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/

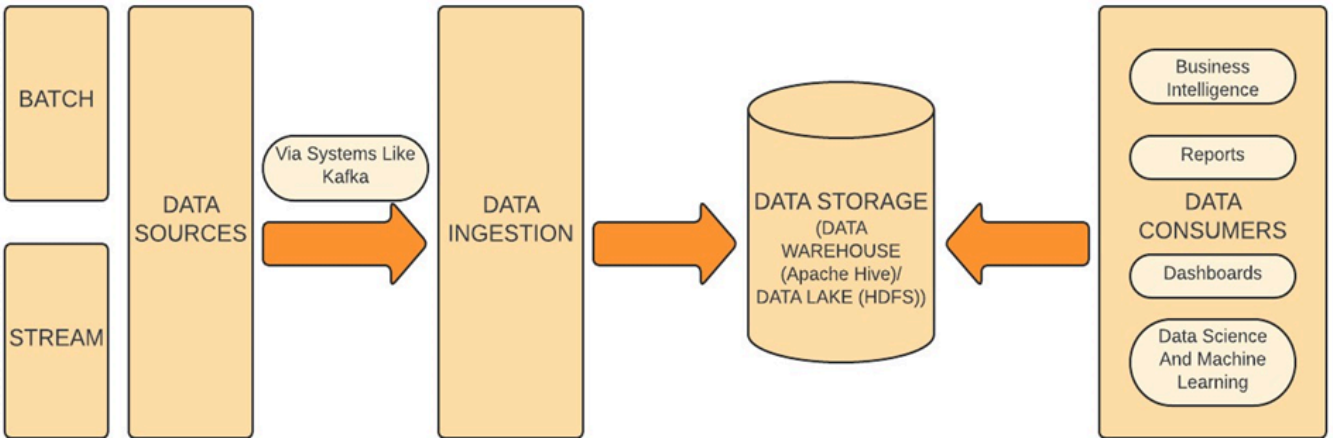
https://www.youtube.com/@GeekySanjay

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

Understanding Big Data Architecture - Oct 4

- **Data Sources**
- **Data Ingestion**
- **Data Storage**

In the previous lecture, we explored the basics of big data and what defines it. Today, we'll look at the high-level architecture of big data technologies and how they fit into the broader ecosystem.



Data Platform Architecture

1. **Data Sources**
   To understand data engineering, we first need to understand data itself. Data sources can be categorized based on the type of data or how it's delivered.
   - **Data Categorization by Type**: This aligns with the different types of big data we discussed.
     - **Structured Data**: Highly organized and easily searchable data, often stored in relational databases like MySQL or PostgreSQL.
     - **Semi-Structured Data**: Not fully organized like relational databases but has some structure, such as JSON or XML files.
     - **Unstructured Data**: Data without a predefined model, like log files, social media feeds, videos, and images.
   - **Data Categorization by Delivery**: Data can also be categorized by how it's delivered—either in real-time or in batches.
     - **Real-Time Streaming Data**: Generated continuously from sources like sensors, logs, user interactions, financial transactions, and social media. This data arrives in small, often irregular chunks that require immediate, low-latency processing to enable quick decisions and actions.

- **Batch Data**: Data that doesn't require immediate processing, often gathered over time from sources like databases, file systems, and logs. This data is processed later, typically for analysis. Real-time data can also be treated as batch data when processed over a time range.
  Example: Consider a customer ordering a product on Flipkart during a sale. Flipkart leadership might monitor this data in real-time to track the sale's performance. However, after the sale ends, the same data is analyzed as batch data to review overall sales figures, auditing, and monthly reports.

2. Data Ingestion

Once data is created, the next important step is to make sure it reaches the big data system so it can be analyzed. This process is called **data ingestion**. Since there are two main types of data—real-time and batch—there are different ways to handle each type: **real-time processing** (streaming) and **batch processing**.

Batch Processing

Batch processing involves processing large amounts of data collected over a period of time, all at once, usually at scheduled intervals.

Key Characteristics:

