Checks if a `SparkSession` is already active in the environment:
            - If yes, it returns the existing session.
            - If no, it creates a new one with the specified configuration.
        - **Why it's useful**:
            - Prevents creating duplicate sessions.
            - Ensures efficient resource use and avoids conflicts.

What Happens Next?

- The `spark` variable now holds the initialized `SparkSession`.
- Use this `spark` instance to:
    - Create DataFrames.
    - Execute SQL queries.
    - Perform other Spark operations throughout your notebook or script.

Loading JSON Data Files in Spark

To load JSON files into Spark DataFrames, follow these steps:

```
# Load JSON files into DataFrames
user_activity_df = spark.read.json("path/to/user_activity.json")
transactions_df = spark.read.json("path/to/transactions.json")
product_catalog_df = spark.read.json("path/to/product_catalog.json")
user_profile_df = spark.read.json("path/to/user_profile.json")
```

Explanation of the Code

1. Loading JSON into DataFrames

Each line reads a JSON file into a Spark DataFrame:

```
user_activity_df = spark.read.json("path/to/user_activity.json")
```

Detailed Breakdown

1. **`user_activity_df =`**
   - This assigns the resulting DataFrame to the variable `user_activity_df`.
   - The variable name is descriptive, indicating it holds data from the `user_activity.json` file.
   - Using a variable allows you to reference the DataFrame later for data manipulation and analysis.
2. **`spark`**
   - Refers to the `SparkSession` object, which manages the Spark environment.
   - Provides the primary interface for reading data, running queries, and configuring Spark settings.
3. **`.read`**
   - A **DataFrameReader** object that facilitates reading data into a DataFrame.
   - Supports multiple formats such as JSON, CSV, Parquet, and more.
4. **`.json("path/to/user_activity.json")`**
   - Reads a JSON file and loads it into a Spark DataFrame.
   - JSON (JavaScript Object Notation) is commonly used for semi-structured data.
   - The `.json()` method:
     - Parses each line or object in the JSON file as a separate row in the DataFrame.
     - Automatically infers the schema (data types for each column) unless explicitly specified.
5. **`"path/to/user_activity.json"`**
   - This is the file path to the JSON file you want to load.
   - Replace `"path/to/user_activity.json"` with the actual path where your file is stored:
     - Local file system: e.g., `/mnt/data/user_activity.json`.
     - HDFS or cloud storage: e.g., `hdfs://...` or `s3://...`.

Example DataFrame Creation

For example, if `user_activity.json` contains:

```
[
  {"user_id": 1, "activity": "search", "timestamp": "2024-11-21T10:00:00Z"},
  {"user_id": 2, "activity": "purchase", "timestamp": "2024-11-21T11:30:00Z"}
]
```

After loading:

```
user_activity_df.show()
```

The output:

```
+-------+----------+-------------------+
|user_id|  activity|          timestamp|
+-------+----------+-------------------+
|      1|    search|2024-11-21T10:00:00Z|
|      2|  purchase|2024-11-21T11:30:00Z|
+-------+----------+-------------------+
```

This approach lets you efficiently process large JSON datasets for analysis and transformation in Spark.

Data Exploration and Cleaning

Step 1: Verify the Loaded Data
To ensure data is correctly loaded, examine the schema and preview sample records for each DataFrame.

```
# Display schema and sample data for each DataFrame
user_activity_df.printSchema()
user_activity_df.show(5)

transactions_df.printSchema()
transactions_df.show(5)

product_catalog_df.printSchema()
product_catalog_df.show(5)

user_profile_df.printSchema()
user_profile_df.show(5)
```

Detailed Explanation

1. Schema Inspection: `printSchema()`

Command:

```
user_activity_df.printSchema()
```

- **Purpose:** Displays the structure of the DataFrame, including column names, data types, and nullability.
- **Why it's important:**
  - Understand how Spark interprets the data structure from the source file.
  - Ensure that column data types align with your use case (e.g., numeric fields as `IntegerType` or `DoubleType`, text fields as `StringType`).
  - Identify any columns with nullable values for further cleaning.
- **Example Output:**

```
root
 |-- user_id: integer (nullable = true)
 |-- activity: string (nullable = true)
 |-- timestamp: string (nullable = true)
```

2. Preview Data: `show()`

Command:

```
user_activity_df.show(5)
```

- **Purpose:** Displays the first 5 rows of the DataFrame in a readable table format.
- **Why it's important:**
  - Quickly check if the data has loaded correctly.
  - Verify that columns contain expected data and formats.
  - Detect potential anomalies (e.g., incorrect parsing, missing or unexpected values).
- **Behavior:**
  - Defaults to showing 20 rows if no argument is provided (`.show()`).
  - Setting an argument, such as `.show(5)`, limits the number of rows displayed.
- **Example Output:**

```
+-------+----------+-------------------+
|user_id|  activity|          timestamp|
+-------+----------+-------------------+
|      1|    search|2024-11-21T10:00:00Z|
|      2|  purchase|2024-11-21T11:30:00Z|
+-------+----------+-------------------+
```

Step 2: Data Cleaning

1. Data Deduplication

Command:

```
transactions_df = transactions_df.dropDuplicates(["transaction_id"])
```

- **Purpose:** Removes duplicate rows based on the `transaction_id` column.
- **Why it's important:**
  - Duplicate records can skew analysis (e.g., inflating sales or user activity metrics).
  - Ensures accurate data for insights and transformations.

2. Handle Missing Values

Command:

```
user_profile_df = user_profile_df.fillna({"age": 0, "gender": "Unknown",
"region": "Unknown"})
```

- **Purpose:** Fills missing values in the `age`, `gender`, and `region` columns with default values.
- **Why it's important:**
  - Missing values can cause errors or inaccuracies in computations and transformations.
  - Setting default values ensures data consistency and avoids null-related errors.
- **Default Values Used:**
  - `age`: Replaced with `0` to indicate unknown or unspecified age.
  - `gender`: Replaced with `"Unknown"` to handle missing demographic information.
  - `region`: Replaced with `"Unknown"` to account for users with no specified location.

Summary of Commands

- **`printSchema()`**: Examines the structure of the DataFrame.
- **`show()`**: Previews sample rows to validate data.
- **`dropDuplicates()`**: Cleans duplicate entries based on specific columns.
- **`fillna()`**: Replaces missing values with defaults for specific columns.

These steps ensure your data is clean, accurate, and ready for further analysis or transformations.

Data Enrichment and Feature Engineering

We combine datasets using joins to create new features. For example, we'll create an enriched DataFrame by merging user activity, transaction data, product catalog, and user profile.

```
# Join user activity with transaction data, product catalog, and user profile
enriched_df = user_activity_df.join(transactions_df, ["user_id", "product_id"], "left") \
    .join(product_catalog_df, "product_id", "left") \
    .join(user_profile_df, "user_id", "left")

# Show a sample of the enriched DataFrame
enriched_df.show(5)
```

Steps Explained:

Step 1: Join `user_activity_df` and `transactions_df`

```
user_activity_df.join(transactions_df, ["user_id", "product_id"], "left")
```

- **Purpose**: Matches rows based on `user_id` and `product_id` columns.
- **Left Join**: Keeps all records from `user_activity_df` even if there's no match in `transactions_df`.
- **Result**:
    - Adds transaction columns (e.g., `transaction_id`, `purchase_amount`) to the activity data.
    - If there's no matching transaction, transaction columns will show as `null`.

Step 2: Join the result with `product_catalog_df`

```
.join(product_catalog_df, "product_id", "left")
```

- **Purpose**: Adds product details (e.g., `category`, `brand`, `price`) based on the `product_id`.
- **Left Join**: Keeps all rows from the previous step.
- **Result**:
    - Adds product information for each activity or transaction.
    - If a product isn't in the catalog, product columns will show as `null`.

Step 3: Join the result with `user_profile_df`

```
.join(user_profile_df, "user_id", "left")
```

- **Purpose**: Adds user details (e.g., `age`, `gender`, `location`) based on the `user_id`.
- **Left Join**: Ensures all rows from the previous step are retained.
- **Result**:
    - Adds user profile information to the combined data.
    - If a user profile is missing, user-related columns will show as `null`.

Final Output:

- The `enriched_df` contains:
    - **User Activity Data** (e.g., clicks, views).
    - **Transaction Data** (if available).
    - **Product Details** (if available).
    - **User Profiles** (if available).

Calculate Metrics and Aggregations

We calculate important metrics like **Average Session Time**, **Conversion Rate**, and **Top Products** using PySpark functions.

```
from pyspark.sql.functions import avg, count

# Calculate Average Session Time per user
avg_session_time_df = user_activity_df.groupBy("user_id") \
    .agg(avg("session_time").alias("avg_session_time"))
avg_session_time_df.show(5)

# Calculate Conversion Rate
conversion_rate_df = transactions_df.groupBy("user_id") \
    .agg((count("transaction_id") /
count("session_id")).alias("conversion_rate"))
conversion_rate_df.show(5)

# Identify Top Products by Total Sales
top_products_df = transactions_df.groupBy("product_id") \
    .agg(count("transaction_id").alias("total_sales")) \
    .orderBy("total_sales", ascending=False)
top_products_df.show(5)
```

Explanation Internals of the Code “Average session time”

This code computes the **average session time** for each user using a DataFrame `user_activity_df` and stores the result in `avg_session_time_df`. It also displays the first five rows of the result using the `show()` method. Let’s break it down step-by-step:

---

1. `user_activity_df.groupBy("user_id")`

- **Purpose**: Groups the data in the `user_activity_df` DataFrame by the `user_id` column.
  - This means all rows belonging to the same `user_id` will be grouped together.
  - After grouping, aggregate functions like `avg()` can be applied to each group.

---

2. `.agg(avg("session_time").alias("avg_session_time"))`

- **Purpose**: Performs an **aggregation operation** on each group (grouped by `user_id`).
  - `avg("session_time")`: Computes the **average** of the `session_time` column for each group (i.e., each user).
  - `.alias("avg_session_time")`: Renames the resulting column to `avg_session_time` for clarity.

---

3. Resulting DataFrame (`avg_session_time_df`)

- After the aggregation, the result is a DataFrame with two columns:
  - **`user_id`**: Unique identifier for each user.
  - **`avg_session_time`**: The average session time for that user, calculated from the `session_time` values.

---

4. `avg_session_time_df.show(5)`

- **Purpose**: Displays the first 5 rows of the resulting `avg_session_time_df` DataFrame.
  - By default, it also shows column names and formats the data as a table.

How It Works Internally

1. **Data Loading**:
   - Spark reads `user_activity_df`, which is likely a log of user activities with columns like `user_id`, `session_time`, and more.
2. **Grouping**:
   - Spark groups all rows in `user_activity_df` by `user_id`.

For example, if `user_activity_df` contains:

```
user_id   session_time

1         10

1         20

2         15

2         25

3         30
```

After grouping by `user_id`, the data is organized like this:

```
Group 1: user_id = 1 -> [10, 20]

Group 2: user_id = 2 -> [15, 25]

Group 3: user_id = 3 -> [30]
```

    -
3. **Aggregation**:

The `avg("session_time")` function computes the mean for each group:

```
Group 1: Avg = (10 + 20) / 2 = 15

Group 2: Avg = (15 + 25) / 2 = 20

Group 3: Avg = 30 / 1 = 30
```

This results in:

```
user_id   avg_session_time

1         15

2         20

3         30
```

    -
4. **Column Renaming**:
    - The average column is renamed to `avg_session_time`.
5. **Displaying Results**:
    - `show(5)` displays the first 5 rows of the resulting DataFrame.

Example Data and Output

Input Data (`user_activity_df`)

user_id	session_time
1	10
1	20
2	15
2	25
3	30
3	20
4	50

Grouped Data

user_id	session_times
1	[10, 20]
2	[15, 25]
3	[30, 20]
4	[50]

Aggregated Result (`avg_session_time_df`)

user_id	avg_session_time
1	15.0
2	20.0
3	25.0
4	50.0

Why Calculate Average Session Time?

Steps Explained

1. Average Session Time

```
avg_session_time_df = user_activity_df.groupBy("user_id") \
    .agg(avg("session_time").alias("avg_session_time"))
```

- **Purpose**: Calculates the average time each user spends in a session.
- **Steps**:
    - `groupBy("user_id")`: Groups data by `user_id` so calculations are done for each user separately.
    - `avg("session_time")`: Calculates the average of the `session_time` column for each user.
    - `.alias("avg_session_time")`: Names the resulting column as `avg_session_time`.

---

2. Conversion Rate

```
conversion_rate_df = transactions_df.groupBy("user_id") \
    .agg((count("transaction_id") /
count("session_id")).alias("conversion_rate"))
```

- **Purpose**: Calculates the ratio of purchases (transactions) to sessions for each user.
- **Steps**:
    1. **Group Data**:
        - `groupBy("user_id")`: Groups data by `user_id` for calculations per user.
    2. **Count Transactions**:
        - `count("transaction_id")`: Counts how many transactions a user completed.
        - Assumes `transaction_id` is unique and non-null for successful transactions.
    3. **Count Sessions**:
        - `count("session_id")`: Counts the number of sessions each user participated in.
        - Assumes `session_id` tracks user interactions with the platform.
    4. **Calculate Conversion Rate**:
        - `count("transaction_id") / count("session_id")`: Divides transactions by sessions to get the conversion rate.
        - Example:
            - 3 purchases over 10 sessions → 0.3\frac{3}{10} = 0.3103=0.3 or 30%.
        - Ensure `session_id` count is non-zero to avoid division errors.
    5. **Alias**:
        - `.alias("conversion_rate")`: Names the resulting column as `conversion_rate`.

3. Top Products by Total Sales

```
top_products_df = transactions_df.groupBy("product_id") \
    .agg(count("transaction_id").alias("total_sales")) \
    .orderBy("total_sales", ascending=False)
```

- **Purpose**: Identifies products with the highest sales.
- **Steps**:
    1. **Group Data**:
        - `groupBy("product_id")`: Groups data by `product_id` to calculate metrics for each product.
    2. **Count Transactions**:
        - `count("transaction_id")`: Counts the total number of sales (transactions) for each product.
        - Assumes each `transaction_id` corresponds to one sale.
    3. **Sort Results**:
        - `.orderBy("total_sales", ascending=False)`: Orders products by total sales in descending order.
    4. **Alias**:
        - `.alias("total_sales")`: Names the resulting column as `total_sales`.

Final Output

1. **Average Session Time DataFrame (`avg_session_time_df`)**:
    - Contains `user_id` and their average session time.
2. **Conversion Rate DataFrame (`conversion_rate_df`)**:
    - Contains `user_id` and their conversion rate (ratio of purchases to sessions).
3. **Top Products DataFrame (`top_products_df`)**:
    - Contains `product_id` and the total number of sales for each product.

Recommendation System Using ALS

We use the **ALS (Alternating Least Squares)** algorithm to build a recommendation model in PySpark.

```
from pyspark.ml.recommendation import ALS

# Configure ALS model
als = ALS(
    userCol="user_id",
    itemCol="product_id",
    ratingCol="purchase_amount",
    rank=10,
    maxIter=5,
    seed=42
)

# Fit the model on transaction data
als_model = als.fit(transactions_df)

# Generate recommendations for each user
user_recommendations = als_model.recommendForAllUsers(10)
user_recommendations.show(5, truncate=False)
```

Explanation

1. **ALS Model Configuration**:
    - `ALS`: A PySpark class for building collaborative filtering recommendation systems.
    - **Parameters**:
        - `userCol`: Column representing users (e.g., `user_id`).
        - `itemCol`: Column representing items (e.g., `product_id`).
        - `ratingCol`: Column representing user ratings or purchase amounts.
        - `rank`: Number of latent factors for recommendations (e.g., set to `10`).
        - `maxIter`: Maximum number of iterations for optimization (e.g., set to `5`).
        - `seed`: Ensures reproducibility by fixing the random seed.
2. **Model Fitting**:
    - `als.fit(transactions_df)`: Trains the ALS model on the transaction dataset (`transactions_df`).
3. **Generate Recommendations**:
    - `als_model.recommendForAllUsers(10)`: Creates a list of top 10 recommended products for each user.
4. **Display Recommendations**:
    - `user_recommendations.show(5, truncate=False)`: Shows the first 5 recommendations without truncating data.

Note

This example uses **AI modeling**, which is outside the scope of this lecture. You can skip it if you're focusing only on data engineering or preprocessing tasks.

Real-Time Recommendations Based on Recent Activity

To focus on recent user actions, filter the data for activity within the last hour.

```
from pyspark.sql.functions import current_timestamp

# Get recent user activity within the last hour
recent_activity_df = user_activity_df.filter("timestamp >= current_timestamp - interval 1 hour").cache()

# Show the filtered data
recent_activity_df.show(5)
```

Explanation

1. **Filtering Recent Activity**
    - **`filter(...)`**:
        - Filters rows in the DataFrame based on a condition.
        - Here, we keep only rows where the activity happened within the past hour.
        -
    - **Condition Explained**:

        - `timestamp >= current_timestamp - interval 1 hour`:
            - **`timestamp`**: Column storing the activity time.
            - **`current_timestamp`**: Gets the current date and time.
            - **`interval 1 hour`**: Represents a one-hour time frame.
            - The filter ensures the DataFrame includes only activities between one hour ago and the current time.
    - **Example**:
        - If the current time is 12:00, this filter keeps activities from 11:00 to 12:00.
2. **Caching the Filtered Data**
    - **`cache()`**:
        - Saves the filtered DataFrame in memory for quick access.
        - Useful when the data will be used multiple times in the same application.
    - **Why Cache?**:
        - Avoids repeating the filtering process.
        - Improves performance by reducing computation time.
    - **Note**:
        - Cache only when the data is reused, as it uses memory.
3. **Resulting DataFrame**
    - **`recent_activity_df`**:
        - Contains all user activities from the past hour.
        - Optimized for faster processing due to caching.
    - **Usage**:
        - Ideal for tasks like generating real-time recommendations or analyzing recent user behavior.

Shutdown Spark Session

To properly close the Spark session after completing your tasks, use the following:

```
spark.stop()
```

Explanation

- **`spark.stop()`**:
    - This command gracefully terminates the Spark session.
    - It releases resources and prevents potential issues in both local and distributed environments.
- **Best Practice**:
    - It's recommended to always include `spark.stop()` at the end of a Spark application to ensure resources are correctly freed up and managed.

Writing end-to-end Spark Application - Nov 22

- **Importing Required Modules**
- **Initialize Spark Session with Configurations**
- **Data Ingestion and Creating Temporary Views**
- **Data Cleaning and Caching**
- **User Engagement Analysis and Segmentation**
- **Article Popularity Analysis Using Window Functions**
- **Demographic Preferences Analysis**
- **Advanced Aggregation - Average Time Spent by User per Category**
- **Save DataFrames as Permanent Tables**
- **Export Results to External Files**

Data Resources

https://drive.google.com/file/d/16CtbizPtUFISw_eYN7aiXO9QEILYgvrv/view?usp=sharing
https://drive.google.com/file/d/1Cv5cCuBUOV58Bsxt5Q4_IsE0SXavSpUg/view?usp=sharing
https://drive.google.com/file/d/1zGjBwxBeaNG97PUNq4Enots5n-4abvw2/view

End to End Application

Step 1: Importing Required Modules

```
# Importing necessary modules from PySpark
# For creating Spark applications
from pyspark.sql import SparkSession
# For various DataFrame operations
from pyspark.sql.functions import col, count, sum, avg, expr, when, rank  from
# For creating window specifications
pyspark.sql.window import Window
```

Step 2: Initialize Spark Session with Configurations

```
# Step 2: Initialize Spark Session with Configurations
spark = SparkSession.builder \
    .appName("News Platform Analytics") \
    .config("spark.sql.shuffle.partitions", "200") \
    .config("spark.sql.autoBroadcastJoinThreshold", "1000") \
    .getOrCreate()
```

Explanation:

- `SparkSession.builder` initializes the Spark session.
- `.appName("...")` sets a name for the application, useful for tracking logs and monitoring.
- `.config(...)` is used to configure the Spark environment:

- **`spark.sql.shuffle.partitions`**: Adjusts the number of partitions during shuffle operations for optimization.
- **`spark.sql.autoBroadcastJoinThreshold`**: Sets the limit for broadcasting large tables in joins. It is disabled for very large tables here to prevent memory errors.

Step 3: Data Ingestion and Creating Temporary Views

```
# Step 3: Data Ingestion and Creating Temporary Views
user_activity_df = spark.read.csv("new_user_activity.csv", header=True, inferSchema=True)
article_metadata_df = spark.read.csv("article_metadata.csv", header=True, inferSchema=True)
user_profile_df = spark.read.csv("user_profile_new.csv", header=True, inferSchema=True)

# Registering DataFrames as temporary SQL views
user_activity_df.createOrReplaceTempView("user_activity")
article_metadata_df.createOrReplaceTempView("article_metadata")
user_profile_df.createOrReplaceTempView("user_profile")
```

Explanation:

- `spark.read.csv(...)` reads data from CSV files into Spark DataFrames:
  - **`header=True`**: Indicates that the first row of the CSV contains column names.
  - **`inferSchema=True`**: Automatically detects the data types for the columns.
- `.createOrReplaceTempView(...)` creates temporary SQL-like tables in Spark, enabling querying the DataFrames using SQL.

Step 4: Data Cleaning and Caching

```
# Step 4: Data Cleaning and Caching
user_activity_df = user_activity_df.dropDuplicates(["user_id", "article_id", "timestamp"])
user_profile_df = user_profile_df.fillna({"age": 0, "region": "Unknown", "gender": "Unknown"})

# Caching DataFrames to memory for faster access
user_activity_df.cache()
user_profile_df.cache()
article_metadata_df.cache()
```

Explanation:

- `.dropDuplicates(...)` removes duplicate rows based on specified columns, ensuring clean data.
- `.fillna(...)` replaces missing (null) values in specified columns with default values:
  - **`age`**: Default value set to `0`.
  - **`region`**: Default value set to `"Unknown"`.
  - **`gender`**: Default value set to `"Unknown"`.
- `.cache()` stores the DataFrame in memory to improve performance for repeated operations on the same data.

Step 4: User Engagement Analysis and Segmentation

```
# Step 4: User Engagement Analysis and Segmentation
user_engagement_sql = """
    SELECT
        user_id,
        COUNT(CASE WHEN action = 'read' THEN 1 END) AS read_count,
        COUNT(CASE WHEN action = 'like' THEN 1 END) AS like_count,
        COUNT(CASE WHEN action = 'share' THEN 1 END) AS share_count,
        CASE
            WHEN COUNT(action) >= 100 THEN 'Highly Engaged'
            WHEN COUNT(action) BETWEEN 50 AND 99 THEN 'Moderately Engaged'
            ELSE 'Low Engagement'
        END AS engagement_level
    FROM user_activity
    GROUP BY user_id
"""
user_engagement_df = spark.sql(user_engagement_sql)
user_engagement_df.createOrReplaceTempView("user_engagement")
```

Explanation:

- The SQL query categorizes users based on their level of engagement.
- `COUNT(CASE WHEN ...)`:
  - Counts specific user actions like **`read`**, **`like`**, and **`share`**.
- `CASE WHEN`:
  - Defines user engagement levels:
    - **`Highly Engaged`**: Total actions ≥ 100.
    - **`Moderately Engaged`**: Total actions between 50 and 99.
    - **`Low Engagement`**: Total actions < 50.
- `GROUP BY user_id` ensures the aggregation is calculated for each user.

Step 5: Article Popularity Analysis Using Window Functions

```
# Step 5: Article Popularity Analysis Using Window Functions
article_engagement_df = user_activity_df.groupBy("article_id") \
    .agg(
        count(when(col("action") == "read", True)).alias("views"),
        count(when(col("action") == "like", True)).alias("likes"),
        count(when(col("action") == "share", True)).alias("shares")
    )

article_with_metadata_df = article_engagement_df.join(article_metadata_df,
"article_id", "inner")
```

```
# Define a window specification for ranking articles within each category
category_window = Window.partitionBy("category").orderBy(col("views").desc())

# Apply the rank function over the window to assign a rank within each category
ranked_articles_df = article_with_metadata_df.withColumn("rank_in_category",
rank().over(category_window))

# Register ranked articles as a temporary view
ranked_articles_df.createOrReplaceTempView("ranked_articles")
```

Explanation:

- **Step 1: Grouping and Aggregation**
  - `.groupBy("article_id")`: Groups actions by article ID.
  - `.agg(...)`: Aggregates data for each article:
    - **`views`**: Counts `read` actions.
    - **`likes`**: Counts `like` actions.
    - **`shares`**: Counts `share` actions.
  - `when(...)`: Filters specific actions.
- **Step 2: Joining Metadata**
  - `.join(...)`: Combines article engagement data with metadata on `article_id`.
- **Step 3: Window Functions for Ranking**
  - `Window.partitionBy("category")`: Divides data by category for separate rankings.
  - `.orderBy(col("views").desc())`: Orders articles within each category by views (highest first).
  - `.withColumn("rank_in_category", rank().over(category_window))`: Assigns a rank to each article within its category based on views.
- **Step 4: Registering as Temporary View**
  - `.createOrReplaceTempView(...)`: Registers the resulting DataFrame as a temporary view for querying with Spark SQL.

Step 6: Demographic Preferences Analysis

```
# Step 6: Demographic Preferences Analysis
demographic_preferences_sql = """
    SELECT
        p.region,
        p.gender,
        m.category,
        COUNT(ua.user_id) AS engagement_count
    FROM user_profile p
    JOIN user_activity ua ON p.user_id = ua.user_id
    JOIN article_metadata m ON ua.article_id = m.article_id
    GROUP BY p.region, p.gender, m.category
    ORDER BY p.region, p.gender, engagement_count DESC
"""
demographic_preferences_df = spark.sql(demographic_preferences_sql)
demographic_preferences_df.createOrReplaceTempView("demographic_preferences")
```

Explanation:

- **Purpose**: Analyzes popular categories by demographic (region and gender).
- **Joins**:
  - `JOIN user_profile p` with `user_activity ua`: Links user demographics with their activities.
  - `JOIN article_metadata m`: Links activities to article categories.
- **Aggregation**:
  - `COUNT(ua.user_id)`: Counts user engagements per category.
  - `GROUP BY p.region, p.gender, m.category`: Groups the data by region, gender, and category.
- **Ordering**:
  - `ORDER BY engagement_count DESC`: Sorts results by engagement count, highest first.

Step 7: Advanced Aggregation - Average Time Spent by User per Category

```
# Step 7: Advanced Aggregation - Average Time Spent by User per Category
user_time_spent_df = user_activity_df.groupBy("user_id", "article_id") \
    .agg(sum("time_spent").alias("total_time_spent"))
user_time_spent_df.createOrReplaceTempView("user_time_spent")

avg_time_spent_sql = """
    SELECT
        u.user_id,
        m.category,
        AVG(t.total_time_spent) AS avg_time_spent
    FROM user_time_spent t
    JOIN article_metadata m ON t.article_id = m.article_id
    JOIN user_profile u ON t.user_id = u.user_id
    GROUP BY u.user_id, m.category
    ORDER BY avg_time_spent DESC
"""
avg_time_spent_df = spark.sql(avg_time_spent_sql)
avg_time_spent_df.createOrReplaceTempView("avg_time_spent")
```

Explanation:

- **Purpose**: Calculates the average time each user spends on articles in different categories.
- **Step 1: Total Time Calculation**
  - `.groupBy("user_id", "article_id")`: Groups activities by user and article.
  - `.agg(sum("time_spent").alias("total_time_spent"))`: Sums time spent per article.
  - Creates a temporary view (`user_time_spent`).
- **Step 2: Average Time Calculation**
  - `AVG(t.total_time_spent)`: Calculates average time spent per category for each user.
  - `GROUP BY u.user_id, m.category`: Groups results by user and article category.
- **Ordering**:
  - `ORDER BY avg_time_spent DESC`: Sorts by average time, highest first.

Step 8: Save DataFrames as Permanent Tables

```
# Step 8: Save DataFrames as Permanent Tables
user_engagement_df.write.saveAsTable("user_engagement_segmented")
ranked_articles_df.write.saveAsTable("ranked_article_popularity")
demographic_preferences_df.write.saveAsTable("demographic_preferences")
avg_time_spent_df.write.saveAsTable("avg_time_spent_per_user")
```

Explanation:

- **Purpose**: Saves processed DataFrames as managed tables for future querying within Spark.
- **`.write.saveAsTable(...)`**:
  - Saves each DataFrame as a table in Spark's catalog.
  - Examples:
    - `user_engagement_segmented`: Segmented user engagement.
    - `ranked_article_popularity`: Ranked articles within categories.
    - `demographic_preferences`: Preferences by demographics.
    - `avg_time_spent_per_user`: Average time spent per user per category.

Step 9: Export Results to External Files

```
# Step 9: Export Results to External Files
user_engagement_df.write.csv("user_engagement.csv", header=True, mode="overwrite")
ranked_articles_df.write.json("ranked_articles.json", mode="overwrite")
demographic_preferences_df.write.csv("demographic_preferences.csv", header=True,
mode="overwrite")
```

Explanation:

- **Purpose**: Exports results as external files in specified formats for external use.
- **`.write.csv(...)`**:
  - Saves DataFrame to CSV format.
  - Adds headers to the file.
  - Overwrites existing files if necessary.
- **`.write.json(...)`**:
  - Saves DataFrame to JSON format.
- Examples:
  - **`user_engagement.csv`**: Stores user engagement data.
  - **`ranked_articles.json`**: Stores ranked articles.
  - **`demographic_preferences.csv`**: Stores demographic preferences.

Step 10: Stop Spark Session

```
# Step 10: Stop Spark Session
spark.stop()
```

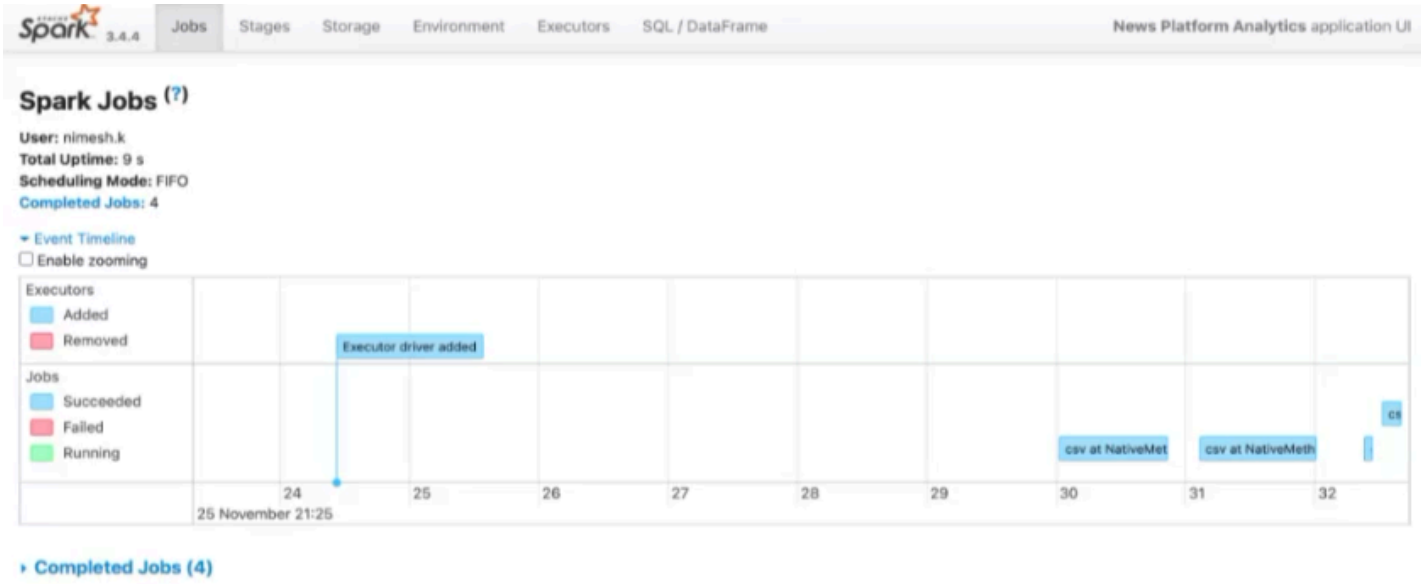
Exploring SHS: Jobs, Stages and Tasks - Nov 25

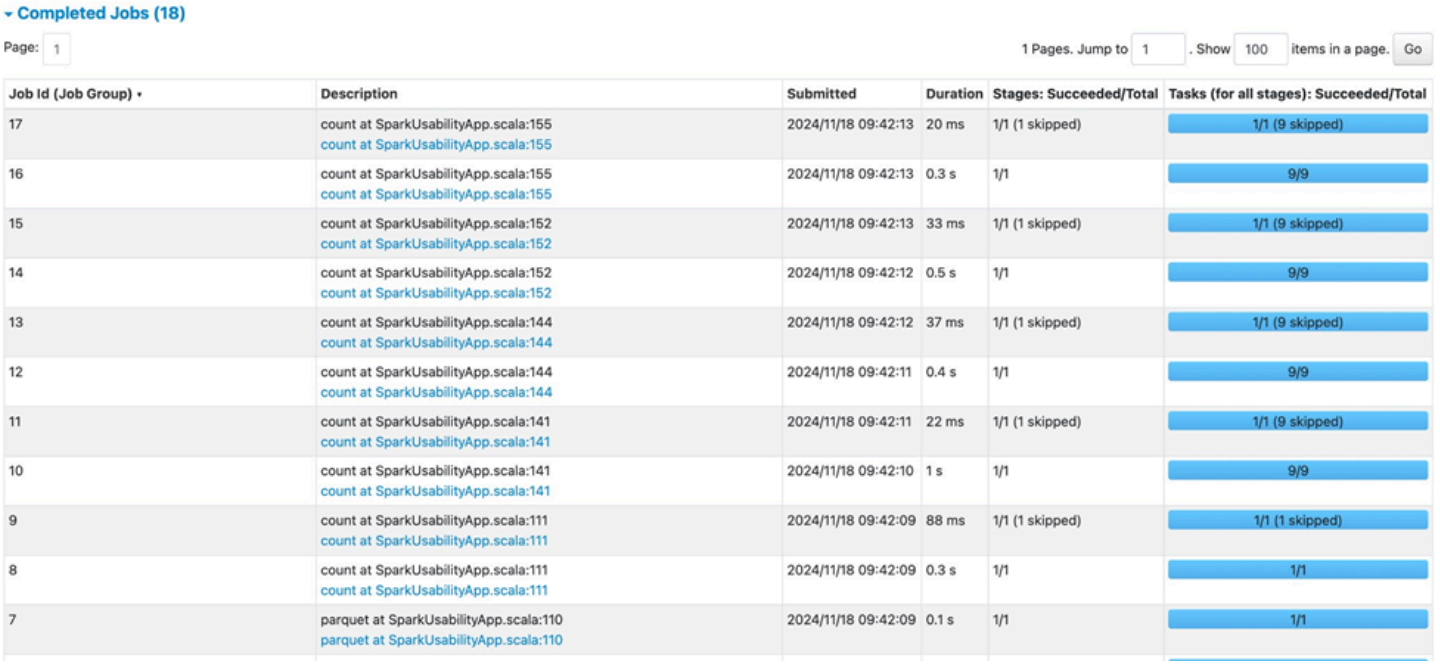
- **Jobs Tab**
    - **User**
    - **Total Uptime**
    - **Scheduling Mode**
    - **Completed Jobs**
    - **Event Timeline**
- **Stage Tab**
    - **Total Time Across All Tasks**
    - **Locality**
    - **Associated Job IDs**
    - **DAG Visualization**
    - **Event Timeline**
    - **Aggregated Metrics by Executors**
    - **Aggregated Metrics by Executors**

In earlier lectures, we have looked at the SHS (Spark History Server) several times to analyze how our application is working. In this lecture, we will explore different parts of SHS in detail.

Jobs Tab ==========

The Jobs tab shows various pieces of information. Let's go through each of them to understand them better.





▾ Completed Jobs (18)

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Job Id (Job Group)	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
17	count at SparkUsabilityApp.scala:155 count at SparkUsabilityApp.scala:155	2024/11/18 09:42:13	20 ms	1/1 (1 skipped)	1/1 (9 skipped)
16	count at SparkUsabilityApp.scala:155 count at SparkUsabilityApp.scala:155	2024/11/18 09:42:13	0.3 s	1/1	9/9
15	count at SparkUsabilityApp.scala:152 count at SparkUsabilityApp.scala:152	2024/11/18 09:42:13	33 ms	1/1 (1 skipped)	1/1 (9 skipped)
14	count at SparkUsabilityApp.scala:152 count at SparkUsabilityApp.scala:152	2024/11/18 09:42:12	0.5 s	1/1	9/9
13	count at SparkUsabilityApp.scala:144 count at SparkUsabilityApp.scala:144	2024/11/18 09:42:12	37 ms	1/1 (1 skipped)	1/1 (9 skipped)
12	count at SparkUsabilityApp.scala:144 count at SparkUsabilityApp.scala:144	2024/11/18 09:42:11	0.4 s	1/1	9/9
11	count at SparkUsabilityApp.scala:141 count at SparkUsabilityApp.scala:141	2024/11/18 09:42:11	22 ms	1/1 (1 skipped)	1/1 (9 skipped)
10	count at SparkUsabilityApp.scala:141 count at SparkUsabilityApp.scala:141	2024/11/18 09:42:10	1 s	1/1	9/9
9	count at SparkUsabilityApp.scala:111 count at SparkUsabilityApp.scala:111	2024/11/18 09:42:09	88 ms	1/1 (1 skipped)	1/1 (1 skipped)
8	count at SparkUsabilityApp.scala:111 count at SparkUsabilityApp.scala:111	2024/11/18 09:42:09	0.3 s	1/1	1/1
7	parquet at SparkUsabilityApp.scala:110 parquet at SparkUsabilityApp.scala:110	2024/11/18 09:42:09	0.1 s	1/1	1/1

1. User

- **Description**: This column or section shows the username or ID of the user who submitted the job.
- **Purpose**: Helps track and assign jobs to specific users in systems with multiple users.
- **Significance**: Useful for auditing, monitoring user activities, or troubleshooting job issues.
    - **Example**: In a corporate setting, this might show individual developers or team service accounts.

---

2. Total Uptime

- **Description**: Displays the total time the Spark application or job has been running.
- **Calculation**: Measured from the job's start time to the current time (for active jobs) or the completion time (for finished jobs).
- **Significance**:
    - Long uptimes might indicate inefficiencies, like resource contention or poor configurations.
    - Short uptimes could mean jobs are well-optimized or lightweight.

---

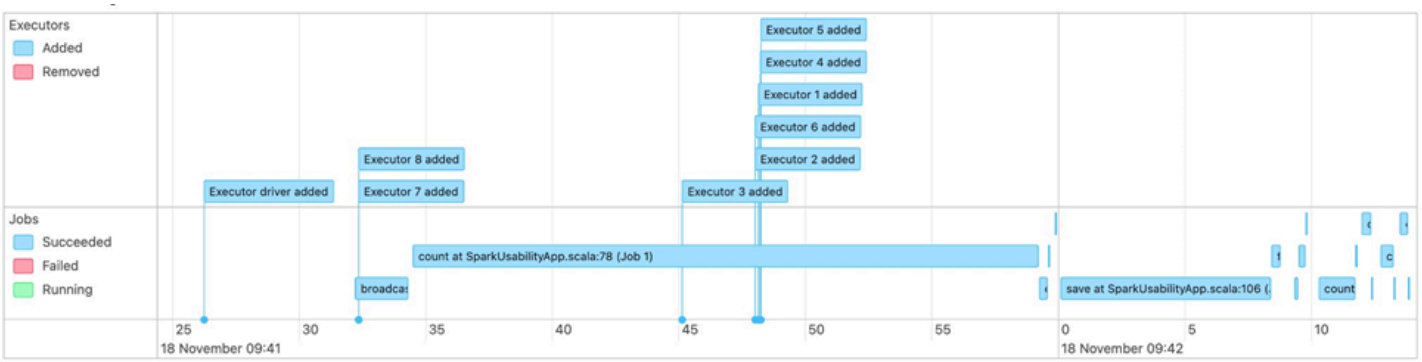
3. Scheduling Mode

- **Description**: Shows how tasks are scheduled and resources are allocated. Common scheduling modes include:
    - **FIFO (First In, First Out)**: Jobs run in the order they are submitted.
    - **FAIR**: Resources are shared fairly among all running jobs.
- **Purpose**: Defines how jobs are prioritized and resources are distributed in multi-job scenarios.
- **Significance**:
    - **FIFO**: Ensures predictable execution for earlier submitted jobs.
    - **FAIR**: Allows multiple jobs to run simultaneously, avoiding resource monopolization (monopoly).

---

4. Completed Jobs

- **Description**: Lists jobs that have finished running.
- **Details Typically Included**:
    - **Job ID**: A unique identifier for each job.
    - **Status**: Indicates if the job succeeded or failed.
    - **Duration**: The total time taken to complete the job.
    - **Stages and Tasks**: Number of stages and tasks executed.
- **Purpose**: Provides a record of completed jobs for auditing, debugging, and performance optimization.
- **Significance**:
    - Enables investigation of failed or inefficient jobs using logs and metrics.
    - Helps as a baseline for improving future workloads.



5. Event Timeline

The Event Timeline is a visual tool that represents the lifecycle of a job or application in a distributed system like Spark. It shows key events such as job submission, stage execution, and task completion in a timeline format, often divided into phases or stages.

Components of the Event Timeline

1. **Job Execution Timeline**
    - Tracks the start and end times of each job.
    - Useful for identifying delays or bottlenecks at the job level.
2. **Stage Timeline**
    - Shows when different stages of a job started and finished.
    - Includes stage dependencies and any overlaps between stages.
3. **Task Timeline**
    - Displays execution times for individual tasks.
    - Highlights task distribution across executors and retries or failures.
4. **Executor/Resource Utilization**
    - Indicates when resources like CPU and memory were active.
    - Helps identify resource contention or underutilization.

How the Event Timeline Helps in Debugging

The Event Timeline provides a detailed breakdown of events during job execution. This is extremely helpful for debugging. Here's how:

1. **Identify Bottlenecks**
   - Find stages or tasks that took much longer than expected.
   - Example: If one stage is significantly slower, it may point to issues like data skew or inefficient partitioning.
2. **Analyze Task Failures**
   - Locate tasks that failed or retried often.
   - Determine if the failures were caused by executor crashes, lack of resources, or bad input data.
3. **Understand Parallelism**
   - Check if tasks are running in parallel or sequentially.
   - Sequential task execution might indicate underuse of cluster resources.
4. **Resource Utilization**
   - Match task execution times with resource usage (e.g., memory spikes or I/O issues).
   - Spot problems like executor slowdowns or memory errors.

Example: Debugging a Job Using the Event Timeline

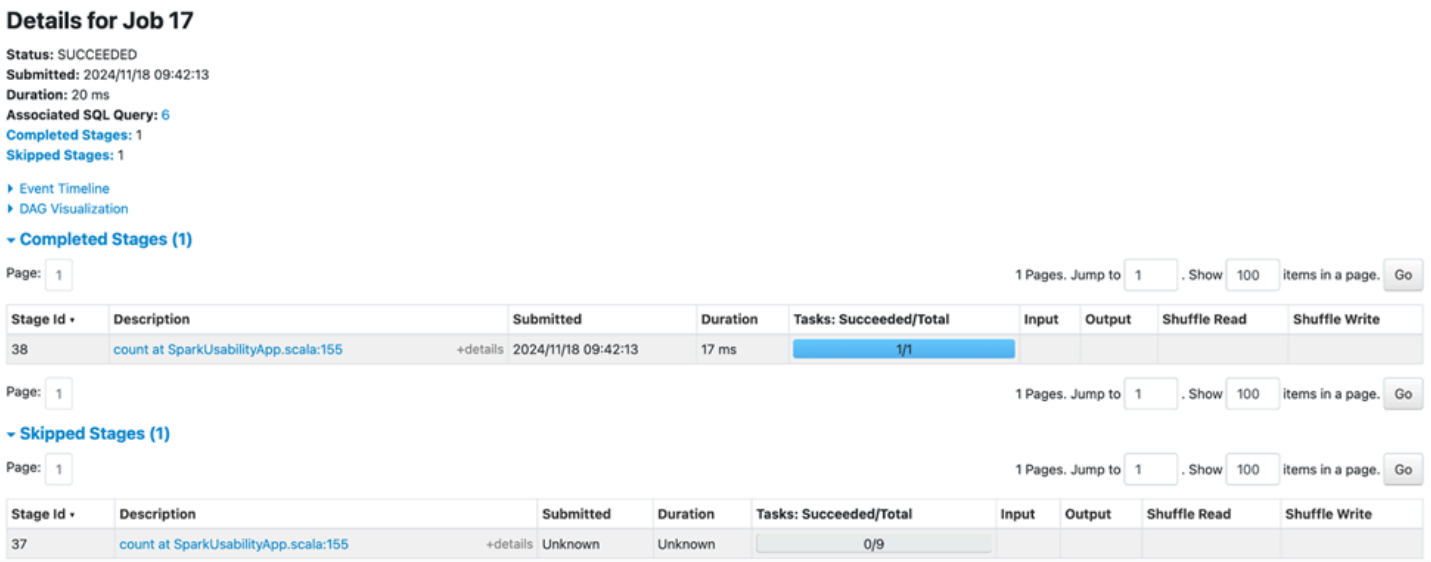
Scenario: A Spark job is running slower than usual, taking 30 minutes instead of the expected 10 minutes.