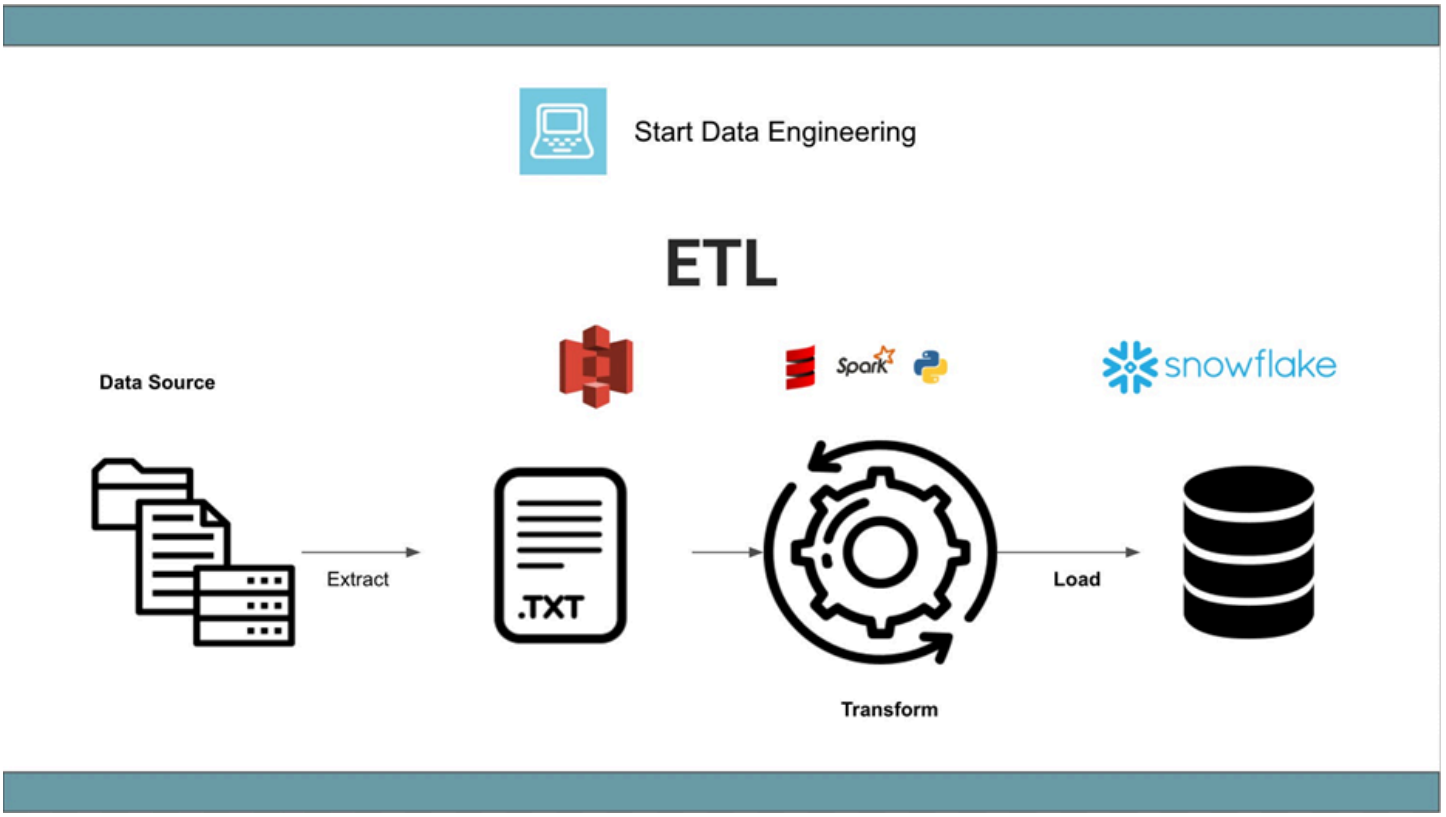
To understand batch processing, think of a company that has thousands of employees and processes payroll at the end of each month. Throughout the month, the company tracks employee hours, bonuses, and deductions. At month's end, it processes all of this data in one go to generate paychecks for each employee.

- **Volume**: Batch processing handles large amounts of data at once. In our example, the payroll system processes salary calculations for thousands of employees all at once. By waiting until the end of the month, the company can process a large volume of data more efficiently, calculating paychecks for everyone in a single run.
- **Latency**: There is usually a delay between data collection and processing. In the payroll example, work hours and bonuses are tracked daily, but processing only happens once a month. This delay (latency) is part of batch processing. So, there's a 30-day delay between data collection (daily logging of hours) and processing (calculating paychecks).
- **Throughput**: Batch processing is designed to process large data sets quickly by handling them all at once. Instead of calculating each paycheck one at a time, the payroll system calculates thousands of paychecks together, achieving high throughput and finishing the process quickly.
- **Complexity**: Batch processing can handle complex calculations since it has all the data at once. Payroll processing includes calculating work hours, bonuses, tax deductions, and more for each employee. With access to the entire data set at the end of the month, the system can perform these calculations in a single batch.

More Use Cases for Batch Processing:

- **Data Warehousing**: Loading and processing large data sets for analysis and reporting.
- **Log Analysis**: Analyzing server logs to create reports on system performance.

Technologies used:



- **Apache Hive on MapReduce (MR)**: A classic tool for batch processing that breaks data into smaller chunks, processes them in parallel, and then combines the results.
- **Apache Spark**: Known for its speed, Spark is also excellent for batch processing with its in-memory data processing capabilities.

Example Scenario for Batch Processing: Imagine a financial firm that needs to analyze daily transactions to generate end-of-day reports. Throughout the day, they collect all transaction data and run a batch job overnight to process and analyze it.