Steps for Debugging:

1. **Inspect the Job Timeline**
   - Look for jobs with unusually long runtimes.
   - Example: Job ID 5 takes 25 minutes, while others are under 5 minutes.
2. **Analyze the Stage Timeline**
   - Focus on the long-running stage in Job ID 5.
   - Identify Stage 3, which takes 22 minutes, as the bottleneck.
3. **Drill Down into Task Execution**
   - Examine the Task Timeline for Stage 3.
   - Notice that 90% of tasks finish quickly, but 10% take much longer.
4. **Diagnose Data Skew**
   - Discover that the slow tasks process large data partitions due to uneven distribution.
   - The timeline shows these tasks as outliers.
5. **Resolution**
   - Adjust the partitioning logic in the Spark job for better data distribution.
   - Rerun the job and verify improvements using the Event Timeline.

By following these steps, you can systematically debug and optimize your Spark job.

Details of One Job



1. **Status**
   - **Description**: Indicates the current state of the job.
     - **Running**: The job is currently executing.
     - **Succeeded**: The job finished successfully without errors.
     - **Failed**: The job encountered errors during execution.
     - **Skipped**: Certain stages or tasks were skipped due to optimizations or failures in dependent stages.
   - **Significance**:
     - Quickly identifies if the job is progressing as expected.
     - Alerts users to failures or partial executions that need attention.
     -
2. **Submitted**
   - **Description**: Shows the timestamp when the job was submitted for execution.
   - **Significance**:
     - Helps track job submission times in production.
     - Correlates job submissions with other system activities, like resource allocation or logs.
3. **Duration**
   - **Description**: Total time taken by the job to complete (or elapsed time for a running job).
     - **Calculation**:

$$\text{Duration} = \text{Completion Time} - \text{Start Time}$$

   - **Significance**:
     - Highlights performance issues.
     - Allows comparison with historical runtimes to identify anomalies.
4. **Associated SQL Query**
   - **Description**: Shows the SQL query (if any) that triggered the job.
   - **Significance**:
     - Links the job to the high-level query logic for debugging.
     - Helps understand how
5. **Completed Stages**
   - **Description**: Lists the stages successfully completed as part of the job.
     - Stages are groups of parallel tasks Spark schedules and executes.
   - **Significance**:
     - Monitors progress in large, multi-stage jobs.
     - Identifies bottlenecks when specific stages take longer to finish.
     -

6. **DAG Visualization**
    - **Description**: Displays a Directed Acyclic Graph (DAG) of the job's execution plan.
        - **Nodes**: Represent stages.
        - **Edges**: Show dependencies between stages.
    - **Significance**:
        - Visualizes data flow and execution logic.
        - Helps detect inefficiencies, like redundant shuffles or wide dependencies.
7. **Skipped Stages**
    - **Description**: Lists stages skipped during job execution.
    - **Why Stages Are Skipped**:
        - **Stage Caching**: Output from a stage is cached, skipping recalculations.
        - **Stage Failures**: Downstream stages are skipped if dependent stages fail.
        - **Stage Sharing**: Multiple actions (e.g., `count()` and `collect()`) on the same RDD/DataFrame share stages, avoiding redundancy.
        - **Result Reuse**: Optimizations like query result caching or reused shuffle outputs skip repeated computations.
    - **Significance**:
        - Improves performance by reusing previously computed results.
        - Reduces resource usage when intermediate data is cached or reused.

Example of Skipped Stages

Scenario:
You execute a Spark SQL query like:

```
SELECT COUNT(*) FROM events WHERE event_type = 'click';
```

This query involves two actions:

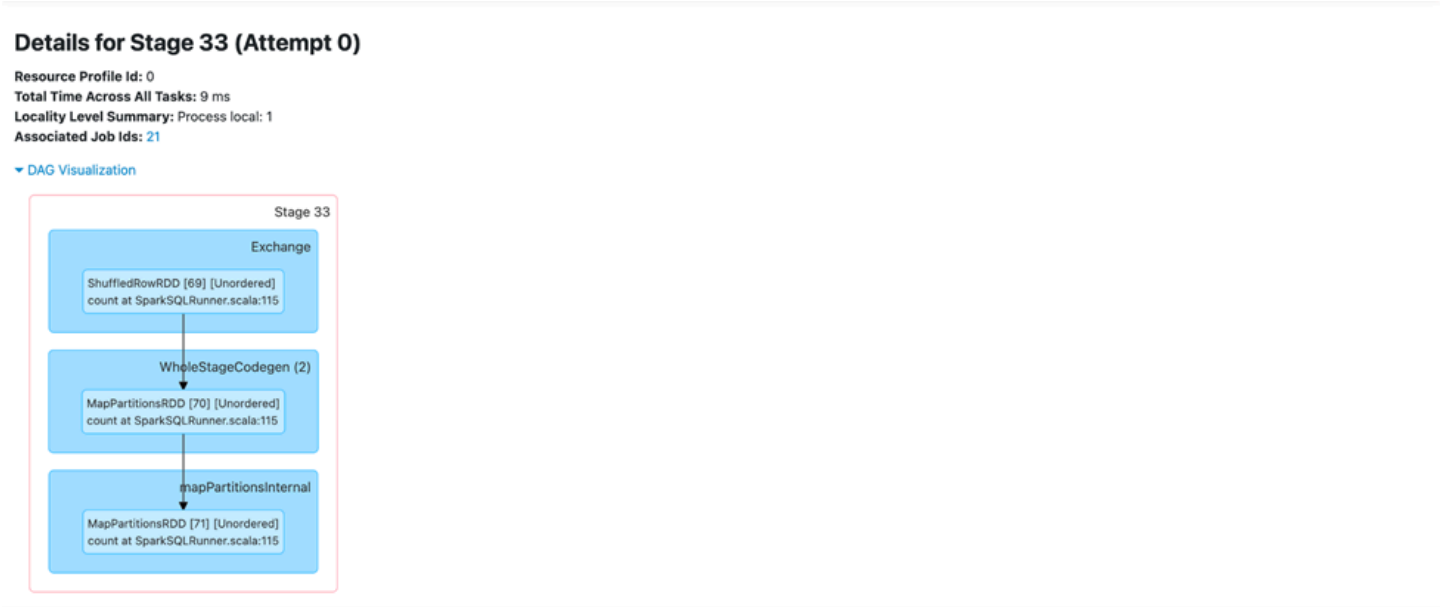
1. **Action 1**: `cache()` is applied to filter results where `event_type = 'click'`.
2. **Action 2**: `COUNT(*)` is performed on the cached results.

In this case:

- The stage responsible for filtering and caching runs once.
- When `COUNT(*)` is executed, the filtering stage is skipped because the results are reused from the cache.

Stage Tab in Spark History Server (SHS) ==========

The **Stage Tab** provides valuable metrics and insights into the performance and execution details of each stage in a Spark application. Below are key terms and their significance:



1. **Total Time Across All Tasks**
    - **Definition**: The cumulative time spent executing all tasks in a stage.
    - **Calculation**:

$$\text{Total Time Across All Tasks} = \sum_{i=1}^{N} (\text{Finish Time}_i - \text{Start Time}_i)$$

      Where N is the total number of tasks in the stage.

    - **Includes**: Both successful and failed tasks.
    - **Importance**:
        - Reflects the computational effort for a stage.
        - Identifies bottlenecks, such as tasks taking unusually long.
        - Useful for comparing stage performance and diagnosing inefficiencies.
2. **Locality**
    - **Definition**: Indicates how close a task is to the data it processes (data locality).
    - **Locality Levels**:
        - **PROCESS_LOCAL**: Task runs on the same node as the data.
        - **NODE_LOCAL**: Task runs on the same node but fetches data over the network.
        - **RACK_LOCAL**: Task runs on a different node within the same rack.
        - **ANY**: Task runs anywhere in the cluster.
    - **Why It Matters**:
        - Tasks closer to the data reduce data transfer overhead.
        - High percentages of **PROCESS_LOCAL** tasks indicate efficient scheduling.
        - Poor locality (e.g., many **ANY** tasks) may signal data skew or storage issues.

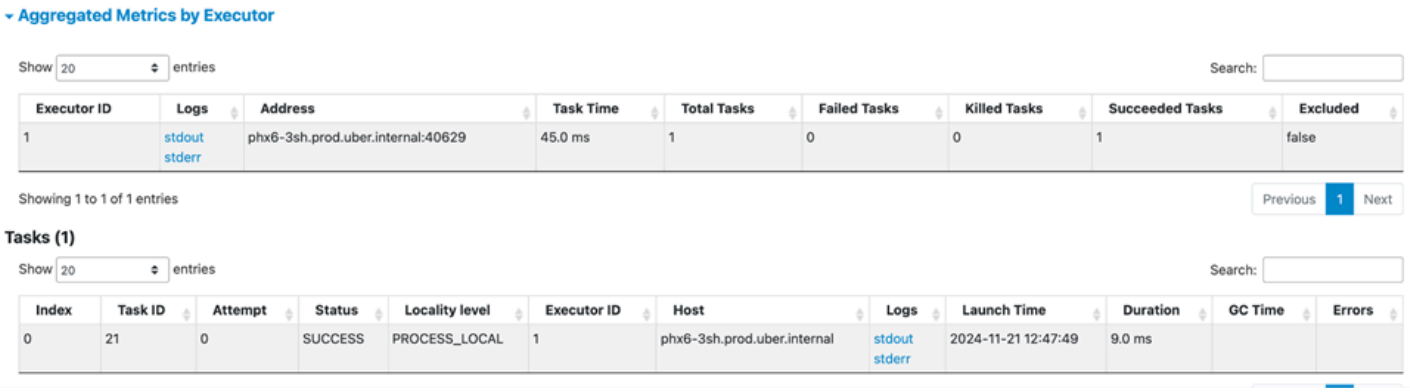
3. **Associated Job IDs**
    - **Definition**: Each stage is linked to one or more Spark jobs based on dependencies.
    - **Where to Find**: The Stage Tab shows associated Job IDs in a dedicated column.
    - **Purpose**:
        - Tracks which job a stage belongs to, useful in multi-job applications.
        - Provides context for the stage's role in the application.
        - Links to the **Jobs Tab** for a broader view when you click on a Job ID.

4. DAG Visualization

- **Definition**: Spark generates a Directed Acyclic Graph (DAG) to represent the sequence of computations, including transformations and actions.
- **In the Stage Tab**:
    - The DAG visualization for the selected stage is available.
    - Highlights relationships between RDDs, stages, and parent-child dependencies.
    - **Interactivity**:
        - Hovering over a stage node reveals details like task count and shuffle dependencies.
        - Identifies bottlenecks such as expensive shuffles in the execution flow.
- **Usefulness**:
    - Provides a visual summary of the execution plan.
    - Aids in debugging performance issues like data skew or redundant operations.

5. Event Timeline

- **Definition**: A graphical representation of timing information for the stage and its tasks.
- **Components**:
    - **X-Axis**: Displays elapsed time (milliseconds or seconds).
    - **Bars/Lines**: Represent each task's execution time, with colors indicating task states (e.g., running, successful, failed).
- **Insights Provided**:
    - Highlights the distribution of task durations, revealing potential data skew if some tasks are much slower.
    - Identifies gaps or overlaps in task execution, which may point to issues like resource contention or executor failures.
    - Helps correlate task delays with locality issues or shuffle overhead.



6. Aggregated Metrics by Executors

This section aggregates metrics at the executor level to assess performance during a stage.

- **Key Metrics**:
    1. **Executor ID**:
        - Identifies the executor running the tasks.
        - Clicking the ID provides more details about the executor.
    2. **Task Time**:
        - Total time spent by tasks executed on the executor.
        - **Insights**: High task times might indicate uneven task distribution or slower hardware.
    3. **Shuffle Read/Write**:
        - **Shuffle Read**:
            - Amount of data read by the executor during shuffles.
            - High values may signal large shuffle stages or data skew.
        - **Shuffle Write**:
            - Data written to shuffle storage.
            - Large values suggest expensive operations like wide transformations.
    4. **Input Bytes**:
        - Amount of input data read by the executor (e.g., from HDFS).
        - Uneven values may indicate partitioning issues or an unbalanced workload.
    5. **Output Bytes**:
        - Data written to the next stage or output storage by the executor.
    6. **GC Time**:
        - Time spent in garbage collection (GC).
        - High GC times may indicate memory pressure or inefficient usage.
    7. **Peak Memory Usage**:
        - Maximum memory used by the executor during task execution.
        - Useful for diagnosing out-of-memory (OOM) errors or memory-intensive workloads.

7. Aggregated Metrics by Tasks

This section provides metrics aggregated at the task level, offering insights into task performance within a stage.

Key Metrics:

1. **Task ID**
    - A unique identifier for each task in the stage.
2. **Duration**
    - Total time taken by the task, including:
        - Execution time.
        - Time spent waiting for resources (e.g., scheduler delays).
    - **Insights**:
        - Longer durations might indicate:
            - **Data skew**: Uneven distribution of data across tasks.
            - **Stragglers**: Slower tasks compared to others.
            - **Network delays**: Common in shuffle-heavy stages.
3. **Task Deserialization Time**
    - Time spent deserializing the task on the executor.
    - **High deserialization times** may signal:
        - Inefficient serialization.
        - Large task payloads or closures.

4. **GC Time**
    - Time spent in garbage collection (GC) during the task.
    - **High GC times** might indicate:
        - Excessive object creation.
        - Inefficient memory usage.
5. **Shuffle Read/Write**
    - **Shuffle Read**:
        - Data read during shuffles by the task.
        - **High values** could point to uneven partitioning.
    - **Shuffle Write**:
        - Data written to shuffle storage by the task.
        - **Large values** suggest costly operations or wide transformations.
6. **Input/Output Bytes**
    - **Input Bytes**: Data read by the task from storage.
    - **Output Bytes**: Data written by the task to downstream stages or final output.
7. **Locality Level**
    - The proximity of the task to the data it processes (e.g., process-local, node-local, rack-local, or any).
    - **Poor locality** may cause delays due to increased data transfer.
8. **Task Result Fetch Time**
    - Time taken to fetch task results from the executor back to the driver.

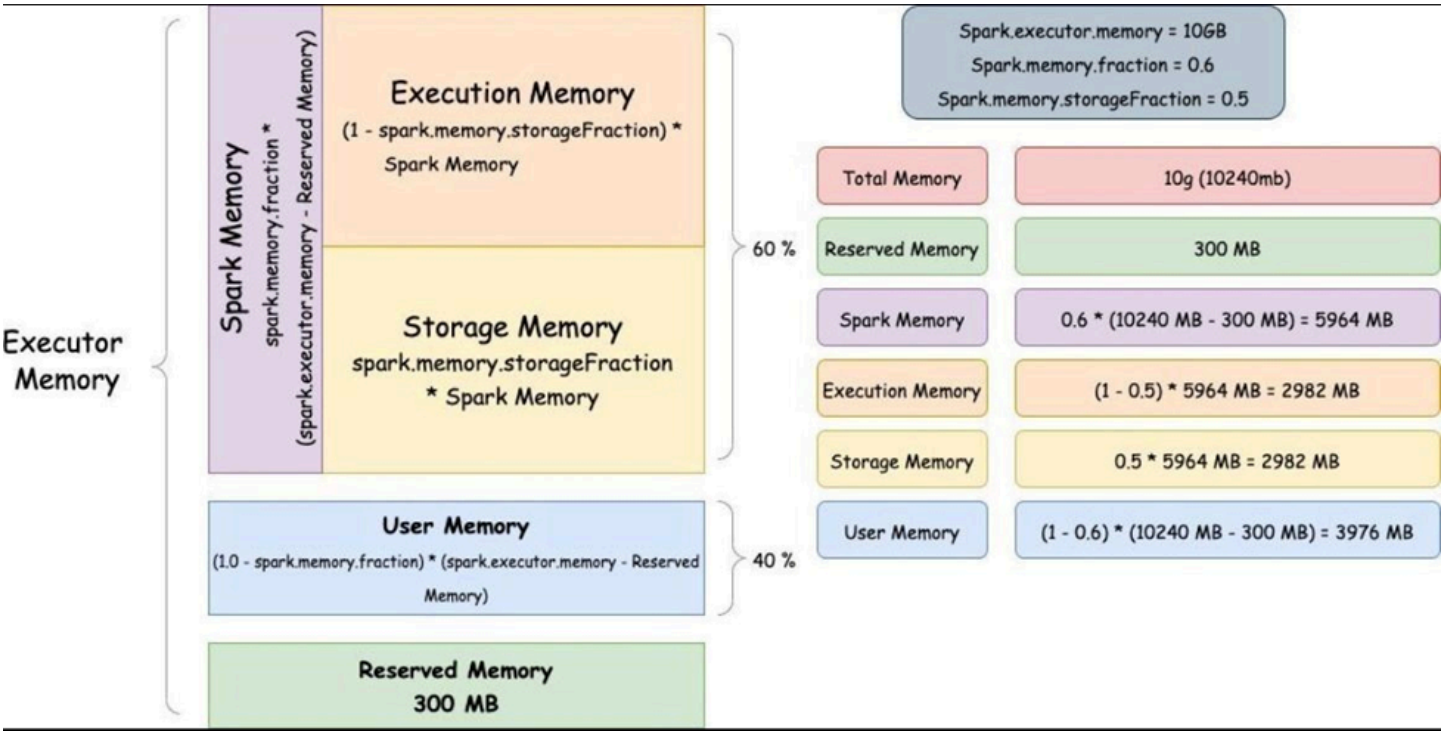
How to Use These Metrics:

1. **Identifying Executor Bottlenecks**:
    - **Uneven Task Distribution**:
        - Look for executors with significantly higher task times or shuffle read/write values.
        - Address with data repartitioning or skew optimization.
    - **Memory Issues**:
        - High GC times or peak memory usage suggest memory pressure.
        - Adjust executor memory settings for better management.
2. **Diagnosing Task Stragglers**:
    - **Outliers in Task Duration**:
        - Identify tasks taking much longer than others.
        - Causes may include data skew, poor locality, or resource contention.
    - **Shuffle Performance**:
        - High shuffle read/write values indicate expensive operations, often from wide transformations.
    - **Deserialization Delays**:
        - Tasks with long deserialization times may have inefficient serialization or excessively large closures.

Debugging issues in Spark - Out of memory - Nov 27

- **Heap Memory**
- **Off-Heap Memory**
- **Memory Manager**
- **GC (Garbage Collection)**
- **Memory Tuning**
- **Common OOM Errors and Resolutions**
- **Best Practices to Avoid OOM Errors**
- **Case 1: OOM Due to Large Shuffle Operations**
- **Case 2: OOM Due to Skewed Data**
- **Case 3: OOM Due to Too Many Concurrent Tasks (High Parallelism)**

One of the most common errors you will see while working with Spark applications is OOM (Out of Memory) errors. To understand and fix these errors, let's first learn how memory management works in Spark.



Heap Memory

Heap memory is the portion of memory allocated by the JVM for objects and data structures. Spark uses JVM heap memory for different purposes, and it is split into two main parts:

1. **Storage Memory:**
    - Stores cached data like RDDs, DataFrames, and Datasets.
    - Managed by Spark's `MemoryManager`.
    - Uses part of the total heap memory, controlled by the `spark.memory.storageFraction` property.
    - When storage memory exceeds its limit, data is removed using the Least Recently Used (LRU) algorithm.

2. **Execution Memory:**
    - Used for task execution, shuffle operations, and temporary data structures (e.g., intermediate results, serialized data).
    - Shares the heap with storage memory and competes for resources if needed.
    - Controlled by the `spark.memory.fraction` property.

By default, Spark splits heap memory into storage and execution memory using a fixed ratio:

- `spark.memory.fraction` = 0.6
- `spark.memory.storageFraction` = 0.5

You can adjust these values based on your workload needs.

Off-Heap Memory

Off-heap memory is memory located outside the JVM's heap space. It helps reduce Garbage Collection (GC) overhead by keeping certain data and cached structures outside the JVM heap.

1. **Benefits of Off-Heap Memory:**
    - Reduces GC overhead since it operates outside the JVM heap.
    - Improves performance for caching and shuffle operations.
2. **Configuration:**
    - Enable off-heap memory: Set `spark.memory.offHeap.enabled` to `true`.
    - Specify the size of off-heap memory: Use `spark.memory.offHeap.size` (in bytes).

Off-heap memory is especially useful for memory-heavy applications where reducing GC pauses boosts performance.

Memory Manager

The **MemoryManager** in Spark controls memory allocation and tracks memory usage across tasks. It ensures efficient usage and prevents components from exceeding memory limits.

The MemoryManager divides memory into three main categories:

1. **Storage Memory:**
    - Used for caching data, like RDDs, DataFrames, and Datasets.
    - Can use both heap and off-heap memory.
    - Cached data is removed using the Least Recently Used (LRU) policy when memory limits are reached.
2. **Execution Memory:**
    - Allocated for temporary data during task execution.
    - Includes space for serialized task outputs, shuffle data, and intermediate results.
    - Dynamically shared with storage memory and managed by the MemoryManager.
3. **User Memory:**
    - Reserved for user-defined structures, accumulators, and broadcast variables.
    - Separate from storage and execution memory.

GC (Garbage Collection)

GC is responsible for freeing memory used by objects that are no longer needed in the JVM heap. However, too much GC activity can slow down performance.

1. **Impact on Spark:**
   - Long GC pauses can delay task execution and create cluster resource contention.
   - Proper memory tuning helps reduce GC overhead.
2. **Strategies to Minimize GC Impact:**
   - Use off-heap memory for large datasets to avoid JVM GC.
   - Adjust heap size (`spark.executor.memory`) and GC settings (`spark.executor.extraJavaOptions`).

Memory Tuning

Proper memory tuning ensures Spark applications run efficiently. Key parameters to configure include:

1. **Heap Memory Settings:**
   - `spark.executor.memory`: Total heap memory for each executor.
   - `spark.driver.memory`: Total heap memory for the driver.
2. **Storage and Execution Ratios:**
   - `spark.memory.fraction`: Portion of JVM heap reserved for storage and execution.
   - `spark.memory.storageFraction`: Portion of `spark.memory.fraction` allocated to storage memory.
3. **Off-Heap Settings:**
   - `spark.memory.offHeap.enabled`: Enable/disable off-heap memory.
   - `spark.memory.offHeap.size`: Size of off-heap memory.
4. **Memory Overhead:**
   - `spark.executor.memoryOverhead`: Additional memory for Spark's internal operations.
   - `spark.driver.memoryOverhead`: Overhead memory for the driver.
5. **Garbage Collection Options:**
   - Tweak GC settings using `spark.executor.extraJavaOptions` (e.g., `-XX:+UseG1GC`).

Common OOM Errors and Resolutions

1. **`java.lang.OutOfMemoryError: Java heap space`**
   - **Cause:**
     - Insufficient heap memory for the driver or executor.
     - Objects or data structures are too large for the allocated heap.
   - **Resolution:**
     - Increase heap size (`spark.executor.memory` or `spark.driver.memory`).
     - Optimize the application to use less memory.
     - Use off-heap memory (`spark.memory.offHeap.enabled`).
2. **`java.lang.OutOfMemoryError: GC overhead limit exceeded`**
   - **Cause:**
     - Too much time spent on GC with little memory freed.
     - Small heap memory or excessive fragmentation.
   - **Resolution:**
     - Increase heap memory (`spark.executor.memory`).
     - Use an efficient GC algorithm (e.g., G1GC: `-XX:+UseG1GC`).
     - Optimize memory usage by reducing dataset size or avoiding large objects.

3. **`java.lang.OutOfMemoryError: Direct buffer memory`**
   - **Cause:**
     - Not enough off-heap memory for shuffle or caching tasks.
     - NIO (Non-blocking I/O) buffers exceed the configured size.
   - **Resolution:**
     - Increase off-heap memory size (`spark.memory.offHeap.size`).
     - Enable off-heap memory (`spark.memory.offHeap.enabled=true`).
4. **`java.lang.OutOfMemoryError: Metaspace`**
   - **Cause:**
     - Metaspace (used for class metadata) is exhausted.
     - Happens when too many classes are loaded, often due to excessive JAR dependencies.
   - **Resolution:**
     - Increase Metaspace size (`-XX:MaxMetaspaceSize`).
     - Reduce the number of loaded classes by minimizing dependencies.
5. **`ExecutorLostFailure (executor <ID>): Container killed by YARN for exceeding memory limits`**
   - **Cause:**
     - Executor exceeds allocated physical memory or overhead.
   - **Resolution:**
     - Increase executor memory (`spark.executor.memory`).
     - Adjust memory overhead (`spark.executor.memoryOverhead`).
6. **`Driver memory overhead exceeded`**
   - **Cause:**
     - Driver exceeds its allocated memory or overhead.
   - **Resolution:**
     - Increase driver memory (`spark.driver.memory`).
     - Adjust driver memory overhead (`spark.driver.memoryOverhead`).
7. **`java.lang.OutOfMemoryError: Unable to create new native thread`**
   - **Cause:**
     - System runs out of threads, often due to too many parallel tasks or low system resources.
   - **Resolution:**
     - Reduce concurrent tasks (`spark.executor.cores` or `spark.task.cpus`).
     - Ensure the OS allows sufficient threads (check `ulimit -u` on Linux).
8. **`org.apache.spark.shuffle.FetchFailedException`**
   - **Cause:**
     - Not enough memory for shuffle data during large joins, groupBy, or aggregations.
   - **Resolution:**
     - Increase shuffle memory (`spark.shuffle.memoryFraction` or `spark.memory.fraction`).
     - Optimize data structures or repartition to reduce shuffle data size.
9. **`java.lang.OutOfMemoryError: Requested array size exceeds VM limit`**
   - **Cause:**
     - Application tries to create an array larger than the JVM limit.
   - **Resolution:**
     - Split large datasets or use smaller partitions.
     - Ensure adequate executor memory is available.

Best Practices to Avoid OOM Errors

1. **Partitioning:**
   - Use appropriate partition sizes with `repartition()` or `coalesce()`.
   - Avoid processing overly large partitions to prevent memory overload.
2. **Memory Configuration:**
   - Allocate sufficient memory for executors, drivers, and overhead.
   - Enable and configure off-heap memory when necessary.
3. **Garbage Collection:**
   - Use optimized GC settings (e.g., G1GC) for better handling of large heaps.
4. **Caching and Persisting:**
   - Avoid caching datasets unless necessary.
   - Choose the right storage level for caching, such as `MEMORY_AND_DISK`.
5. **Data Optimization:**
   - Filter out unnecessary data as early as possible in the processing pipeline.
   - Use efficient serialization formats like Kryo to minimize memory usage.
6. **Shuffle Optimizations:**
   - Minimize wide transformations (e.g., `groupBy`, `join`) when possible.
   - Properly configure `spark.sql.shuffle.partitions` and `spark.default.parallelism` to balance the workload.

Case 1: OOM Due to Large Shuffle Operations

Scenario:
An application performs a wide transformation (e.g., `groupBy()`, `join()`, or `reduceByKey()`) on a dataset with hundreds of gigabytes of data. The shuffle stage fails because the default partitioning creates a few partitions that are too large to fit into memory on the executors, causing an OutOfMemoryError (OOM).

Root Cause:
By default, Spark uses 200 partitions for shuffle operations (`spark.sql.shuffle.partitions = 200`). For very large datasets, this can lead to:

- Some partitions becoming very large, especially if the data is unevenly distributed across keys.
- Executors running out of memory while processing these large partitions.

Solution:
Adjust `spark.sql.shuffle.partitions` to increase the number of partitions and distribute data more evenly, reducing memory pressure per partition.

Steps to Fix:

1. Analyze the dataset size and determine the target partition size.
   - Example: For a 500 GB dataset and a target partition size of 128 MB:

$$\text{Number of Partitions} = \frac{500\,\text{GB}}{128\,\text{MB}} = 4000$$

2. Set `spark.sql.shuffle.partitions` to a higher value (e.g., 4000).
   `spark.conf.set("spark.sql.shuffle.partitions", "4000")`

3. Run the application and monitor memory usage during shuffle stages.

Result:
Increasing partitions balances data across the cluster, reduces the memory load per executor, and avoids OOM errors during shuffle-heavy operations.

Case 2: OOM Due to Skewed Data

Scenario:
An application performs a join operation between two datasets, but one dataset has highly skewed keys (e.g., one key accounts for 90% of the records). During the shuffle stage, Spark attempts to allocate all records for the skewed key into a single partition, causing OOM errors.

Root Cause:
Skewed data results in:

- Imbalanced partition sizes, where some partitions are disproportionately large.
- Excessive memory consumption for large partitions, even if `spark.sql.shuffle.partitions` is set to a high value.

Solution:
Use Spark's data skew handling techniques to manage partition size and memory usage effectively.

Techniques to Mitigate Data Skew:

1. **Enable Adaptive Query Execution (AQE):**
    - Property: `spark.sql.adaptive.skewJoin.enabled=true`
    - **Effect:**
        - Spark dynamically detects and splits large partitions for skewed keys into smaller chunks during execution.
    - **Result:**
        - This reduces memory load and prevents OOM errors.
2. **Broadcast Smaller Dataset:**
    - Property: `spark.sql.autoBroadcastJoinThreshold`
    - **Effect:**
        - If the smaller dataset can fit into memory, Spark broadcasts it to all executors, avoiding shuffle entirely.
    - **Default:** 10 MB (adjust if needed, e.g., `spark.sql.autoBroadcastJoinThreshold=100MB`)
    - **Result:**
        - Removes shuffle stages, significantly reducing memory pressure.
3. **Salting the Skewed Key:**
    - **Process:**
        - Add a random "salt" to the skewed key to distribute records more evenly across partitions.

Example:

```
data1 = data1.withColumn("salt", F.rand())
data1 = data1.withColumn("new_key", F.concat(F.col("key"), F.col("salt")))
data2 = data2.withColumn("new_key", F.concat(F.col("key"), F.col("salt")))
joined_data = data1.join(data2, "new_key")
```

    - **Result:**
        - Redistributes records of the skewed key, balancing partition sizes.

4. **Reduce Shuffle Data Size in Flight:**
    - Property: `spark.reducer.maxSizeInFlight=16MB`
    - **Effect:**
        - Lowers the maximum size of shuffle data transferred between executors.
        - Default: 48 MB (reduce to limit memory usage).
    - **Result:**
        - Prevents memory overload during shuffle operations.

Outcome:

- Splitting skewed keys into smaller partitions using AQE reduces memory pressure.
- Broadcasting smaller datasets avoids shuffle altogether.
- Salting redistributes data evenly across partitions, preventing OOM errors.

Case 3: OOM Due to Too Many Concurrent Tasks (High Parallelism)

Scenario:
An application processes a large dataset with high parallelism, using many cores per executor (e.g., 8 cores per executor). Each core runs a task, and all tasks share the executor's memory. When too many tasks run concurrently, they consume more memory than the executor can provide, causing OutOfMemoryError (OOM).

Root Cause:

- Each task uses part of the executor's memory for temporary data, intermediate results, and shuffle operations.
- High concurrency leads to memory overcommitment.
- Memory is divided into execution memory, storage memory, and overhead, leaving insufficient space for all tasks.

Solution: Reduce Concurrent Tasks by Adjusting Cores per Executor

1. **Lower `spark.executor.cores`:**
    - Reduce the number of cores per executor to limit concurrent tasks.
    - Example:
        - If executor memory is 8 GB and `spark.executor.cores` is 8, reduce it to 4 to halve the concurrent tasks.
2. **Recalculate Executors and Cores:**
    - Maintain total cluster capacity by increasing the number of executors when reducing cores.
    - Example:
        - Before: 10 executors × 8 cores = **80 total cores**.
        - After: 20 executors × 4 cores = **80 total cores**.

Result:

- Fewer concurrent tasks reduce memory contention.
- Each task gets more memory to operate, preventing OOM errors.
- Stability improves with better memory management.

Example:

- **Before:**
  - Executor memory: 8 GB
  - Executor cores: 8
  - Each task gets approximately: [memory per task]

$$\text{Memory per task} = \frac{\text{Executor Memory - Memory Overhead}}{\text{Cores}} \approx \frac{8\,\text{GB} - 1\,\text{GB}}{8} = 0.875\,\text{GB/task}.$$

  - Tasks fail due to insufficient memory.
- **After:**
  - Executor cores reduced to 4

$$\text{Memory per task} = \frac{8\,\text{GB - 1 GB}}{4} = 1.75\,\text{GB/task}.$$

  - Tasks succeed with increased memory per task.

Benefits:

1. **Reduced Memory Contention:**
   - Fewer tasks share the same memory, lowering the risk of memory exhaustion.
2. **Improved Task Throughput:**
   - With fewer tasks running concurrently, each task completes faster due to less memory contention and overhead.
3. **Better GC Performance:**
   - Reducing the number of concurrent tasks decreases the frequency and intensity of garbage collection.

Other Properties to Tune:

1. **spark.executor.memoryOverhead:**
   - Increase memory overhead if tasks require additional memory for shuffle operations or large intermediate data.
2. **spark.dynamicAllocation.enabled:**
   - Enable dynamic allocation to scale the number of executors based on workload:
   - `spark.dynamicAllocation.enabled=true`

When to Use This Solution:

- If your application processes large datasets with complex transformations (e.g., joins, aggregations).
- If monitoring tools (e.g., Spark UI or logs) indicate memory contention due to too many concurrent tasks.
- If you observe frequent OOM errors even when executor memory and shuffle partitions are optimally configured.

By reducing the number of cores per executor, you balance memory utilization and task execution, preventing OOM errors caused by excessive parallelism.

Debugging issues in Spark - Contd - Nov 29

- **What is SLA in Data Pipelines?**
- **Why is SLA Important in Data Pipelines?**
- **Causes of Delays in Task Execution - Data Skewness**
- **How to Solve Data Skewness**
- **Parallelism in Spark**
- **Tune the Parameter**
- **Experiment and Benchmark**

In the last lecture, we learned how to debug out-of-memory errors in Spark. Another common issue you might face while working with Spark applications is slowness. Before diving into slowness in Spark, let's first understand why slowness matters in Spark applications.

What is SLA in Data Pipelines?

SLA stands for **Service Level Agreement**. It is a documented agreement between a service provider and a client, defining the expected performance and service levels. In data pipelines, an SLA usually includes:

1. **Timeliness:** The deadline for delivering or processing data (e.g., a daily ETL job must finish by 6 AM).
2. **Accuracy:** Ensuring the data meets specific quality or correctness standards.
3. **Availability:** The uptime or reliability of the pipeline or its components.
4. **Throughput:** The amount of data the pipeline must process within a given time.
5. **Error Tolerance:** The acceptable limits for failures or delays (e.g., retry policies for temporary issues).

Why is SLA Important in Data Pipelines?

SLAs are essential for data pipelines because they:

1. **Ensure Reliability**
   A well-defined SLA guarantees timely data delivery, ensuring downstream systems or stakeholders can work without interruptions. For example, late data might delay reports or decision-making processes.
2. **Set Clear Expectations**
   SLAs define what is expected from the data pipeline, helping providers and consumers align on goals like delivery time, quality, and reliability.
3. **Facilitate Monitoring and Accountability**
   SLAs help teams establish **key performance indicators (KPIs)** to monitor the pipeline's health. If a pipeline repeatedly fails to meet its SLA, it highlights areas for improvement.
4. **Improve Operational Efficiency**
   By setting deadlines and failure limits, SLAs push teams to optimize workflows, enhance infrastructure, and handle errors more effectively.
5. **Mitigate Risks**
   SLAs act as a safety net to minimize the business impact of data delays, errors, or downtime. For instance, missing an SLA in a real-time fraud detection system could cause financial losses.
6. **Build Trust**
   Consistently meeting SLAs builds confidence between the service provider and consumers, ensuring trust in the pipeline's performance.
7. **Aid Compliance**
   For pipelines in regulated environments, SLAs help meet compliance standards like GDPR or HIPAA, which may require specific timelines for data delivery or accuracy levels.

Understanding and maintaining SLAs is crucial to ensuring smooth operations in Spark applications and data pipelines.

Causes of Delays in Task Execution: Data Skewness

One common issue that can delay task execution in distributed systems is **data skewness**. This happens when data is not evenly distributed across partitions. In a well-balanced system, all nodes should have roughly the same amount of data to process. However, data skewness can occur due to:

- **Uneven grouping keys** in operations like `GROUP BY`, `JOIN`, or `REDUCE`.
- **Imbalanced input datasets**.
- **Poor partitioning logic**.

Example: Imagine processing sales data grouped by region. If one region (e.g., "North America") contains 80% of the data, the node responsible for "North America" will do most of the work, while other nodes process only a small amount of data.

How Does Data Skewness Affect Task Performance?

Data skewness can create several Problems in distributed systems:

1. **Uneven Load Distribution**
    - Some tasks process much more data than others, creating **slow tasks**.
    - **Example:** A `reduce` task handling a key with millions of records, while other tasks process only a few records.
2. **Slow Tasks**
    - Nodes handling larger partitions take longer to finish, which delays the entire job.
3. **High Network Overhead**
    - During shuffle operations, large partitions generate excessive network traffic, slowing down the system.
4. **Wasted Resources**
    - Some nodes remain idle, waiting for slow tasks to complete, leading to inefficient resource usage.

Understanding and addressing data skewness is key to improving performance in distributed systems.

How to Solve Data Skewness?

Here are strategies to reduce or fix data skewness:

A. Pre-Processing and Data Balancing

1. **Analyze Data Distribution**
    - Use tools like `Spark's DataFrame.describe()` or `Hive's ANALYZE TABLE` to find skewed keys.
    - **Example:** Check which keys dominate in a `GROUP BY`.
2. **Filter or Sample Data**
    - Remove or sample skewed data during development and testing to achieve a more balanced dataset.

B. Partitioning Strategies

1. **Custom Partitioning**
   - Use a custom partitioning function to evenly distribute data.
   - **Example:** Hash keys intelligently to balance the load.
2. **Salting Skewed Keys**
   - Add random suffixes (salt) to skewed keys to spread them across partitions.
   - **Example:**
     - Original key: `"North America"`
     - Salted keys: `"North America_1"`, `"North America_2"`.
     - Remove the salt during aggregation to combine results.

C. Algorithmic Adjustments

1. **Skewed Join Optimization**
   - Use **Broadcast Joins** for small datasets:
     - Broadcast the smaller dataset to all nodes to avoid expensive shuffles.

     Example in Spark:
     ```
     df_large.join(broadcast(df_small), "key")
     ```

2. **Skew Join Hints**
   - Provide hints to the framework for handling skewed joins.

     Example in SQL:
     ```
     SELECT /*+ SKEW('key') */ *
     FROM table1
     JOIN table2
     ON table1.key = table2.key;
     ```

D. Data Sampling and Bucketing

1. **Data Sampling**
   - Test performance using a representative subset of data to minimize skewness effects.
2. **Bucketing in Hive or Spark**
   - Divide the dataset into smaller, evenly distributed buckets based on skewed keys.

E. Configuration Tuning

1. **Increase Parallelism**
   - Adjust the number of partitions or reduce tasks to balance the workload.

Example in Spark:
```
df.repartition(100)
```

   -
2. **Memory and Shuffle Settings**

- Increase shuffle partition size or executor memory to handle larger tasks.

These strategies can help manage data skewness, ensuring smoother and faster task execution.

Real-World Example: Handling Skewed Clickstream Data

Scenario:
You are processing clickstream data using a `GROUP BY` on `user_id`. Popular users (e.g., influencers) dominate the dataset, creating skewness.

Solution:

1. Analyze the distribution of `user_id` to identify skewed keys.
2. Use **salting** to spread high-volume users across partitions.
   - For example, append random suffixes to `user_id` to balance the load.
3. Combine salting with **broadcast joins** for small reference data to avoid costly shuffles.
4. Optimize **shuffle** and **executor memory settings** for better performance.

Parallelism in Spark

Parallelism in Spark can be managed and improved by tuning the `spark.sql.shuffle.partitions` configuration. This parameter controls the number of partitions used during shuffle operations, such as joins and aggregations, which involve data movement across nodes.

Why Is `spark.sql.shuffle.partitions` Important?

By default, Spark sets `spark.sql.shuffle.partitions = 200`. While this works for small to medium datasets, it can lead to:

1. **Underutilization of Resources**
   - On large clusters, 200 partitions may not fully use all available nodes and cores.
2. **Skewed Task Durations**
   - If partitions hold too much data, some tasks take significantly longer to finish, causing **slow tasks** that slow down the job.

Benefits of Adjusting `spark.sql.shuffle.partitions`:

1. **Balanced Data Distribution:**
   - Divides data into an appropriate number of partitions for the dataset size.
2. **Manageable Task Sizes:**
   - Ensures each executor processes a reasonable chunk of data.
3. **Optimized Task Execution:**
   - Reduces task durations and minimizes stragglers, speeding up job completion.

Adjusting parallelism and partitioning settings can significantly improve the efficiency of Spark jobs, especially when handling skewed datasets.

How to Solve Task Slowness Using `spark.sql.shuffle.partitions`

Step 1: Understand Your Dataset

Before tuning `spark.sql.shuffle.partitions`, analyze your dataset and workload:

- **Dataset Size:** Larger datasets need more partitions.
- **Cluster Resources:** Ensure the number of partitions balances with your cluster's cores and executors.

Step 2: Tune the Parameter

Set the number of shuffle partitions based on your workload and cluster size.

Code Example:

```
from pyspark.sql import SparkSession

# Create SparkSession
spark = SparkSession.builder \
    .appName("Increase Parallelism") \
    .config("spark.sql.shuffle.partitions", "500") \  # Increase shuffle
partitions
    .getOrCreate()

# Example transformation
df = spark.read.csv("large_dataset.csv")
# Perform a shuffle operation (e.g., GROUP BY)
result = df.groupBy("column_name").count()
result.show()
```

Step 3: Experiment and Benchmark

Test your pipeline with different `spark.sql.shuffle.partitions` values:

- **Small Workload:** Use fewer partitions (e.g., 100–200).
- **Large Workload:** Use more partitions (e.g., 500–1000+).

Use Spark's Web UI to Monitor:

1. **Task Durations:** Check if tasks complete in reasonable time.
2. **Executor Utilization:** Ensure all cores are active.

Benefits of Tuning `spark.sql.shuffle.partitions`

1. **Increased Parallelism:**
   - More partitions create more tasks, which can run in parallel.
2. **Reduced Task Slowness:**
   - Smaller partitions mean faster task execution as each processes less data.
3. **Better Resource Utilization:**
   - Ensures CPU and memory are fully utilized.
4. **Improved Scalability:**
   - Allows efficient handling of larger datasets.

Best Practices for Setting `spark.sql.shuffle.partitions`

1. **Align with Cluster Size:**
   - Set partitions ≈ 2–4 × total cores in the cluster.
2. **Avoid Too Many Partitions:**
   - Too many small tasks increase coordination and shuffle overhead.
3. **Monitor Shuffle Metrics:**
   - Use Spark UI to check shuffle read/write sizes and ensure even data distribution.
4. **Enable Adaptive Execution (If Available):**

For Spark 3.0+, dynamically adjust partitions with Adaptive Query Execution (AQE):
`spark.conf.set("spark.sql.adaptive.enabled", "true")`

Real-World Example

Scenario:
You're processing a **10 TB dataset** with a `GROUP BY` operation. The default `200 partitions` results in:

- Each partition handling **50 GB of data**.
- Slow tasks delaying the pipeline.

Solution:
Increase `spark.sql.shuffle.partitions` to `1000`:

- Each partition handles **10 GB of data**.
- Tasks run in parallel across all cores, minimizing delays.

Using Dynamic Allocation with Minimum and Maximum Executors to Solve Application Slowness

Dynamic Allocation allows Spark to adjust the number of executors at runtime based on workload requirements, providing flexibility and resource efficiency.

Key Features of Dynamic Allocation

1. **Scale Up:** Add more executors during high workloads.
2. **Scale Down:** Release idle executors when they're no longer needed, saving resources.

Configuring Minimum and Maximum Executors

1. **`spark.dynamicAllocation.minExecutors`:**
   - Sets the minimum number of executors that the application will always maintain.
   - Ensures baseline parallelism to avoid resource starvation.
2. **`spark.dynamicAllocation.maxExecutors`:**
   - Sets the maximum number of executors the application can scale up to.
   - Prevents over-provisioning, avoiding unnecessary cluster load.

How Minimum and Maximum Executors Address Slowness

A. Prevent Under-Provisioning

- Without enough executors (`minExecutors`), tasks may run serially or with limited resources, causing slow execution.
- Setting an appropriate `minExecutors` ensures sufficient resources for smaller workloads.

B. Handle Sudden Workload Spikes

- For high workloads, the application dynamically scales up to `maxExecutors`, enabling faster processing with additional resources.
- This reduces bottlenecks caused by large tasks or heavy shuffles.

C. Optimize Resource Utilization

- When workloads decrease, Spark scales down to the `minExecutors`, freeing resources for other applications.
- Ensures efficient use of the cluster.

D. Eliminate Idle Executor Time

- Idle executors are terminated automatically when not required, improving application efficiency and reducing overall execution time.

Example Configuration

```
from pyspark.sql import SparkSession

# Create SparkSession with dynamic allocation settings
spark = SparkSession.builder \
    .appName("Dynamic Allocation Example") \
    .config("spark.dynamicAllocation.enabled", "true") \
    .config("spark.dynamicAllocation.minExecutors", "2") \  # Minimum executors
    .config("spark.dynamicAllocation.maxExecutors", "10") \  # Maximum executors
    .config("spark.executor.idleTimeout", "60s") \  # Terminate idle executors
after 60 seconds
    .getOrCreate()
```

```
# Example transformation
df = spark.read.csv("dataset.csv")
result = df.groupBy("column_name").count()
result.show()
```

Benefits of Dynamic Allocation

1. **Adaptability:** Adjust resources dynamically based on workload.
2. **Faster Processing:** Scale up resources for large tasks or shuffles.
3. **Resource Efficiency:** Release idle executors to avoid wastage.
4. **Cost Optimization:** Minimize resource consumption when workloads decrease.

By carefully setting `minExecutors` and `maxExecutors`, you ensure your Spark application runs efficiently under varying workloads.

Configuring Minimum and Maximum Executors in Spark

Here's how you can configure the number of executors for dynamic allocation in Spark.

Example Configuration

```
from pyspark.sql import SparkSession

# Create SparkSession with dynamic allocation settings
spark = SparkSession.builder \
    .appName("DynamicAllocationExample") \
    .config("spark.dynamicAllocation.enabled", "true") \  # Enable dynamic
allocation
    .config("spark.dynamicAllocation.minExecutors", "5") \  # Minimum number of
executors
    .config("spark.dynamicAllocation.maxExecutors", "50") \  # Maximum number of
executors
    .config("spark.dynamicAllocation.initialExecutors", "10") \  # Initial
executors
    .config("spark.executor.cores", "4") \  # Cores per executor
    .config("spark.executor.memory", "8g") \  # Memory per executor
    .getOrCreate()
```

Explanation of Parameters

1. **`minExecutors=5`:**
    - Ensures the application always has at least 5 executors, even for light workloads, providing a baseline for processing.
2. **`maxExecutors=50`:**
    - Allows the application to scale up to 50 executors to handle heavy workloads effectively.
3. **`initialExecutors=10`:**

- Starts the application with 10 executors and adjusts dynamically based on workload requirements.

Understanding Scheduler Delays

What Are Scheduler Delays?

Scheduler delays occur when tasks wait in a queue before being assigned resources for execution.

Causes of Scheduler Delays:

1. **Insufficient Resources:**
   - If there aren't enough executors or CPU cores, tasks must wait for resources to become available.
2. **Imbalanced Resource Distribution:**
   - Overloaded executors handle more tasks while others remain underutilized.
3. **High Task Count:**
   - A large number of small tasks increases scheduling overhead and coordination time.
4. **Resource Contention:**
   - Multiple jobs or applications compete for cluster resources, causing delays.

Impact of Scheduler Delays:

- Increases overall job latency.
- Reduces cluster throughput and efficiency.

Understanding Scanned Data

What Is Scanned Data?

This refers to the total volume of data read from the storage layer (e.g., HDFS, S3) by Spark for processing.

Causes of Excessive Scanning and Delays:

1. **Large Input Data Size:**
   - Large datasets generate more partitions, increasing the number of tasks and scheduling overhead.
2. **Data Skew:**
   - Uneven distribution of data across partitions creates slow task, delaying subsequent stages.
3. **Inefficient Data Pruning:**
   - Missing optimizations like partition pruning or predicate pushdown results in unnecessary data being scanned.
4. **Slow Storage Systems:**
   - A storage layer that cannot serve data fast enough delays task execution.

Impact of Excessive Scanned Data:

- Slower task execution.
- Increased resource contention.
- Higher scheduling latency for subsequent tasks.

Key Takeaways

1. Configure **minimum and maximum executors** to ensure efficient resource allocation and scaling.
2. Monitor and optimize **scheduler delays** by balancing resources and reducing small tasks.
3. Reduce **scanned data** by using optimizations like data pruning and addressing data skew.
4. Use tools like the Spark UI to monitor performance and fine-tune configurations.

Failures and Reattempts in Spark

Failures and task reattempts are common in Spark but can cause delays and inefficiencies. Here's a breakdown of the issue and its impact.

What It Is

When a task fails, Spark retries the task, adding scheduling overhead and delaying overall job execution.

Causes of Scheduler Delays

1. **Executor Failures:**
    - When an executor crashes or is lost due to hardware or network issues, all tasks running on it must be rescheduled.
2. **Memory Overflows:**
    - Tasks handling large partitions may exceed memory limits, leading to failures and retries.
3. **Shuffle Errors:**
    - Failures during shuffle operations (e.g., issues reading from or writing to shuffle files) trigger task reattempts.
4. **Transient Network Issues:**
    - Temporary network interruptions during data transfers between nodes can result in retries.
5. **Data Skew:**
    - Uneven data distribution causes some partitions to be overloaded, leading to timeouts or task failures.
6. **Checkpoint or Savepoint Issues:**
    - If tasks rely on checkpointed or savepoint data that becomes unavailable, they fail and must restart from earlier stages.

Impact of Task Failures and Retries

1. **Delays in Task Completion:**
    - Frequent retries extend the time required to finish a job.
2. **Increased Resource Consumption:**
    - Retrying tasks consumes additional CPU, memory, and network resources.
3. **Cluster Bottlenecks:**
    - Failures can cause resource contention, delaying other tasks or jobs.

Key Insights

To minimize delays caused by failures:

- Optimize memory usage to prevent overflows.
- Address data skew with techniques like salting or repartitioning.
- Monitor shuffle operations and ensure reliable data transfer between nodes.
- Use fault-tolerant storage systems for checkpoint and savepoint data.

Proactive measures can reduce retries and improve overall job performance.

Advanced optimization techniques for Spark - Dec 02

- **Problems with Static Planning**
- **Inflexibility in Handling Runtime Variability**
- **Suboptimal Join Strategy Selection**
- **Solution: Adaptive Query Execution (AQE)**
- **Static Query Plan**
- **Dynamic Query Plan with AQE**
- **Performance Comparison: Before and After AQE**
- **Core AQE Configuration Options in Apache Spark**

In earlier lessons, we learned about different optimizations for Spark applications to make them faster and fix common errors. While these techniques work well, they have some limitations. One main issue is **static query execution**, which uses fixed query plans created during the query compilation stage. These plans are based on pre-execution estimates, which can often be wrong. This causes poor performance, especially for large or complex workloads.

Let's look at these limitations in more detail:

Problems with Static Planning

Inaccurate Query Planning

Static query plans use metadata and statistics about the data to decide how to run the query. This includes details like:

- **Table cardinality**: The number of rows in a dataset.
- **Data distribution**: Whether the data is evenly spread out or uneven.
- **Partition sizes**: The estimated size of each chunk of data.

Challenges with Estimation

1. **Data Skew**:
    - If one partition has much more data than others, tasks for that partition take longer, creating unbalanced workloads.
    - Example: Joining a sales dataset with a product catalog where one product (e.g., `product_id = 1001`) makes up 70% of the sales.
2. **Unbalanced Partitions**:
    - Partitions with uneven sizes can cause:
        - **Small partitions**: Wasting resources by running many tiny tasks.
        - **Large partitions**: Overloading tasks, increasing runtime, or causing failures.
3. **Incorrect Cardinality**:
    - Overestimating or underestimating row counts can:
        - Lead to bad decisions for join strategies or partitioning.
        - Cause unnecessary shuffling and I/O operations.

These limitations can hurt the performance of your Spark application, so understanding them is key to improving efficiency.

Inflexibility in Handling Runtime Variability

Static query planning is hard because it assumes that data characteristics will match the initial estimates during the entire execution. However, real-world scenarios often bring unexpected changes, such as:

1. **Small Data Processing**:
    - If the actual data size is much smaller than estimated, unnecessary shuffling or partitioning can occur, causing task scheduling overhead.
    - Example: A query assumes 200 partitions for a group-by operation, but the data only needs 10 partitions.
2. **Unpredicted Data Skew**:
    - Some partitions might have much more data than others, creating "hotspots."
    - Tasks handling these larger partitions become slow (stragglers (slow task)), delaying the whole job.
3. **Unbalanced Resource Allocation**:
    - When data distribution changes, static planning cannot adjust resources dynamically, leading to bottlenecks.

Result:

The static nature of query execution cannot adapt to:

- Changes in dataset sizes.
- Skewed or unpredictable data distributions.
- Real-time resource availability or workload changes.

Suboptimal Join Strategy Selection

Choosing the right join strategy is crucial for efficient query execution. This depends on accurate dataset size and distribution estimates. Common join strategies are:

1. **Broadcast Join**:
    - Copies a smaller dataset to all nodes, avoiding shuffles.
2. **Shuffle Join**:
    - Redistributes both datasets across the cluster, causing significant shuffle and I/O costs.

Static Execution Problem:

Join strategy selection happens before execution, based on estimated dataset sizes:

- If a dataset is **overestimated**, Spark may use a shuffle join unnecessarily.
- If a dataset is **underestimated**, Spark might attempt a broadcast join, leading to memory issues.

Example:

Joining two tables:

- `orders` with 100 million rows.
- `customers` with 10,000 rows.

If Spark fails to recognize `customers` as small enough for broadcasting, it will use an expensive shuffle join instead.

These limitations show how static planning struggles to adapt, affecting performance and efficiency.

==========

Detailed Explanation of application.properties

Visit

https://youtu.be/_MmSSunw_gM

Like | Comment | Subscribe

Wasted Resources

1. **Over-Provisioned Partitions**:
    - Static query planning often creates too many partitions for small datasets.
    - Example: For a dataset of 500 MB, planning for 200 shuffle partitions results in tiny tasks, wasting CPU cycles and adding unnecessary overhead.
2. **Long-Running Tasks for Skewed Partitions**:
    - When data is skewed, some tasks handle much larger partitions, becoming **stragglers**.
    - Slower tasks delay the entire stage, leaving other resources underused.
3. **Inflexible Resource Utilization**:
    - Static plans cannot reallocate resources dynamically.
    - This leads to:
        - Wasted compute power on tiny tasks.
        - Idle nodes waiting for straggler tasks to finish.

Real-World Scenario Example

Imagine a query on an e-commerce analytics platform:

```
SELECT category, COUNT(*)
FROM sales
GROUP BY category;
```

Challenges in Static Query Execution:

1. **Inaccurate Estimates**:
    - The size of the `sales` dataset may not account for skew in the `category` column (e.g., one category makes up 60% of sales).
2. **Excessive Shuffle Partitions**:
    - Over-partitioning creates hundreds of tiny shuffle files for small categories, wasting resources.
3. **Data Skew**:
    - A single category with most of the data may create one very large partition, causing a long-running task.
4. **Suboptimal Join (if joined with another table)**:
    - A static plan might select a shuffle join even if one table is small enough for a broadcast join.

Result

The query execution becomes inefficient:

- **Long-running tasks** due to skewed data.
- **Wasted resources** on over-partitioned small tasks.
- **Suboptimal join strategies**, leading to excessive shuffling and I/O.

Solution: Adaptive Query Execution (AQE)

AQE solves these problems by dynamically adjusting the query plan during execution to handle skewed data, optimize partitions, and improve resource utilization.

What is AQE (Adaptive Query Execution)?

Adaptive Query Execution (AQE) is a feature in Apache Spark that optimizes query plans dynamically during runtime. Unlike static query execution, where the plan is fixed before execution, AQE adjusts the plan based on real-time data and execution metrics. This helps Spark handle unpredictable data, like skewed partitions or changing data sizes, and improves performance.

Key Features of AQE

1. **Runtime Optimization**:
    - AQE changes the query execution plan during execution using real-time metrics, such as shuffle size and partition data.
2. **Stage-Level Execution**:
    - Spark splits a query into stages separated by shuffle boundaries.
    - AQE applies optimizations after each stage using the results of the previous stage.
3. **Incremental Plan Replacement**:
    - Spark refines and replaces the physical query plan during execution based on runtime statistics.
    - Completed stages are not re-executed.
4. **Seamless Integration**:
    - Fully compatible with SQL and DataFrame APIs.

Enabled with a simple configuration:

```
spark.sql.adaptive.enabled = true
```

How AQE Works

AQE dynamically optimizes queries at runtime by:

- Using **logical planning** to create an initial plan.
- Monitoring runtime metrics like shuffle size and partition data.
- Re-optimizing the plan based on this information.
- Iteratively refining the plan for efficiency.

This adaptability helps Spark perform well with complex and large-scale workloads.

Query Exampl

Consider the following SQL query:

```
SELECT o.customer_id, SUM(o.order_amount) AS total_spent
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
WHERE c.region = 'North America'
GROUP BY o.customer_id;
```

With AQE, Spark can dynamically:

- Handle data skew in the `orders` table.
- Adjust partitioning for efficient group-by operations.
- Optimize the join strategy based on the size of the `customers` table.

This flexibility ensures efficient query execution, even with changing or unexpected data characteristics.

Static Query Plan

Logical Plan

1. **Read Data**: Load the `orders` and `customers` tables.
2. **Apply Filter**: Filter `customers` where `region = 'North America'`.
3. **Perform Join**: Join the tables on `customer_id`.
4. **Group and Aggregate**: Group by `o.customer_id` and calculate the total order amount.

Static Physical Plan

1. **Join Strategy**:
   - The static optimizer assumes no precise size statistics for the `customers` and `orders` tables.
   - It plans a **shuffle hash join**, meaning both tables are shuffled and repartitioned on `customer_id`.
2. **Partitioning**:
   - The static plan uses the default shuffle partition configuration, often set to 200 partitions.
3. **Execution Sequence**:
   - The filter is applied to `customers` first.
   - Both datasets are shuffled and joined.
   - The result is grouped by `customer_id` and aggregated.

Issues in Static Plan

1. **Suboptimal Join Strategy**:
   - If `customers` is a small table, a **broadcast join** would be more efficient, but static planning cannot determine this without accurate size statistics.
2. **Excessive Shuffling**:
   - Both datasets are fully shuffled before the join.
   - If one dataset (e.g., `customers`) is small, this shuffle is unnecessary and adds overhead.
3. **Inefficient Partitioning**:
   - Using a fixed 200 shuffle partitions can lead to:
     - **Over-partitioning** small datasets, wasting resources.
     - **Under-partitioning** large datasets, causing straggler tasks that delay execution.

By addressing these issues with dynamic optimization like AQE, Spark can significantly improve query performance.

Dynamic Query Plan with AQE

1. Join Strategy

- **Dynamic Adjustment**: During execution, Spark measures the actual size of the `customers` and `orders` tables.
- **Optimization**: If the `customers` table is small (e.g., 10 MB), AQE switches to a **broadcast join**, avoiding unnecessary shuffling.

2. Partition Coalescing (merging)

- **Evaluation**: After the shuffle stage, Spark checks partition sizes.
- **Optimization**: If partitions are too small (e.g., <10 MB), AQE merges them into fewer, larger partitions, reducing task scheduling overhead.

3. Dynamic Partition Pruning (Filter the data while reading, retrieving only the required items**)**

- **Reduction**: The filter `c.region = 'North America'` on the `customers` table reduces rows significantly (e.g., from 1M to 100K).
- **Optimization**: AQE prunes irrelevant partitions in the `orders` table, limiting the data scanned and shuffled.

4. Skew Join Optimization

- **Detection**: If runtime statistics show skewed partitions in `orders` (e.g., one `customer_id` dominates), AQE splits large partitions into smaller chunks.
- **Benefit**: This balances the workload across tasks, avoiding delays caused by straggler tasks.

Performance Impact of AQE Optimizations

Optimization	Static Plan Impact	AQE Plan Impact	Performance Benefit
Join Strategy	Shuffle join for both datasets.	Broadcast join for the small `customers` table.	Eliminates unnecessary shuffle for `customers`.
Partition Coalescing	Fixed 200 partitions.	Fewer, larger partitions (e.g., ~50).	Reduces task scheduling overhead.
Dynamic Partition Pruning	Full scan of `orders`.	Reads only relevant partitions for North America.	Avoids scanning irrelevant data.
Skew Join Optimization	Skewed partitions cause long tasks.	Splits large partitions, balancing execution.	Prevents delays from straggler tasks.

Performance Comparison: Before and After AQE

Before AQE

- **Execution Plan**:
  - Shuffle join between `orders` and `customers`.
  - Fixed 200 partitions for all shuffle stages.
  - Full scan of the `orders` table.
- **Performance Metrics**:
  - **Total Execution Time**: 10 minutes.
  - **Resource Utilization**:
    - Excessive shuffle overhead.
    - Wasted compute resources on small partitions.
    - Straggler tasks caused by data skew.

After AQE

- **Execution Plan**:
  - Broadcast join for `customers` and `orders`.
  - Dynamically pruned irrelevant partitions in the `orders` table.
  - Reduced shuffle partitions to ~50.
  - Split skewed partitions into balanced tasks.
- **Performance Metrics**:
  - **Total Execution Time**: 3 minutes.
  - **Resource Utilization**:
    - Reduced shuffle I/O.
    - Balanced task execution across the cluster.
    - Optimized use of resources.

Core AQE Configuration Options in Apache Spark

1. Enable or Disable AQE

- **Property**: `spark.sql.adaptive.enabled`
- **Default**: `false`
- **Description**: Turns Adaptive Query Execution on or off.
- **Usage**:
  - Set to `true` to enable runtime query plan optimizations.
  - Particularly useful for workloads with unpredictable data sizes or skew, significantly enhancing performance.

Example:

```
spark.sql.adaptive.enabled = true
```

Adaptive Query Execution (AQE) Configuration Options

2. Dynamic Partition Pruning

2.1 Enable/Disable Dynamic Partition Pruning

- **Property**: `spark.sql.dynamicPartitionPruning.enabled`
- **Default**: `true`
- **Description**: Activates dynamic partition pruning to minimize the data scanned during query execution.
- **Usage**:
  - Dynamically identifies and prunes irrelevant partitions based on runtime filter values (e.g., values derived from subqueries).
  - Enhances performance for queries with filters on partition columns.

2.2 Fallback to Static Partition Pruning

- **Property**: `spark.sql.dynamicPartitionPruning.fallbackFilterRatio`
- **Default**: `0.5` (50%)
- **Description**: Defines the threshold for reverting to static partition pruning when runtime pruning is insufficient.
- **Usage**:
  - If dynamic pruning cannot prune enough partitions, Spark uses this threshold to decide when to apply static pruning instead.

3. Coalescing Shuffle Partitions

3.1 Enable/Disable Coalescing Shuffle Partitions

- **Property**: `spark.sql.adaptive.coalescePartitions.enabled`
- **Default**: `true`
- **Description**: Merges small shuffle partitions into larger ones to reduce task overhead.
- **Usage**:
    - Useful for queries with excessively small shuffle partitions, which can lead to too many tasks.
    - Dynamically adjusts partition sizes to optimize resource usage and performance.

3.2 Target Partition Size

- **Property**: `spark.sql.adaptive.coalescePartitions.minPartitionSize`
- **Default**: `1MB`
- **Description**: Specifies the minimum size for coalesced shuffle partitions.
- **Usage**:
    - Helps determine the degree to which Spark consolidates smaller partitions into larger ones.

3.3 Initial Shuffle Partition Count

- **Property**: `spark.sql.adaptive.coalescePartitions.initialPartitionNum`
- **Default**: Same as `spark.sql.shuffle.partitions` (e.g., 200).
- **Description**: Sets the initial number of shuffle partitions before AQE coalesces them.
- **Usage**:
    - Ensures a balanced approach to task parallelism and partition sizing during the initial shuffle phase.

4. Skew Join Optimization

4.1 Enable/Disable Skew Join Optimization

- **Property**: `spark.sql.adaptive.skewJoin.enabled`
- **Default**: `true`
- **Description**: Identifies and addresses skewed partitions during shuffle joins.
- **Usage**:
    - Splits large, skewed partitions into smaller chunks to distribute the workload more evenly across tasks.

4.2 Skew Partition Threshold

- **Property**: `spark.sql.adaptive.skewJoin.skewedPartitionThresholdInBytes`
- **Default**: `256MB`
- **Description**: Defines the size threshold for detecting skewed partitions.
- **Usage**:
    - Partitions exceeding this size are considered skewed and are split into smaller sub-partitions.

4.3 Skew Partition Split Factor

- **Property**: `spark.sql.adaptive.skewJoin.splitFactor`
- **Default**: `2`
- **Description**: Determines the number of splits for skewed partitions.
- **Usage**:
    - A higher value results in splitting large skewed partitions into more sub-partitions for better distribution.

---

5. Join Strategy Switching

5.1 Enable/Disable Join Strategy Switching

- **Property**: `spark.sql.adaptive.localShuffleReader.enabled`
- **Default**: `true`
- **Description**: Enables dynamic switching between shuffle join and broadcast join based on data size.
- **Usage**:
    - If one side of the join is small, Spark automatically switches to a broadcast join to avoid unnecessary shuffling.

5.2 Broadcast Join Threshold

- **Property**: `spark.sql.adaptive.localShuffleReader.thresholdInBytes`
- **Default**: `64MB`
- **Description**: Sets the maximum dataset size for it to be eligible for a broadcast join.
- **Usage**:
    - Datasets larger than this threshold are handled using shuffle joins unless AQE decides they can be broadcasted.

---

6. Query Stage Execution

6.1 Query Stage Materialization Timeout

- **Property**: `spark.sql.adaptive.forceApplyTimeout`
- **Default**: `300s` (5 minutes)
- **Description**: Sets the timeout for materializing query stages during AQE.
- **Usage**:
    - Prevents delays from excessive waiting during stage reoptimization.

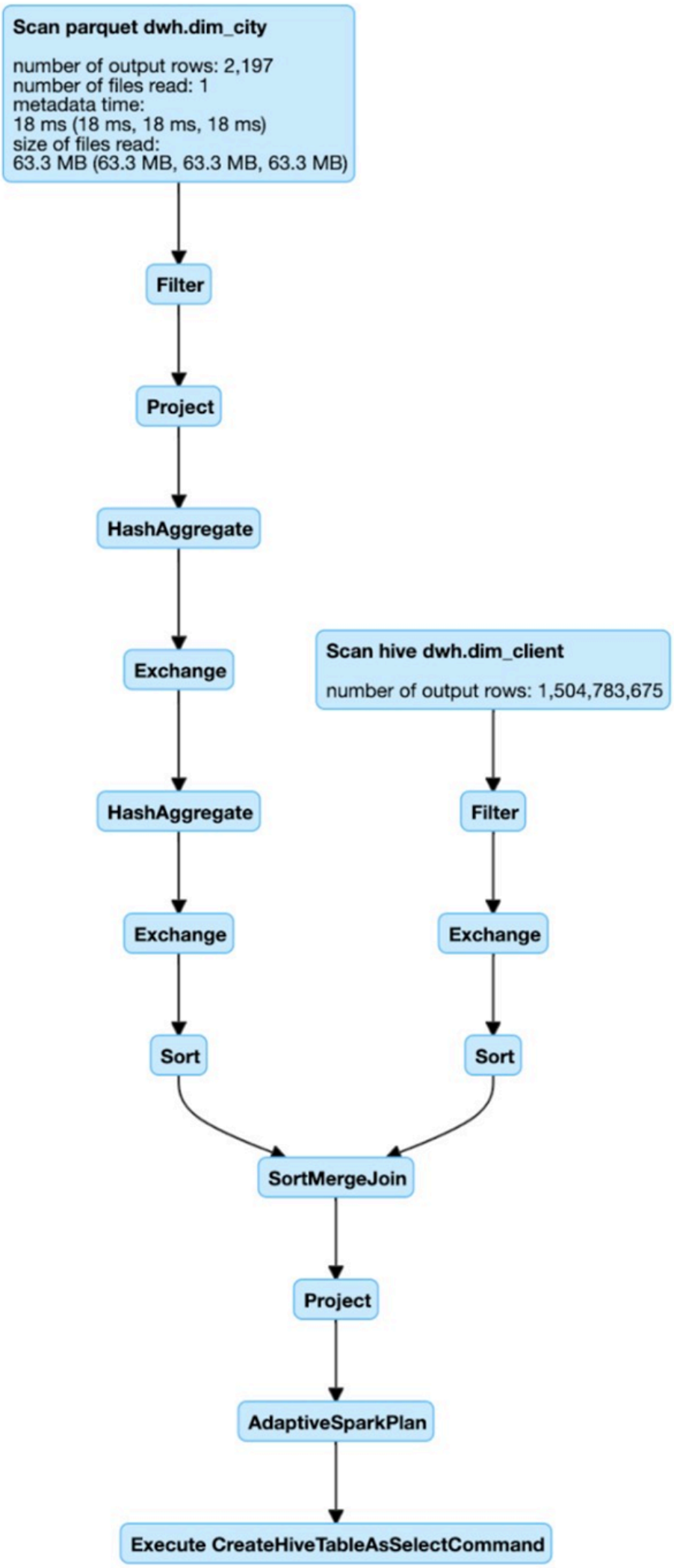
7. Miscellaneous Configurations

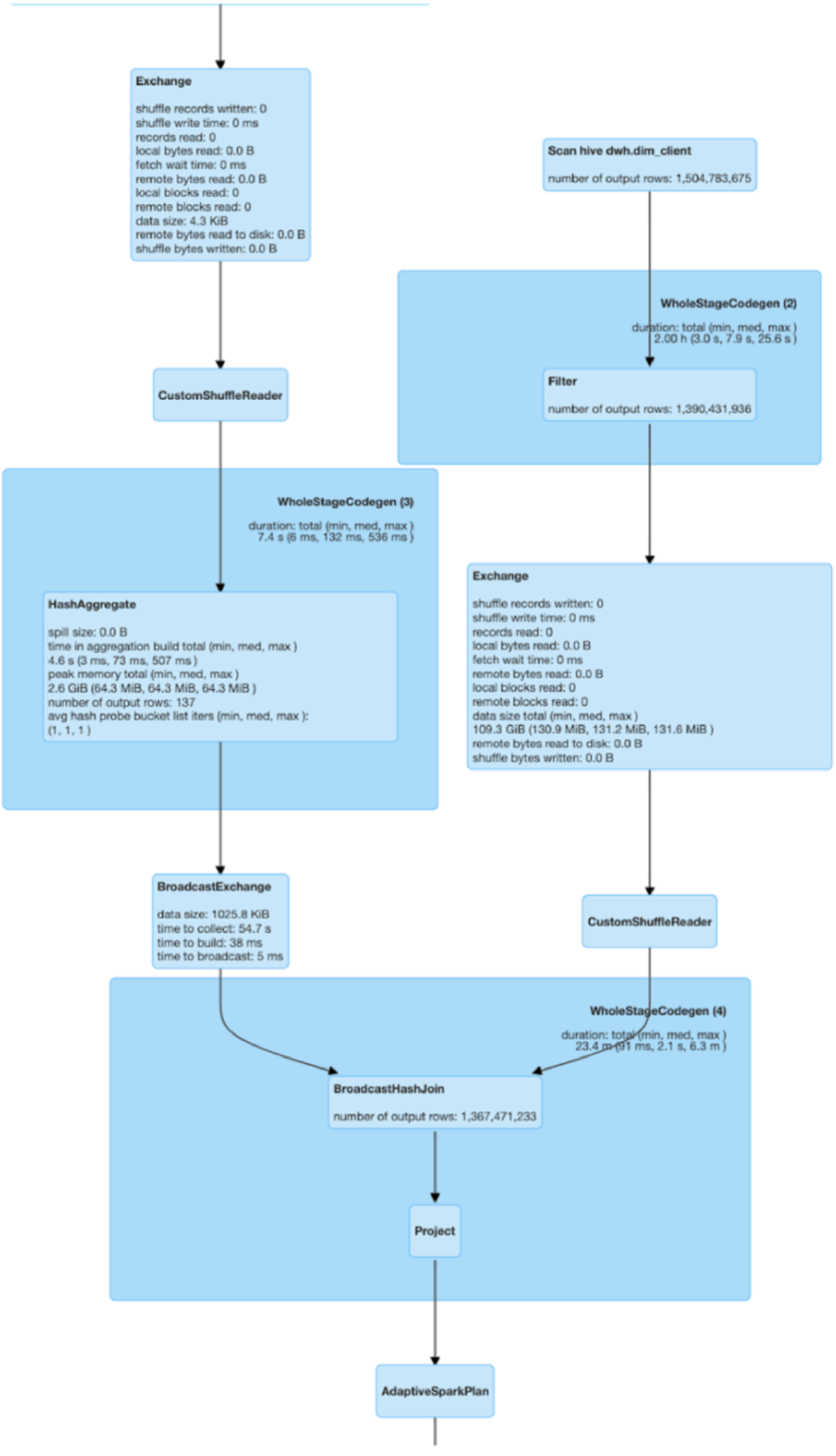
7.1 Maximum Number of Optimized Plans

- **Property**: `spark.sql.adaptive.maxNumPlans`
- **Default**: `10`
- **Description**: Specifies the maximum number of query plans AQE can generate during reoptimization.
- **Usage**:
    - Ensures AQE does not continuously refine the query plan, providing an upper limit to the number of iterations.

7.2 Intermediate Stage Metrics

- **Property**: `spark.sql.adaptive.reportIntermediateStages`
- **Default**: `false`
- **Description**: Enables reporting of metrics for intermediate stages during AQE reoptimization.
- **Usage**:
    - Useful for debugging and monitoring the performance of query execution during optimization.


Scan parquet dwh.dim_city
number of output rows: 2,197
number of files read: 1
metadata time:
18 ms (18 ms, 18 ms, 18 ms)
size of files read:
63.3 MB (63.3 MB, 63.3 MB, 63.3 MB)
Filter
Project
HashAggregate
Exchange
HashAggregate
Exchange
Sort
Scan hive dwh.dim_client
number of output rows: 1,504,783,675
Filter
Exchange
Sort
SortMergeJoin
Project
AdaptiveSparkPlan
Execute CreateHiveTableAsSelectCommand


Exchange
shuffle records written: 0
shuffle write time: 0 ms
records read: 0
local bytes read: 0.0 B
fetch wait time: 0 ms
remote bytes read: 0.0 B
local blocks read: 0
remote blocks read: 0
data size: 4.3 KiB
remote bytes read to disk: 0.0 B
shuffle bytes written: 0.0 B
Scan hive dwh.dim_client
number of output rows: 1,504,783,675
WholeStageCodegen (2)
duration: total (min, med, max)
2.00 h (3.0 s, 7.9 s, 25.6 s)
CustomShuffleReader
Filter
number of output rows: 1,390,431,936
WholeStageCodegen (3)
duration: total (min, med, max)
7.4 s (6 ms, 132 ms, 536 ms)
HashAggregate
spill size: 0.0 B
time in aggregation build total (min, med, max)
4.6 s (3 ms, 73 ms, 507 ms)
peak memory total (min, med, max)
2.6 GiB (64.3 MiB, 64.3 MiB, 64.3 MiB)
number of output rows: 137
avg hash probe bucket list iters (min, med, max):
(1, 1, 1)
Exchange
shuffle records written: 0
shuffle write time: 0 ms
records read: 0
local bytes read: 0.0 B
fetch wait time: 0 ms
remote bytes read: 0.0 B
local blocks read: 0
remote blocks read: 0
data size total (min, med, max)
109.3 GiB (130.9 MiB, 131.2 MiB, 131.6 MiB)
remote bytes read to disk: 0.0 B
shuffle bytes written: 0.0 B
BroadcastExchange
data size: 1025.8 KiB
time to collect: 54.7 s
time to build: 38 ms
time to broadcast: 5 ms
CustomShuffleReader
WholeStageCodegen (4)
duration: total (min, med, max)
23.4 m (91 ms, 2.1 s, 6.3 m)
BroadcastHashJoin
number of output rows: 1,367,471,233
Project
AdaptiveSparkPlan

Cloud Offerings For Data WarehousesRedshift and BigQuery - Dec 06

- **Why use the cloud?**
- **How to Sign Up for BigQuery's Free Trial**
- **Key Features of BigQuery**
- **Ways to Connect to BigQuery**
- **How BigQuery Works**
- **BigQuery’s Resource Management Approach**
- **Advantages of BigQuery’s Architecture**
- **Query Example: Reading from an S3 External Table**
- **Live Demo Screens**

All the technologies we have learned so far are open-source. This means they can be used by deploying them on servers, either on-premises or on cloud hosts provided by cloud providers. However, not everyone uses these technologies directly; some prefer solutions offered by cloud providers. The most popular providers are Google Cloud and AWS. For example, Google Cloud offers **BigQuery**, and AWS offers **Redshift**.

In the next few lectures, we will learn how these cloud technologies work.

Why use the cloud?

1. **Scalability and Flexibility**
    - **Elastic Resources**: The cloud allows businesses to scale resources up or down based on demand without buying physical hardware.
    - **Flexibility**: Companies can choose from different models like IaaS, PaaS, or SaaS based on their needs.
2. **Cost Efficiency**
    - **Pay-as-you-go**: You only pay for what you use, which reduces upfront costs for hardware and infrastructure.
    - **Operational Savings**: Less need for maintaining physical infrastructure or hiring IT staff for hardware management.
3. **Better Performance and Reliability**
    - **High Availability**: Cloud providers often guarantee high uptime with Service Level Agreements (SLAs).
    - **Global Network**: Cloud providers have data centers worldwide, offering faster and more reliable services close to users.

How to Sign Up for BigQuery's Free Trial

You can try BigQuery for free using the **BigQuery sandbox**. It doesn’t require a credit card or billing account. The sandbox provides:

- **10 GB** of active storage
- **1 TB** of processed query data per month

Steps to Access BigQuery Sandbox:

1. **Open BigQuery**
    - Go to the BigQuery page in the **Google Cloud Console**.
    - Log in with your Google account or create a new one if needed.

2. **Create a Project**
    - Click on **"Create Project"**.
    - Enter a project name and organization (if required).
    - Click **"Create"** to set up your project.
3. **Start Using BigQuery**
    - Once your project is created, you'll see the BigQuery interface. Here, you can run queries and explore datasets.

Key Features of BigQuery

1. **Serverless Architecture**
    - No need to manage infrastructure or storage. Focus on querying and analyzing data.
2. **Massive Scalability**
    - Handles datasets at the **petabyte scale** with ease.
3. **SQL-Like Interface**
    - Supports SQL queries, making it user-friendly for those familiar with relational databases.
4. **Separation of Storage and Compute**
    - Data is stored in a **columnar format** (Capacitor storage engine), and compute resources are allocated dynamically when running queries.
5. **Real-Time Analytics**
    - Supports streaming data for near real-time insights.
6. **Seamless Integration**
    - Works well with Google Cloud services like Google Sheets, Data Studio, and AI/ML tools like TensorFlow.
7. **Machine Learning (BigQuery ML)**
    - Run machine learning models directly within BigQuery using SQL.
8. **Security and Compliance**
    - Provides fine-grained access control and meets data protection standards like GDPR and HIPAA.

Ways to Connect to BigQuery

1. **BigQuery Web UI**
    - **Description**: Use the Google Cloud Console to interact with BigQuery directly.
    - **Use Case**: Best for running quick queries, managing datasets, and exploring schemas.
    - **How to Use**:
        1. Log in to the **Google Cloud Console**.
        2. Go to the **BigQuery** section.
        3. Use the query editor to write and execute SQL queries.
2. **BigQuery Command Line Interface (CLI)**
    - **Description**: Use the `bq` command-line tool for interacting with BigQuery.
    - **Use Case**: Ideal for scripting, automation, and command-line enthusiasts.
    - **How to Use**:
        1. Install the **Google Cloud SDK**.
        2. Authenticate with `gcloud auth login`.

Run commands like:

```
bq query "SELECT * FROM dataset.table LIMIT 10"
```

3. **BigQuery Client Libraries**
    - **Description**: SDKs in various programming languages enable programmatic access to BigQuery.
    - **Use Case**: Great for application integration, automation, or creating custom analytics pipelines.
    - **Supported Languages**: Python, Java, Node.js, C#, Go, Ruby, PHP, and more.

Example (Python):

```
from google.cloud import bigquery

client = bigquery.Client()
query = "SELECT * FROM `project.dataset.table` LIMIT 10"
results = client.query(query).to_dataframe()
print(results)
```

4. **JDBC/ODBC Drivers**
    - **Description**: Use JDBC/ODBC drivers to connect BigQuery with BI tools or custom applications.
    - **Use Case**: Perfect for integration with tools like Tableau, Power BI, or legacy systems.
    - **How to Use**:
        1. Download the official drivers from the Google BigQuery website.
        2. Configure the connection string with your project details.
5. **APIs (REST/JSON or gRPC)**
    - **Description**: Use BigQuery's REST API or gRPC endpoints for direct interaction.
    - **Use Case**: Ideal for custom integrations and workflows not supported by other methods.
    - **How to Use**:
        1. Authenticate using **OAuth 2.0** or a **service account**.
        2. Make API calls to interact with BigQuery services.

Example (REST API with `curl`)

```
curl -X POST \
  -H "Authorization: Bearer $(gcloud auth application-default print-access-token)" \
  -H "Content-Type: application/json" \
  -d '{"query": "SELECT * FROM dataset.table LIMIT 10"}' \
  https://bigquery.googleapis.com/bigquery/v2/projects/PROJECT_ID/queries
```

How BigQuery Works

1. **Data Storage**
    - Data is stored in a **highly optimized, columnar format** that enhances performance for analytical queries.
2. **Query Execution**
    - BigQuery uses a **distributed execution engine** to process queries across multiple nodes for high speed and scalability.
3. **On-Demand or Flat-Rate Pricing**
    - Users can:
        - Pay per query (on-demand).
        - Subscribe to flat-rate pricing for predictable costs.
4. **Streaming Data**
    - BigQuery supports **real-time ingestion of data streams** through APIs, enabling quick insights.

BigQuery's Resource Management Approach

BigQuery uses serverless architecture and a **slot-based model** to manage storage, compute, and execution resources.

1. **Serverless Resource Management**
    - No need to manage servers or clusters; BigQuery automatically handles infrastructure.
    - Resources are dynamically allocated based on workload demands.
2. **Slot-Based Compute Model**
    - A **slot** is the basic unit of computational capacity in BigQuery.
    - Queries are executed using slots, dynamically assigned by the BigQuery scheduler.
    - Organizations can purchase dedicated slots and allocate them using **reservations** for specific projects or workloads.
3. **Dynamic Query Scheduling**
    - BigQuery's scheduler prioritizes queries based on resource availability and query complexity.
    - Users can set priorities for their queries (e.g., batch vs. interactive) to optimize resource usage.
4. **Reservations and Capacity Management**
    - Flat-rate pricing allows organizations to buy a **pool of slots** and assign them to workloads.
    - Slots can be managed using reservations, enabling resource allocation for teams or projects.
    - Unlike systems like YARN, which use queues and resource allocation rules, BigQuery abstracts these details, relying on intelligent and dynamic allocation.

1. Core Components of BigQuery

BigQuery's architecture consists of the following key components:

a. Storage Layer

- **Columnar Storage**:
    - Data is stored in a proprietary columnar format called **Capacitor**.
    - Optimized for analytical queries, allowing fast access to specific columns without reading the entire table.
- **Separation of Storage and Compute**:
    - Storage and compute are **independently scalable**, meaning storage scales for data size while compute scales for query processing.

b. Compute Layer

- **Query Execution Engine**:
    - Uses a **massively parallel processing (MPP)** model to break queries into smaller tasks.
    - Tasks are distributed across thousands of nodes for efficient execution.
- **Slot-Based Model**:
    - Compute capacity is measured in **slots**, which represent units of processing power.
    - Slots are dynamically allocated by BigQuery's internal scheduler.

c. Metadata and Control Plane

- **Metadata Management**:
    - Tracks datasets, tables, schemas, and partitions.
    - Optimizes query plans and execution strategies.
- **Job Scheduler**:
    - Schedules query execution and assigns slots based on resource availability and priorities.
- **Access Control**:
    - Uses Google Cloud Identity and Access Management (IAM) for fine-grained security controls.

2. Query Execution Workflow

1. **Query Submission**:
    - SQL queries can be submitted through the BigQuery web interface, CLI, API, or client libraries.
2. **Query Parsing**:
    - Queries are parsed and optimized into an execution plan by the BigQuery engine.
3. **Query Execution**:
    - The plan is executed across multiple nodes.
    - Each node processes a portion of the data in parallel, leveraging columnar storage for efficiency.
4. **Result Aggregation**:
    - Partial results from all nodes are combined and returned to the user.

---

3. BigQuery's Distributed Architecture

a. Distributed Storage

- Data is distributed across multiple Google Cloud regions and zones.
- Automatic **replication** ensures durability and availability.

b. Distributed Compute

- Uses Google's **Dremel technology** to process queries across thousands of parallel nodes.
- Columnar data format enhances speed and efficiency during query execution.

---

4. Streaming and Batch Ingestion

BigQuery supports two main modes for data ingestion:

- **Batch Ingestion**:
    - Bulk upload data using Cloud Storage or APIs.
- **Streaming Ingestion**:
    - Stream real-time data via BigQuery's streaming API for near real-time analytics.

---

5. BigQuery ML and BI Engine

- **BigQuery ML**:
    - Machine learning capabilities are integrated directly into BigQuery.
    - Users can build and deploy ML models using SQL.
- **BI Engine**:
    - An in-memory analysis service designed to accelerate dashboards in tools like Google Data Studio.

---

6. High Availability and Fault Tolerance

- Data is replicated across regions to ensure **high availability**.
- **Automatic failover mechanisms** maintain query execution even during hardware or software failures.

7. Security and Compliance

- **Encryption**:
  - Data is encrypted both **at rest** and **in transit**.
- **Access Control**:
  - Permissions can be assigned at the **organization**, **project**, **dataset**, or **table** level.
- **Compliance**:
  - BigQuery meets key standards like **GDPR**, **HIPAA**, and **ISO/IEC 27001**.

Advantages of BigQuery's Architecture

- **Scalability**: Seamlessly handles datasets of petabyte scale.
- **Flexibility**: Decoupled storage and compute resources allow independent scaling.
- **Performance**: Columnar storage and distributed query execution deliver rapid results.
- **Ease of Use**: Serverless model eliminates the need for infrastructure management.

---

1. Reading Data Directly from Amazon S3

BigQuery can access data stored in Amazon S3 through **BigLake** or by creating an **external table**. This allows querying data in S3 without importing it into BigQuery storage.

Key Steps and How It Works

1. **Data Format Compatibility**:
   - BigQuery supports files stored in S3 in the following formats:
     - CSV
     - JSON
     - Parquet
     - ORC
     - Avro
2. **Access Configuration**:
   - A **Google Cloud service account** is granted permissions to access the S3 bucket using AWS IAM.
   - Alternatively, the **Google Cloud Storage Transfer Service** can connect S3 to Google Cloud for data access.
3. **External Table Creation**:
   - Create an external table in BigQuery pointing to the S3 bucket and specifying the data format.
   - The table serves as a virtual link to the S3 data, enabling SQL queries directly on the external storage.
4. **Query Execution**:
   - BigQuery reads the data from S3 in a **read-only** manner.
   - Data remains in S3 while query results are computed in BigQuery's execution layer.

2. Loading Data from S3 into BigQuery

To permanently move data into BigQuery storage, you can use Google Cloud Storage as an intermediary.

Key Steps and How It Works

1. **Transfer Data to Google Cloud Storage**:
   - Use **Google Cloud Storage Transfer Service** to copy data from S3 to a Google Cloud Storage bucket.
   - Specify the source S3 bucket and the target Cloud Storage bucket in the transfer settings.
2. **Load Data into BigQuery**:
   - Once the data is in Google Cloud Storage, use the **BigQuery Data Transfer Service** or the `LOAD DATA` SQL command to import it.
   - Specify the dataset, table, and data format during the process.
3. **Data Storage in BigQuery**:
   - Data is stored in BigQuery's **columnar storage**, optimized for fast analytics.