Stream Processing

Stream Processing handles data in real-time as it arrives, processing each data point immediately without waiting for large amounts of data to accumulate.

Key Characteristics of Stream Processing

Let's look at an example of an online banking system that processes thousands of transactions per second. The bank uses a stream processing system to detect fraud, like unusual login attempts or suspicious transactions, in real-time.

- **Real-Time Processing**: Stream processing allows for instant analysis and response. In the bank's fraud detection system, each transaction is analyzed immediately. If a suspicious pattern (e.g., an unusual transaction or access from an unfamiliar location) is detected, the system can trigger an alert or block the transaction instantly.
- **Low Latency**: Stream processing has very minimal delay. As soon as a transaction is initiated, the fraud detection system processes it within milliseconds, allowing the bank to respond almost instantly.
- **Scalability**: Stream processing systems must handle varying data volumes, especially with high-velocity data like thousands of transactions per second. During peak times, the system scales to handle increased traffic, ensuring real-time fraud detection without delays.

- **Event-Driven**: Each event (like a login or transaction) triggers specific actions in the system. In the bank's fraud detection system, every transaction is treated as an event that triggers checks to validate its legitimacy. If the transaction exceeds a certain limit, it might trigger an additional verification step, like sending a one-time password (OTP) to the user.

More Use Cases for Stream Processing:

- **Real-Time Analytics**: Monitoring system performance or user activity in real-time to make quick decisions.
- **IoT Data**: Processing sensor data in real-time to monitor conditions and take immediate actions.

Technologies for Stream Processing:

- **Apache Kafka**: Used as a messaging system to handle high-speed data streams.
- **Apache Flink**: A framework for stateful processing and event-driven applications.
- **Apache Spark Streaming**: An extension of Spark for processing live data streams with a similar API to batch processing.

Example Scenario for Stream Processing: Consider a streaming service like Netflix, which processes user interactions in real-time. As users watch shows, the system instantly processes this data to update recommendations and optimize the viewing experience.

Data Storage

Let's look at the difference between a **data warehouse** and a **data lake** with a simple example.

Imagine you have a huge collection of books:

- A **data warehouse** is like a well-organized library where every book is carefully categorized and placed on the right shelf. Everything is neatly arranged, making it easy to find specific information or analyze certain topics.
- A **data lake** is more like a giant storage room where all kinds of books, documents, and papers are kept, but without any specific order. You can store anything here—even if you don't know how you'll use it yet. It's flexible and good for storing everything, but finding exactly what you need might take more time.

So, a **data warehouse** is structured and organized for quick access, while a **data lake** stores everything in its raw form, offering flexibility for future use.

Examples:

- **Data Lake**: Suitable for raw, unstructured, or semi-structured data (e.g., AWS S3, Azure Data Lake, HDFS).
- **Data Warehouse**: Best for structured data, optimized for analytical queries (e.g., Amazon Redshift, Google BigQuery, Apache Hive).

Data Lake: HDFS

HDFS (Hadoop Distributed File System) is a popular technology used in data lakes.

To understand how HDFS works, let's start with a simple example:

- Imagine you have a single computer with **1 TB** of storage. This works fine when your data needs are small. But as you need more storage, you may run out of space.

Vertical Scaling

One solution is to add more storage to this computer, known as **vertical scaling**. Vertical scaling means upgrading a single machine's resources—like adding more disk space or processing power.

However, vertical scaling has limits. Eventually, adding more resources becomes costly or impossible, so we need another approach.

Horizontal Scaling

In **horizontal scaling**, instead of adding resources to a single computer, you add more computers (or nodes). For instance, if you add **100 nodes**, each with **1 TB** of storage, you now have a **cluster** with **100 TB** of usable storage.

In HDFS, data is stored across multiple nodes in this cluster, allowing you to store and retrieve data efficiently, even when dealing with huge volumes.

Data Warehouse: Apache Hive

Apache Hive is a data warehouse built on top of HDFS. It's both a storage system and a processing engine that makes working with big data easier by using a SQL-like language called **HiveQL**.

Hive organizes data into databases and tables, allowing users to query and manage large datasets easily. Even people without extensive knowledge of big data systems can use Hive to access and analyze data stored in HDFS.

We'll explore Hive's architecture in more detail in future sections.

Data Processing

Now that we've learned about data, its sources, entry into the big data environment, and storage, the next question is: **How is this data processed so analysts and others can use it?**

Raw data is often not very useful in its original form. It might be incomplete, messy, or unstructured. For example, an e-commerce platform like Flipkart stores order details in a MySQL database, including information like customer name, contact info, email, products ordered, product price, and delivery address. But when storing this data in a big data system, not all details are needed, and additional information like product category and metadata might be required for better analysis. So, to store only the useful data, we need to **transform** it as it's being ingested. This is where **data processing** in big data becomes important.