3. Security and Permissions

To enable BigQuery to read data from S3:

1. **AWS IAM Permissions**:
   - Grant necessary permissions to the S3 bucket for the Google Cloud service account.
   - Example: Provide `s3:GetObject` permissions for specific files or buckets.
2. **Cross-Cloud Authentication**:
   - Use a **Service Account Key** or configure an **AWS IAM Role** with Google Cloud Identity Federation for secure access.

Advantages of Each Method

Method	Advantages
External Table	- No data duplication.
	- Cost-effective for querying data as-is directly from S3.
Load Data to BigQuery	- Optimized for performance.
	- Best for repeated or intensive queries, as data is stored in BigQuery.

Query Example: Reading from an S3 External Table

If an external table is created, you can query it as you would a regular BigQuery table:

```
SELECT *
FROM `my_project.my_dataset.my_external_table`
WHERE column_name = 'example_value';
```

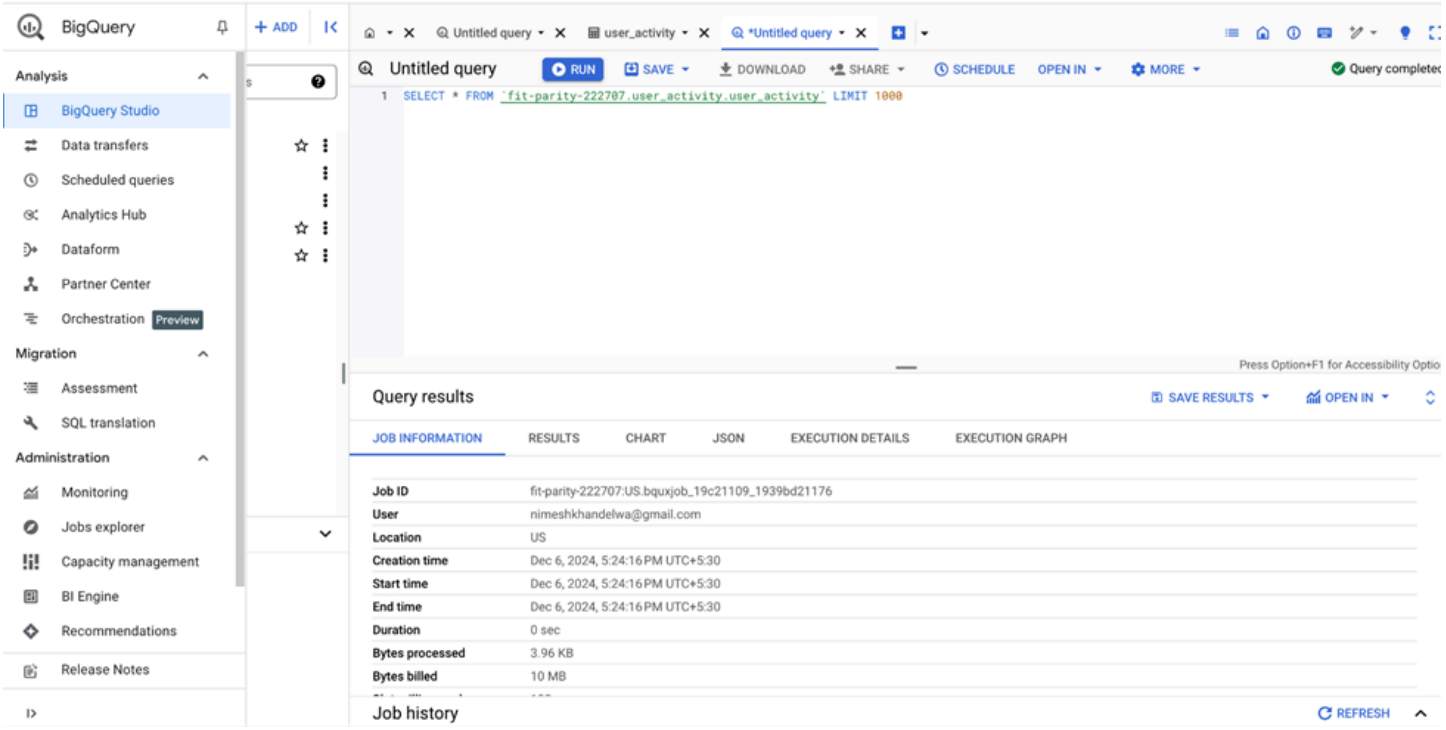
Tools and Integrations

1. **BigLake**: For querying S3 and other external storage with unified access.
2. **Storage Transfer Service**: For automated data movement between Amazon S3 and Google Cloud Storage (GCS).
3. **BigQuery Data Transfer Service**: For loading data from GCS into BigQuery storage.

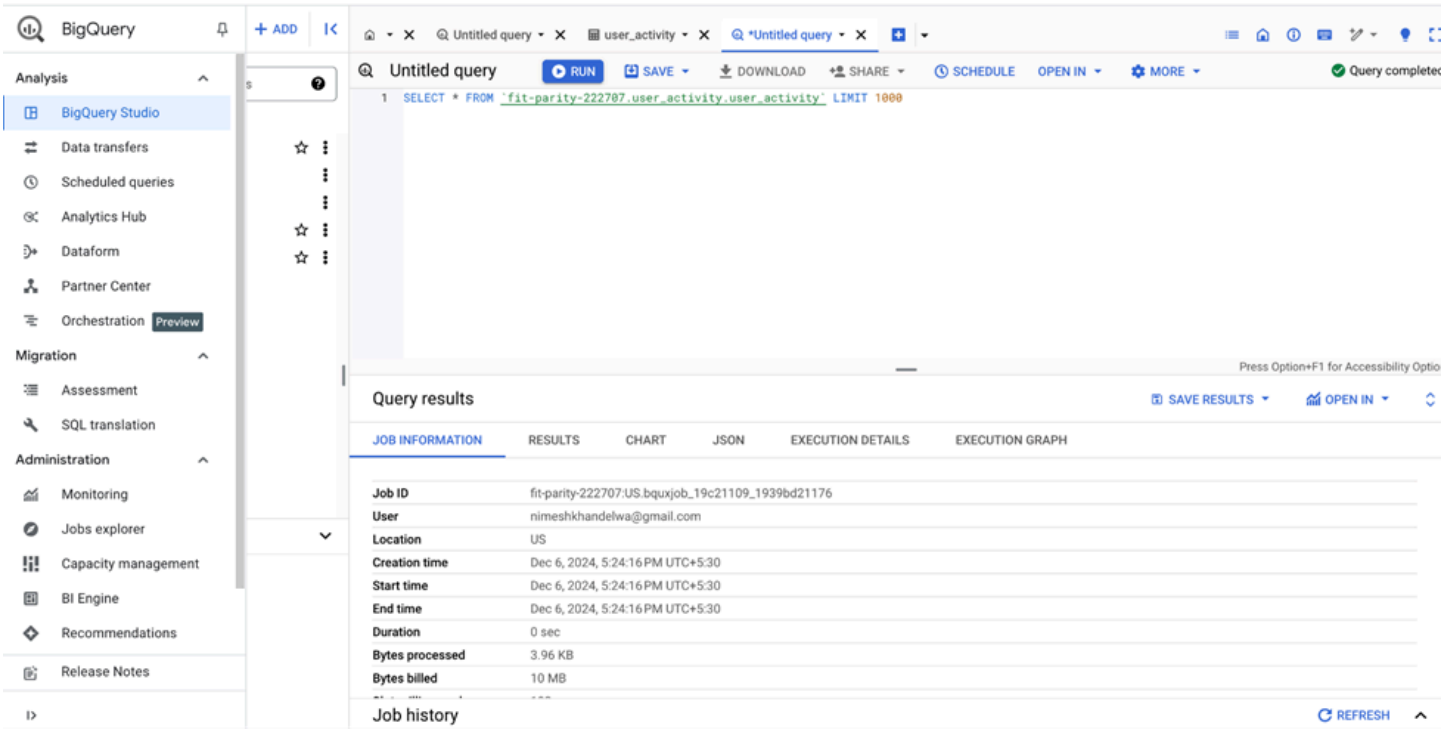
By integrating Amazon S3 with BigQuery, you can combine the strengths of both platforms, enabling efficient and scalable cross-cloud analytics.

Live Demo

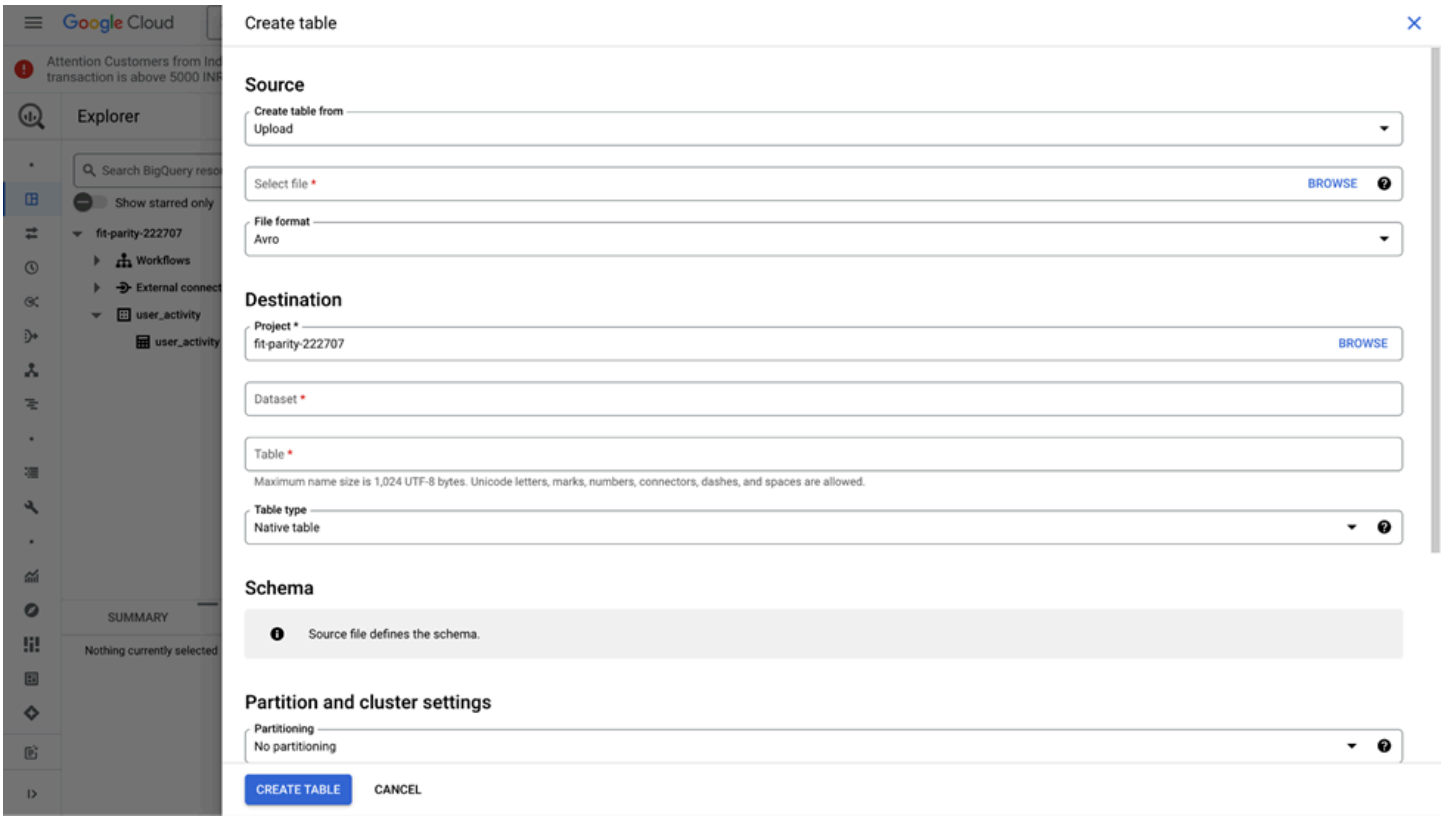
Intro screen



Load data



Upload Local



Query Results

Query results

SAVE RESULTS | OPEN IN

JOB INFORMATION | RESULTS | CHART | JSON | EXECUTION DETAILS | EXECUTION GRAPH

Job ID	fit-parity-222707:US.bquxjob_19c21109_1939bd21176
User	nimeshkhandelwa@gmail.com
Location	US
Creation time	Dec 6, 2024, 5:24:16 PM UTC+5:30
Start time	Dec 6, 2024, 5:24:16 PM UTC+5:30
End time	Dec 6, 2024, 5:24:16 PM UTC+5:30
Duration	0 sec
Bytes processed	3.96 KB
Bytes billed	10 MB

Job history — REFRESH

Query results

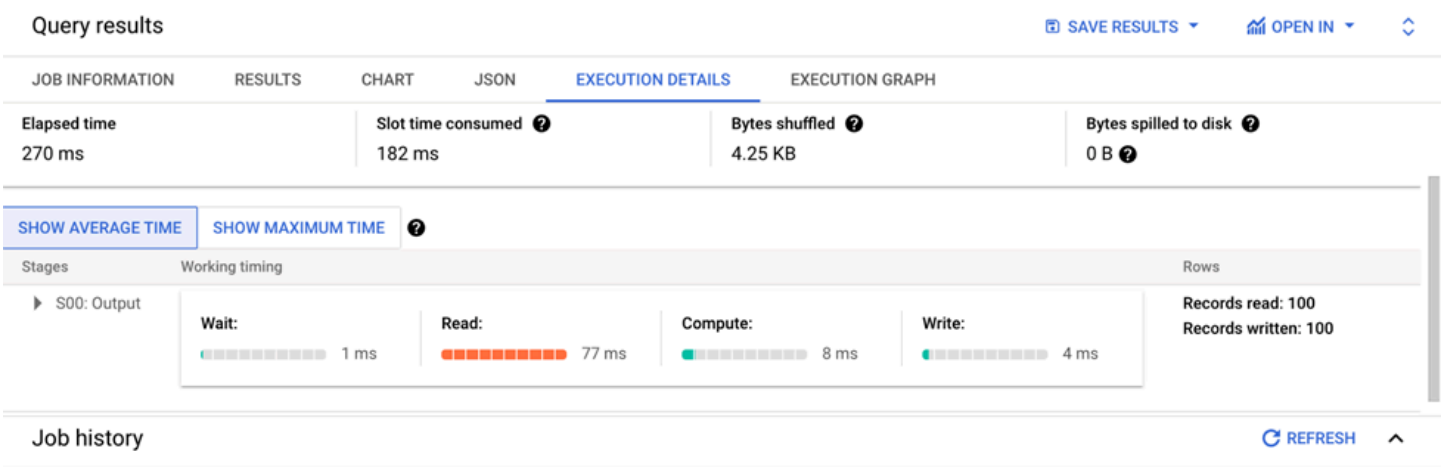
SAVE RESULTS | OPEN IN

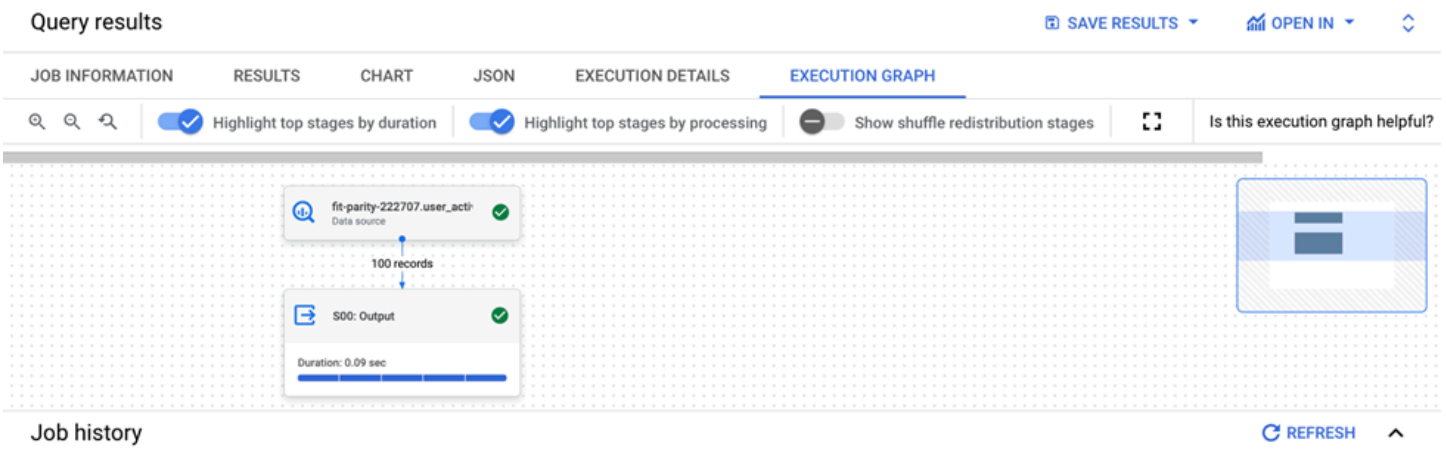
JOB INFORMATION | RESULTS | CHART | JSON | EXECUTION DETAILS | EXECUTION GRAPH

Row	user_id	product_id	action	timestamp	session_time
1	2	23	view	2024-11-04 15:38:35 UTC	192.53
2	2	8	view	2024-10-30 18:41:35 UTC	76.83
3	4	38	view	2024-10-13 03:03:35 UTC	190.92
4	7	30	view	2024-11-01 18:18:35 UTC	199.06
5	8	34	view	2024-10-27 12:35:35 UTC	125.27
6	10	49	view	2024-11-04 13:24:35 UTC	39.85
7	10	7	view	2024-10-20 08:48:35 UTC	57.37
8	11	19	view	2024-10-15 11:32:35 UTC	90.16

Results per page: 50 — 1 – 50 of 100

Job history — REFRESH





Cloud Offerings For Data WarehousesRedshift and BigQuery-2 - Dec 09

- **Key features of Redshift**
- **Redshift Architecture**
- **Ways to Connect to Amazon Redshift**
- **Web-Based Query Editor (AWS Management Console)**
- **Amazon Redshift Query Editor**

In the last lecture, we learned about Google BigQuery. This lecture focuses on Amazon Redshift, a cloud-based service from AWS. Amazon Redshift is a fully managed, petabyte-scale cloud data warehouse that is designed to process and analyze large amounts of data efficiently. Its design is meant for high performance, scalability, and reliability.

Key features of Redshift:

- **Massively Parallel Processing (MPP)**: Distributes query tasks across multiple nodes for faster processing.
- **Columnar Storage**: Stores data in columns to improve query performance and reduce I/O.
- **Data Compression**: Uses advanced compression techniques to save storage space and speed up queries.
- **Scalability**: Supports horizontal scaling (adding nodes) and independent scaling with RA3 nodes.
- **Concurrency Scaling**: Automatically adds resources during busy times to handle many queries at once.

- **Redshift Spectrum**: Allows querying data directly from Amazon S3 without loading it into the cluster.
- **Automated Backups**: Regularly backs up data to Amazon S3 for reliability and disaster recovery.
- **Fault Tolerance**: Replicates data within the cluster and can fix itself if issues occur.
- **Query Optimization**: Uses efficient execution plans, result caching, and a query optimizer to improve performance.
- **Workload Management (WLM)**: Allocates resources to prioritize different workloads or user groups.
- **Data Distribution Styles**: Optimizes data placement using KEY, EVEN, or ALL distribution styles.
- **AWS Integration**: Works well with S3, Glue, CloudWatch, IAM, and other AWS services.
- **Security**: Provides data encryption (both at rest and in transit), IAM-based access control, and VPC support.
- **Elastic Resize**: Quickly changes the cluster size to match workload changes.
- **BI Tool Compatibility**: Works with Tableau, Power BI, Looker, and other visualization tools.
- **Cost Efficiency**: Offers pay-as-you-go pricing, reserved instances, and cost-optimized RA3 nodes.
- **High Availability**: Ensures 99.9% uptime with data replication and automatic failover.
- **Materialized Views**: Provides pre-computed results for faster repeated query execution.
- **Data Sharing**: Allows sharing data between Redshift clusters without duplication.
- **Advanced Analytics**: Supports complex SQL queries, joins, subqueries, and window functions.

Redshift Architecture

1. Core Components of Redshift Architecture
   Redshift's architecture has several key components:
   a. **Leader Node**
      - **Role**: The leader node is the main part of the Redshift cluster. It handles query planning and coordination.
      - **Functions**:
        - Accepts client connections and query requests.
        - Parses and optimizes SQL queries.
        - Creates query execution plans.
        - Coordinates work distribution to compute nodes.
        - Collects results from compute nodes and sends them back to the client.

   b. **Compute Nodes**

      - **Role**: Compute nodes execute the queries by processing and storing data.
      - **Structure**:
        - A cluster can have 1 to 128 compute nodes.
        - Each compute node has multiple slices (virtual partitions) for parallel processing.
      - **Functions**:
        - Store data on their local storage.
        - Perform data transformations, filtering, and aggregations.
        - Return intermediate and final results to the leader node.

   c. **Node Slices**

      - **Role**: Each compute node is divided into slices, which are independent processing units that improve parallel processing.
      - **Structure**: The number of slices per node depends on the type of node.
        - For example, Dense Compute (DC2) nodes usually have more slices than Dense Storage (DS2) nodes.
      - **Functions**:
        - Store a subset of data from a table.

        - Process the subset in parallel with other slices.

d. **Storage Layer**

- Redshift uses a columnar storage model, storing data in columns instead of rows. This improves compression and read performance since only the relevant columns are read for a query.
- Data is distributed among the compute nodes based on a distribution key (hash, key, or even distribution).

2. **Key Architectural Features**

a. **Massively Parallel Processing (MPP)**

- Redshift achieves high performance with MPP, which splits queries into smaller tasks spread across slices on multiple nodes.
- Each slice processes its data independently, allowing Redshift to scale efficiently with more nodes.

b. **Data Distribution**

- Data is distributed across slices using several distribution styles:
    - KEY: Rows with the same distribution key are stored on the same node.
    - EVEN: Rows are distributed evenly across all slices.
    - ALL: A full copy of the table is given to all nodes (for small lookup tables).

c. **Result Caching**

- Redshift caches query results, which allows repeated queries to be answered quickly without re-executing them.

d. **Columnar Compression**

- Columnar storage lets Redshift compress data effectively, lowering storage needs and improving I/O performance.
- Compression methods (like LZO, ZSTD) are automatically selected during data loading.

e. **Fault Tolerance**

- Data is copied within the cluster for redundancy.
- Automatic recovery systems ensure high availability and reliability.

3. **Data Loading and Query Execution**

a. **Data Loading**

- Data is loaded into Redshift from sources like Amazon S3, DynamoDB, or other databases. This can be done using tools like AWS Glue, the COPY command, or ETL tools.
- Redshift Spectrum allows querying external data in S3 without loading it into the cluster.

b. **Query Execution Process**

- The client sends a SQL query to the Leader Node.
- The leader node parses, optimizes, and builds a query execution plan.
- The leader node assigns query tasks to slices across Compute Nodes.
- Compute nodes execute the query tasks in parallel, working on their data parts.
- Intermediate results go back to the leader node for aggregation.
- The leader node combines the results and sends them back to the client.

4. **Scalability**

Redshift offers two main scaling options:

- Horizontal Scaling: Add more compute nodes to increase processing power.

- Vertical Scaling: Choose a larger instance type for higher capacity (e.g., DC2 vs. RA3 nodes).

5. **Advanced Features**
   a. **Concurrency Scaling** - Automatically adds computer resources to handle many requests at once without affecting the main cluster's performance.

   b. **RA3 Nodes** - Redshift RA3 nodes separate storage and compute, allowing independent scaling of storage capacity.

   c. **Redshift Spectrum** - Lets you query data directly in S3 using the same Redshift SQL interface.

   d. **Elastic Resize** - Quickly adjusts the cluster size to meet changing workloads or storage needs.

6. **Redshift Performance Optimization**
   - **Distribution Keys**: Choose the right distribution keys to prevent data skew and improve query performance.
   - **Sort Keys**: Define sort keys to optimize data retrieval.
   - **Vacuuming and Analyzing**: Regular maintenance to reclaim storage and update statistics.
   - **Workload Management (WLM)**: Allocates resources to different user groups or workloads for better prioritization.

Ways to Connect to Amazon Redshift

a. **Using SQL Client Tools (e.g., DBeaver, pgAdmin, SQL Workbench/J)**

1. Download and Install a Client Tool:
   - Popular tools include SQL Workbench/J, DBeaver, and pgAdmin.
2. Download the Amazon Redshift JDBC/ODBC Driver:
   - Get it from the AWS Redshift Driver page.
3. Configure the Client Tool:
   - Create a new connection using the Redshift JDBC/ODBC driver.
   - Provide the following details:
     - Host: Cluster Endpoint (found in the AWS Redshift Console under Cluster Properties).
     - Port: 5439 (or your custom port if you set one).
     - Database: Name of your database.
     - Username: Your Redshift database username.
     - Password: Your Redshift database password.
   - Test the connection and save it.

b. **Using Python (psycopg2 Library)**

1. **Install psycopg2:**
   - Run: `pip install psycopg2-binary`

   Write Python Code:

```
import psycopg2

# Define connection details
conn = psycopg2.connect(
    host='your-cluster-endpoint',
    port=5439,
    database='your-database-name',
    user='your-username',
    password='your-password'
```

```
)

# Test connection
cursor = conn.cursor()
cursor.execute("SELECT current_date;")
print("Connection Successful:", cursor.fetchone())

# Close connection
cursor.close()
conn.close()
```

2. **Run the Script:**
   - Replace `your-cluster-endpoint`, `your-database-name`, `your-username`, and `your-password` with actual values.
   - Execute the script to test the connection.

c. Using AWS CLI

1. Install and Configure AWS CLI:
   - Install AWS CLI by following the AWS CLI Installation Guide.
   - Configure the CLI with your credentials by running: `aws configure`
2. Connect to Redshift:
   - Use the `aws redshift` command for management tasks.
   - To connect interactively, use `psql` or another PostgreSQL client tool.

d. Using JDBC/ODBC Applications

- JDBC Connection String: `jdbc:redshift://your-cluster-endpoint:5439/your-database-name`
- ODBC Connection:
  - Configure the Redshift ODBC driver on your machine.
  - Use a client tool to establish a connection.

Web-Based Query Editor (AWS Management Console)

Amazon Redshift offers an easy-to-use Query Editor in the AWS Management Console, enabling you to execute queries without needing to install any additional tools.

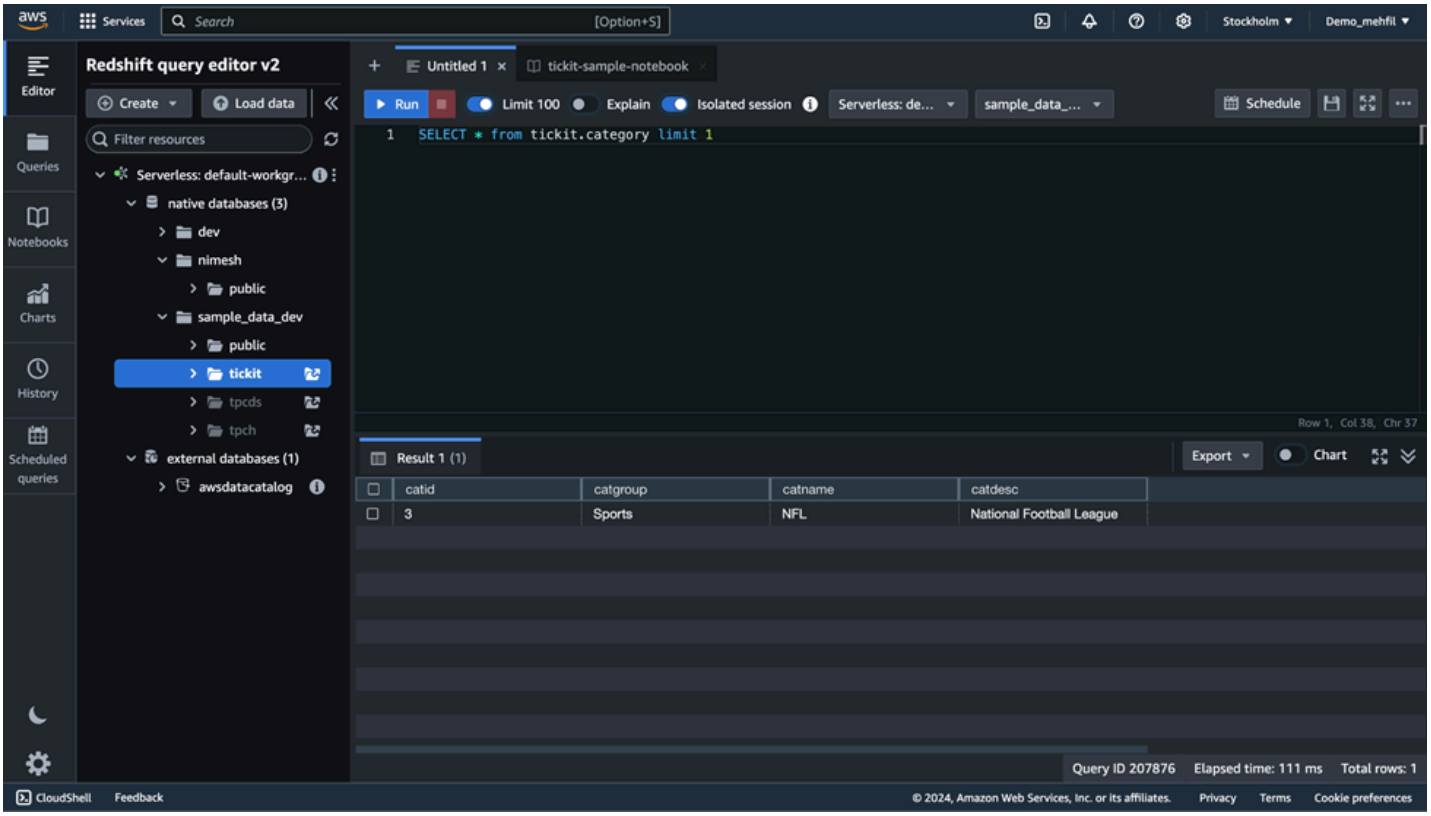
Steps to Use the Query Editor:

1. Log In to the AWS Management Console:
   - Go to the AWS Console.
   - Navigate to the Amazon Redshift service.
2. Access the Query Editor:
   - In the Redshift console, select Query Editor from the navigation pane.
3. Authenticate:
   - Choose the Redshift cluster you want to connect to.
   - Provide your database credentials:
     - Database Name: The name of your database in the Redshift cluster.
     - Database User: Your Redshift username.
     - Password: Your database password.

- Organize queries into projects or folders as needed.

4. Execute Queries:
   - Use the query editor to:
     - Run SQL commands.
     - Explore tables, schemas, and query results directly in your browser.
   - Query results are displayed in a table format for easy viewing.
5. Save and Manage Queries:
   - Save frequently used queries for future use.

Live Demo



Amazon Redshift Query Editor

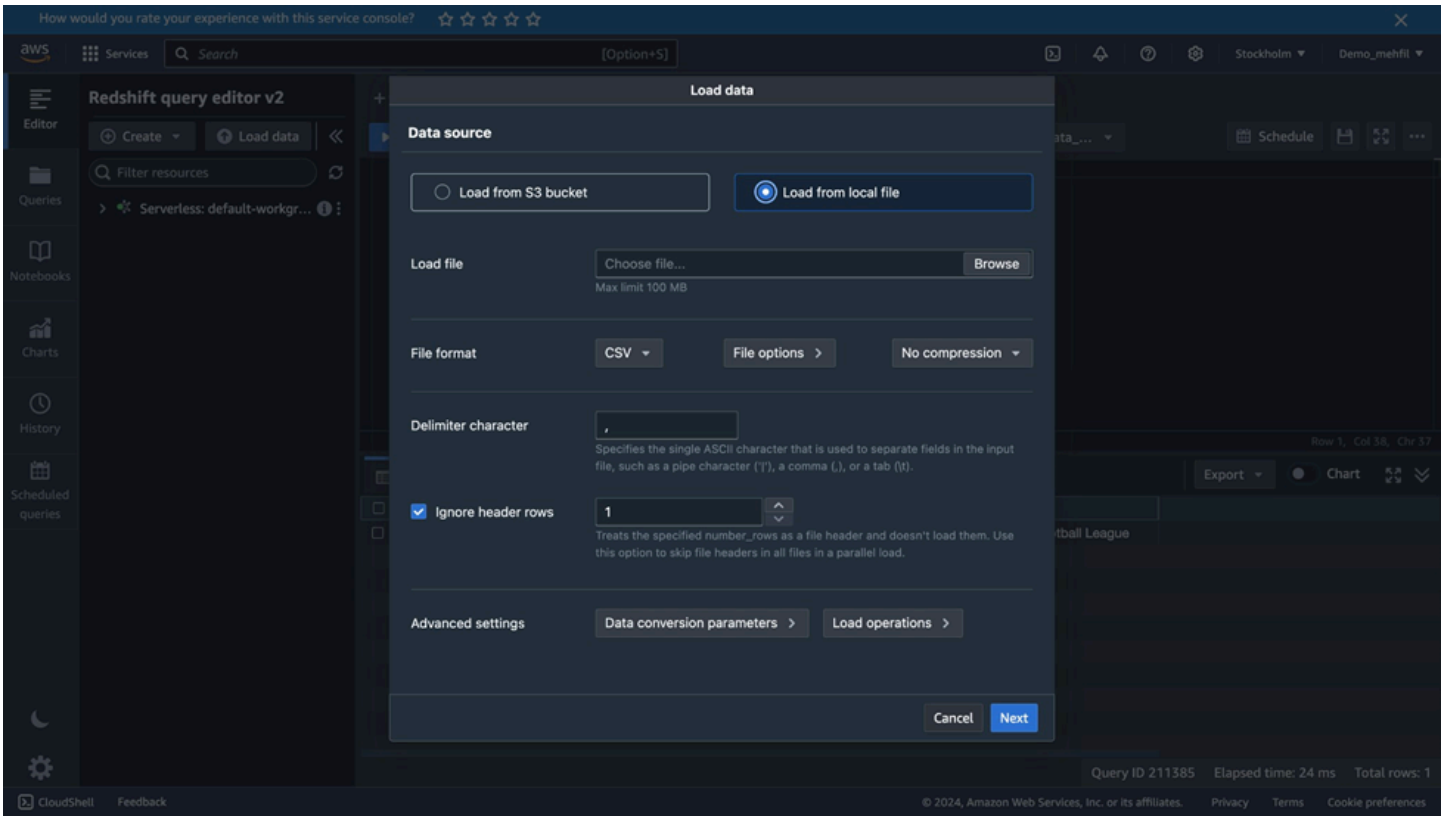
Amazon Redshift's Query Editor is a web-based interface within the AWS Management Console that enables users to interact with their Redshift databases. It provides a straightforward and efficient way to execute SQL queries, explore database objects, and manage data. Here's a detailed overview of the components of the Redshift Query Editor:

1. **Navigation Pane**
   The navigation pane offers access to database resources and tools:
   - Cluster Selection:
     - Lists all available Redshift clusters.
     - Allows selection of the cluster to query.
   - Database Explorer:
     - Displays a hierarchical view of databases, schemas, tables, and other objects.
     - Provides quick access to table structures, column details, and relationships.
2. **Connection Settings**
   Essential settings for connecting to the database:
   - Cluster Name: The Redshift cluster you are connecting to.
   - Database Name: The specific database within the selected cluster.
   - User Credentials: Username and password required for authentication.

- Region Selection: The AWS region where the cluster is located.

3. **Query Editor Workspace**
   The main area for writing and executing SQL queries:
   - **SQL Editor:**
     - A text editor with syntax highlighting for SQL.
     - Supports writing and editing complex queries.
   - **Toolbar:**
     - Options include:
       - Run Query: Executes the SQL query.
       - Stop Query: Cancels a running query.
       - Save Query: Saves queries for later use.
       - Load Query: Opens previously saved queries.
       - Clear Editor: Clears the workspace.
   - Auto-Completion: Suggests keywords, table names, and columns as you type.
4. **Results Panel**
   Displays the results of query execution:
   - Result Table:
     - Shows data retrieved from the query in a tabular format.
     - Supports sorting and filtering on columns.
   - Export Options:
     - Export query results to CSV for offline analysis.
   - Error Messages:
     - Displays errors or warnings if the query fails, along with detailed debugging messages.
5. **Query History**
   Allows tracking and management of previous queries:
   - Executed Queries:
     - Lists recently executed queries with their execution times.
   - Query Status:
     - Indicates the success or failure of each query.
   - Re-execute Queries:
     - Quickly rerun a query from history.
6. **Database Object Inspector**
   Provides metadata about database objects:
   - Schemas: Displays all schemas in the selected database.
   - Tables: Lists tables with detailed structure information (e.g., column names, data types, constraints).
   - Views: Displays all available views in the database.
   - Stored Procedures: Lists stored procedures and allows viewing their definitions.
   - Functions: Provides details about built-in or user-defined functions.
7. **Query Settings**
   Additional options to customize query execution:
   - Timeout Settings:
     - Set maximum query execution time to avoid long-running queries.
   - Resource Limits:
     - Control query resource usage to prevent overloading the cluster.
8. **Visual Tools**
   Provides visual aids for better interaction:
   - Table Previews: Quickly preview table contents and structure without writing SQL.
   - Schema Visualization: Some advanced versions include schema diagrams for exploring relationships between tables.

9. **Integration with AWS Services**
   The query editor integrates seamlessly with other AWS services:
   - IAM Authentication:
     - Uses IAM roles for secure and managed access.
   - CloudWatch Integration:
     - Monitor query execution metrics through CloudWatch logs.
10. **Settings and Preferences**
    Customize the behavior of the query editor:
    - Theme Settings:
      - Choose between light or dark themes for the interface.
    - Execution Preferences:
      - Configure default settings like auto-commit, query limits, and result display preferences.



DataBricks Hands on - Dec 11

- **Key Advantages of Databricks' Spark-as-a-Service**
- **Databricks Pricing**
- **How DBU Consumption Works**
- **Storage Options in Databricks**
- **Resource Managers in Databricks**
- **What Is Photon?**
- **Live Demo: Creating a Databricks Community Edition Account**
- **Cluster Management Tabs Overview**

In the previous lecture, we learned about two popular cloud solutions for handling big data. But what if we want to use Spark on the cloud? That's where Databricks comes in. Databricks provides Apache Spark as a fully managed service, making it easier to deploy and manage Spark clusters. This removes the challenges of setting up, configuring, and maintaining clusters, so users can focus on processing and analyzing data.

Key Advantages of Databricks' Spark-as-a-Service:

1. **Ease of Use**:
   Databricks offers a simple interface for creating and managing Spark clusters. Users can set up clusters with just a few clicks, speeding up project launches and reducing operational work.
2. **Optimized Performance**:
   Databricks enhances Apache Spark with features like Photon, an optimized version of Spark written in C++. These improvements ensure better performance for Spark workloads.
3. **Scalability**:
   Databricks supports auto-scaling, which adjusts resources automatically based on workload demands. This ensures both high performance and cost-efficiency.
4. **Collaboration**:
   Databricks provides collaborative notebooks that support multiple languages like Python, R, SQL, and Scala. This makes it easy for teams of data engineers, scientists, and analysts to work together.
5. **Integration with Ecosystem**:
   Databricks integrates smoothly with various data sources and tools, making it highly versatile within existing data infrastructures.

By offering Spark as a service, Databricks allows organizations to use Apache Spark without worrying about the operational challenges. This enables faster and more efficient data processing and analytics.

Now, let's explore the different features of Databricks. But before that, let's look at Databricks pricing.

Databricks Pricing:

Databricks uses a flexible pricing model based on compute usage, measured in **Databricks Units (DBUs)**. Here's a closer look at how pricing works and how resources are managed:

- The cost depends on the type of cluster and workload.
- Users can optimize costs by efficiently managing cluster sizes and usage.

Databricks Pricing Model

A **Databricks Unit (DBU)** is a way to measure the processing power used in the Databricks platform. It is the foundation for how Databricks charges for its services. The DBU simplifies pricing by focusing on compute usage instead of the complexity of cloud infrastructure.

Key Characteristics of DBUs:

1. **Definition**:
    - A DBU represents the processing capacity per hour, based on the workload type and the cluster configuration running in Databricks.
2. **Granularity**:
    - DBU usage is billed per second, but it is typically calculated on an hourly basis.
3. **Workload-Specific Pricing**:
    - Different workloads (e.g., interactive analysis or batch processing) use varying amounts of DBUs, depending on the task's resource needs.
4. **Cluster Configuration Influence**:
    - The type and size of the virtual machine (VM) or instance running the cluster affect DBU usage. More powerful or larger instances consume more DBUs per hour.

DBU Costs Depend On:

- **Cloud Provider**: Costs differ for AWS, Azure, and Google Cloud.
- **Pricing Tier**: Options include Standard, Premium, and Enterprise.
- **Compute Type**: Costs vary based on whether the workload is interactive, job-based, or light.

How DBU Consumption Works

1. **Cluster Size**:
    - Larger clusters with more nodes consume more DBUs per hour.
    - Example:
        - A cluster with 4 worker nodes uses more DBUs than a cluster with 2 nodes.
2. **VM Type**:
    - The type of instance affects DBU consumption.
    - Powerful instances, such as compute-optimized or memory-optimized VMs, use more DBUs.
3. **Workload Duration**:
    - The longer a cluster runs, the more DBUs it consumes.
    - Using auto-termination features can help reduce unnecessary usage.

Example of DBU Calculation

1. **Cluster Configuration**:
    - **Driver Node**: Medium instance using 1 DBU/hour.
    - **Worker Nodes**: 4 workers, each using 2 DBUs/hour.
2. **DBU Consumption for 1 Hour**:
    - Driver: 1 DBU/hour.
    - Workers: 4 × 2 DBUs/hour = 8 DBUs/hour.
    - **Total**: 1 + 8 = 9 DBUs/hour.
3. **Cost Estimation**:
    - If 1 DBU costs $0.15, the total cost per hour = 9 × $0.15 = **$1.35/hour**.

Advantages of the DBU Model

1. **Transparency**:
    - Organizations can estimate costs easily by monitoring DBU usage for each workload.
2. **Flexibility**:
    - DBU pricing separates usage from specific infrastructure, making it simple to scale workloads.
3. **Control**:
    - Features like auto-scaling and auto-termination help reduce unnecessary DBU consumption.

Now, let's explore the infrastructure components of Databricks.

Storage Options in Databricks

1. **Cloud Object Storage Integration (Recommended)**:
    - Databricks works seamlessly with cloud storage services from major providers:
        - **AWS S3**: Direct access to S3 buckets using IAM roles or credentials.
        - **Azure Data Lake Storage (ADLS) & Azure Blob Storage**: For Azure users, integrates smoothly with ADLS and Blob Storage.
        - **Google Cloud Storage (GCS)**: Supports storing and retrieving data on GCS.
    - These cloud storage options serve as a "Data Lake" to store raw, processed, and curated data.
2. **Advantages**:
    - **Scalability**: Storage scales as your data grows.
    - **Cost-Effectiveness**: Pay-as-you-go pricing for cloud storage.
    - **Native Integrations**: Optimized connectors for efficient data read/write operations.
3. **Databricks File System (DBFS)**:
    - An abstraction layer that simplifies interaction with cloud storage by presenting it like a local file system.
    - Operates over your configured cloud storage account (e.g., S3, ADLS).
4. **Common Use Cases**:
    - Temporary storage for intermediate data during processing.
    - Easy access to external datasets for users.
5. **Delta Lake**:
    - A storage layer built on top of cloud storage that adds **ACID properties** and transactional consistency.
6. **Key Features**:
    - **Versioned Data**: Supports time travel and rollback.
    - **Schema Enforcement**: Maintains data integrity.
    - **Data Indexing**: Speeds up read/write operations.
7. **Local/On-Premises Storage**:
    - Databricks can connect to on-premises storage systems using protocols like NFS or SMB or via tools like HDFS.
    - Requires additional setup, such as secure networking and gateway configurations.
8. **Note**: Databricks is optimized for cloud environments, making cloud storage the preferred option.

Resource Managers in Databricks

Databricks employs various resource managers to allocate compute, memory, and storage resources efficiently across different cloud platforms. Here's an overview of the resource management strategies:

1. **Apache Spark Standalone Cluster Manager**
    - **Where Used**: Default on AWS and Azure for Databricks clusters.
    - **Description**:
        - Built into Apache Spark, it handles the distribution of driver and executor nodes.
        - Databricks clusters consist of:
            - **Driver Node**: Manages SparkContext and schedules tasks.
            - **Worker Nodes (Executors)**: Perform distributed computations and store intermediate results.
        - Containers (e.g., LXC) isolate driver and executor processes.
    - **How It Manages Resources**:
        - **Cluster Sizing**: Users define worker nodes and instance types during cluster creation.
        - **Auto-Scaling**: Automatically adjusts the number of workers based on workload demands.
        - **Auto-Termination**: Clusters shut down after inactivity to save costs.

2. **Kubernetes for Resource Management (GCP)**
    - **Where Used**: Databricks on Google Cloud Platform (GCP).
    - **Description**:
        - Utilizes Google Kubernetes Engine (GKE) for orchestrating resources.
        - Node pools in Kubernetes are configured as:
            - **Driver Nodes**: Run the Spark master process.
            - **Worker Nodes (Executors)**: Execute distributed tasks.
    - **Benefits**:
        - **Elasticity**: Scales node pools automatically based on workload.
        - **Efficiency**: Optimizes containerized workload allocation.
        - **Fault Tolerance**: Restarts failed nodes or containers, ensuring high availability.

---

3. **Databricks Serverless Resource Management**
    - **Where Used**: Available on AWS and Azure.
    - **Description**:
        - Abstracts cluster management, allowing Databricks to handle infrastructure provisioning, scaling, and termination.
        - Users simply run workloads without specifying instance types or configurations.
    - **How It Works**:
        - **Elastic Compute**: Resources are provisioned dynamically as needed.
        - **Cost Optimization**: Compute costs are incurred only during active job execution, with idle resources automatically released.
        - **Security**: Workloads run in isolated environments for data privacy.

---

4. **Integration with Cloud Providers' Resource Managers**
    - Databricks integrates with the resource management systems of underlying cloud platforms:
        - **AWS Auto Scaling Groups**: Adjusts EC2 instance pools for clusters dynamically.
        - **Azure VM Scale Sets (VMSS)**: Manages and scales Azure VM instances for clusters.
        - **GCP Managed Instance Groups (MIGs)**: Works with Kubernetes for VM provisioning.

---

5. **User-Defined Resource Management**
    - **Cluster Policies**:
        - Administrators set policies to control cluster size, instance types, and features for cost management and compliance.
    - **Task Parallelism**:
        - Configures the level of parallelism in Spark jobs to optimize resource usage.
    - **Caching and Persistence**:
        - Supports in-memory dataset caching to reduce redundant computations and improve performance.

Databricks' resource management approaches ensure flexibility, cost efficiency, and scalability across diverse workloads.

What Is Photon?

Photon is a high-performance, C++-based query execution engine developed by Databricks to replace the default Apache Spark execution engine (written in Scala and Java). It focuses on low-level optimizations to deliver significantly faster query performance, particularly for SQL and DataFrame workloads.

Key Features of Photon

1. **Vectorized Query Execution**
    - Processes multiple rows of data simultaneously using **SIMD (Single Instruction, Multiple Data)** operations.
    - This approach leverages modern CPUs to execute operations on data blocks efficiently, outperforming traditional row-by-row processing.
2. **Native Code Generation**
    - Utilizes **LLVM**, a compiler framework, to generate native machine code for query execution at runtime.
    - By avoiding JVM interpretation, Photon delivers performance close to hardware-level speed.
3. **Optimized for Modern Hardware**
    - Designed to leverage modern CPU architectures, including:
        - Multi-core processors.
        - Advanced instruction sets like **AVX-512**.
    - Efficient memory usage avoids bottlenecks like garbage collection, common in JVM-based engines.
4. **Support for ANSI SQL**
    - Fully compatible with **ANSI SQL**, making it ideal for SQL-based workloads and BI tools.
    - Supports advanced SQL operations, including complex joins, aggregations, and window functions.
5. **Seamless Integration with Delta Lake**
    - Tightly integrated with **Delta Lake**, enabling high-speed processing for structured, semi-structured, and unstructured data.
    - Enhances performance for Delta Lake features like:
        - **ACID transactions**.
        - **Schema enforcement**.
        - **Data versioning**.
6. **Out-of-the-Box Compatibility**
    - Automatically enabled on supported clusters without requiring workload or configuration changes.

Photon optimizes Databricks for modern data processing needs, delivering faster query execution and better hardware utilization.

Performance Improvements

Photon significantly enhances performance for SQL and BI workloads through the following:

1. **Speed Improvements**
    - Queries run **2x–8x faster** compared to the default Spark SQL engine.
    - The boost is most noticeable for operations like aggregations, joins, and filtering.
2. **Lower Cost**
    - Faster execution reduces resource usage, lowering overall compute costs for the same workload.
3. **Scalability**
    - Photon handles large datasets efficiently, ensuring high throughput and smooth scalability for analytics workloads.

How Photon Works

1. **Execution Flow**
    - Photon converts SQL queries or DataFrame operations into an **optimized execution plan**.
    - This plan is compiled into native machine code using **LLVM**, bypassing the need for JVM interpretation.
    - Queries are executed in a **vectorized manner**, processing batches of data in parallel for faster performance.
2. **Interoperability with Spark**
    - Photon integrates seamlessly with existing **Spark infrastructure** in Databricks.
    - For unsupported tasks (e.g., UDFs or specific machine learning operations), Photon automatically falls back to the standard Spark engine.

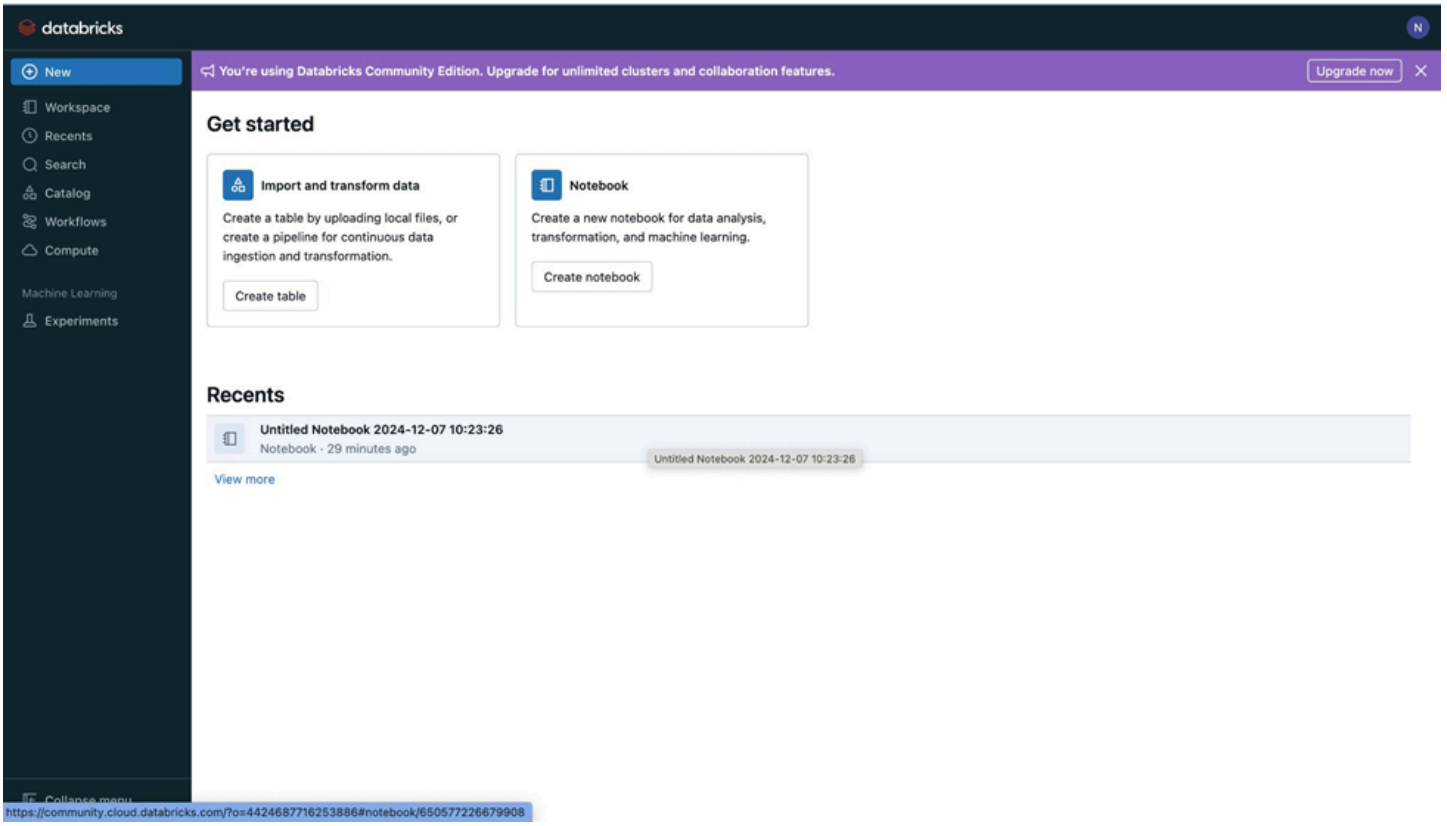
With these features, Photon accelerates workloads, reduces costs, and ensures flexibility for various analytics tasks.

Live Demo: Creating a Databricks Community Edition Account

The Databricks Community Edition is a free version of the platform designed for learning and experimentation. Follow these simple steps to create an account:

Steps to Create a Databricks Community Edition Account

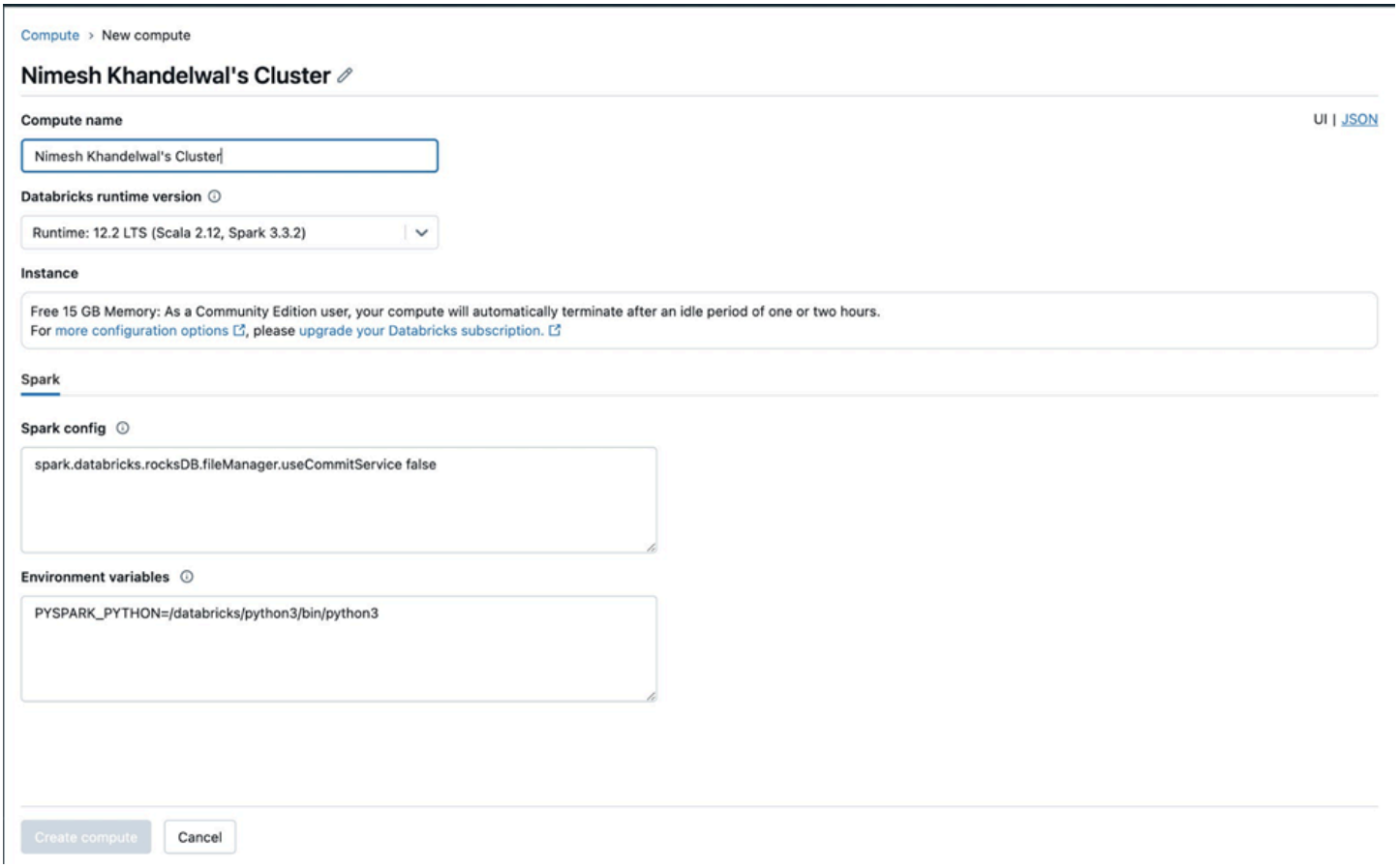
1. **Access the Signup Page**
    - Open your browser and go to the Databricks Community Edition Signup page.
2. **Complete the Signup Form**
    - **First Name**: Enter your first name.
    - **Last Name**: Enter your last name.
    - **Email Address**: Provide a valid email address (use personal or professional, not temporary).
    - **Password**: Choose a secure password.
    - **Role/Title**: Optionally specify your role (e.g., Data Engineer, Data Scientist).
    - **Organization**: Optionally enter your organization's name.
3. **Agree to Terms**
    - Check the box to accept the Databricks Terms of Service and Privacy Policy.
4. **Click "Get Started for Free"**
    - Submit the form by clicking the button.
5. **Verify Your Email**
    - Check your inbox for a verification email and click the link to confirm your email address.
6. **Log In**
    - Use your registered email and password to log in to the Databricks Community Edition.

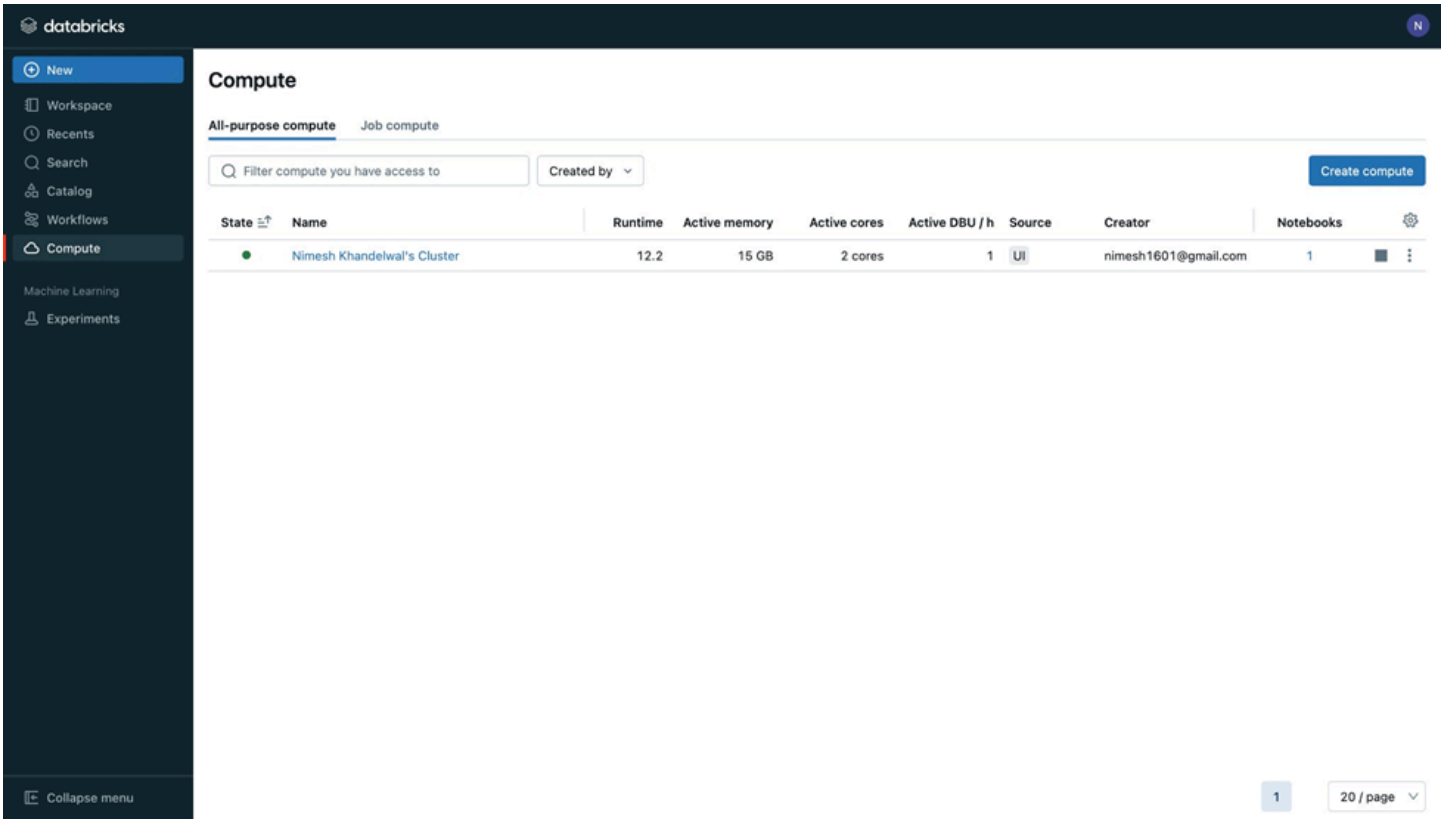


Steps to Create a Compute Cluster

1. **Navigate to the Compute Page**
   - After logging in, go to the **Compute** section in the Databricks workspace.
2. **Create a Cluster**
   - Click the **Create Cluster** button.
3. **Configure the Cluster**
   - **Cluster Name**: Enter a name (e.g., "My First Cluster").
   - **Databricks Runtime Version**: Use the default runtime version pre-configured for the Community Edition.
   - **Node Type**: The Community Edition uses a small, single-node configuration by default.
   - **Autotermination**: Set the cluster to terminate after a period of inactivity (default is 120 minutes).
   - Other settings are locked in the Community Edition and cannot be modified.
4. **Launch the Cluster**
   - Click **Create Cluster** at the bottom of the form.
5. **Wait for Initialization**
   - The cluster will transition from **Pending** to **Running** within a few moments.
   - Once running, your cluster is ready for use.

This setup allows you to explore Databricks' features, including running notebooks and analyzing datasets.





Cluster Management Tabs Overview

Databricks provides various tabs to manage and monitor clusters efficiently. Here's a summary of each tab and its purpose:

1. Configuration Tab

- **Purpose**: View and modify cluster settings.
- **Key Sections**:
    - **Cluster Name**: Displays the cluster's name.
    - **Cluster Mode**: Indicates the mode (e.g., Single-Node for Community Edition, Standard/High Concurrency for paid editions).
    - **Databricks Runtime Version**: Shows runtime details, including Spark version and pre-installed libraries.
    - **Node Type**: Displays the type/size of driver and worker nodes.
    - **Number of Workers**: Shows the worker configuration (fixed in Community Edition).
    - **Autotermination Settings**: Configures the idle timeout period before shutdown.
    - **Advanced Options**: Lets you modify Spark settings, environment variables, and init scripts (unavailable in Community Edition).

---

2. Spark UI Tab

- **Purpose**: Access the Apache Spark Web UI for cluster performance monitoring.
- **Key Features**:
    - **Jobs**: Lists Spark jobs with status, execution time, and stages.
    - **Stages**: Provides task-level insights, including shuffle read/write and durations.
    - **Storage**: Displays memory and disk usage of RDDs.
    - **Environment**: Shows Spark properties, environment variables, and JVM settings.
    - **Executors**: Details driver/executor metrics like memory usage and task completion.

---

3. Metrics Tab

- **Purpose**: Visualize resource usage and performance metrics.
- **Key Metrics**:
    - **CPU Usage**: Tracks CPU utilization over time for cluster nodes.
    - **Memory Usage**: Displays memory consumption by driver and workers.
    - **Disk I/O**: Monitors read/write operations to disk.
    - **Network I/O**: Tracks network throughput for data movement.

---

4. Libraries Tab

- **Purpose**: Manage libraries installed on the cluster.
- **Key Actions**:
    - **View Installed Libraries**: Lists pre-installed libraries (e.g., Spark, MLlib).
    - **Install New Libraries**: Allows installation of libraries from PyPI, Maven, or local files (not available in Community Edition).
    - **Library Status**: Displays installation status and issues.

5. Event Log Tab

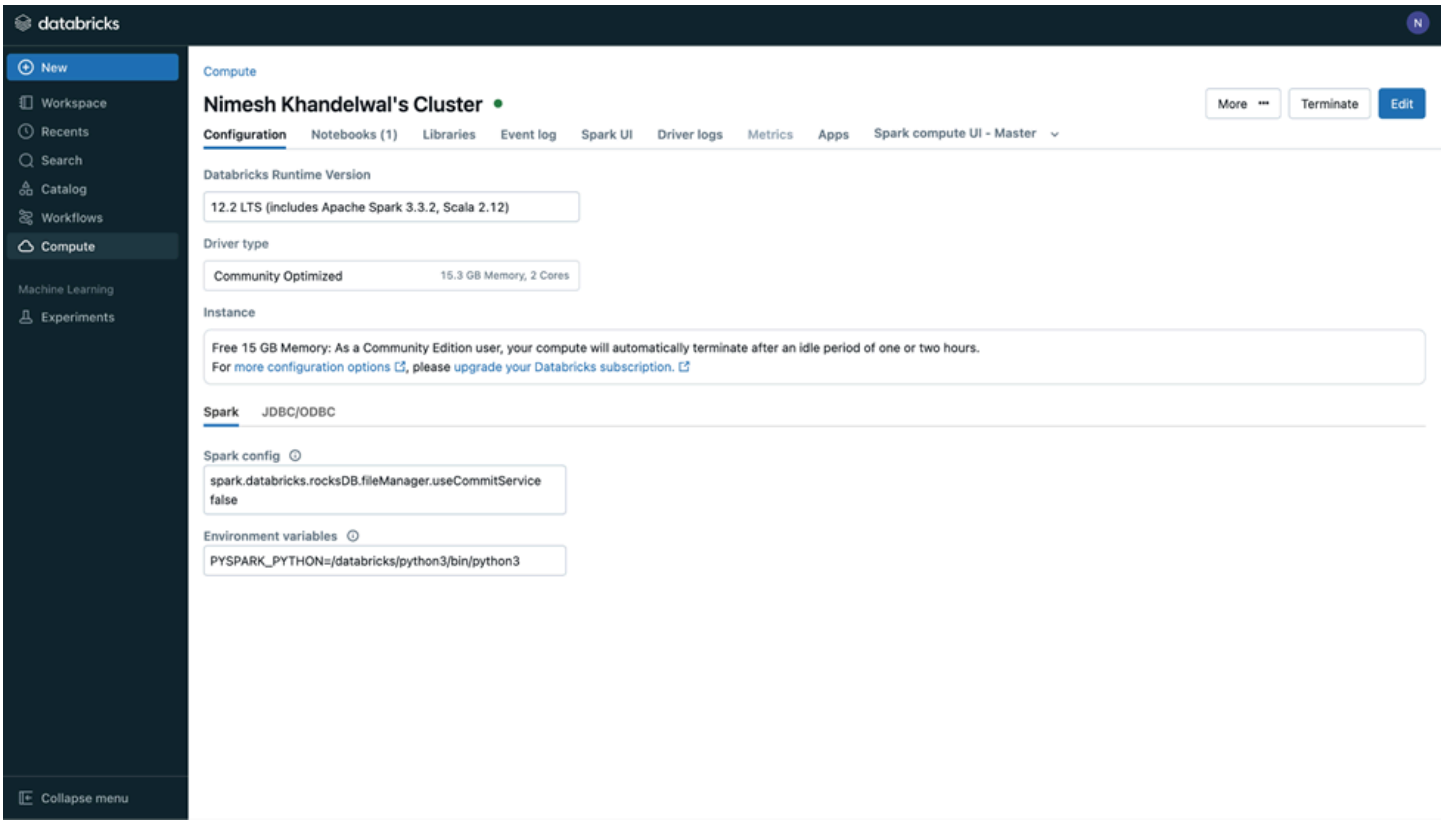
- **Purpose**: Troubleshoot and audit cluster operations using event logs.
- **Key Details**:
    - **Cluster Creation Events**: Logs cluster creation and termination activities.
    - **Autoscaling Events**: Tracks changes in worker nodes (available for paid editions).
    - **Error Messages**: Displays warnings and errors during cluster operations.

---

6. Driver Logs Tab

- **Purpose**: Debug and analyze driver-related performance issues.
- **Key Details**:
    - **Output Logs**: Captures the driver's process output.
    - **Error Logs**: Displays stack traces and errors.
    - **Streaming Logs**: Monitors logs for streaming applications.

This structure helps users efficiently monitor and manage their clusters for optimal performance.



Steps to Create a Table in Databricks

Step 1: Prepare Your Data

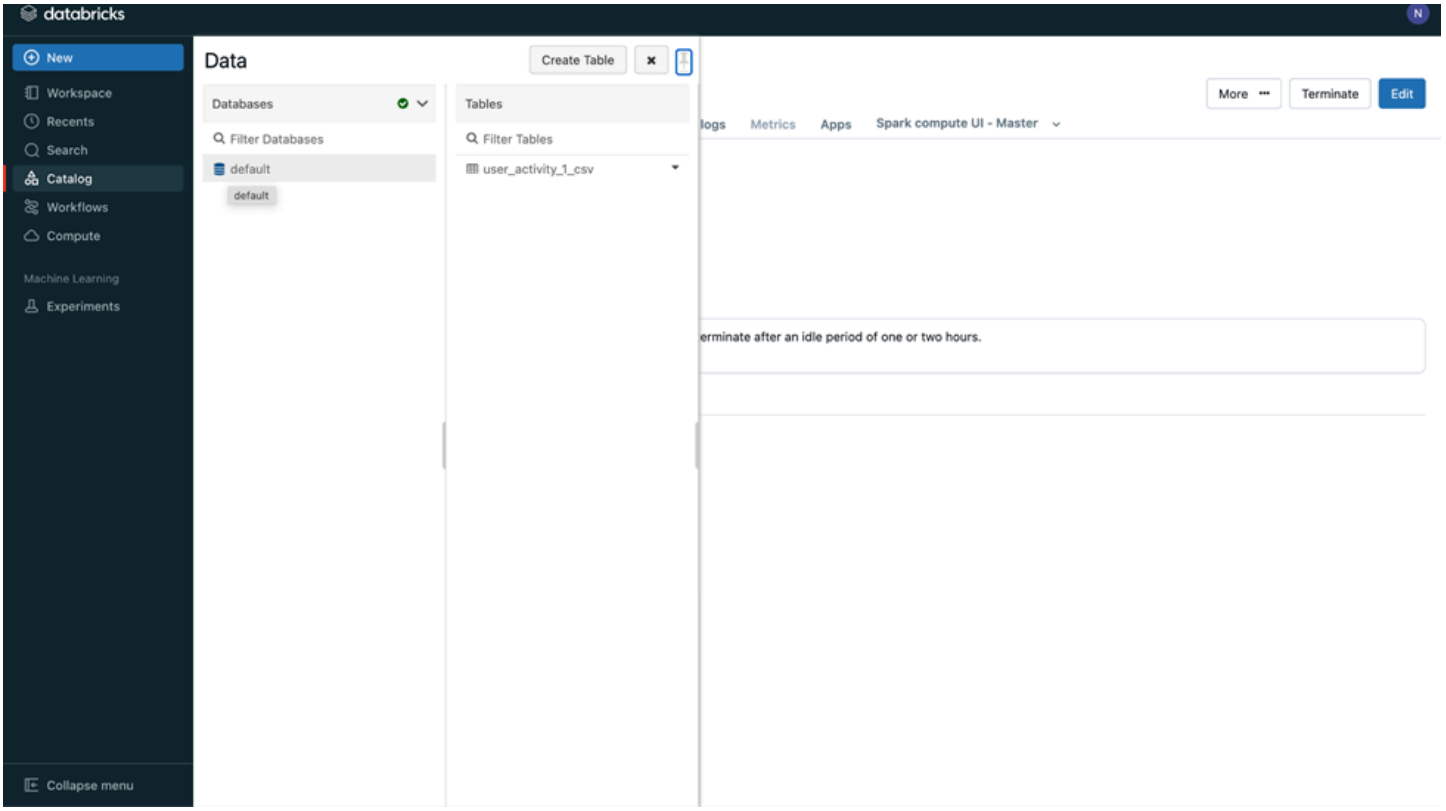
- Ensure your data is in a supported format (e.g., CSV, JSON, Parquet, Delta, ORC, etc.).
- Data can be stored in:
    - Your local machine
    - Cloud storage (e.g., AWS S3, Azure Blob, Google Cloud Storage)
    - Databricks File System (DBFS)

Step 2: Upload Data to Databricks (Optional)

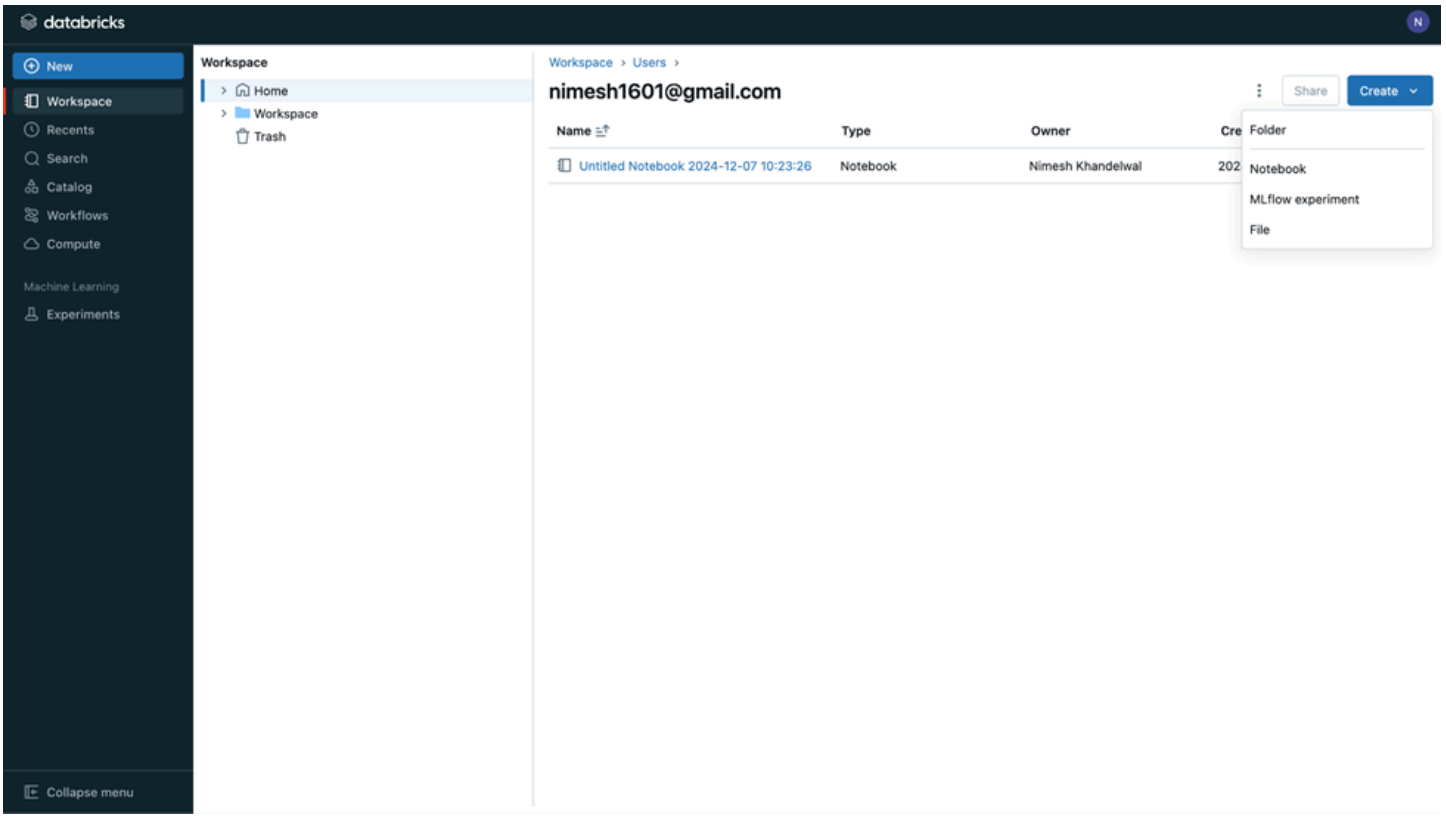
- **Upload from Local Machine**:
  - In Databricks Workspace, go to the **Data** tab in the sidebar.
  - Click **Create Table**.
  - Under **Upload File**, click **Browse** and select your data file.
  - Databricks will upload the file to DBFS.
- **Use External Storage**:
  - If your data is in external storage, configure access (e.g., mount S3 buckets or Azure Blob storage to DBFS).

Step 3: Create a Table

1. **Navigate to the Data Tab**:
   - Open the **Data** section in the sidebar.
2. **Click Create Table**:
   - You will have options to upload data or connect to external storage.
3. **Choose Your Data Source**:
   - Select **Upload File** for local data.
   - Choose **External Storage** for data in S3, ADLS, or GCS.
4. **Configure Table Options**:
   - **File Type**: Select the file format (CSV, JSON, etc.).
   - **Infer Schema**: Enable Databricks to automatically detect column names and types.
   - **Table Name**: Specify the name of the table.
   - **Database Name** (Optional): Choose a database if organizing tables.
   - **Path**: Define the location for table metadata and data.
5. **Preview the Data**:
   - Databricks will generate a preview to verify the data structure.
6. **Create Table**:
   - Click **Create Table** to complete the process.



Creating a Workspace



Steps to Create a Notebook in Databricks

Step 1: Choose the Location

- Navigate to the folder where you want to create the notebook.
- You can use the default folder or create a new one by right-clicking and selecting **Create Folder**.

Step 2: Create the Notebook

- Click the **Create** button at the top right of the Workspace panel.
- From the dropdown menu, select **Notebook**.

Step 3: Configure the Notebook

- **Name**: Provide a name for your notebook (e.g., "My First Notebook").
- **Default Language**: Choose the default language for the notebook:
  - Python
  - Scala
  - SQL
  - R

    (Note: You can switch languages in specific cells using magic commands like `%python`, `%sql`, etc.)
- **Cluster**: Select the cluster you want to attach to the notebook. If no cluster is running, create or start one from the **Compute** tab.

Step 4: Click "Create"

- After configuring the settings, click **Create** to open the new notebook.

Step 5: Use the Notebook

- **Interface Overview**:
  - **Cells**: The notebook consists of cells where you can write code or markdown.
  - **Toolbar**: Includes options to run all cells, clear results, and more.
  - **Cluster Selection**: Ensure the notebook is attached to a running cluster (shown at the top).
- **Add and Run Code**:
  - Write your code in a cell (e.g., for Python: `print("Hello, Databricks!")`).
  - Click the **Play** button next to the cell or press **Shift + Enter** to execute.
- **Add Markdown for Documentation**:
  - Use `%md` to create markdown cells. For example:

    ```
    %md # Welcome to Databricks This is my first notebook.
    ```

Step 6: Save and Organize

- **Saving**:
  - Notebooks are auto-saved in Databricks.
- **Organize**:
  - Move notebooks to folders by right-clicking the notebook name and selecting **Move**.

Tips for Working with Databricks Notebooks

1. **Switch Languages in Cells**:
   - Use magic commands like `%python`, `%sql`, `%scala`, or `%r` to switch between languages in different cells.
   - Example for SQL:

     ```
     %sql SELECT * FROM my_table;
     ```
2. **Visualization**:
   - Use built-in visualization options for SQL queries by clicking the **chart icon**.
3. **Collaborate**:
   - Share the notebook with team members by clicking the **Share** button.
4. **Version Control**:
   - Use **Revision History** to track changes and revert to earlier versions.

==========

Designing Class and Schema Diagrams for Splitwise: A Step-by-Step Guide

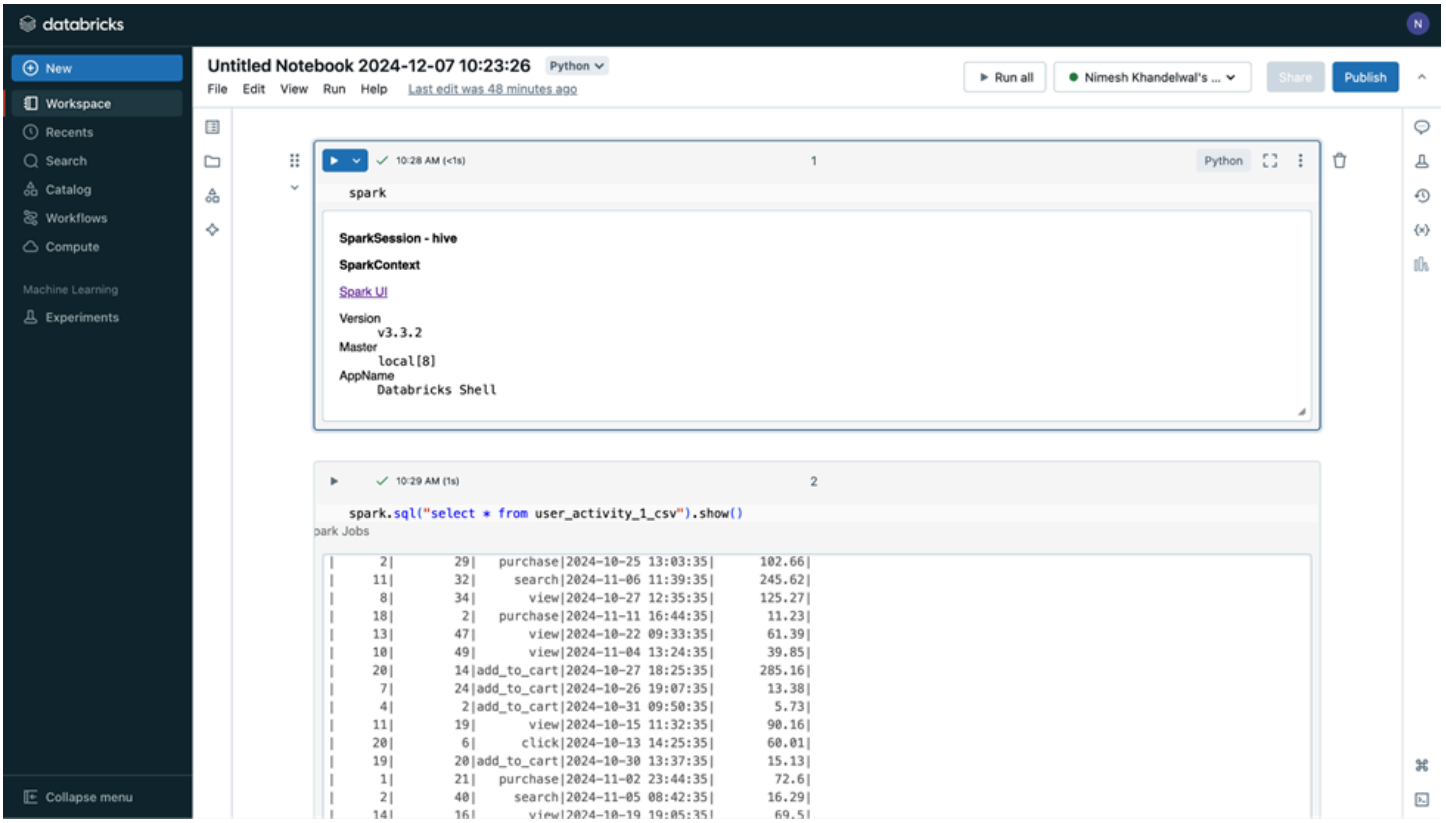
Visit

https://youtu.be/NGsenhTCMoA

Like | Comment | Subscribe

==========| **@GeekySanjay** |==========

==========



Snowflake Hands on - Dec 12

- **Data Storage and Management**
- **Pricing in Snowflake**
- **Snowflake Editions**
- **Cost Management**
- **Internals of Snowflake**
- **Data Processing in Snowflake**
- **Benefits of Snowflake's Architecture**

Snowflake is a cloud-based platform for data warehousing that helps organizations store, manage, and analyze large amounts of data effectively. It is built specifically for the cloud, making it different from traditional data warehouses. Snowflake provides features for better scalability, performance, and ease of use. Here's a summary of its main functionalities:

1. Data Storage and Management:

- **Structured and Semi-Structured Data Support:** Snowflake lets you store and query both structured data (like CSV files) and semi-structured data (like JSON, Avro, and Parquet) in one place.
- **Micro-Partitioning:** Data is automatically divided into small, compressed micro-partitions to improve query speed and storage efficiency.
- **TimeTravel:** This feature allows you to access older versions of your data, making it possible to view or restore data from specific times in the past for recovery and auditing.
- **Zero-Copy Cloning:** You can quickly create clones of databases, schemas, or tables without copying the actual data. This is useful for testing and development.

2. Data Loading and Integration:

- **Bulk Data Loading:** Snowflake supports loading large datasets from sources like Amazon S3, Azure Blob Storage, and Google Cloud Storage.
- **Continuous Data Ingestion (Snowpipe):** This feature loads data in real time by detecting and ingesting new files automatically.
- **Data Transformation:** You can perform SQL-based transformations and integrate with other tools for advanced data processing.

3. Querying and Analytics:

- **SQL Support:** Snowflake supports standard and advanced SQL for tasks like data definition (DDL), data manipulation (DML), and complex queries with window functions and aggregates.
- **Materialized Views:** Precomputed query results are stored to make repetitive queries faster.
- **Stored Procedures and User-Defined Functions (UDFs):** You can write business logic directly in the database using SQL or other languages like JavaScript and Python.

4. Performance Optimization:

- **Automatic Clustering:** Snowflake manages data clustering automatically, improving query performance without manual effort.
- **Result Caching:** Query results are saved to speed up repeated executions of the same query and reduce costs.

5. Security and Governance:

- **Data Encryption:** Ensures data security with encryption for data at rest and in transit.
- **Access Controls:** Role-based access control and multi-factor authentication protect user access.
- **Data Masking and Classification:** Protects sensitive data by masking and classifying it to meet compliance standards.

6. Data Sharing and Collaboration:

- **Secure Data Sharing:** Share data securely across Snowflake accounts without copying or moving it.
- **Data Marketplace:** Discover and share datasets through a marketplace for better collaboration.

7. Development and Extensibility:

- **Snowpark:** A developer framework for writing code in Java, Python, or Scala, which runs within Snowflake's processing engine.
- **Streamlit Integration:** Run Streamlit apps directly in Snowflake to build and share custom tools for machine learning and data science.

8. Monitoring and Management:

- **Snowsight:** A web-based interface to manage resources, monitor system usage, and query data, giving insights into performance.
- **Resource Monitors:** Set up monitors to track and control credit usage for cost and resource management.

Pricing in Snowflake

Snowflake uses a **consumption-based pricing model** with these key components: **compute resources**, **data storage**, **data transfer**, and **cloud services**. Understanding each helps manage costs effectively.

1. Compute Resources (Virtual Warehouses):

Compute costs are measured in **Snowflake credits**, based on the size and usage time of virtual warehouses. These warehouses come in different sizes, with credits consumed per hour as follows:

Warehouse Size	Credits per Hour
X-Small	1
Small	2
Medium	4
Large	8
X-Large	16
2X-Large	32
3X-Large	64
4X-Large	128
5X-Large	256
6X-Large	512

- **Billing:** Charged per second, with a minimum of 60 seconds per usage instance.
- **Benefit:** You pay only for the compute time you use, offering precise cost control.

2. Data Storage:

Storage costs depend on the **average terabytes (TB)** of data stored per month. Rates vary by **cloud provider** and **region**.

- Example: In U.S.-based AWS regions, storage costs around **$23 per TB per month**.

3. Data Transfer:

- **Data Ingress (incoming data):** Free of charge.
- **Data Egress (outgoing data):** Charged based on:
    - Source and destination regions.
    - Cloud providers involved.

4. Cloud Services:

The cloud services layer manages tasks like **authentication**, **metadata management**, and **query optimization**.

- **Charges:** Credits are used, but Snowflake applies a **fair-use policy**:
    - If usage stays below **10% of daily compute credits**, no extra charge is applied.
    - Only usage above the 10% threshold is billed.