The process of preparing this data is called **ETL (Extract, Transform, Load)**.

ETL Tools (Extract, Transform, Load)

ETL is a core process in data management, especially for e-commerce platforms. Let's go through each step with an example from an e-commerce setting.

Extract

The first step is to **extract** data from various sources, which could include:

- **Order databases**: Customer orders, items purchased, prices, quantities, and shipping details.

- **Product catalogs**: Product details, stock levels, and pricing.
- **Customer databases**: Customer profiles, browsing history, and purchase history.
- **Web logs**: Data on user interactions like page views, clicks, and search queries.

For example, if Flipkart wants to analyze buying patterns during a sale, it would pull data from orders, product catalogs, and web logs during that sale period.

Transform

The **transform** step involves cleaning, structuring, and enriching data to make it suitable for analysis. This can include:

- **Data cleaning**: Removing duplicates, handling missing values, and fixing errors.
- **Normalization**: Making data from different sources follow a uniform format and structure.
- **Aggregation**: Summing up data, like total sales by product or customer.
- **Enrichment**: Adding data, such as categorizing products or linking customer demographics.

For example, Flipkart might clean up product name inconsistencies, normalize prices to one currency, and enrich sales data with customer demographics.

Load

The final step is **loading** the transformed data into a **data warehouse** or database, where it can be analyzed.

For example, Flipkart would load the cleaned and aggregated data into its data warehouse, allowing analysts to generate reports on sales performance, customer behavior, and product trends.

Big Data Processing Engines

The ETL process can be handled by **processing engines** like **Apache Hive** and **Apache Spark**. These engines help sort through massive amounts of data, transforming it from raw information into something meaningful and useful. Processing engines provide an easy-to-use interface, often similar to SQL, to perform ETL tasks efficiently.

Data Integration: Data Pipelines

A common question is, **when is the data ingested**? That's where **data pipelines** come in. Data pipelines are automated workflows that move and transform data between systems.

A popular tool for managing data pipelines is **Apache Airflow**. Think of it as a smart scheduler for data tasks. It can handle complex workflows, such as pulling data from various sources, cleaning it, and loading it into a database, all automatically. With Airflow, you set the conditions, timing, and frequency for these tasks—like setting up a to-do list that runs daily or at certain times.

Data Consumers

Once data is ingested and stored, it's ready for use by different **data consumers**:

- **Business Intelligence (BI) Tools**: For querying, visualizing, and analyzing data (e.g., Tableau).
- **Data Science and Machine Learning**: Platforms for building models and advanced analytics (e.g., Jupyter Notebooks, Databricks).
- **Dashboards**: Real-time visual displays of key metrics and trends.
- **Reports**: Summaries and detailed views of data insights.

These consumers use the processed data to gain insights, track performance, and make data-driven decisions.

==========

Mastering GitHub and Git: A Comprehensive Tutorial for All Skill Levels

Download notes from the video description

https://youtu.be/AkqOtUig5p4

Like | Comment | Subscribe

==========

Creating a Spring Boot Project with Spring Initializer

Visit

https://youtu.be/hTtO_sOW-dI

Like | Comment | Subscribe

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

==========

Hadoop Ecosystem Fundamentals Of Distributed Systems - Oct 7

- **Why distributed system is required**
- **Installation**
- **Internals of hdfs**
- **Play with hdfs**

Hadoop Installation Guide
https://docs.google.com/document/d/1QXi1EMv42RN3yhATLIC5X0t_6Zi9AUcqdoxFe8j7jC8/edit?tab=t.0

Data Engineering Extra resource
https://drive.google.com/drive/u/0/folders/1eOrjAkr5uyegYJ0x9qcXNZZ8VYeV2ubU

Why distributed system is required
A distributed system is a model where components on different networked computers communicate and coordinate their actions by exchanging messages

The issue with Traditional systems

Traditional systems face several challenges when dealing with large data:

1. **Limited Storage**: Traditional systems often rely on a single server or a small set of servers, which limits the amount of data they can store. As data grows into **terabytes or petabytes**, a single server's storage capacity becomes insufficient.
2. **Performance Bottlenecks**: As data size increases, reading, writing, and processing large amounts of data on a single server slows down. Traditional systems struggle to process massive datasets efficiently, leading to **slower query responses** and increased latency.
3. **Single Point of Failure**: If the main server in a traditional system fails, the entire system can go down. This lack of **fault tolerance** means the system is less reliable when dealing with large data volumes.
4. **Scalability Issues**: Traditional systems typically scale **vertically** (adding more power to a single machine). However, vertical scaling has limits due to hardware constraints. As the data grows, scaling up a single server becomes expensive and impractical.
5. **High Costs**: Handling large data on traditional systems often requires **high-performance hardware**, which is expensive. These systems are not optimized to use clusters of cheaper machines like distributed systems do.
6. **Inability to Process Unstructured Data**: Traditional systems, such as relational databases, are designed to handle **structured data** (like rows and columns in a database). They struggle with **unstructured data** like logs, images, and social media data, which make up a large portion of big data today.

7. **Limited Parallelism**: Traditional systems usually process tasks in a **sequential manner**, which is slow when dealing with large datasets. They can't easily split tasks across multiple machines to process data in parallel, as distributed systems do.

These limitations make traditional systems unsuitable for modern applications that deal with **big data**. Distributed systems like Hadoop solve these issues by spreading the data and processing across many machines, allowing for better performance, scalability, and reliability.

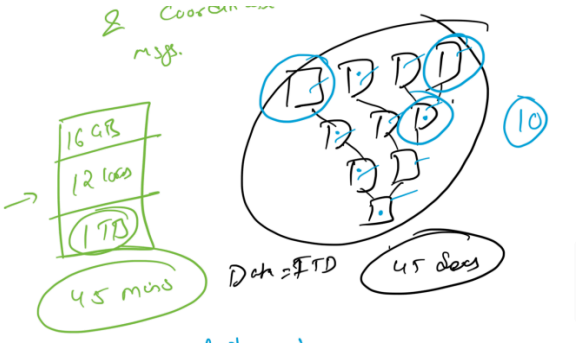
Why Bigdata is required

In software engineering, distributed systems like Big Data frameworks (e.g., Hadoop) are important because they help solve problems related to scalability, performance, and data processing. Here's why:

1. Handling Large Data: As data grows, traditional systems can't store or process it efficiently. Hadoop allows data to be spread across many machines, making it possible to handle terabytes or even petabytes of data.
2. Scalability: Distributed systems can scale horizontally by adding more machines (nodes) to the network. This way, when more data or traffic comes in, you can expand the system without changing the core infrastructure.
3. Fault Tolerance: Hadoop replicates data across multiple nodes, so if one machine fails, the system can still work. This ensures the system stays available and reliable even if some parts fail.
4. Parallel Processing: Hadoop uses frameworks like MapReduce, which break down tasks into smaller chunks and process them in parallel across different nodes. This makes data processing much faster compared to traditional systems.
5. Cost Efficiency: Instead of investing in expensive supercomputers, distributed systems can run on clusters of low-cost machines. This makes it cost-effective for processing large datasets.

Overall, Hadoop and similar distributed systems are essential for managing and analyzing massive datasets efficiently in software engineering.

As shown in the images below, the first image represents a traditional system, while the second image shows a distributed system. Our goal is to process 1TB of data. In the traditional system, processing takes 45 minutes due to sequential processing. However, with the distributed system, the same data is processed in 45 seconds because tasks are done in parallel using multiple machines.