Pricing Models:

1. **On-Demand Pricing:**
    - Pay for storage and compute based on actual usage.
    - Billing happens monthly.
    - Flexible option for unpredictable workloads.
2. **Capacity Pricing:**
    - Commit upfront to a specific usage level for discounts.
    - Ideal for predictable workloads, saving costs over time.

Snowflake Editions:

Snowflake offers different editions with unique features and pricing:

- **Standard Edition:** Includes basic features for general needs.
- **Enterprise Edition:** Adds advanced features like multi-cluster warehouses and extended time travel.
- **Business Critical Edition:** Provides extra security and compliance features.
- **Virtual Private Snowflake (VPS):** Offers dedicated infrastructure for maximum security and performance.

Each edition has a different credit cost, reflecting the value of its features.

Cost Management

Snowflake provides tools to help manage and optimize costs effectively, including:

- **Resource Monitors:** Track consumption, set thresholds, and receive alerts.
- **Detailed Usage Views:** Gain insights into usage patterns to adjust operations.

These tools enable organizations to align usage with operational needs and budget constraints, ensuring proactive cost control.

Internals of Snowflake

Snowflake is a **cloud-based data warehousing platform** designed for scalable and efficient data storage and processing. Its architecture separates **storage**, **compute**, and **cloud services**, offering high flexibility and performance.

Data Storage in Snowflake

The centralized storage layer organizes data into **databases**, **schemas**, and **tables**, handling both **structured** and **semi-structured data formats** (e.g., JSON, Avro, Parquet).

Key Features:

1. **Micro-Partitioning:** Automatically divides data into optimized partitions for efficient retrieval and query performance.
2. **Columnar Storage:** Data is stored in columns, enhancing compression and speeding up analytical queries.
3. **Automatic Clustering:** Snowflake manages clustering automatically, ensuring optimal organization without manual intervention.

Data Processing in Snowflake

The compute layer consists of **virtual warehouses**, which are independent clusters performing data processing tasks.

Key Features:

1. **Elastic Scalability:**
    - Resize or suspend warehouses dynamically to match workload needs.
    - Storage and compute are independent, allowing flexibility.
2. **Concurrency Handling:**
    - Multiple warehouses process diverse workloads simultaneously without contention.
3. **Support for Multiple Programming Languages:**
    - Snowflake's **Snowpark** framework enables writing data processing code in **Python**, **Java**, and **Scala**, executed directly within Snowflake's engine.

Snowflake Cloud Services Layer

This layer manages essential platform operations, including:

- **Authentication**
- **Metadata Management**
- **Query Optimization**
- **Access Control**

By orchestrating interactions between storage and compute, the cloud services layer ensures seamless operation and a cohesive user experience.

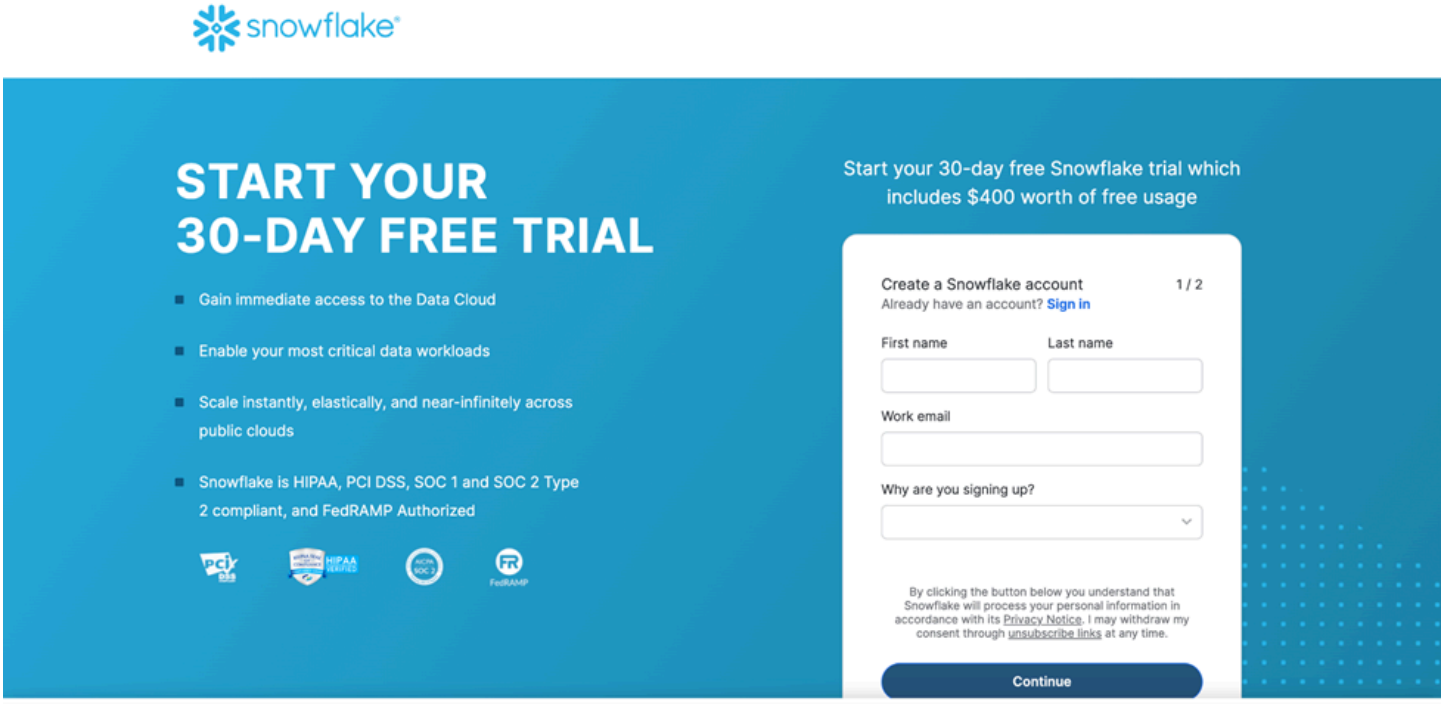
Benefits of Snowflake's Architecture

Snowflake's separation of **storage** and **compute** resources provides a flexible and efficient environment for:

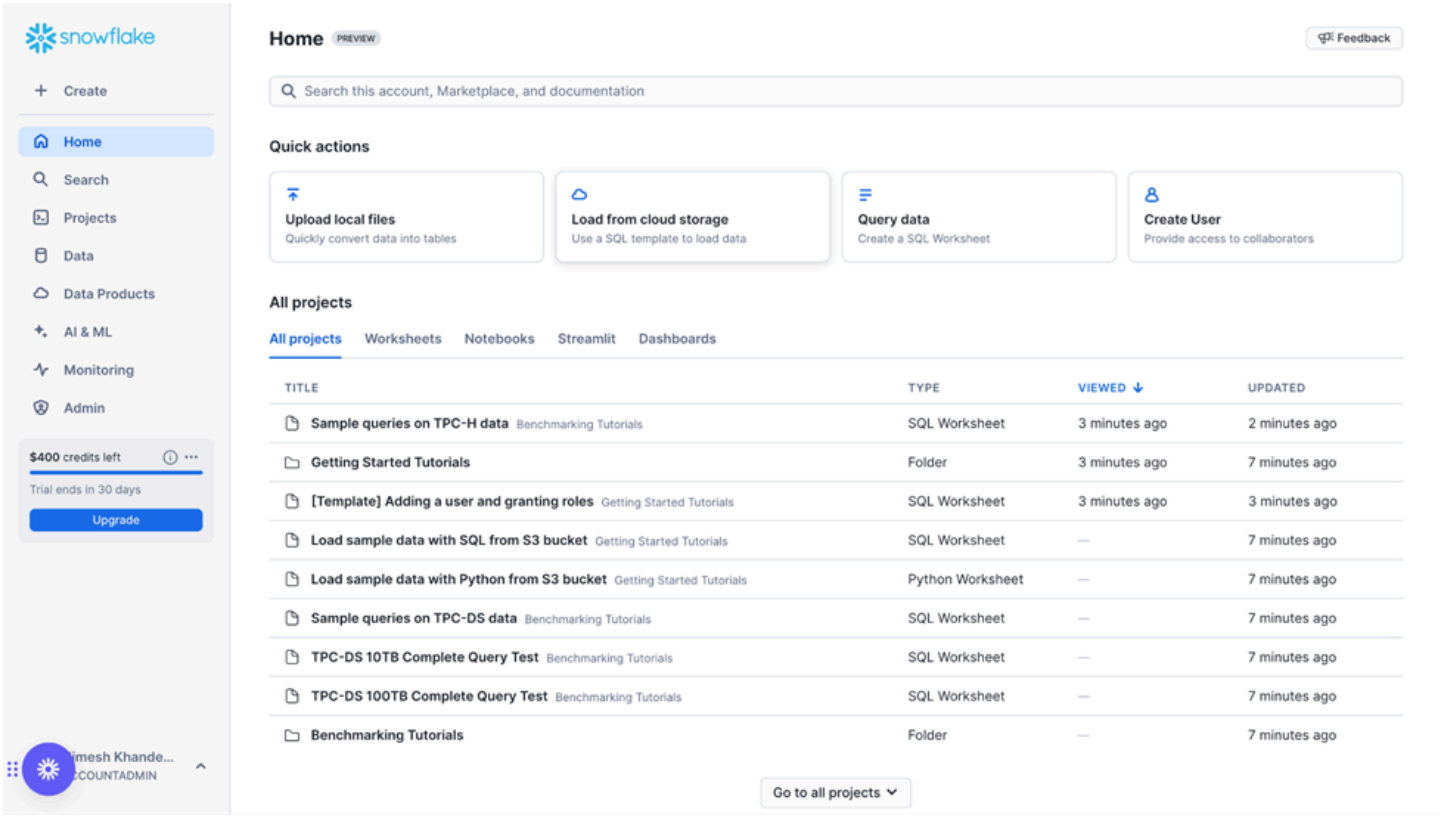
- Scalable **data storage**.
- High-performance **data processing**.
- Supporting a wide range of **analytical** and **operational workloads**.

Live Demo: Sign Up

1. **Visit the Snowflake Trial Page**
    - Navigate to the Snowflake trial signup page.
2. **Complete the Registration Form**
    - Provide your name, email, company, and other required details.
3. **Select Your Cloud Provider and Region**
    - Choose from **AWS**, **Azure**, or **Google Cloud**, and specify the closest region.
4. **Activate Your Account**
    - Check your email for an activation link. Follow the instructions to set up a password and access your account.

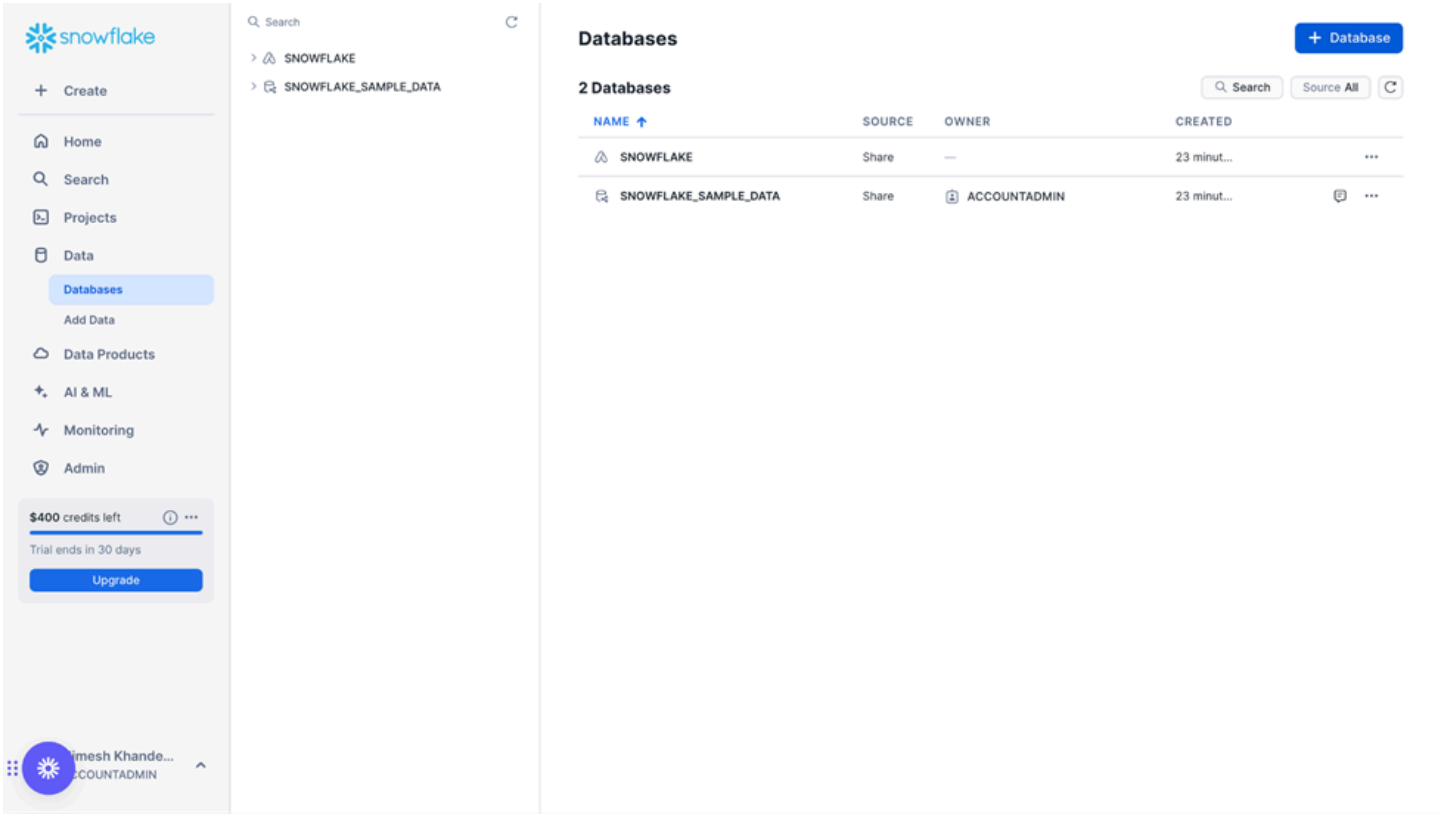


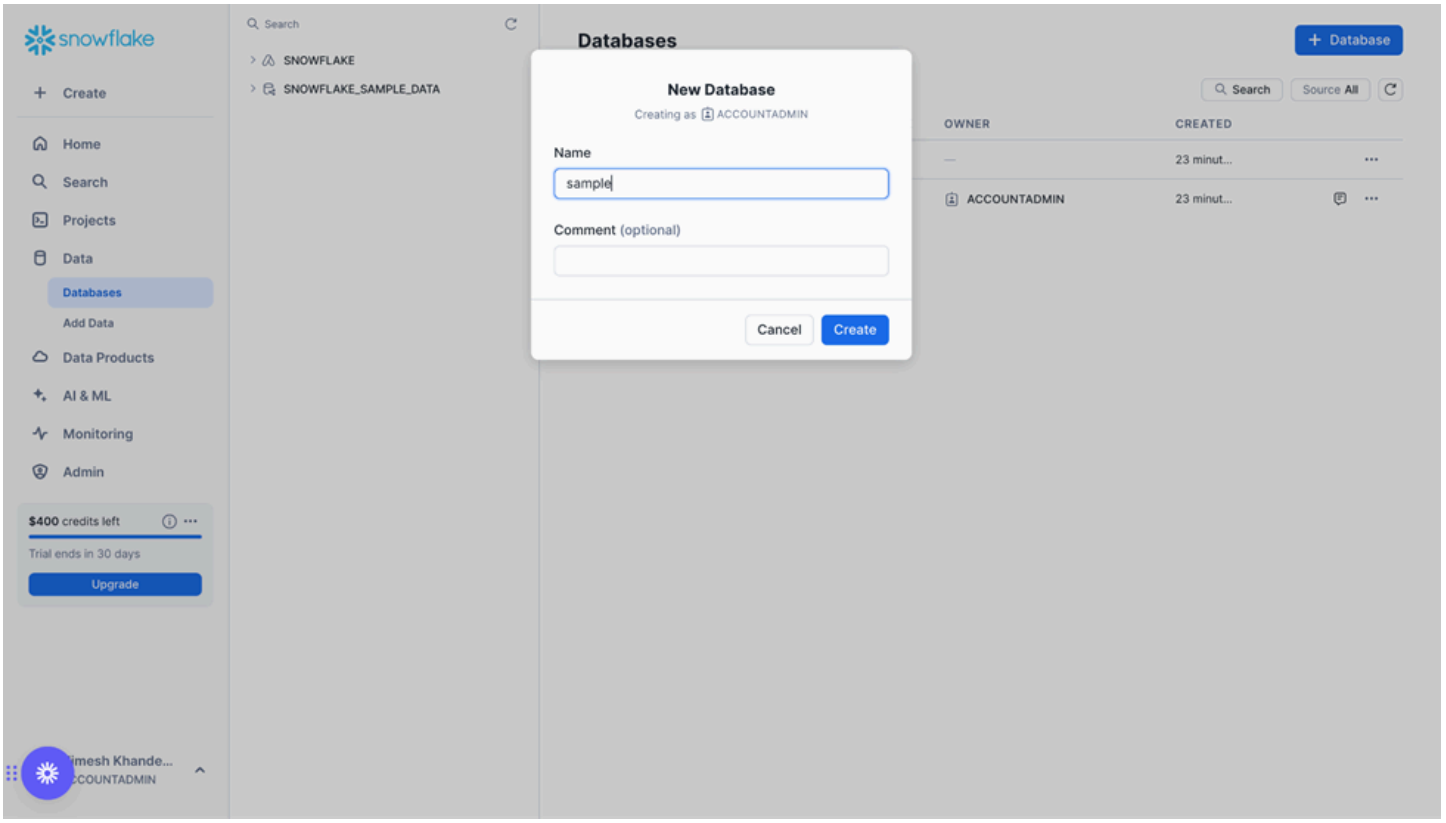
Home Screen

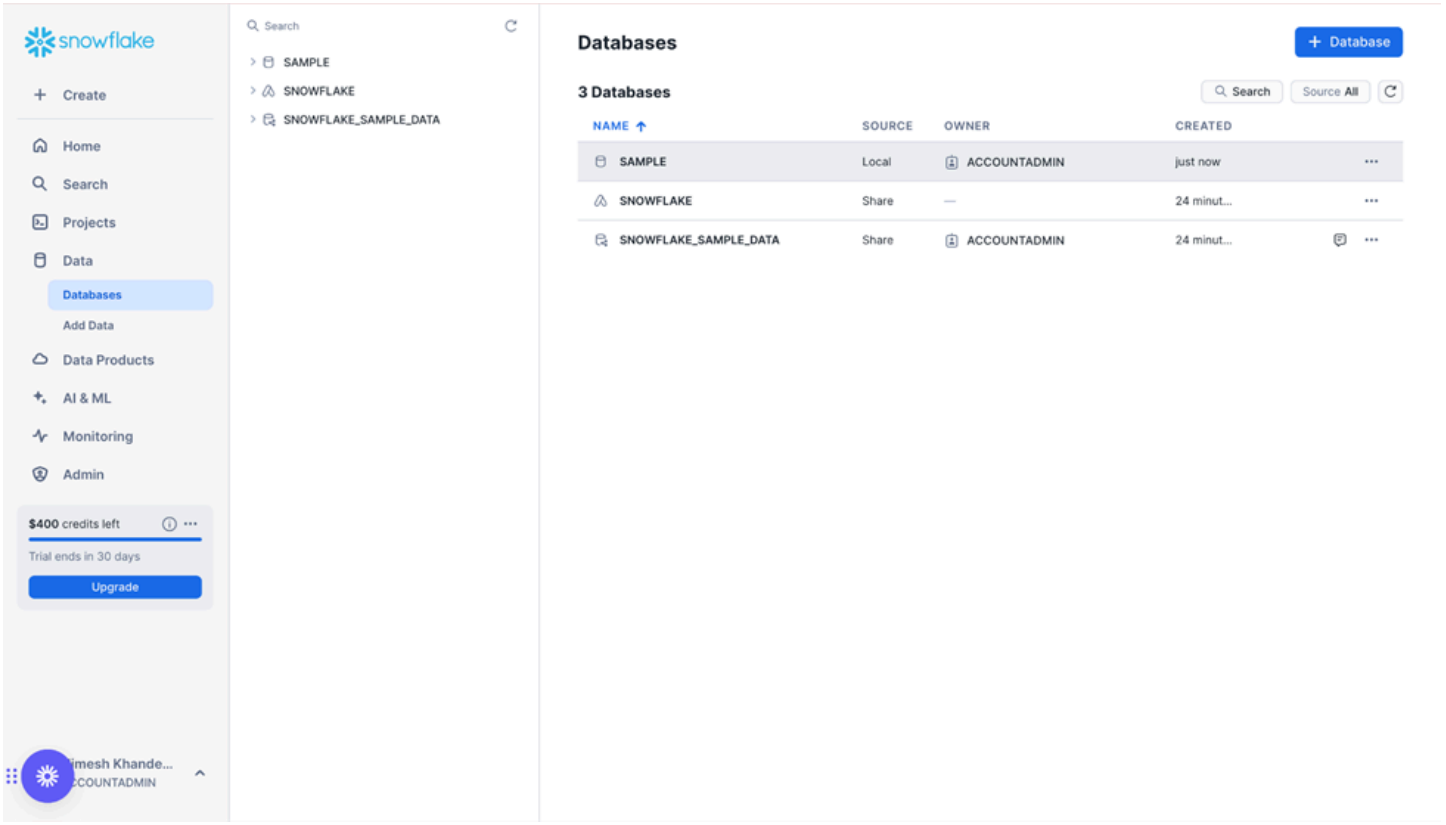


Home Screen: Create a Database

1. **Navigate to the Databases Section**
   - On the left-hand navigation pane, under the **Data** tab, click **Databases**.
2. **Create a New Database**
   - Click **Create** and provide the following details:
     - **Database Name:** A unique name (e.g., `sample`).
     - **Comment (Optional):** Add a description of the database.
3. **Save the Database**
   - Click **Create** to finalize. The database will appear in the list.

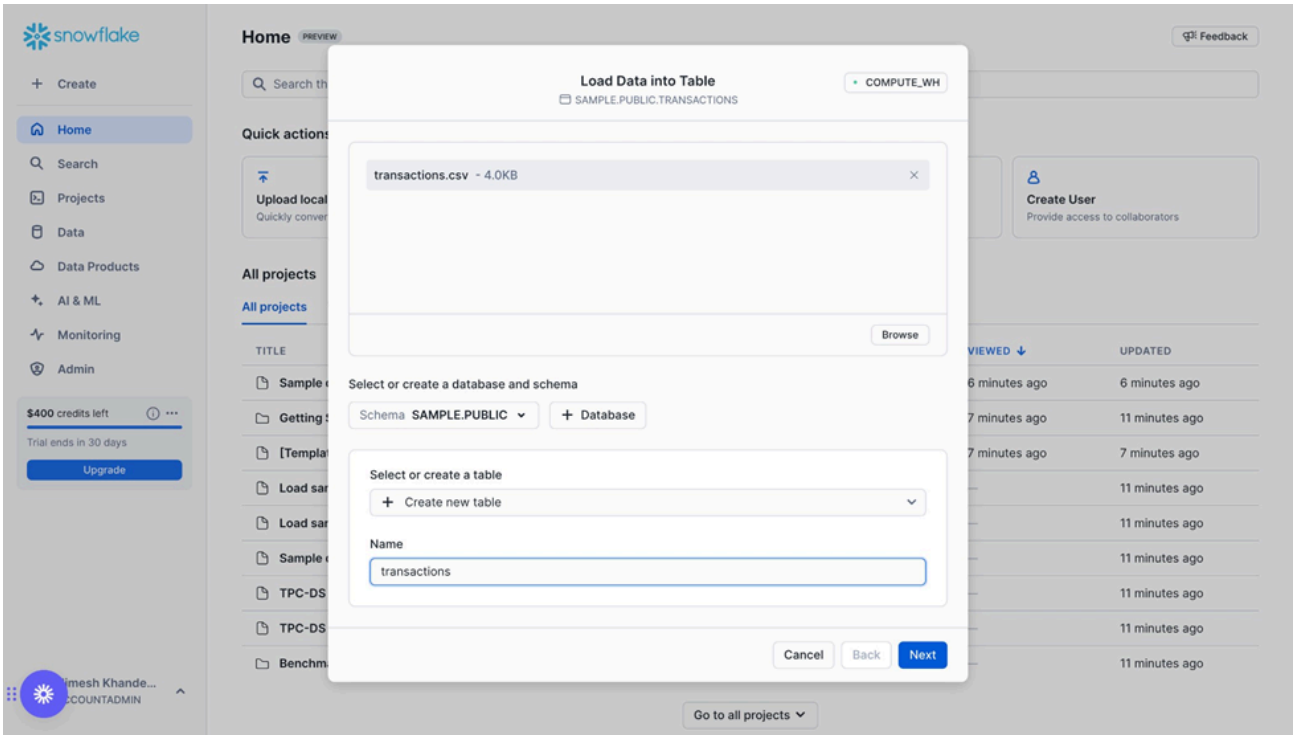

snowflake
Create
Home
Search
Projects
Data
Databases
Add Data
Data Products
AI & ML
Monitoring
Admin
$400 credits left
Trial ends in 30 days
Upgrade
Search
SNOWFLAKE
SNOWFLAKE_SAMPLE_DATA
Databases
+ Database
2 Databases
Search
Source All
NAME
SOURCE
OWNER
CREATED
SNOWFLAKE
Share
23 minut...
SNOWFLAKE_SAMPLE_DATA
Share
ACCOUNTADMIN
23 minut...

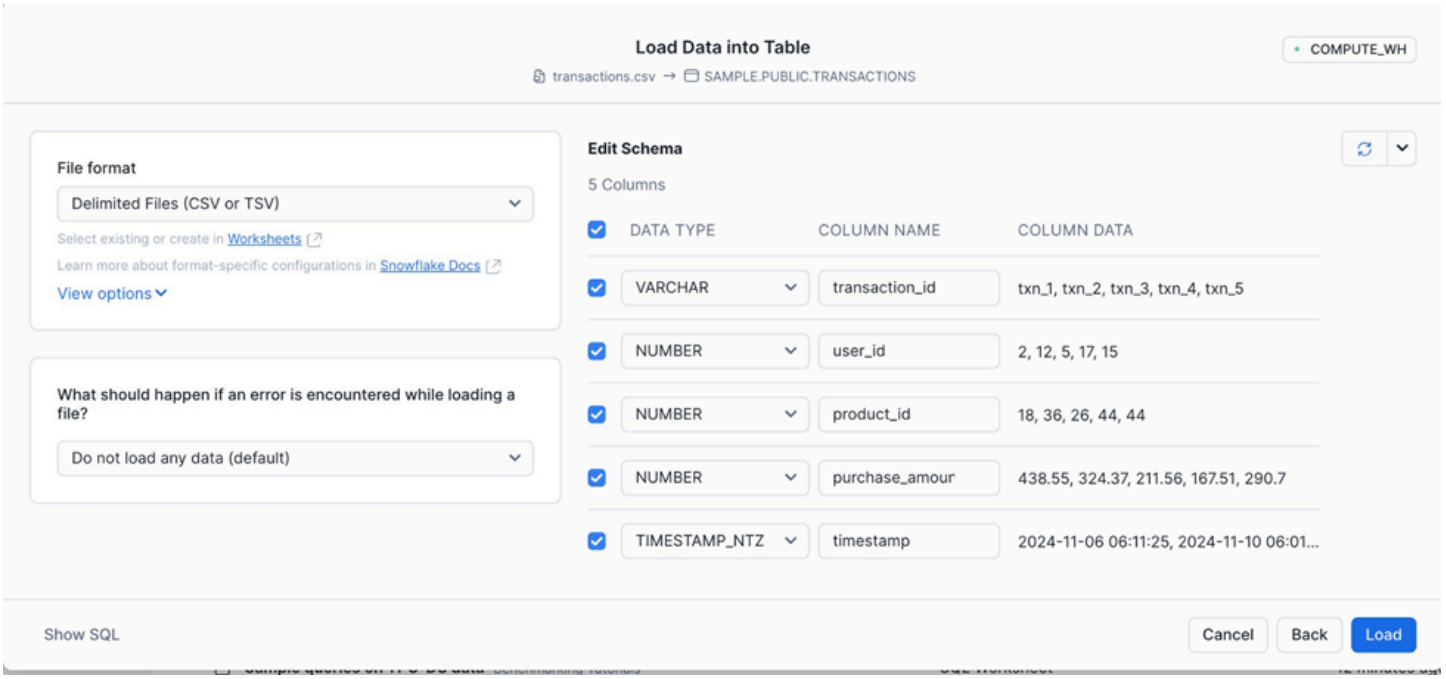

snowflake
Create
Home
Search
Projects
Data
Databases
Add Data
Data Products
AI & ML
Monitoring
Admin
$400 credits left
Trial ends in 30 days
Upgrade
Search
SNOWFLAKE
SNOWFLAKE_SAMPLE_DATA
Databases
+ Database
Search
Source All
OWNER
CREATED
23 minut...
ACCOUNTADMIN
23 minut...
New Database
Creating as ACCOUNTADMIN
Name
sample
Comment (optional)
Cancel
Create



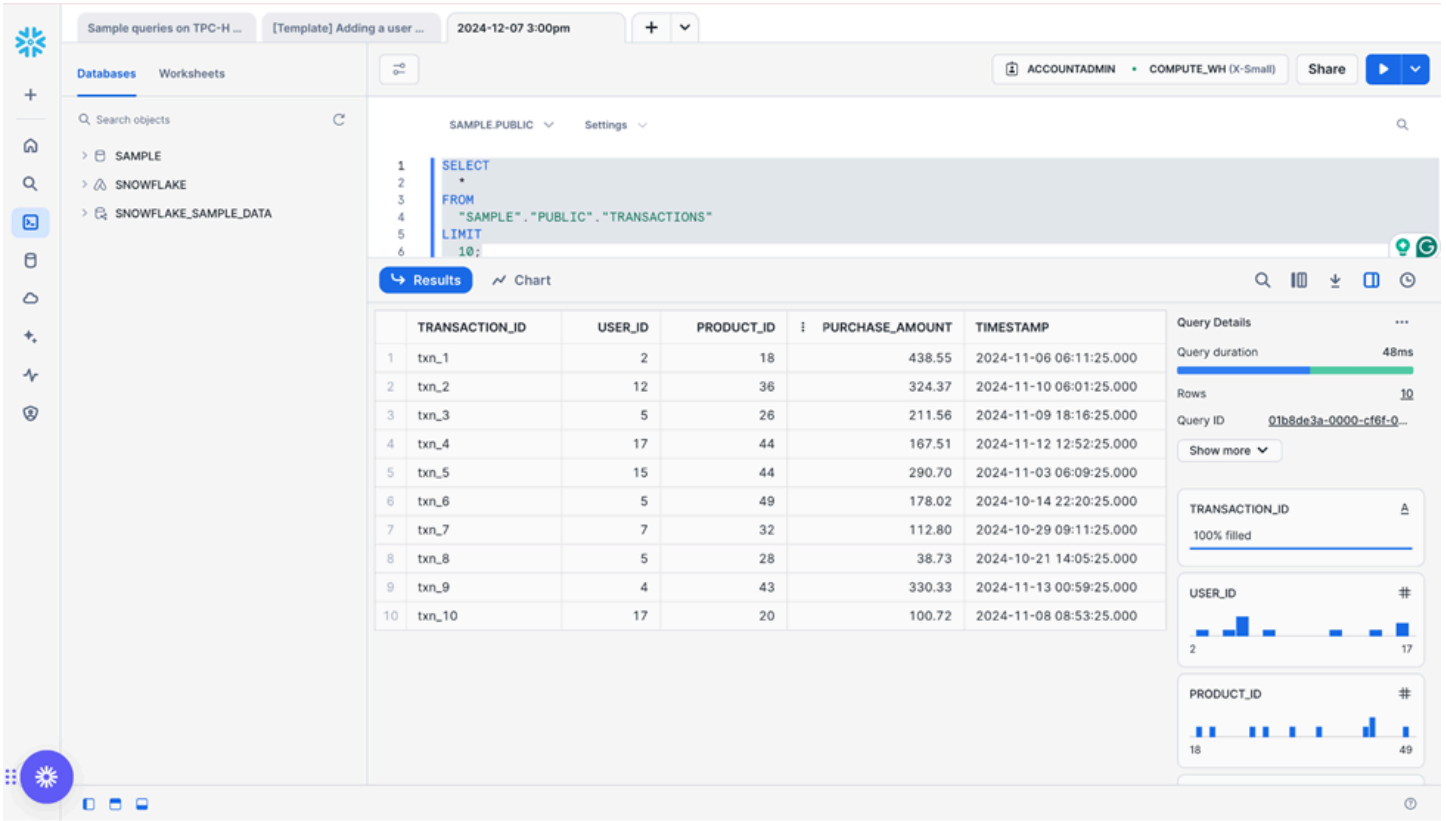
Load a Table

1. **Click the Load Table Button**
   - In the table view, click **Load Table**.
2. **Select a File**
   - Choose a file from your local machine and specify options like delimiter, header row, and file format.
3. **Preview the Data**
   - Review the preview to ensure correct parsing.
4. **Upload the Data**
   - Click **Load** to upload the data into the table.

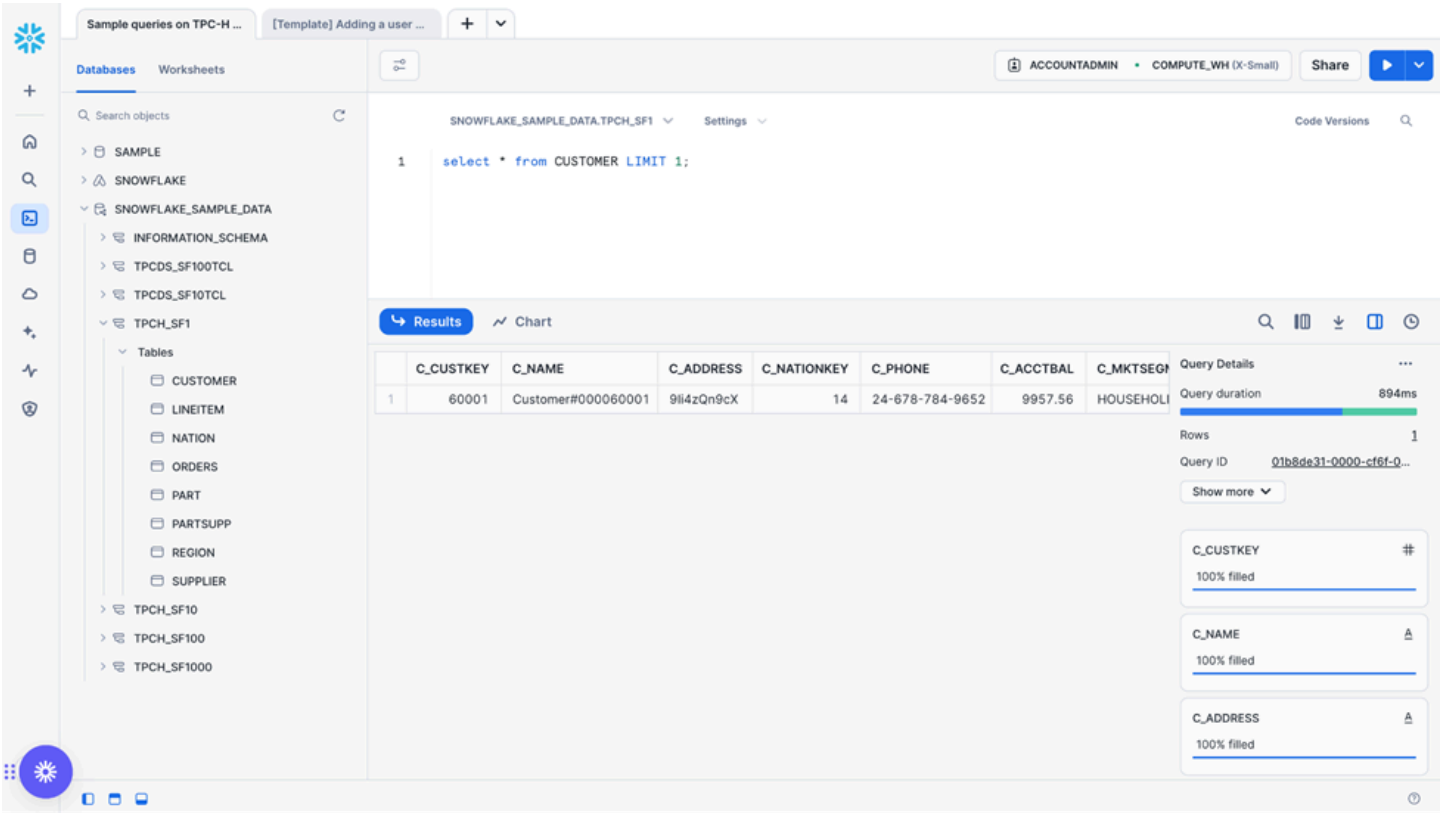




Query the Data



You can run sample **TPC-H queries** or write custom queries to analyze the uploaded data.



Choosing a Cloud-Based Solution

Here's a comparison of **Snowflake**, **Databricks**, **Google BigQuery**, and **Amazon Redshift**:

Feature	Snowflake	Databricks	Google BigQuery	Amazon Redshift
Architecture	Multi-cluster, shared-data; decoupled storage & compute.	Built on Apache Spark; unified platform for data engineering, analytics, and ML.	Serverless, fully managed; optimized for real-time analytics.	Cluster-based; leader and compute nodes.
Data Types	Structured, semi-structured (e.g., JSON, Parquet).	Handles structured, semi-structured, and unstructured data.	Optimized for structured and semi-structured data.	Best for structured data; less optimized for semi-structured data.
Performance	Auto-scaling, high concurrency with multi-cluster warehouses.	Elastic scaling; excels in big data and ML tasks.	Auto-scaling; ideal for rapid ad-hoc queries on large datasets.	High performance for structured data; requires manual scaling.
Scalability	Independent scaling of storage and compute.	Elastic and seamless for large datasets.	Automatic, serverless scaling.	Manual cluster resizing required.

Integration	Integrates with BI tools; supports AWS, Azure, Google Cloud.	Supports multiple languages; integrates with various data sources.	Integrates with Google Cloud ecosystem; supports standard SQL.	Works well with AWS services.
Pricing	Pay-as-you-go or capacity pricing.	Includes compute, storage, and software; less transparent pricing.	Pay-per-query; cost-effective for intermittent workloads.	On-demand clusters; discounts with reserved instances.
Security	End-to-end encryption, role-based access, compliance with GDPR, HIPAA.	Enterprise-grade security; supports industry standards.	Strong security; compliance with multiple regulations.	Encryption, network isolation, GDPR, HIPAA compliance.
Use Cases	Fully managed, scalable data warehouses for diverse data types.	Data engineering, advanced analytics, and machine learning.	Real-time analytics and ad-hoc SQL querying.	Structured data analytics within AWS-integrated environments.

Choosing the Right Solution

Consider your organization's **data processing**, **analytics needs**, and **budget**. This table highlights each platform's strengths to help align with your requirements.

Spark Streaming - Dec 14

- **Structured Streaming**
- **Architecture of Spark Streaming**
- **Spark Streaming Workflow**
- **Transformations in Spark Streaming**
- **Data Generator: Simulate Real-Time Data Streaming**
- **Structured Streaming Application**
- **How Structured Streaming Differs from Batch Processing**

Spark Streaming is a part of Apache Spark that helps process real-time data streams. It is made for handling live data and lets developers create scalable and fault-tolerant streaming applications. Spark Streaming processes continuous data streams almost in real-time and works well with other parts of the Spark system for deeper analysis. However, Spark Streaming has been deprecated in favor of Structured Streaming.

In this lecture, we will focus on Structured Streaming in Spark.

Structured Streaming

Structured Streaming is a scalable and fault-tolerant stream processing engine built on Apache Spark. It was introduced in Spark 2.0 and allows real-time data processing using a high-level API similar to Spark SQL and DataFrames. Unlike traditional Spark Streaming, which uses a micro-batch model, Structured Streaming processes data continuously while still keeping Spark's batch-processing strengths.

Key Features of Structured Streaming:

1. **Declarative API**
    - Uses the same DataFrame/Dataset API as batch processing, making it easy to use for those familiar with Spark SQL.
2. **Event-Time Processing**
    - Handles data with timestamps using event-time rules, letting users write logic based on when events actually happened.
3. **Fault Tolerance**
    - Ensures exactly-once guarantees by using checkpointing and write-ahead logs (WAL).
4. **Incremental Query Model**
    - Treats streaming tasks as incremental queries, updating the output continuously as new data arrives.
5. **Seamless Integration**
    - Works with many input sources like Kafka, Kinesis, and file streams, and supports outputs like databases, HDFS, and message queues.
6. **Backpressure Handling**
    - Automatically adjusts the processing speed based on how fast data is being received.
7. **Output Modes**
    - **Append**: Adds only new rows to the output.
    - **Update**: Updates only the rows that changed in the result.
    - **Complete**: Writes the entire result to the output sink.

Input Sources in Structured Streaming

1. **Kafka**: Reads data from Kafka topics for high-throughput, real-time ingestion.
2. **File Source**: Monitors a directory for new files and processes them as they arrive.
3. **Socket**: Reads data from a TCP socket connection.

4. **Rate Source**: Generates data at a constant rate for testing or benchmarking purposes.
5. **Custom Sources**: Users can implement custom connectors for specialized input sources.

Output Sinks in Structured Streaming

1. **Console**: Outputs results to the console for debugging and testing.
2. **Kafka**: Writes processed data back to Kafka topics.
3. **File**: Saves results to a directory in formats like Parquet, JSON, or CSV.
4. **Database**: Writes results to relational databases like MySQL, Postgres, etc.
5. **Memory**: Stores results in memory for interactive analysis (debugging only).
6. **Custom Sink**: Users can define their own sink for specific use cases.

Architecture of Spark Streaming

1. **Input Sources**:
   - Data is ingested from sources like Kafka, Kinesis, HDFS, Flume, or sockets.
   - Spark Streaming reads data in real-time from these sources.
2. **Batching**:
   - The ingested data is divided into micro-batches based on the user-defined batch interval.
   - For example, if the interval is set to 1 second, all data received in that second is grouped into one RDD.
3. **Processing Engine**:
   - Each batch (RDD) is processed using Spark's core APIs, such as `map`, `reduce`, `join`, or `filter`.
   - Spark Streaming allows the use of transformations and output operations on DStreams.
4. **Output Sinks**:
   - The processed data can be sent to output sinks like HDFS, databases, dashboards, or Kafka.

Spark Streaming Workflow

Step 1: Setup the Spark Context

A `StreamingContext` is created on top of the `SparkContext` with a defined batch interval.

```
from pyspark import SparkContext
from pyspark.streaming import StreamingContext

sc = SparkContext("local[2]", "StreamingApp")
ssc = StreamingContext(sc, 1)  # Batch interval: 1 second
```

Step 2: Connect to the Input Source

Connect Spark Streaming to an input source such as Kafka, sockets, or files.

```
lines = ssc.socketTextStream("localhost", 9999)  # Connect to a TCP socket
```

Step 3: Transform the Data

Apply transformations like `map`, `filter`, and `reduceByKey` to process each batch.

```
words = lines.flatMap(lambda line: line.split(" "))
word_counts = words.map(lambda word: (word, 1)).reduceByKey(lambda x, y: x + y)
```

Step 4: Output the Processed Data

Send the processed data to an output sink or print it to the console.

```
word_counts.pprint()
```

Step 5: Start Streaming

Start the streaming job and wait for it to finish.

```
ssc.start()
ssc.awaitTermination()
```

Transformations in Spark Streaming

Transformations in Spark Streaming are operations that work on DStreams (Discretized Streams) to produce new DStreams. Transformations can be either **stateless** or **stateful**:

Stateless Transformations

These transformations are independent of previous batches. Examples include `map`, `flatMap`, `filter`, and `reduceByKey`.

Example:

```
lines = ssc.socketTextStream("localhost", 9999)  # Input source
words = lines.flatMap(lambda line: line.split(" "))  # Split lines into words
word_counts = words.map(lambda word: (word, 1)).reduceByKey(lambda x, y: x + y)
# Count words
```

Stateful Transformations

These transformations maintain state across batches. For example, `updateStateByKey` can keep track of running totals.

Example:

```
def update_function(new_values, running_total):
    return sum(new_values) + (running_total or 0)  # Update state

running_counts = word_counts.updateStateByKey(update_function)  # Maintain
running totals
```

Windowed Operations

Windowed operations allow aggregation over a sliding window of time, making them ideal for metrics like averages, sums, or counts over the last N seconds or minutes.

Example: Calculate Word Counts Over a Sliding Window

```
word_counts = words.map(lambda word: (word, 1))
windowed_word_counts = word_counts.reduceByKeyAndWindow(
    lambda x, y: x + y,  # Reduce function
    lambda x, y: x - y,  # Inverse function
    30,  # Window duration (30 seconds)
    10   # Slide duration (10 seconds)
)
```

Next Steps

Let's create an application to stream data and then write a Spark application to read and process this stream.

Data Generator: Simulate Real-Time Data Streaming

This Python script simulates real-time data generation and streams it through a TCP socket to be consumed by a Spark Structured Streaming application.

Code

```
import socket
import time
import random

# Define server host and port
HOST = 'localhost'  # Localhost for testing
PORT = 9999         # Port for the TCP socket
# Create a TCP socket
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind((HOST, PORT))
server_socket.listen(1)  # Listen for 1 connection
print(f"Server is running on {HOST}:{PORT}")
# Accept client connection
client_socket, addr = server_socket.accept()
print(f"Connection established with {addr}")
# Generate and send data continuously
try:
    while True:
        # Generate random words as streaming data
        data = f"word{random.randint(1, 100)} word{random.randint(1, 100)}\n"
        client_socket.send(data.encode("utf-8"))  # Encode and send
        print(f"Sent: {data.strip()}")
        time.sleep(1)  # Wait 1 second before sending the next batch
except BrokenPipeError:
    print("Client disconnected.")
finally:
    client_socket.close()
    server_socket.close()
```

Line-by-Line Explanation

1. `import socket`
   - Provides functionality to create and manage a TCP socket.
2. `import time`
   - Adds delay (sleep) between data transmissions.
3. `import random`
   - Generates random words to simulate data.
4. `HOST = 'localhost'`
   - Sets the server hostname to `localhost` for local testing.
5. `PORT = 9999`
   - Specifies the port on which the server will listen.
6. `server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)`
   - Creates a TCP socket using IPv4 (`AF_INET`) and `SOCK_STREAM` for TCP.
7. `server_socket.bind((HOST, PORT))`
   - Binds the socket to the specified host and port.
8. `server_socket.listen(1)`
   - Configures the server to listen for a single incoming connection.
9. `print(f"Server is running on {HOST}:{PORT}")`
   - Logs the server's startup details.
10. `client_socket, addr = server_socket.accept()`
    - Waits for a client to connect and returns the client socket and address.
11. `print(f"Connection established with {addr}")`
    - Logs details of the client connection.
12. `try:`
    - Begins a block to handle any runtime errors during data transmission.
13. `while True:`
    - Starts an infinite loop to continuously send data.
14. `data = f"word{random.randint(1, 100)} word{random.randint(1, 100)}\n"`
    - Creates a string with two random words to simulate streaming data.
15. `client_socket.send(data.encode("utf-8"))`
    - Encodes the string as bytes and sends it to the client.
16. `print(f"Sent: {data.strip()}")`
    - Logs the data sent to the client.
17. `time.sleep(1)`
    - Introduces a delay of 1 second before sending the next batch.
18. `except BrokenPipeError:`
    - Handles the scenario where the client disconnects unexpectedly.
19. `finally:`
    - Ensures proper cleanup of resources, regardless of any errors.
20. `client_socket.close()`
    - Closes the client socket connection.
21. `server_socket.close()`
    - Shuts down the server socket.

This script continuously generates random data and sends it over a TCP connection, creating a real-time data stream for testing. It includes error handling to manage disconnections gracefully.

Structured Streaming Application

This Spark Structured Streaming application reads data from a TCP socket, processes it in real-time, and outputs word counts.

Code

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode, split

# Initialize Spark session
spark = SparkSession.builder \
    .appName("StructuredStreamingExample") \
    .getOrCreate()

# Define the input source: Read data from the TCP socket
lines = spark.readStream \
    .format("socket") \
    .option("host", "localhost") \
    .option("port", 9999) \
    .load()

# Process the data: Split lines into words
words = lines.select(
    explode(split(lines.value, " ")).alias("word")
)

# Perform aggregations: Count occurrences of each word
word_counts = words.groupBy("word").count()

# Define the output sink: Write results to a file
output_path = "word_count_output"
query = word_counts.writeStream \
    .outputMode("complete") \
    .format("csv") \
    .option("path", output_path) \
    .option("checkpointLocation", "word_count_checkpoint") \
    .start()

# Wait for the streaming query to terminate
query.awaitTermination()
```

Line-by-Line Explanation

1. `from pyspark.sql import SparkSession`
   - Imports `SparkSession`, the entry point for any Spark application.
2. `from pyspark.sql.functions import explode, split`
   - Imports:
     - `explode`: Flattens arrays into individual rows.
     - `split`: Splits a string into an array based on a delimiter.
3. `spark = SparkSession.builder.appName("StructuredStreamingExample").getOrCreate()`
   - Initializes a Spark session with the name *StructuredStreamingExample*.
4. `lines = spark.readStream.format("socket").option("host", "localhost").option("port", 9999).load()`
   - Configures the input source to read data from a TCP socket:
     - `format("socket")`: Specifies the data source as a socket.
     - `option("host", "localhost")`: Sets the server's hostname.
     - `option("port", 9999)`: Sets the server's port.
5. `words = lines.select(explode(split(lines.value, " ")).alias("word"))`
   - Processes incoming data:
     - `split(lines.value, " ")`: Splits each line into an array of words.
     - `explode(...)`: Converts each array element into a separate row.
     - `alias("word")`: Names the column `word`.
6. `word_counts = words.groupBy("word").count()`
   - Groups the data by the `word` column and counts occurrences of each word.
7. `query = word_counts.writeStream.outputMode("complete").format("csv").option(...).start()`
   - Configures the output:
     - `outputMode("complete")`: Writes the entire word count table for each update.
     - `format("csv")`: Writes the output to CSV files.
     - `option("path", "word_count_output")`: Specifies the output directory.
     - `option("checkpointLocation", "word_count_checkpoint")`: Sets a checkpoint directory for fault tolerance.
     - `start()`: Starts the streaming query.
8. `query.awaitTermination()`
   - Keeps the streaming job running until manually terminated.

This script continuously processes real-time data, performs word counts, and writes the results to a file system. It ensures fault tolerance with a checkpoint directory and real-time updates using the complete output mode.

How It All Fits Together

1. **Data Generator**
   - Streams random text data to a TCP socket continuously.
2. **Structured Streaming Application**
   - Reads the streamed data from the TCP socket, splits it into words, and counts the occurrences of each word in real-time.
3. **Output**
   - The word counts are displayed in the console after processing each batch of data.

Example Console Output

When the Spark application is running, the console may display the following:

```
-------------------------------------------
Batch: 1
-------------------------------------------
+-------+-----+
|  word |count|
+-------+-----+
| word42|    3|
| word76|    2|
| word89|    1|
+-------+-----+
```

This output updates with every new batch processed, reflecting real-time word counts.

How Structured Streaming Differs from Batch Processing

Structured Streaming is designed for real-time data processing, while batch processing works with static datasets. Below is a comparison of their key differences and why Structured Streaming achieves lower latency:

---

1. Data Handling

- **Batch Processing**
  - Operates on a fixed dataset (e.g., files in HDFS or a database table).
  - Entire dataset must be fully available before processing begins.
  - Example: A daily job processes log files collected over the past 24 hours.
- **Structured Streaming**
  - Works with unbounded data streams, treating them as continuously growing tables.
  - New data is incrementally added to the input table and processed as it arrives.
  - Example: Real-time log analysis processes data as it is generated.

---

2. Processing Model

- **Batch Processing**
  - Processes the entire dataset in one operation.
  - Requires significant resources to handle large datasets since transformations are applied to all rows at once.
- **Structured Streaming**
  - Processes data incrementally (in micro-batches or continuous mode).
  - Only new rows since the last trigger are processed, avoiding reprocessing old data.
  - Efficient and resource-friendly.

3. Query Execution Flow

Batch Processing Execution

1. **Data Loading -** Entire dataset is loaded into memory or read from disk.
2. **Query Optimization -** Catalyst Optimizer creates a plan for processing the entire dataset.
3. **Execution -** Applies transformations and actions to all rows simultaneously.
4. **Result Generation -** Outputs the final result after processing is complete.

Structured Streaming Execution

1. **Data Ingestion -** New data is appended to an unbounded input table.
2. **Incremental Processing -** Processes only new rows since the last trigger.
3. **Query Optimization -** Query plan is reused for each micro-batch or event.
4. **State Updates -** Maintains intermediate states for aggregations or joins.
5. **Incremental Output -** Outputs updated results in near real-time.

4. Resource Utilization

- **Batch Processing**
  - Consumes significant memory and CPU resources, as the entire dataset is processed at once.
  - Higher risk of memory bottlenecks with large datasets.
- **Structured Streaming**
  - Processes small chunks of data (micro-batches) or individual events.
  - Resource usage is spread over time, making it suitable for high-throughput data streams.

5. Latency Comparison: Why Structured Streaming is Faster

1. **Incremental Processing**
   - **Structured Streaming**: Processes only the new data during each trigger.
   - **Batch Processing**: Processes the entire dataset every time, even for minor updates.
2. **Small Processing Units**
   - **Structured Streaming**: Handles data in micro-batches (e.g., 1-second intervals) or event-by-event.
   - **Batch Processing**: Operates on the entire dataset, causing delays with large volumes.
3. **State Management**
   - **Structured Streaming**: Maintains intermediate results (e.g., running totals), avoiding recomputation of the entire dataset.
4. **Reusing Query Plans**
   - **Structured Streaming**: Reuses the same query plan for every trigger, reducing overhead.
   - **Batch Processing**: Re-optimizes the query plan for every new dataset.
5. **Trigger-Based Execution**
   - **Structured Streaming**: Executes only for new data since the last trigger, minimizing delays.
   - **Batch Processing**: Waits for the full dataset before starting, leading to longer processing times.

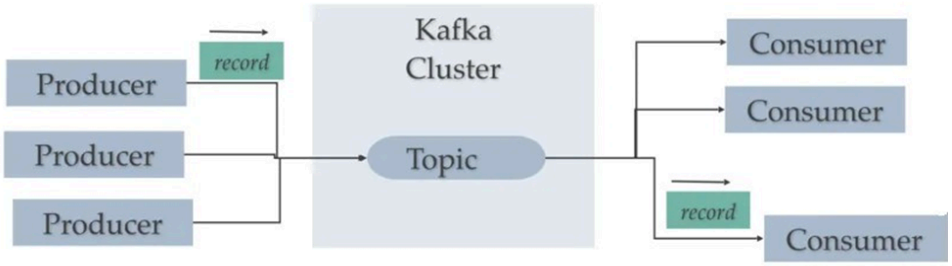
Conclusion

Structured Streaming is optimized for real-time data processing, making it ideal for applications requiring low latency and efficient resource usage. Batch processing, while powerful for static datasets, struggles with the demands of continuous data streams.

Kafka Ingestion Architecture - Dec 16

- **Partitions**
- **Replicas**
- **Key Components of Kafka Architecture**
- **How Big Data Flows Through Kafka**
- **Kafka Installation**
- **Spark Application Code**
- **Revised Explanation for Streaming Data from Kafka Using Spark**

Kafka uses the pub/sub model, similar to other messaging systems. Producers (applications) send messages (records) to a Kafka broker (node), and these messages are processed by consumers (other applications). Messages are stored in a topic, and consumers can subscribe to a topic to listen for those messages. These messages can include things like sensor readings, meter data, or user actions.



A **topic** can be thought of as a category or feed name where messages are stored and published.

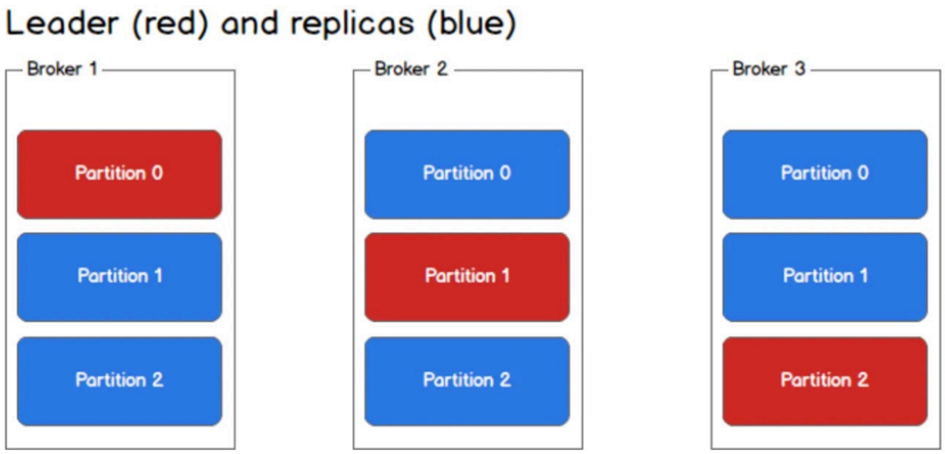
Partitions

Kafka topics can grow very large, making it hard to store all the data for a topic on one node. To solve this, topics are divided into **partitions**. Partitions allow the topic's data to be spread across multiple brokers (Kafka nodes). This makes it possible for multiple consumers to read from the topic at the same time, in parallel. Each partition can be stored on a separate machine.

Partitioning can be done in various ways, like by the order of arrival, hashing, IDs, etc.

Replicas

To ensure high availability, each partition has **replicas**. Replicas are copies of the partition stored on different brokers.



Let's look at an example with a Kafka cluster of 3 brokers:

- A topic is split into 3 partitions: Partition 0, Partition 1, and Partition 2.
- Each broker has a copy of each partition.
- One copy of each partition is chosen as the **leader**, and the others are **followers** that stay synchronized with the leader.

All writes and reads for a partition go through its leader. The leader ensures replicas are updated with new data. If a leader fails, one of the replicas is promoted to the new leader.

Key Components of Kafka Architecture

1. **Producers**:
    - Applications, devices, or systems that create and send data to Kafka.
    - Producers send messages to specific Kafka topics.
    - Messages can include logs, metrics, or transactional data in formats like JSON, Avro, or binary.
2. **Topics**:
    - Logical channels where producers send messages, and consumers fetch them.
    - Topics are divided into **partitions** for scalability:
        - Partitioning enables parallel processing by splitting data across multiple partitions.
        - Each partition is an append-only log.
    - Producers determine which partition a message goes to, usually by using a key or a round-robin strategy.
3. **Brokers**:
    - Kafka runs on a cluster of servers, with each server called a broker.
    - Brokers store topic partitions and handle message storage and delivery.
    - Kafka automatically distributes partitions across brokers for load balancing and fault tolerance.
4. **ZooKeeper (or Kafka Metadata Service)**:
    - Manages metadata for the Kafka cluster, such as broker locations, partition assignments, and leader elections.
    - Starting with Kafka 2.8, **KRaft (Kafka Raft Metadata mode)** replaces ZooKeeper.
5. **Consumers**:
    - Applications that subscribe to topics and retrieve data for processing.
    - Kafka's **consumer groups** allow multiple consumers to read from a topic's partitions in parallel:
        - Each consumer in a group reads from a subset of the topic's partitions.
6. **Partitions**:
    - A unit of parallelism in Kafka.
    - Each partition stores data sequentially and ensures message ordering within the partition.
7. **Offsets**:
    - Every message in a partition has a unique identifier called an **offset**.
    - Consumers use offsets to track which messages have been processed.
8. **Replication**:
    - Kafka replicates each partition across multiple brokers for fault tolerance.
    - One broker acts as the **leader** for each partition, while others are **followers**.

How Big Data Flows Through Kafka

1. Data Ingestion into Kafka

1. **Producers Push Data**:
    - Producers send data (e.g., application logs, sensor readings, event streams) to Kafka using the Kafka Producer APIs.
    - Each message is sent to a specific topic and partition, determined by:
        - A key (e.g., user ID, transaction ID).
        - A round-robin strategy (random assignment).

2. **Partitioning for Scalability**:
    - Kafka partitions the topic data to enable parallel writes and reads.
    - Example:
        - A topic named `sensor-data` with 3 partitions can store messages from multiple producers simultaneously.
3. **Brokers Store the Data**:
    - Each partition is stored on a broker.
    - Kafka writes messages to the partition as an append-only log for fast sequential writes.
    - Messages are retained based on configuration:
        - **Time-based retention** (e.g., 7 days).
        - **Size-based retention** (e.g., 10 GB).
4. **Replication for Fault Tolerance**:
    - Kafka replicates each partition across multiple brokers.
    - If a broker fails, Kafka elects a new leader for the partition.

---

2. Data Consumption from Kafka

1. **Consumers Subscribe to Topics**:
    - Consumers use the Kafka Consumer APIs to subscribe to topics.
    - Kafka's poll-based model lets consumers fetch data at their own pace.
2. **Consumer Groups**:
    - Kafka distributes partitions among consumers within a group to enable parallel processing.
    - Example:
        - A topic named `click-stream` has 6 partitions.
        - Consumer Group A has 3 consumers, with each consumer reading 2 partitions.
        - This allows efficient processing of large amounts of data.
3. **Message Offsets**:
    - Consumers track message offsets to ensure no data is missed or processed more than once.
    - Kafka retains consumed messages, enabling consumers to reprocess data if needed.
4. **Stream Processing**:
    - Consumers can process data using:
        - **Kafka Streams API** for real-time processing.
        - External frameworks like Apache Spark, Apache Flink, or Storm for advanced processing and analytics.

Kafka Installation

1. **Download Kafka**:
   `wget https://downloads.apache.org/kafka/3.7.2/kafka_2.12-3.7.2.tgz`
2. **Extract the Tar File**:
   `tar -xvzf kafka_2.12-3.7.2.tgz`
3. **Move Kafka to /opt Directory**:
   `sudo mv kafka_2.12-3.7.2 /opt/kafka/`
4. **Create Necessary Directories**:
   `mkdir /tmp/zookeeper`
   `mkdir /tmp/kafka-logs`
5. **Start ZooKeeper**:
   `/opt/kafka/bin/zookeeper-server-start.sh /opt/kafka/config/zookeeper.properties`
6. **Start Kafka Broker**:
   `/opt/kafka/bin/kafka-server-start.sh /opt/kafka/config/server.properties`

7. **Create a Topic**:
   ```
   /opt/kafka/bin/kafka-topics.sh --create --topic test-topic
   --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1
   ```
8. **List Topics**:
   ```
   /opt/kafka/bin/kafka-topics.sh --list --bootstrap-server localhost:9092
   ```
9. **Start a Producer**:
   ```
   /opt/kafka/bin/kafka-console-producer.sh --topic test-topic
   --bootstrap-server localhost:9092
   ```

Demo

1. **Start ZooKeeper**:
   ```
   /opt/kafka/bin/zookeeper-server-start.sh
   /opt/kafka/config/zookeeper.properties
   ```
2. **Start Kafka Broker**:
   ```
   /opt/kafka/bin/kafka-server-start.sh /opt/kafka/config/server.properties
   ```
3. **Create a Kafka Topic**:
   ```
   /opt/kafka/bin/kafka-topics.sh --create --topic test-topic
   --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1
   ```
4. **Verify Topic Creation**:
   ```
   /opt/kafka/bin/kafka-topics.sh --list --bootstrap-server localhost:9092
   ```
5. **Start a Producer**:
   ```
   /opt/kafka/bin/kafka-console-producer.sh --topic test-topic
   --bootstrap-server localhost:9092
   ```
6. **Push Event**:
   ```
   {"id": 1, "name": "Test Event", "timestamp": "2024-12-09T08:00:00Z"}
   ```

Spark Application Code

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import from_json, col
from pyspark.sql.types import StructType, StringType, IntegerType, TimestampType

# Initialize Spark session
spark = SparkSession.builder \
    .appName("KafkaStructuredStreaming") \
    .getOrCreate()

# Define schema
schema = StructType() \
    .add("id", IntegerType()) \
    .add("name", StringType()) \
    .add("timestamp", StringType())
```

```
# Read data from Kafka
kafka_stream = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("subscribe", "test-topic") \
    .option("startingOffsets", "earliest") \
    .load()

# Parse the value column as JSON
parsed_stream = kafka_stream.selectExpr("CAST(value AS STRING)") \
    .select(from_json(col("value"), schema).alias("data")) \
    .select("data.*")

# Convert timestamp to proper format
parsed_stream = parsed_stream.withColumn("timestamp",
col("timestamp").cast(TimestampType()))

# Write the stream to the console
query = parsed_stream.writeStream \
    .outputMode("append") \
    .format("console") \
    .start()

query.awaitTermination()
```

Command to Run Spark Application

```
spark-submit --packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.4.4
spark-kafka-stream.py
```

Detailed Breakdown of the Code

1. **SparkSession Initialization**:
   - Initializes a Spark session for the streaming job.
2. **Schema Definition**:
   - Defines the structure of incoming JSON data with fields like `id`, `name`, and `timestamp`.
3. **Reading from Kafka**:
   - Configures the Kafka consumer to read from the `test-topic` topic.
   - Reads messages from the earliest offset.
4. **Parsing Kafka Data**:
   - Converts the Kafka `value` field to a string and parses it as JSON using the defined schema.
5. **Timestamp Conversion**:
   - Casts the `timestamp` field to a proper `TimestampType` for easier processing.
6. **Writing to Console**:
   - Writes the parsed stream to the console in append mode.
7. **Application Execution**:
   - Use `spark-submit` to run the script, ensuring the required Kafka-Spark connector is included.

Revised Explanation for Streaming Data from Kafka Using Spark

---

1. Importing Required Libraries

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import from_json, col
from pyspark.sql.types import StructType, StringType, IntegerType, TimestampType
```

- **`SparkSession`**: Entry point to configure and run the Spark application.
- **`from_json`**: Parses JSON strings into a structured format (like a table).
- **`col`**: References DataFrame columns for operations.
- **Schema Types**:
  - **`StructType`**: Defines the schema of structured data.
  - **`StringType`**, **`IntegerType`**, **`TimestampType`**: Specify data types for fields.

---

2. Initializing the Spark Session

```
spark = SparkSession.builder \
    .appName("KafkaStructuredStreaming") \
    .getOrCreate()
```

- **`appName`**: Sets the name for the Spark application in the UI.
- **`getOrCreate`**: Creates a new Spark session or retrieves an existing one.

---

3. Defining the Schema

```
schema = StructType() \
    .add("id", IntegerType()) \
    .add("name", StringType()) \
    .add("timestamp", StringType())
```

- Defines the structure of incoming JSON messages from Kafka.
- **Fields**:
  - `id`: Integer field.
  - `name`: String field.
  - `timestamp`: String field representing the time of the event.

Example JSON:

```
{"id": 1, "name": "Test Event", "timestamp": "2024-12-09T08:00:00Z"}
```

4. Reading Data from Kafka

```
kafka_stream = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("subscribe", "test-topic") \
    .option("startingOffsets", "earliest") \
    .load()
```

- **`readStream`**: Reads data in real-time.
- **`format("kafka")`**: Specifies Kafka as the source.
- **Options**:
  - `kafka.bootstrap.servers`: Address of Kafka cluster (e.g., `localhost:9092`).
  - `subscribe`: Kafka topic to read from (e.g., `test-topic`).
  - `startingOffsets`: Start reading from:
    - `earliest`: Beginning of the topic.
    - `latest`: Only new messages.

Default Kafka DataFrame Columns:

- `key`, `value`, `topic`, `partition`, `offset`, `timestamp`, etc.

---

5. Parsing Kafka Data

```
parsed_stream = kafka_stream.selectExpr("CAST(value AS STRING)") \
    .select(from_json(col("value"), schema).alias("data")) \
    .select("data.*")
```

- **Step 1**: Cast the binary `value` column to a string.
- **Step 2**: Parse the JSON string in `value` using the schema.
  - `from_json(col("value"), schema)`: Converts JSON strings to structured format.
  - `alias("data")`: Assigns the parsed data a temporary name (`data`).
- **Step 3**: Extract individual fields (`id`, `name`, `timestamp`).

---

6. Converting the Timestamp

```
parsed_stream = parsed_stream.withColumn(
    "timestamp", col("timestamp").cast(TimestampType())
)
```

- **`withColumn`**: Adds or updates a column.
- **`cast(TimestampType())`**: Converts the `timestamp` field from a string to a proper timestamp format.

7. Writing the Stream to the Console

```
query = parsed_stream.writeStream \
    .outputMode("append") \
    .format("console") \
    .start()
```

- **`writeStream`**: Writes the processed data to a destination.
- **`outputMode`**: Specifies how results are written:
    - `append`: Outputs only new rows.
    - `update`: Outputs only updated rows.
    - `complete`: Outputs the entire table (used for aggregations).
- **`format("console")`**: Sends output to the console (useful for debugging).

8. Keeping the Stream Active

```
query.awaitTermination()
```

- **`awaitTermination`**: Keeps the streaming process active until stopped manually.

Example Workflow Summary

1. **Initialize Spark Session**: Create a Spark application.
2. **Read from Kafka**: Connect to the Kafka topic and start consuming messages.
3. **Parse JSON**: Convert raw Kafka data into structured format using the schema.
4. **Write to Console**: Print the results to the console in real time.
5. **Keep Streaming**: Use `awaitTermination()` to keep the application running.

Execution Flow

1. **Reading Data from Kafka**: The application consumes messages from the Kafka topic `test-topic`.
2. **Parsing Kafka Messages**:
    - The `value` column containing binary data is cast to strings.
    - The strings are parsed as JSON using the defined schema.
3. **Transforming Data**:
    - The parsed JSON is converted into a structured DataFrame with fields: `id`, `name`, and `timestamp`.
4. **Converting Timestamp**: The `timestamp` field is cast to a proper `TimestampType` for time-based operations.
5. **Writing to Console**: The processed data is displayed in real-time on the console.

Output Example

Input Message (sent to the Kafka topic):

```
{"id": 1, "name": "Test Event", "timestamp": "2024-12-09T08:00:00Z"}
```

Console Output:

```
+---+-----------+-------------------+
| id|       name|          timestamp|
+---+-----------+-------------------+
|  1| Test Event|2024-12-09 08:00:00|
+---+-----------+-------------------+
```

Input console

```
nimesh.k@nimesh-k-M94Q15F2JH ~ % /opt/kafka/bin/kafka-console-prod
ucer.sh --topic test-topic --bootstrap-server localhost:9092
>{"id": 1, "name": "Test Event", "timestamp": "2024-12-09T08:00:00
Z"}
>
>
```

Output console

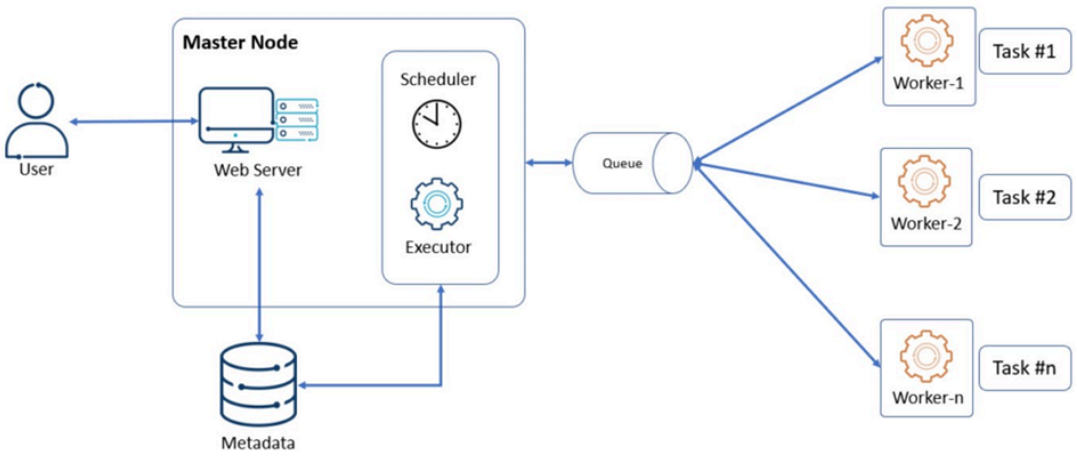
```
-------------------------------------------
Batch: 1
-------------------------------------------
24/12/09 13:53:19 INFO CodeGenerator: Code gener
ated in 31.644333 ms
24/12/09 13:53:19 INFO CodeGenerator: Code gener
ated in 22.863917 ms
+----+----------+-------------------+
|  id|      name|          timestamp|
+----+----------+-------------------+
|   1|Test Event|2024-12-09 13:30:00|
|null|      null|               null|
+----+----------+-------------------+

24/12/09 13:53:19 INFO WriteToDataSourceV2Exec:
Data source write support MicroBatchWrite[epoch:
 1, writer: ConsoleWriter[numRows=20, truncate=t
rue]] committed.
24/12/09 13:53:19 INFO CheckpointFileManager: Wr
iting atomically to file:/private/var/folders/4l
/4lqkszcj5jx446ljmn1phnmr0000gq/T/temporary-4063
a3e1-0292-4fd0-a0ed-b130ed6665c0/commits/1 using
 temp file file:/private/var/folders/4l/4lqkszcj
5jx446ljmn1phnmr0000gq/T/temporary-4063a3e1-0292
-4fd0-a0ed-b130ed6665c0/commits/.1.8b982724-7d48
-4941-8971-a15402049ccc.tmp
24/12/09 13:53:19 INFO CheckpointFileManager: Re
named temp file file:/private/var/folders/4l/4lq
kszcj5jx446ljmn1phnmr0000gq/T/temporary-4063a3e1
-0292-4fd0-a0ed-b130ed6665c0/commits/.1.8b982724
-7d48-4941-8971-a15402049ccc.tmp to file:/privat
```

Scheduling Data Processing Jobs Using Cron AndApacheAirflow - Dec 20

- **Key Features of Airflow**
- **Core Concepts in Airflow**
- **Example Workflow: ETL Workflow**
- **Installation**
- **Live Demo**
- **Key Concepts Explained**
- **Create a Spark Job Script**
- **Trigger the DAG**
- **Commands Cheat Sheet**
- **Airflow Web UI Overview**

Apache Airflow is an open-source platform for programmatically creating, scheduling, and monitoring workflows as Directed Acyclic Graphs (DAGs). It allows users to define workflows in Python and manage task execution, making it a valuable tool for data pipelines, ETL processes, and machine learning workflows.



Key Features of Airflow

1. **Workflow as Code**
   - Workflows are defined as Python scripts, making them flexible, version-controlled, and testable.
2. **Dynamic Workflow Management**
   - DAGs and tasks can be dynamically created, updated, or extended based on logic, input, or runtime parameters.
3. **Task Scheduling**
   - Airflow schedules tasks using cron-like syntax or preset intervals, running them at specific times or based on dependencies.
4. **Scalability**
   - Airflow supports distributed execution using a queue and worker system, scaling horizontally across multiple servers.
5. **Extensibility**
   - Airflow provides built-in operators like `BashOperator` and `PythonOperator`, and integrates with tools like Hadoop, Spark, Hive, PostgreSQL, AWS, and Google Cloud.
   - Custom operators can be created for unique needs.
6. **Monitoring and Logging**
   - The web-based UI offers real-time monitoring, task logs, and options to retry or rerun tasks.

Core Concepts in Airflow

1. **Directed Acyclic Graph (DAG)**
    - A DAG represents a workflow where:
        - **Nodes** are tasks.
        - **Edges** define dependencies between tasks.
    - DAGs must be acyclic, meaning no task can depend on itself, directly or indirectly.
2. **Tasks**
    - Individual units of work defined in a DAG.
    - Tasks can execute shell commands, Python scripts, database queries, API calls, etc.
3. **Operators**
    - Predefined task templates for common operations:
        - **BashOperator**: Executes a shell command.
        - **PythonOperator**: Executes a Python function.
        - **EmailOperator**: Sends emails.
        - **CustomOperator**: For user-defined logic.
4. **Task Dependencies**
    - Tasks in a DAG can have dependencies to control the execution order:
        - **Sequential**: Task B runs after Task A.
        - **Parallel**: Tasks B and C run simultaneously after Task A.
5. **Scheduler**
    - Ensures tasks are executed in the correct order and at the specified time.
6. **Executor**
    - Manages task execution (e.g., locally, via Celery, Kubernetes).
7. **Metadata Database**
    - Airflow uses a database (e.g., SQLite, PostgreSQL, MySQL) to store metadata about DAGs, tasks, and their states.
8. **Web UI**
    - A graphical interface to visualize workflows, monitor runs, view logs, and manage tasks.

Example Workflow: ETL Workflow

Steps:

1. Extract data from an API.
2. Transform data using Spark.
3. Load the processed data into a PostgreSQL database.

DAG Example:

```
from airflow import DAG
from airflow.operators.bash import BashOperator
from datetime import datetime

# Define the DAG
with DAG(
    dag_id='etl_workflow',
    start_date=datetime(2023, 1, 1),
    schedule_interval='@daily',
    catchup=False,
) as dag:
```

```
# Tasks
extract = BashOperator(
    task_id='extract',
    bash_command='python extract_data.py',
)

transform = BashOperator(
    task_id='transform',
    bash_command='spark-submit transform_data.py',
)

load = BashOperator(
    task_id='load',
    bash_command='python load_to_db.py',
)

# Define task dependencies
extract >> transform >> load
```

Explanation:

1. **DAG Definition:**
   - `dag_id`: Unique name for the DAG.
   - `start_date`: Specifies when the DAG should start.
   - `schedule_interval`: Runs daily (`@daily`).
   - `catchup`: Disables backfilling for missed runs.
2. **Tasks:**
   - **`extract`:** Runs a Python script to extract data.
   - **`transform`:** Submits a Spark job for data transformation.
   - **`load`:** Executes a script to load data into PostgreSQL.
3. **Task Dependencies:**
   - **`extract >> transform >> load`:** Ensures sequential execution of tasks.

Installation

Steps:

1. **Install Python Environment:**

   Airflow requires Python (3.7+). Check your Python version:
   ```
   python --version
   ```

2. **Create a Virtual Environment (Optional but Recommended):**

   Create and activate a virtual environment:
   ```
   python -m venv airflow_env
   source airflow_env/bin/activate
   ```

3. **Install Apache Airflow:**

   Install Airflow and its dependencies:
   ```
   pip install apache-airflow
   ```

4. **Initialize the Airflow Database:**

   Airflow uses a database to store metadata. Initialize it:
   ```
   airflow db init
   ```

5. **Create an Admin User:**

   Set up an admin user for the Airflow web interface:
   ```
   airflow users create \
   --username admin \
   --password admin \
   --firstname Admin \
   --lastname User \
   --role Admin \
   --email admin@example.com
   ```

6. **Start Airflow Services:**

   Start the Airflow web server:
   ```
   airflow webserver
   ```

7. **Start the Scheduler:**

   Open another terminal and start the scheduler:
   ```
   airflow scheduler
   ```

8. **Access the Web Interface:**
   - Open your browser and go to:
     `http://localhost:8080`

Live Demo

Steps to Run a DAG in Airflow:

Create the `airflow/dags` Directory:
If not already present, create the directory:
```
mkdir -p airflow/dags
```

Create a Python File for the DAG:
Save the following DAG file in your `airflow/dags` folder (default: `~/airflow/dags`).

```
from airflow import DAG
from airflow.operators.bash import BashOperator
from datetime import datetime
```

```
# Define the DAG
with DAG(
    dag_id='spark_job_example',
    start_date=datetime(2023, 1, 1),
    schedule_interval=None,  # Trigger manually
    catchup=False,
) as dag:

# Task: Run a Spark job
run_spark_job = BashOperator(
task_id='run_spark_job',
bash_command='spark-submit --master local ~/airflow/dags/spark_example.py',)
```

Key Concepts Explained

1. Importing Required Libraries:

```
from airflow import DAG
from airflow.operators.bash import BashOperator
from datetime import datetime
```

- **DAG**: Represents a workflow in Airflow, used to define tasks and dependencies.
- **BashOperator**: Executes a bash command as a task in the DAG.
- **datetime**: Defines the `start_date` of the DAG.

2. Defining the DAG:

```
with DAG(
    dag_id='spark_job_example',
    start_date=datetime(2023, 1, 1),
    schedule_interval=None,  # Trigger manually
    catchup=False,
) as dag:
```

- **dag_id**: Unique identifier for the DAG (`spark_job_example`).
- **start_date**: The date the DAG becomes valid. Should always be in the past.
- **schedule_interval**: Specifies how often the DAG runs.
    - **`None`**: Runs only when triggered manually.
    - **Cron expressions**: E.g., `"0 12 * * *"` for daily runs at noon.
- **catchup=False**: Prevents backdated runs between `start_date` and the current date.

3. Defining a Task:

```
run_spark_job = BashOperator(
    task_id='run_spark_job',
    bash_command='spark-submit --master local ~/airflow/dags/spark_example.py',
)
```

    - **task_id**: Unique name for the task (`run_spark_job`).
    - **bash_command**: Command executed by the task.
        - **`spark-submit`**: Runs the Spark job.
        - **`--master local`**: Runs the job locally.
        - **`~/airflow/dags/spark_example.py`**: Path to the Python script for the Spark job.

Create a Spark Job Script:

Save the following file as `spark_example.py` in the same folder (`~/airflow/dags`).

```
from pyspark.sql import SparkSession

# Initialize Spark Session
spark = SparkSession.builder \
    .appName("SampleSparkJob") \
    .getOrCreate()

# Sample Data Processing
data = [(1, "Alice"), (2, "Bob"), (3, "Charlie")]
columns = ["ID", "Name"]
df = spark.createDataFrame(data, columns)

# Perform a Transformation
df_transformed = df.withColumnRenamed("Name", "FullName")

# Save Output
df_transformed.show()
df_transformed.write.mode("overwrite").csv("/tmp/spark_output")

# Stop Spark Session
spark.stop()
```

Trigger the DAG

Open the Airflow web interface:
`http://localhost:8080`

1. Enable the `spark_job_example` DAG.
2. Manually trigger the DAG using the **"Trigger DAG"** button.
3. Monitor the DAG run in the **"Graph View"** or **"Task Instance Details"**.

Commands Cheat Sheet

Restart Airflow Services:

```
airflow webserver
airflow scheduler
```

List DAGs:

```
airflow dags list
```

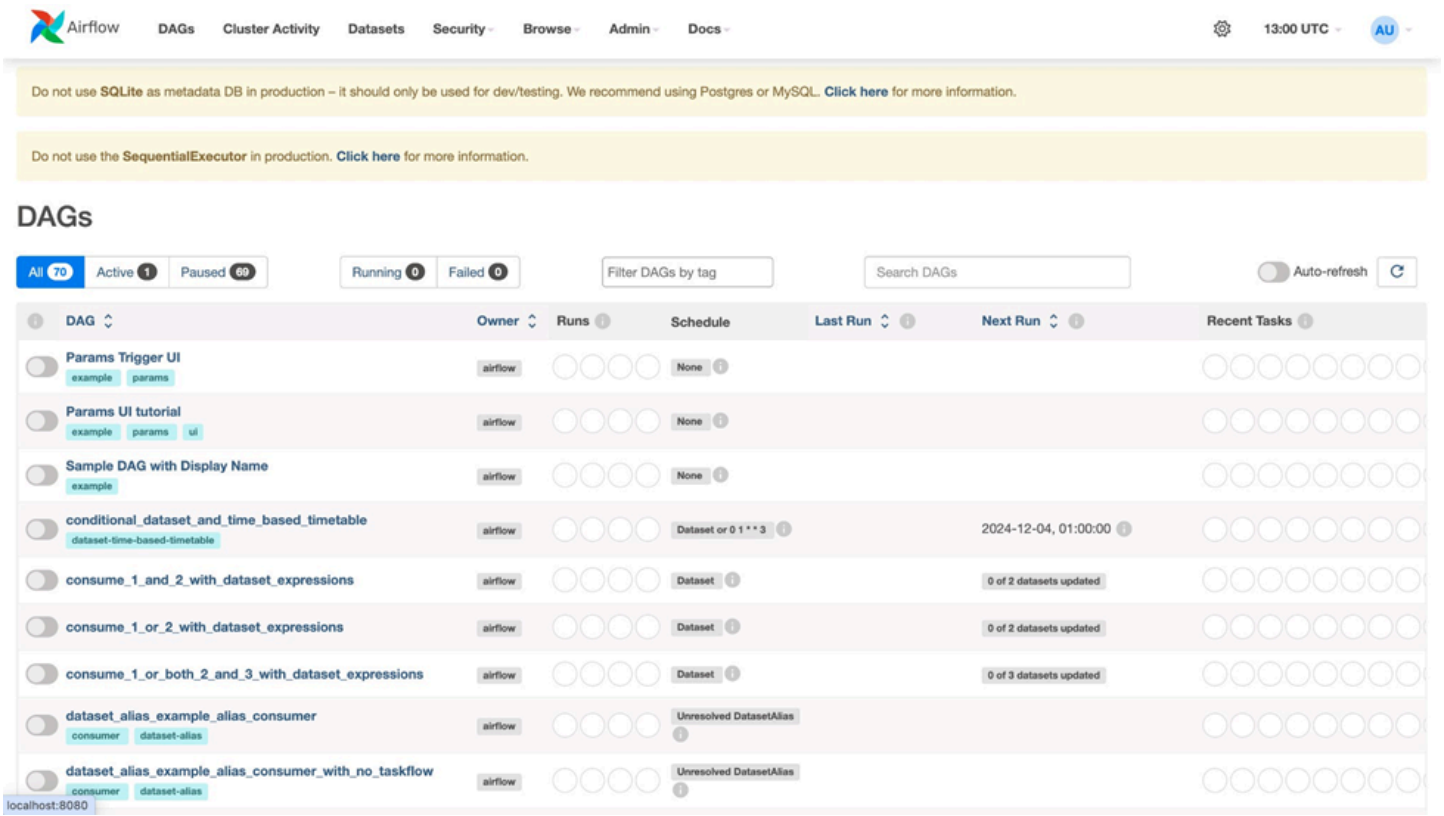
Trigger a DAG:

```
airflow dags trigger spark_job_example
```

Monitor DAG Runs:

```
airflow tasks list spark_job_example
```

Airflow Web UI Overview



1. DAGs Tab

The default screen in Airflow Web UI, displaying all available DAGs.

- **Purpose**: Manage and monitor DAGs.
- **Key Features**:
    - **DAG ID**: Unique identifiers for each DAG.
    - **Status Toggles**: Enable or disable DAGs (disabling prevents execution).
    - **Links**:
        - **Trigger DAG**: Manually start a DAG run.
        - **View Details**: Access DAG information and monitoring tools.
        - **Graph View, Tree View, etc.**: Quick visualizations.
    - **Last Run**: Shows the most recent execution time.
    - **Next Run**: Indicates the next scheduled execution.

2. Browse Tab

Tools to explore historical data and logs.

- **DAG Runs**:
    - **Purpose**: View past DAG runs.
    - **Features**: Status (success, failed, running), execution dates, detailed views.
    - **Use Case**: Diagnose failures or delays.
- **Task Instances**:
    - **Purpose**: View task execution history.
    - **Features**: Task status, duration, logs, search.
    - **Use Case**: Analyze task performance or debug issues.
- **Logs**:
    - **Purpose**: Access detailed task logs.
    - **Use Case**: Debug failures with error logs.
    -

- **Jobs**:
  - **Purpose**: Monitor system-level jobs (e.g., scheduler, tasks).
  - **Use Case**: Check ongoing activities.
- **SLA Misses**:
  - **Purpose**: Track tasks that missed their SLA.
  - **Use Case**: Improve SLA compliance.
- **XComs**:
  - **Purpose**: View data passed between tasks.
  - **Use Case**: Debug cross-task communication.

3. Data Profiling Tab

Analyze task and workflow performance.

- **Gantt Chart**:
  - **Purpose**: Visualize task start and end times.
  - **Use Case**: Identify bottlenecks.
- **Task Duration**:
  - **Purpose**: Show task execution durations over time.
  - **Use Case**: Monitor performance trends.
- **Landing Times**:
  - **Purpose**: Track delays between scheduled and actual start times.
  - **Use Case**: Pinpoint delays due to resource constraints.
- **Code**:
  - **Purpose**: View DAG source code.
  - **Use Case**: Quickly verify or inspect DAG logic.

4. Admin Tab

Manage system configurations and metadata.

- **Connections**:
  - **Purpose**: Manage external system connections (e.g., databases).
  - **Use Case**: Add credentials for AWS, MySQL, etc.
- **Variables**:
  - **Purpose**: Manage global variables for workflows.
  - **Use Case**: Store environment configurations.
- **Pools**:
  - **Purpose**: Manage resource pools for tasks.
  - **Use Case**: Limit resource access to prevent overloading.
- **Configuration**:
  - **Purpose**: View current `airflow.cfg` settings.
  - **Use Case**: Verify parallelism or executor configurations.
- **XComs**:
  - **Purpose**: Same as **Browse > XComs**.
  - **Use Case**: Inspect task-shared data.

5. Docs Tab

- **Purpose**: Access Airflow documentation and references.
- **Use Case**: Look up operators, APIs, and features.

6. Graph View

Accessed from the DAGs Tab or DAG Details page.

- **Purpose**: Visualize the task flow in a DAG.
- **Features**:
    - Show dependencies and execution status (success, failed).
    - Click tasks for logs or re-runs.
- **Use Case**: Understand and debug workflow logic.

7. Tree View

Accessed from the DAGs Tab or DAG Details page.

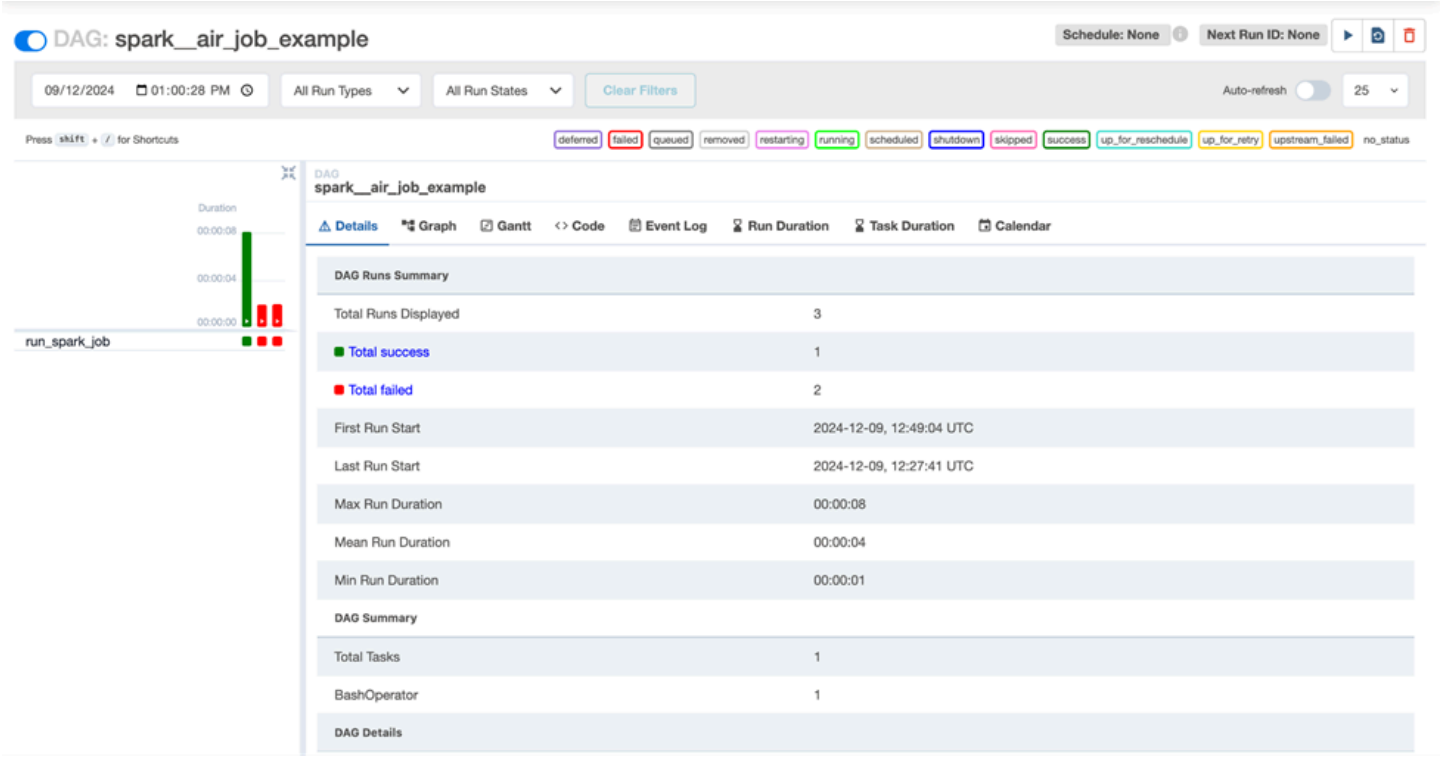
- **Purpose**: Hierarchical view of task executions.
- **Features**:
    - Task statuses for each run.
    - Links to logs and details.
- **Use Case**: Quickly identify failing tasks.

8. Task Instance Details

Accessed by clicking a task in any view (Graph, Tree, etc.).

- **Purpose**: Detailed task execution info.
- **Features**:
    - Logs, start/end time, retries, execution duration.
- **Use Case**: Diagnose task-specific issues effectively.

Dag Screen



Quick Tip:

Explore these tabs regularly to monitor DAGs, troubleshoot issues, and optimize performance.

Case Study: End-to-End Use Case - 24 Dec 24

Scenario:

You are working as a Big Data Engineer for an e-commerce platform that handles two types of event streams:

1. **User Click Events**: Captures user interactions on the website, such as pages visited, devices used, and locations.
2. **Order Details Events**: Tracks orders placed, including details like order amount, payment method, and purchased items.

Goal:

1. Process these streams and write data into respective locations.
2. Develop a Spark pipeline to query these locations and create other useful tables for data analytics.

Step 1: Event Generator

Before processing and analyzing the data using Spark and Hive, we simulate real-time data streams since there is no actual application producing the data. The event generator will:

1. Simulate **User Click Events** and **Order Placed Events**.
2. Send these events to respective Kafka topics: `user_click` and `order_details_topic`.

Complete Code: Event Generator

```
from confluent_kafka import Producer
import json
import random
import time
from faker import Faker

# Initialize Faker for data generation
fake = Faker()

# Kafka Configuration
KAFKA_BROKER = 'localhost:9092'  # Update with your Kafka broker address
USER_CLICK_TOPIC = 'user_click'
ORDER_DETAILS_TOPIC = 'order_details_topic'
```

```
# Kafka Producer Initialization
producer = Producer({'bootstrap.servers': KAFKA_BROKER})

# Function to simulate user click events
def produce_user_click_event(event_count):
    for _ in range(event_count):
        event_data = {
            "event_type": "user_click",
            "user_id": random.randint(1, 1000),
            "timestamp": time.time(),
            "page": fake.uri_path(),
            "device": random.choice(["Desktop", "Mobile", "Tablet"]),
            "location": fake.city()
        }
        print(f"Producing User Click Event: {json.dumps(event_data, indent=2)}")
        producer.produce(USER_CLICK_TOPIC, json.dumps(event_data))
    producer.flush()
    print(f"{event_count} User Click Events Produced Successfully!")