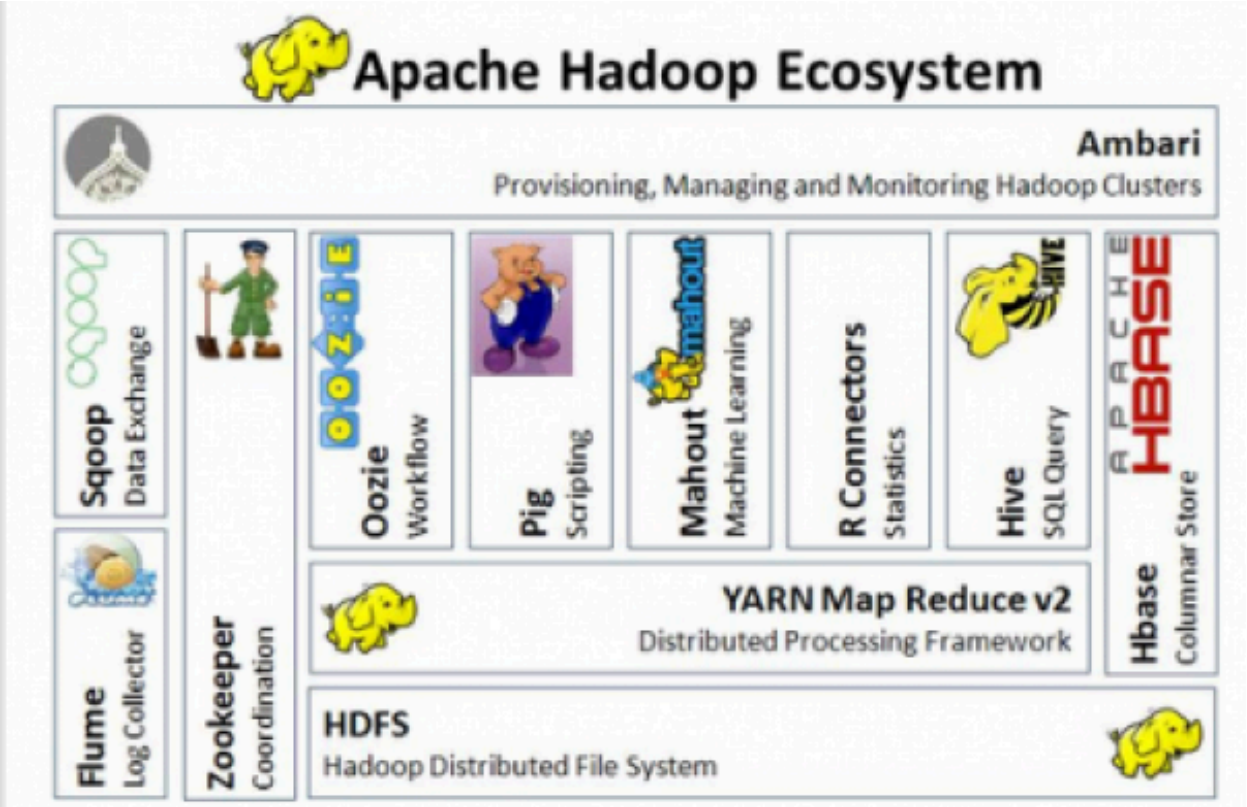


Issues with distributed systems:

- System Failure: Since multiple systems are involved, some may fail at times, meaning we can't always utilize 100% of the system's capacity.
- Limited Bandwidth: Transferring data between servers over the network faces bandwidth limitations, which can slow down communication.
- High Programming Complexity: Managing data processing across multiple machines and ensuring smooth operation requires more complex coding and coordination.

The solution: An introduction to Apache Hadoop for big data

Hadoop is a **framework** that enables the **distributed processing** of large datasets across **commodity computers**(clusters of low-cost computers or traditional computers) using **simple programming** models.



https://opensource.com/life/14/8/intro-apache-hadoop-big-data

Hadoop Systems Overview: Hadoop helps manage and process large data across many computers.

- **HDFS**: Stores large data across multiple computers.
- **YARN**: Manages how data is processed on different machines.
- **HBase**: A NoSQL database for fast data read/write.
- **Hive**: A tool for querying and managing data like a data warehouse.
- **R Connectors**: Lets Hadoop work with R for statistical analysis.
- **Mahout**: Provides machine learning tools.
- **Pig**: A tool for analyzing big data using scripts.
- **Oozie**: Manages and schedules Hadoop jobs.
- **Ambari**: A user-friendly interface to manage Hadoop.

- **Sqoop**: Transfers. Import data between Hadoop and databases.
- **Flume**: Moves/Import real-time data into Hadoop.
- **Zookeeper**: Keeps distributed systems in sync and working together.

Characteristics of Hadoop:

- **Scalable**: Supports both vertical and horizontal scaling by adding more machines.
- **Highly Reliable**: Ensures reliability by storing copies of data through replication.
- **Economical**: Uses cheaper hardware due to horizontal scaling.
- **Flexible**: It can store any type of data. For unstructured data, use HBase and HDFS; for structured data, use Hive SQL; and for semi-structured data, use a combination of Pig and Hive

Hadoop Core Services:

- HDFS: Acts like a distributed storage system where data is stored for processing.
- Yarn: Manages system resources like CPU and memory for running applications.
- MapReduce: Handles the processing engine that performs data computation.

HDFS (Hadoop Distributed File System): It provides access to data across Hadoop clusters. It's open-source with no upfront management costs. HDFS is very fast, capable of processing up to 1 TB per second, and automatically creates copies of data for reliability.

Popular Big Data Tools:

- Cloudera Distribution
- Apache Hadoop
- AWS EMR
- GCP Dataflow
- Azure HDInsight

HDFS Architecture:

It works on a master-slave model. The NameNode acts as the master, while the DataNodes are the slaves. Each DataNode communicates with the NameNode over TCP, sending a message called a heartbeat every 3 seconds to signal that it's active. When a client requests to store data, HDFS checks for available space, splits the file into chunks, and stores the first chunk on a DataNode. It then creates two additional replicas on other DataNodes, storing 3 copies of each chunk in total (as the default replication factor is 3). These replicas are distributed across different racks for fault tolerance.

Data Modeling - Oct 9

- **Data Warehousing**
- **Hive’s Architecture**
- **Key Components**
- **Flow of Query Execution**
- **Commonly used HQL commands**

Introduction to OLAP Systems

Let’s start by understanding OLAP systems and their importance. Imagine you run a business with a huge amount of data coming in daily—sales transactions, customer interactions, inventory changes, and more. To make smart decisions, like knowing which products to stock or which customers to target, you need to analyze all this data.

This is where OLAP, or Online Analytical Processing, comes in. OLAP systems are designed to help you explore your data deeply, allowing you to run complex queries, generate reports, and extract insights necessary for informed decision-making. Think of OLAP as the brain behind your business intelligence, transforming raw data into actionable insights**.**

OLAP vs. OLTP: What’s the Difference?

You might be curious about how OLAP systems differ from OLTP, or Online Transaction Processing systems. OLTP acts as the workhorse for your business's day-to-day operations. It’s like the cash register in a store, efficiently processing each transaction in real time.

In contrast, OLAP is not about managing individual transactions as they happen; it's focused on analyzing the broader trends. OLAP systems collect all those transactions, store them, and allow you to analyze them to identify trends, patterns, and opportunities. So, while OLTP prioritizes speed and accuracy in processing transactions, OLAP focuses on providing deep insights over time.

Diving Into OLAP Data Modeling

Now that we’ve covered the basics, let’s discuss how to structure the data in an OLAP system. This is where data modeling comes into play, starting with two key components: **Fact Tables** and **Dimension Tables**.

1. Fact Tables: The Heart of Your Data Model

Imagine you’re managing a chain of retail stores and want to analyze your sales data. The **Fact Table** is where the action happens—it’s where all your core data is stored. Think of it as the table that holds all the important details about your business metrics, such as sales, revenue, or quantities sold.

A **fact table** primarily stores quantitative data—often referred to as measures—that can be analyzed. These measures typically represent business metrics like sales, revenue, or quantities, which are the focus of queries and analysis. The fact table contains the core data you want to analyze, and its meaning is represented by a dimension table.

Example of a Fact Table

Here's an example to illustrate this concept: Let's say you have a table called **SalesFact** in your data warehouse. It might look something like this:

Date	ProductID	StoreID	SalesAmount	QuantitySold
2024-08-01	1001	1	500	10
2024-08-01	1002	2	300	5
2024-08-02	1003	1	450	8

In this example, **SalesAmount** and **QuantitySold** are measures—the numbers you want to analyze. The other columns, like **ProductID** and **StoreID**, are foreign keys that connect this fact table to other tables (which we'll discuss shortly).

Key Takeaways About Fact Tables:

- **Foreign Keys**: These links connect your fact table to other tables, providing context to your data.
- **Measurable Data**: Fact tables store the key numbers—like sales, revenue, or quantities—that you want to analyze.
- **Lots of Rows**: Fact tables usually contain many rows because they capture every transaction or event in detail.
- **Grain**: The grain of a fact table is the level of detail it captures. In our example, each row represents a single sale.

2. Dimension Tables: Adding Context to Your Data

Now, let's discuss **Dimension Tables**. If the fact table is the heart of your data model, then dimension tables are like the veins and arteries—they provide the context needed to make sense of the raw data in your fact table. Dimension tables contain descriptive attributes (often called dimensions) that relate to the fact table and help categorize, filter, and aggregate the data in meaningful ways.

Example Time

Let's revisit our **SalesFact** table. To fully understand the sales data, you would want to know more about the products, stores, and dates involved. This is where dimension tables come into play.

You might have a **ProductDimension** table that looks like this:

ProductID	ProductName	Category	Brand
1001	Laptop	Electronics	TechCorp
1002	Smartphone	Electronics	PhoneCo
1003	Desk Chair	Furniture	ComfortPlus

And a **StoreDimension** table like this:

StoreID	StoreName	Location	Manager
1	Downtown	New York	John Smith
2	Mall Branch	Los Angeles	Sarah Brown

And a **DateDimension** table like this:

Date	DayOfWeek	Month	Quarter	Year
2024-08-01	Thursday	August	Q3	2024
2024-08-02	Friday	August	Q3	2024

These dimension tables help you drill down into the specifics of your data. For instance, you can see that ProductID 1001 in the fact table corresponds to a Laptop from TechCorp.

Why Dimension Tables Matter:

- **Descriptive Data**: They store information such as product names, categories, locations, and dates.
- **Primary Keys**: Each record in a dimension table has a unique identifier that is referenced in the fact table.
- **Smaller Size**: Dimension tables usually contain fewer rows than fact tables since they summarize data.
- **Enabling Analysis**: They allow you to filter, group, and label data in the fact table for more meaningful analysis.

Putting It All Together: Fact and Dimension Tables in Action

So, how do fact and dimension tables work together? For example, if you want to find out, “What was the total sales amount for each product category in July 2024?” you would join the **SalesFact** table with the **ProductDimension** and **DateDimension** tables. You’d use the **DateDimension** to filter records to July 2024, and the **ProductDimension** to group the results by product category.

Alternatively, if you’re curious about, “Which store had the highest sales on August 1, 2024?” you’d join the **SalesFact** table with the **StoreDimension** and **DateDimension** tables to find your answer.

In both cases, the fact table provides the numbers, while the dimension tables offer the context, enabling you to analyze the data and gain the insights you need.

Schema Design in OLAP: Star, Snowflake, and Galaxy

When building an OLAP system, organizing your data is crucial. This is where schema design comes into play. Let’s explore a few common types of schemas: **Star Schema**, **Snowflake Schema**, and **Galaxy Schema**.