# Function to simulate order placed events
def produce_order_placed_event(event_count):
    for _ in range(event_count):
        order_data = {
            "event_type": "order_placed",
            "order_id": fake.uuid4(),
            "user_id": random.randint(1, 1000),
            "order_amount": round(random.uniform(10, 5000), 2),
            "transaction_id": fake.uuid4(),
            "payment_method": random.choice(["Credit Card", "Debit Card", "UPI",
"Net Banking"]),
            "timestamp": time.time(),
            "items": [
                {
                    "item_id": random.randint(1, 10000),
                    "item_name": fake.word(),
                    "price": round(random.uniform(5, 100), 2)
                } for _ in range(random.randint(1, 5))
            ]
        }
        print(f"Producing Order Placed Event: {json.dumps(order_data, indent=2)}")
        producer.produce(ORDER_DETAILS_TOPIC, json.dumps(order_data))
    producer.flush()
    print(f"{event_count} Order Placed Events Produced Successfully!")
```

```
# Main Function to Trigger Events
def main():
    print("Event Producer for Kafka Started...")
    print("Available Events:")
    print("1. User Click Event")
    print("2. Order Placed Event")
    print("3. Exit")

    while True:
        choice = input("\nEnter the number of the event to trigger (1/2/3): ").strip()
        if choice in ["1", "2"]:
            try:
                event_count = int(input("How many events do you want to generate?
").strip())
                if event_count <= 0:
                    print("Please enter a positive number!")
                    continue
            except ValueError:
                print("Invalid input! Please enter a valid number.")
                continue
            if choice == "1":
                produce_user_click_event(event_count)
            elif choice == "2":
                produce_order_placed_event(event_count)
        elif choice == "3":
            print("Exiting...")
            break
        else:
            print("Invalid choice! Please enter 1, 2, or 3.")

if __name__ == "__main__":
    main()
```

Explanation:

1. **Simulating Data**:
   - Faker library generates realistic data like city names, user IDs, and UUIDs.
   - Random functions are used to simulate variability in devices, pages, and items.
2. **Kafka Integration**:
   - Events are sent to Kafka topics using the Confluent Kafka Producer.
3. **User Interaction**:
   - The script prompts the user to choose between generating click or order events and how many events to produce.
4. **Real-time Streaming**:
   - Events are continuously streamed into Kafka topics, mimicking a real-world e-commerce platform.

This setup is the foundation for further processing with Spark pipelines and Hive analytics.

Code Explanation in Chunks

---

1. Imports and Initializations

```
from confluent_kafka import Producer
import json
import random
import time
from faker import Faker

fake = Faker()
```

- **Purpose**: Import libraries for Kafka integration, data generation, and time manipulation.
- **`Faker`**: Generates realistic fake data like user IDs, city names, UUIDs, etc.

2. Kafka Configuration

```
KAFKA_BROKER = 'localhost:9092'
USER_CLICK_TOPIC = 'user_click'
ORDER_DETAILS_TOPIC = 'order_details_topic'

producer = Producer({'bootstrap.servers': KAFKA_BROKER})
```

- **Kafka Broker**: Specifies the address of the Kafka server (`localhost:9092` in this case).
- **Topics**: Predefined Kafka topics:
  - `user_click`: For user interaction events.
  - `order_details_topic`: For order-related events.
- **Producer**: Initialized with the Kafka broker to publish messages to topics.

3. Simulating User Click Events

```
def produce_user_click_event(event_count):
    for _ in range(event_count):
        event_data = {
            "event_type": "user_click",
            "user_id": random.randint(1, 1000),
            "timestamp": time.time(),
            "page": fake.uri_path(),
            "device": random.choice(["Desktop", "Mobile", "Tablet"]),
            "location": fake.city()
        }
        print(f"Producing User Click Event: {json.dumps(event_data, indent=2)}")
        producer.produce(USER_CLICK_TOPIC, json.dumps(event_data))
    producer.flush()
```

- **Functionality**:
  - Creates `event_count` random user click events.
  - Each event contains details like:
    - **Event type**: `user_click`.
    - **User ID**: Randomly generated.
    - **Timestamp**: Current time in seconds.
    - **Page**: Random URI path.
    - **Device**: Random device type.
    - **Location**: Random city name.
  - Sends each event to the Kafka topic `user_click`.

4. Simulating Order Placed Events

```
def produce_order_placed_event(event_count):
    for _ in range(event_count):
        order_data = {
            "event_type": "order_placed",
            "order_id": fake.uuid4(),
            "user_id": random.randint(1, 1000),
            "order_amount": round(random.uniform(10, 5000), 2),
            "transaction_id": fake.uuid4(),
            "payment_method": random.choice(["Credit Card", "Debit Card", "UPI",
"Net Banking"]),
            "timestamp": time.time(),
            "items": [
                {
                    "item_id": random.randint(1, 10000),
                    "item_name": fake.word(),
                    "price": round(random.uniform(5, 100), 2)
                } for _ in range(random.randint(1, 5))
            ]
        }
        print(f"Producing Order Placed Event: {json.dumps(order_data,
indent=2)}")
        producer.produce(ORDER_DETAILS_TOPIC, json.dumps(order_data))
    producer.flush()
```

- **Functionality**:
  - Creates `event_count` random order-placed events.
  - Each event contains:
    - **Event type**: `order_placed`.
    - **Order ID** and **Transaction ID**: Unique UUIDs.
    - **User ID**: Randomly generated.
    - **Order Amount**: Random amount within a range.
    - **Payment Method**: Random choice from available options.
    - **Timestamp**: Current time in seconds.
    - **Items**: A nested list of order items with:
      - **Item ID**: Random unique ID.
      - **Item Name**: Random word.
      - **Price**: Randomly generated item price.
  - Sends each event to the Kafka topic `order_details_topic`.

5. Main Function

```
def main():
    while True:
        choice = input("\nEnter the number of the event to trigger (1/2/3): ").strip()
        if choice in ["1", "2"]:
            try:
                event_count = int(input("How many events do you want to generate?
").strip())
                if event_count <= 0:
                    print("Please enter a positive number!")
                    continue
            except ValueError:
                print("Invalid input! Please enter a valid number.")
                continue
            if choice == "1":
                produce_user_click_event(event_count)
            elif choice == "2":
                produce_order_placed_event(event_count)
        elif choice == "3":
            print("Exiting...")
            break
        else:
            print("Invalid choice! Please enter 1, 2, or 3.")
```

- **Purpose**:
  - Provides an interactive command-line interface (CLI) to:
    - Trigger **User Click Events**.
    - Trigger **Order Placed Events**.
    - Exit the program.
  - Ensures valid inputs (e.g., positive numbers for event counts).

Summary:

This program efficiently simulates real-time user activity and order data streams, sending them to Kafka topics for further processing. Each function focuses on one specific task, ensuring modularity and clarity.

==========

Detailed Explanation of application.properties

Visit

https://youtu.be/_MmSSunw_gM

Like | Comment | Subscribe

==========| **@GeekySanjay** |==========

Step 2: Kafka Setup

This step ensures that Kafka is properly set up, the required topics are created, and events are functioning as expected.

Step-by-Step Instructions

---

1. Start Kafka

- Navigate to your Kafka installation directory and start the required services:

Start ZooKeeper:
`bin/zookeeper-server-start.sh config/zookeeper.properties`
Start Kafka Broker:
`bin/kafka-server-start.sh config/server.properties`

---

2. Create Kafka Topics

- Use the following commands to create the required topics (`user_click` and `order_details_topic`) if they do not already exist:

Create `user_click` topic:
`bin/kafka-topics.sh --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --topic user_click`
Create `order_details_topic`:
`bin/kafka-topics.sh --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --topic order_details_topic`

---

3. Verify Topics

Check if the topics were successfully created:
`bin/kafka-topics.sh --list --bootstrap-server localhost:9092`

You should see the following output:
```
user_click
order_details_topic
```

---

4. Run the Event Generator Script

Execute the Python script created in Step 1 to generate and send events to Kafka:
`python event_generator.py`

- **Instructions**:
  - Choose the event type:
    - 1 for User Click Events.
    - 2 for Order Placed Events.
  - Enter the number of events to generate (e.g., 10).

5. Consume Messages from Kafka Topics

- Open a new terminal to start a Kafka consumer and verify if events are being received:

Consume messages from `user_click`:

```
bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic
user_click --from-beginning
```

Consume messages from `order_details_topic`:

```
bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic
order_details_topic --from-beginning
```

By following these steps, you ensure that Kafka is correctly set up and can produce, send, and consume events seamlessly.

Step 3: Streaming Applications

In this step, we'll write and execute Spark Streaming applications to consume data from Kafka topics (`user_click` and `order_details_topic`), process the data, and save it as Parquet files.

Streaming Application 1: User Click Events

This Spark application consumes user click events from Kafka, processes them, and writes the data to a Parquet file.

Spark Application Code

```
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField, StringType, LongType,
TimestampType
from pyspark.sql.functions import from_json, col

# Kafka Configuration
KAFKA_BROKER = "localhost:9092"
USER_CLICK_TOPIC = "user_click"

# Initialize Spark Session
spark = SparkSession.builder \
    .appName("User Click Streaming Application") \
    .getOrCreate()

# Define Schema for User Click Events
user_click_schema = StructType([
    StructField("event_type", StringType(), True),
    StructField("user_id", LongType(), True),
    StructField("page", StringType(), True),
    StructField("device", StringType(), True),
    StructField("location", StringType(), True),
    StructField("timestamp", LongType(), True)
])
```

```
# Step 1: Read Data from Kafka Topic
raw_stream_df = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", KAFKA_BROKER) \
    .option("subscribe", USER_CLICK_TOPIC) \
    .load()

# Step 2: Deserialize JSON and Apply Schema
user_click_df = raw_stream_df.selectExpr("CAST(value AS STRING)") \
    .select(from_json(col("value"), user_click_schema).alias("data")) \
    .select(
        col("data.event_type"),
        col("data.user_id"),
        col("data.page"),
        col("data.device"),
        col("data.location"),
        (col("data.timestamp") / 1000).cast(TimestampType()).alias("event_time")
    )

# Step 3: Write Data to Parquet Files
checkpoint_dir = "/tmp/spark_checkpoints/user_click"
output_path = "/tmp/spark_output/user_click"

query = user_click_df.writeStream \
    .outputMode("append") \
    .format("parquet") \
    .option("path", output_path) \
    .option("checkpointLocation", checkpoint_dir) \
    .start()

query.awaitTermination()
```

Code Explanation

1. Kafka Configuration

- **`KAFKA_BROKER`**: Specifies the Kafka server (localhost:9092).
- **`USER_CLICK_TOPIC`**: Kafka topic to subscribe to for user click events.

2. Spark Session Initialization

- Initializes the Spark application and assigns a name: "**User Click Streaming Application**".

3. Schema Definition

- Defines the structure of user click events with fields such as `event_type`, `user_id`, `page`, `device`, `location`, and `timestamp`.

4. Reading Data from Kafka

- Uses `spark.readStream` to consume data from the `user_click` Kafka topic.
- Reads Kafka messages as raw byte streams.

5. Data Processing

- **Deserialize JSON**: Converts Kafka message values from JSON format to a structured schema.
- **Schema Application**: Maps JSON data to predefined fields and formats the `timestamp` as a `TimestampType`.

6. Writing Data to Parquet

- **Checkpoint Directory**: Ensures data recovery in case of job failure.
- **Output Path**: Specifies the directory where processed data is saved as Parquet files.
- Uses the `append` mode to add new data continuously.

7. Query Execution

- Starts the streaming query and writes data in real-time to the specified Parquet directory.

Running the Application

1. **Start Kafka**: Ensure Kafka is running (refer to Step 2).

Execute Spark Application: Run the script using the following command:
`spark-submit user_click_streaming.py`

2. **Monitor Output**:
    - Verify the Parquet files in `/tmp/spark_output/user_click`.
    - Checkpointing data will be saved in `/tmp/spark_checkpoints/user_click`.

This completes the setup and execution of the Spark Streaming application for processing `user_click` events. Follow a similar approach for the `order_details_topic`.

==========

Designing Class and Schema Diagrams for Splitwise: A Step-by-Step Guide

==========

Streaming Application 2: Order Details Events

This application processes order details events from the Kafka topic `order_details_topic` and writes the output to Parquet files.

Spark Application Code (Simplified Explanation)

```
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField, StringType, LongType,
DoubleType, TimestampType, ArrayType
from pyspark.sql.functions import from_json, col, explode

# Kafka Configuration
ORDER_DETAILS_TOPIC = "order_details_topic"


# Step 1: Initialize Spark Session
spark = SparkSession.builder \
    .appName("Order Details Streaming Application") \
    .getOrCreate()

# Step 2: Define Schema for Order Details Events
order_items_schema = StructType([
    StructField("item_id", LongType(), True),
    StructField("item_name", StringType(), True),
    StructField("price", DoubleType(), True)
])

order_details_schema = StructType([
    StructField("event_type", StringType(), True),
    StructField("order_id", StringType(), True),
    StructField("user_id", LongType(), True),
    StructField("order_amount", DoubleType(), True),
    StructField("transaction_id", StringType(), True),
    StructField("payment_method", StringType(), True),
    StructField("timestamp", DoubleType(), True),
    StructField("items", ArrayType(order_items_schema), True)
])

# Step 3: Read Data from Kafka Topic
raw_order_stream_df = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("subscribe", ORDER_DETAILS_TOPIC) \
    .load()
```

```
# Step 4: Deserialize JSON and Apply Schema
order_details_df = raw_order_stream_df.selectExpr("CAST(value AS STRING)") \
    .select(from_json(col("value"), order_details_schema).alias("data")) \
    .select(
        col("data.event_type"),
        col("data.order_id"),
        col("data.user_id"),
        col("data.order_amount"),
        col("data.transaction_id"),
        col("data.payment_method"),
        (col("data.timestamp") / 1000).cast(TimestampType()).alias("event_time"),
        col("data.items")
    )

# Step 5: Flatten the Items Array for Analysis
flattened_order_details_df = order_details_df \
    .withColumn("item", explode(col("items"))) \
    .select(
        col("event_type"),
        col("order_id"),
        col("user_id"),
        col("order_amount"),
        col("transaction_id"),
        col("payment_method"),
        col("event_time"),
        col("item.item_id").alias("item_id"),
        col("item.item_name").alias("item_name"),
        col("item.price").alias("item_price")
    )

# Step 6: Write Data to Parquet Files
order_checkpoint_dir = "/tmp/spark_checkpoints/order_details"
order_output_path = "/tmp/spark_output/order_details"

order_query = flattened_order_details_df.writeStream \
    .outputMode("append") \
    .format("parquet") \
    .option("path", order_output_path) \
    .option("checkpointLocation", order_checkpoint_dir) \
    .start()

order_query.awaitTermination()
```

Explanation in Simple Steps

1. **Kafka Configuration**
   - Topic: `order_details_topic`
   - Kafka server address: `localhost:9092`
2. **Define Schema**
   - **Order Items Schema**: Each order contains items with `item_id`, `item_name`, and `price`.
   - **Order Details Schema**: Includes `order_id`, `user_id`, `order_amount`, and `items` (array of `order_items_schema`).
3. **Read Data from Kafka**
   - Use Spark to read messages from the Kafka topic.
   - Messages are in JSON format, so they are deserialized into structured data.
4. **Flatten Items for Analysis**
   - Use `explode` to break the `items` array into individual rows, making it easier to analyze.
5. **Write to Parquet**
   - Processed data is saved to Parquet files for storage and later use.
   - Checkpoints are used to track progress and ensure fault tolerance.

Output

- **Checkpoint Directory**: `/tmp/spark_checkpoints/order_details`
- **Parquet Files Location**: `/tmp/spark_output/order_details`

This setup processes streaming order events in real-time and saves them in a structured format.

How to Run the Applications

Step 1: Submit the Spark Applications

Save the two applications as `user_click_stream.py` and `order_details_stream.py`.

Run them using the `spark-submit` command:

```
spark-submit user_click_stream.py
spark-submit order_details_stream.py
```

Step 2: Verify the Output

1. Check the output directories for generated Parquet files:
   - **User Click Events**: `/tmp/spark_output/user_click`
   - **Order Details Events**: `/tmp/spark_output/order_details`

   Use `ls` to verify the files in these directories:

   ```
   ls /tmp/spark_output/user_click
   ls /tmp/spark_output/order_details
   ```

Step 4: Creating External Hive Tables

Create external Hive tables for querying the Parquet files generated by the Spark applications without moving or managing the data in Hive.

External Table for User Click Events

Table Creation Query
Run this query in Hive or an SQL editor connected to your Hive Metastore:

```
CREATE EXTERNAL TABLE user_click (
    event_type STRING,
    user_id BIGINT,
    page STRING,
    device STRING,
    location STRING,
    event_time TIMESTAMP
)
STORED AS PARQUET
LOCATION '/tmp/spark_output/user_click';
```

Explanation of the Query

1. **Columns**:
    - Matches the schema of `user_click_df` in the Spark application.
    - `event_time` is stored as a TIMESTAMP.
2. **STORED AS PARQUET**:
    - Specifies that the table reads data stored in Parquet format.
3. **LOCATION**:
    - Points to the directory where Spark writes the `user_click` Parquet files.
4. **EXTERNAL TABLE**:
    - Ensures Hive maps the table to existing data without owning or deleting it when the table is dropped.

This approach allows you to query the streaming data directly from Hive while keeping the data under Spark's control.

External Table for Order Details Events

Table Creation Query

Run the following query in Hive to create the external table:

```
CREATE EXTERNAL TABLE order_details (
    event_type STRING,
    order_id STRING,
    user_id BIGINT,
    order_amount DOUBLE,
    transaction_id STRING,
    payment_method STRING,
    event_time TIMESTAMP,
    item_id BIGINT,
    item_name STRING,
    item_price DOUBLE
)
STORED AS PARQUET
LOCATION '/tmp/spark_output/order_details';
```

Explanation of the Query

1. **Columns**:
    - Matches the schema of `flattened_order_details_df` from the Spark application.
    - Includes `item_id`, `item_name`, and `item_price` for detailed item-level analysis.
2. **STORED AS PARQUET**:
    - Indicates that the data is stored in Parquet file format.
3. **LOCATION**:
    - Specifies the directory where Spark writes the `order_details` Parquet files.
4. **EXTERNAL TABLE**:
    - Ensures the table maps to the existing data without modifying or deleting the Parquet files when the table is dropped.

Verify the Tables

After creating the external tables, verify they are mapped correctly to the Parquet files.

List Tables in Hive:
Use this command to check the created tables:
`SHOW TABLES;`

1. **Describe the Schema**:
   Verify the structure of the tables:
   `DESCRIBE user_click;`
   `DESCRIBE order_details;`
2. **Query the Tables**:
   Retrieve a sample of the data to confirm the setup:
   `SELECT * FROM user_click LIMIT 10;`
   `SELECT * FROM order_details LIMIT 10;`

This process ensures the external tables are properly set up and ready for querying in Hive.

Step 5: Enhancing Data

In this step, we will create a Spark application to join the two Hive tables (`user_click` and `order_details`) on the `user_id` field. The resulting dataset will include `user_id` and `location` from the `user_click` table, and all other fields from the `order_details` table. The enhanced data will be written to a new Hive table called `order_details_enhanced`.

Spark Application Code

```
from pyspark.sql import SparkSession

# Initialize Spark Session with Hive Support
spark = SparkSession.builder \
    .appName("Enhance Order Details Data") \
    .config("spark.sql.catalogImplementation", "hive") \
    .enableHiveSupport() \
    .getOrCreate()

# Step 1: Read Data from Hive Tables
print("Reading data from Hive tables...")
user_click_df = spark.sql("SELECT user_id, location FROM user_click")
order_details_df = spark.sql("SELECT * FROM order_details")

# Step 2: Join Data on user_id
print("Joining data...")
order_details_enhanced_df = order_details_df \
    .join(user_click_df, "user_id", "left") \
    .select(
        order_details_df["user_id"],
        user_click_df["location"],
        order_details_df["event_type"],
        order_details_df["order_id"],
        order_details_df["order_amount"],
        order_details_df["transaction_id"],
        order_details_df["payment_method"],
        order_details_df["event_time"],
        order_details_df["item_id"],
        order_details_df["item_name"],
        order_details_df["item_price"]
    )

# Step 3: Write Enhanced Data to Hive Table
print("Writing enhanced data to Hive table...")
order_details_enhanced_df.write.mode("overwrite").saveAsTable("order_details_enha
nced")
print("Enhanced data written successfully!")
```

Explanation

1. **Read Data from Hive Tables**:
   We read the `user_click` and `order_details` data from Hive tables using Spark SQL.
2. **Join Data on `user_id`**:
   We join the two dataframes (`user_click_df` and `order_details_df`) using the `user_id` column. The join type is `left`, meaning all records from the `order_details` table will be included, along with matching data from the `user_click` table.
3. **Write Enhanced Data to Hive**:
   The resulting `order_details_enhanced_df` dataframe, which contains both the `location` field from `user_click` and all the fields from `order_details`, is saved as a new Hive table called `order_details_enhanced`.

This process enhances the data by adding user location details to the order details.

Step 6: Create Airflow DAG to Schedule the Spark Application

In this step, we will create an Airflow DAG to schedule the `order_details_enhanced.py` Spark application to run every hour.

Airflow DAG Code

```
from airflow import DAG
from airflow.operators.bash import BashOperator
from datetime import datetime, timedelta

# Default arguments for the DAG
default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}

# Define the DAG
dag = DAG(
    'order_details_enhanced_dag',
    default_args=default_args,
    description='Run Spark application to create order_details_enhanced table',
    schedule_interval='@hourly',  # Runs every hour
    start_date=datetime(2024, 1, 1),  # Replace with the desired start date
    catchup=False
)
```

```
# Task to run the Spark application
run_spark_job = BashOperator(
    task_id='run_order_details_enhanced',
    bash_command='spark-submit /path/to/order_details_enhanced.py',
    dag=dag
)

# Define task dependencies (if any)
# In this case, there's only one task, so no dependencies are needed
```

Explanation

1. **Default Arguments for the DAG**:
   The `default_args` dictionary sets up the owner, retry behavior, and other default configurations for the DAG.
2. **Define the DAG**:
   The `DAG` is created with a name (`order_details_enhanced_dag`), a description, and a schedule. In this case, the DAG runs every hour (`@hourly`), starting from `2024-01-01`.
3. **Task to Run Spark Application**:
   The `BashOperator` is used to run the `order_details_enhanced.py` Spark application using the `spark-submit` command.
4. **Task Dependencies**:
   Since this DAG has only one task, no task dependencies are needed.

This Airflow DAG ensures that the Spark application runs every hour to create the `order_details_enhanced` table.

Interview Questions Preparation - 3 Jan 25

- **Why is GROUP BY faster than DISTINCT for deduplication in Hive?**
- **What happens if you use both SORT BY and ORDER BY in the same query?**
- **What is the purpose of ANALYZE TABLE in Hive, and how does it affect query optimization?**
- **When would you use a Common Table Expression (CTE) over a Temporary Table in Hive?**
- **How does Hive handle NULL values in GROUP BY, JOIN, and ORDER BY operations?**
- **What are the differences between LIMIT, FETCH, and OFFSET in Hive queries? When would you use each?**
- **What are the advantages and disadvantages of enabling vectorization in Hive? When might vectorization not be suitable?**
- **What happens when you create a Hive table with a schema different from the schema of the underlying file? How does Hive handle schema mismatch?**
- **Explain the Role of Hive's Dynamic Partition Mode. What Are the Risks of Setting It to NONSTRICT?**

Q1: Why is GROUP BY faster than DISTINCT for deduplication in Hive?

Solution:

1. **Execution Difference**:
   - **DISTINCT**: In Hive, using `DISTINCT` involves a single Reduce phase to remove duplicates. All unique values are collected and shuffled to the reducers. This process can be slow and use many resources, especially for large datasets with high cardinality (many unique values).
   - **GROUP BY**: When using `GROUP BY`, Hive first aggregates data in the Map phase (local aggregation). This reduces the amount of data shuffled to the Reduce phase, making `GROUP BY` more efficient for deduplication.
2. **Performance Implications**:
   - `DISTINCT` skips the local aggregation step, so it involves more shuffling of data.
   - `GROUP BY` uses local aggregation to minimize the amount of data shuffled.

   Example:
   ```
   -- Using DISTINCT
   SELECT DISTINCT user_id FROM transactions;

   -- Using GROUP BY
   SELECT user_id FROM transactions GROUP BY user_id;
   ```
3. **Conclusion**:
   For large datasets, the `GROUP BY` query is usually faster because the Map phase pre-aggregates the data.
4. **Recommendation**:
   If you only need deduplication, prefer `GROUP BY` for better performance.

Q2: What happens if you use both SORT BY and ORDER BY in the same query?

Solution:

1. **Definitions**:
   - **SORT BY**: Sorts data within each reducer. The output is not globally sorted but is partially sorted within each partition.
   - **ORDER BY**: Ensures a globally sorted output by using a single reducer. This can cause slow performance for large datasets.
2. **When Used Together**:
   - `SORT BY` happens first, sorting data locally within each reducer.
   - `ORDER BY` collects the output from all reducers and performs a global sort.
3. **Implications**:
   - Using both is uncommon because `ORDER BY` already handles global sorting.
   - Adding `SORT BY` before `ORDER BY` may increase complexity without significant benefits unless there is a specific reason to pre-sort data.

   Example:
   ```
   SELECT * FROM employees SORT BY department_id ORDER BY salary;
   ```
4. **Outcome**: First, rows are sorted by `department_id` within each reducer. Then, rows are globally sorted by `salary`.
5. **Recommendation**: Avoid combining `SORT BY` and `ORDER BY` unless you have a clear use case, as it increases complexity and can reduce efficiency.

Q3: What is the purpose of ANALYZE TABLE in Hive, and how does it affect query optimization?

Solution:

1. **Purpose**:
   `ANALYZE TABLE` collects statistics about a table or partition in Hive. These statistics are saved in the Hive Metastore and help the Hive query planner create better execution plans.
2. **Statistics Collected**:
   - Row counts.
   - Data size (total size in bytes).
   - Number of files and partitions.
   - Column-level statistics (e.g., distinct values, min/max values).

     Usage:
     ```
     ANALYZE TABLE sales COMPUTE STATISTICS;
     ANALYZE TABLE sales COMPUTE STATISTICS FOR COLUMNS;
     ```

3. **Impact on Query Optimization**:
   - The Hive optimizer uses these statistics to choose efficient join methods, like map-side join or reduce-side join.
   - It helps in partition pruning, dynamic partitioning, and sorting strategies.
   - Without accurate statistics, the query planner might create inefficient plans, leading to slower queries.
4. **Best Practices**:
   - Always run `ANALYZE TABLE` after loading or updating large amounts of data.
   - Update statistics regularly to keep query plans efficient as data changes.

Q4: When would you use a Common Table Expression (CTE) over a Temporary Table in Hive?

Solution:

1. **Common Table Expression (CTE)**:
   - **Definition**:
     A CTE is a named subquery created using the `WITH` clause and works only within the query where it is defined.
   - **When to Use**:
     - To improve query readability and reuse.
     - When the intermediate result is used multiple times within the same query.
     - When the intermediate result is not needed outside the query.

       Example:
       ```
       WITH top_customers AS (
         SELECT customer_id, SUM(amount) AS total_spent
         FROM transactions
         GROUP BY customer_id
         HAVING total_spent > 1000
       )
       SELECT * FROM top_customers ORDER BY total_spent DESC;
       ```

2. **Temporary Table**:
    - **Definition**: A temporary table is created by the user and exists for the whole session.
    - **When to Use**:
        - For large intermediate results that will be reused in multiple queries.
        - When data needs to be shared across queries in the same session.
        - For debugging or when the intermediate data is too large to keep in memory.

        Example:
        ```
        CREATE TEMPORARY TABLE temp_top_customers AS
        SELECT customer_id, SUM(amount) AS total_spent
        FROM transactions
        GROUP BY customer_id
        HAVING total_spent > 1000;
        SELECT * FROM temp_top_customers ORDER BY total_spent DESC;
        ```

3. **Comparison**:

Feature	CTE	Temporary Table
Scope	Single query	Entire session
Reusability	Within the query	Across multiple queries
Performance	Faster for in-memory	Slower due to disk I/O
Ease of Use	Inline with query	Needs explicit creation

4. **Recommendation**:
    - Use **CTEs** for simple, single-query scenarios for better readability.
    - Use **Temporary Tables** when handling large datasets or sharing results across multiple queries.

Q5: How does Hive handle NULL values in GROUP BY, JOIN, and ORDER BY operations?

GROUP BY:

- Hive treats `NULL` as a distinct value in the `GROUP BY` clause.
- Rows with `NULL` in the grouped column(s) are aggregated into their own group.

Example:
```
SELECT department_id, COUNT(*)
FROM employees
GROUP BY department_id;
```

If some `department_id` values are `NULL`, the result might look like:

department_id	count
NULL	5
101	20
102	15

JOIN:

- Hive follows standard SQL behavior:
  - **INNER JOIN**: Rows with `NULL` in the join key are excluded because `NULL = NULL` is evaluated as `FALSE`.
  - **OUTER JOIN types**:
    - **LEFT OUTER JOIN**: Rows with `NULL` in the join key from the left table are included, with `NULL` values in the right table's columns.
    - **RIGHT OUTER JOIN**: Rows with `NULL` in the join key from the right table are included, with `NULL` values in the left table's columns.
    - **FULL OUTER JOIN**: Rows with `NULL` in the join key from either table are included.

Example:

```
SELECT e.employee_id, d.department_name
FROM employees e
LEFT JOIN departments d
ON e.department_id = d.department_id;
```

ORDER BY:

- By default, Hive places `NULL` values:
  - At the **beginning** for ascending order (`ASC`).
  - At the **end** for descending order (`DESC`).
- This behavior can be changed using `NULLS FIRST` or `NULLS LAST`.

Example:

```
SELECT employee_name, department_id
FROM employees
ORDER BY department_id ASC NULLS LAST;
```

Best Practices:

- Be cautious with `NULL` values in `JOIN` conditions to avoid unintended row exclusions.
- Use functions like `COALESCE` to handle `NULL` values explicitly.

Example:

```
SELECT COALESCE(department_id, -1) AS department_id
FROM employees;
```

Q6: What are the differences between `LIMIT`, `FETCH`, and `OFFSET` in Hive queries? When would you use each?

LIMIT:

- Restricts the number of rows returned in the query result.
- **Use case**: Sampling data or debugging.

Syntax:
```
SELECT * FROM employees LIMIT 10;
```

- **Behavior**: Simple and direct, but no control over skipping rows (pagination).

FETCH:

- An ANSI-compliant way to limit rows, introduced in Hive 3.0.
- Can specify both the number of rows and an offset when combined with `OFFSET`.
- **Use case**: Fetching a precise subset of rows.

Syntax:
```
SELECT * FROM employees ORDER BY employee_id FETCH FIRST 10 ROWS ONLY;
```

OFFSET:

- Skips the first `n` rows of the query result.
- Typically used with `FETCH` for pagination.
- **Use case**: Implementing pagination in dashboards or reports.

Syntax:
```
SELECT * FROM employees ORDER BY employee_id OFFSET 5 ROWS FETCH NEXT 10 ROWS ONLY;
```

- **Behavior**: Skips rows before returning results.

Comparison Table

Feature	`LIMIT`	`FETCH`	`OFFSET`
Introduced	Early Hive versions	Hive 3.0 and later	Hive 3.0 and later
Pagination	No	Yes (with `OFFSET`)	Yes
Standards	Non-standard SQL	ANSI-compliant SQL	ANSI-compliant SQL
Use Case	Quick sampling/debugging	Precise row selection	Pagination with row skipping

Recommendation:

- Use `LIMIT` for quick testing or sampling data.
- Use `FETCH` with `OFFSET` for advanced scenarios like paginated dashboards.

Q7: What are the advantages and disadvantages of enabling vectorization in Hive? When might vectorization not be suitable?

What is Vectorization in Hive?

- Vectorization allows Hive to process **batches of rows** (e.g., 1024 rows at a time) instead of processing one row at a time.
- It operates on **columnar data formats** (e.g., ORC, Parquet).
- Improves efficiency by reducing overhead from function calls and optimizing CPU utilization.

Advantages

1. **Performance Boost**:
    - Faster query execution, especially for arithmetic or aggregation-heavy queries.
2. **Efficient CPU Utilization**:
    - Processes batches of rows in a single operation, reducing repetitive function calls.
3. **Reduced I/O Overhead**:
    - Works well with columnar storage formats, enhancing data locality.
4. **Scalability**:
    - Handles large datasets effectively by cutting down execution time.

Disadvantages

1. **Compatibility Issues**:
    - Not all queries or functions are supported.
        - Complex UDFs may not be vectorized.
        - Joins with non-equijoin conditions might fail to vectorize.
2. **Memory Overhead**:
    - Batch processing consumes more memory, potentially leading to issues in resource-constrained environments.
3. **File Format Dependency**:
    - Performs best with **ORC** and **Parquet**.
    - Limited performance gains with formats like **text** or **CSV**.

When is Vectorization Not Suitable?

1. Queries involving unsupported operations (e.g., specific UDFs).
2. Environments with limited memory, where batch processing can cause **Out of Memory (OOM)** errors.
3. Using non-columnar storage formats like **text** or **CSV**.

Best Practices

- Enable vectorization for tables stored in **ORC** or **Parquet** formats.
- Ensure sufficient memory allocation for batch processing.

  Configure Hive settings to enable vectorization:

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

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

Q8: What happens when you create a Hive table with a schema different from the schema of the underlying file? How does Hive handle schema mismatch?

In Hive, the table schema defines how data is interpreted during query execution. When the table schema and the underlying file schema differ, several outcomes occur depending on the mismatch type and query execution context.

Scenario 1: Column Count Mismatch

1. **Table schema has fewer columns than the file:**
   - Hive **ignores extra columns** in the file.
   - Only columns defined in the table schema are read and returned.
     Example:
     ```
     CREATE TABLE employees (id INT, name STRING)
     ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
     ```

     File content:
     ```
     1,John,HR
     2,Jane,Finance
     ```
     Query:
     ```
     SELECT * FROM employees;
     ```
     Output:
     ```
     1, John
     2, Jane
     ```

2. **Table schema has more columns than the file:**
   - Additional table columns are **filled with NULL values.**
     Example:
     ```
     CREATE TABLE employees (id INT, name STRING, department STRING)
     ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
     ```

     File content:
     ```
     1,John
     2,Jane
     ```
     Query:
     ```
     SELECT * FROM employees;
     ```
     Output:
     ```
     1, John, NULL
     2, Jane, NULL
     ```

Scenario 2: Data Type Mismatch

- Hive **attempts to cast** file values to the table schema's data type.
- If casting fails, Hive **returns NULL** for the affected values.

  Example:

  ```
  CREATE TABLE employees (id INT, name STRING)
  ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
  ```

File content:

```
a1,John
2,Jane
```

Query:

```
SELECT * FROM employees;
```

Output:

```
NULL, John
2, Jane
```

Scenario 3: Partitioning and Schema Mismatch

- If a **partitioned table's schema** does not match the file structure, the data for the partition is interpreted using the table schema.
- If the mismatch involves the **partition column itself,** queries may fail or return incorrect results.

Best Practices

1. **Use Schema Evolution:**
   - Hive supports schema evolution for formats like **Avro** and **Parquet**, ensuring backward compatibility for schema updates.
2. **Validate Data:**
   - Verify that the data matches the schema before loading files into Hive tables.
3. **Use External Tables:**
   - For loosely structured data, use `EXTERNAL` tables to avoid data loss during schema changes.
4. **Enable Strict Schema Checks:**
   - Use the following configuration to enforce schema compatibility:

     ```
     SET hive.strict.schema = true;
     ```

Q9: Explain the Role of Hive's Dynamic Partition Mode. What Are the Risks of Setting It to `NONSTRICT`?

Dynamic Partitioning in Hive allows data insertion into partitioned tables without specifying all partition column values. Hive dynamically determines the partition based on input data.

Modes of Dynamic Partitioning

1. **STRICT Mode (Default):**
   - **Requirement:** At least one partition column must be statically specified in the `INSERT` query.
   - **Benefit:** Prevents excessive partition creation, ensuring better resource management.
     Example:

     ```
     SET hive.exec.dynamic.partition.mode = strict;

     INSERT INTO TABLE sales PARTITION (year, month)
     SELECT product_id, amount, year, month
     FROM raw_sales
     WHERE year = 2023; -- Static partition on 'year'
     ```

2. **NONSTRICT Mode:**
    - **Requirement:** No need to statically specify any partition column.
    - **Behavior:** All partitions are created dynamically based on input data.
      Example:
      ```
      SET hive.exec.dynamic.partition.mode = nonstrict;

      INSERT INTO TABLE sales PARTITION (year, month)
      SELECT product_id, amount, year, month
      FROM raw_sales;
      ```

Advantages of Dynamic Partitioning

1. **Ease of Use:**
    - Simplifies data loading by removing the need to manually specify partition values.
2. **Scalability:**
    - Supports automated partition management, making it ideal for large-scale data ingestion.

Risks of Using `NONSTRICT` Mode

1. **Uncontrolled Partition Creation:**
    - **Cause:** Input data with many distinct values for partition columns.
    - **Consequences:**
        - **Metadata Overhead:** Excessive partitions strain the Hive Metastore.
        - **Performance Issues:** Increased partitions result in slower query execution due to more I/O and scanning.
          Example:
          ```
          -- Input data with 100,000 distinct 'year' and 'month'
          combinations

          INSERT INTO TABLE sales PARTITION (year, month)
          SELECT product_id, amount, year, month
          FROM raw_sales;
          ```
2. **Disk Space and Resource Exhaustion:**
    - Too many partitions can overwhelm storage systems, leading to inefficient resource use.
3. **Error-Prone Data Loading:**
    - Without validation, incorrect input (e.g., empty strings or `NULL` values) can create unintended partitions, causing data inconsistencies.

Best Practices

- Use **STRICT mode** for better control over partition creation.
- Validate input data to ensure correctness before loading into partitioned tables.
- Regularly monitor the Hive Metastore and partition count to manage metadata overhead.