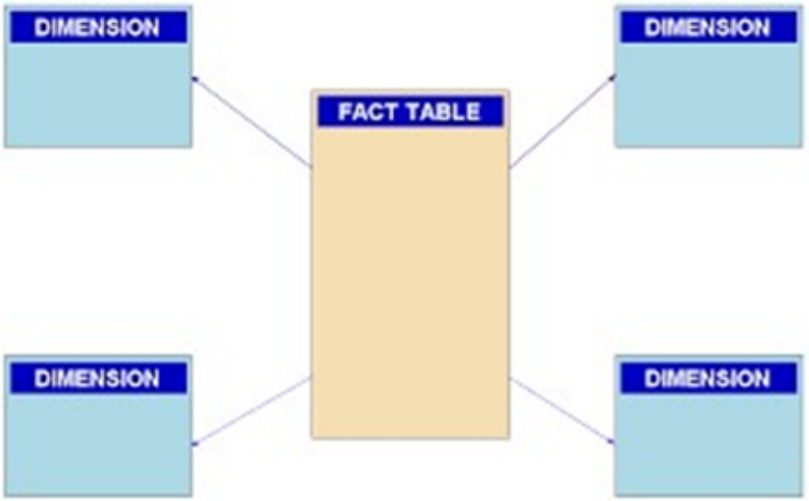
Star Schema: The Simple and Intuitive Choice

Visualize a star shape. In the center, you have your fact table, and surrounding it, like the points of a star, are your dimension tables. This is the **Star Schema**, one of the simplest and most intuitive ways to structure your data.

Here's how it looks:

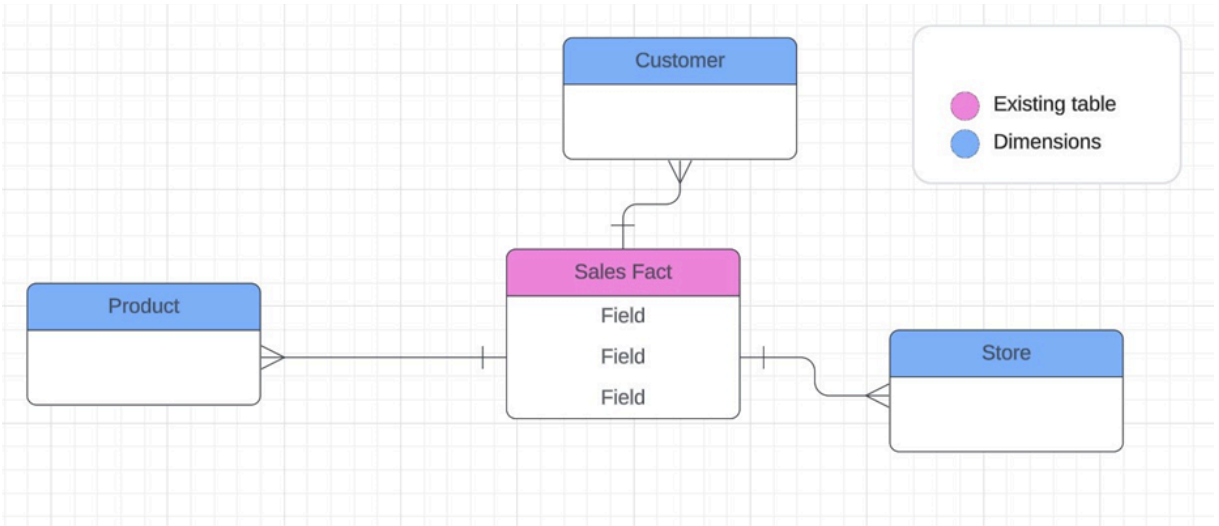
- **Fact Table**: This central table contains all your measurable data—such as sales amounts and quantities sold.
- **Dimension Tables**: These tables surround the fact table, providing context with details like product names, customer information, and store locations.

In this schema, the fact table is at the center, and each dimension table is directly connected to it, resembling the points of a star.



Example:

Consider a data warehouse for a retail store. The **SalesFact** fact table captures all the sales data, while the **ProductDimension**, **CustomerDimension**, and **StoreDimension** dimension tables offer additional details about the products, customers, and store locations.



- **SalesFact**: Contains sales data (e.g., SalesAmount, QuantitySold).
- **ProductDimension**: Includes details about products (e.g., ProductID, ProductName).
- **CustomerDimension**: Holds information about customers (e.g., CustomerID, CustomerName).
- **StoreDimension**: Lists details about stores (e.g., StoreID, StoreLocation).

Why Use a Star Schema?

- **Easy to Understand**: Its simple structure makes it easy to design and query.
- **Fast Performance**: Queries run faster due to fewer joins (connections between tables).
- **Some Redundancy**: Data may be duplicated in dimension tables, but this trade-off enhances performance.

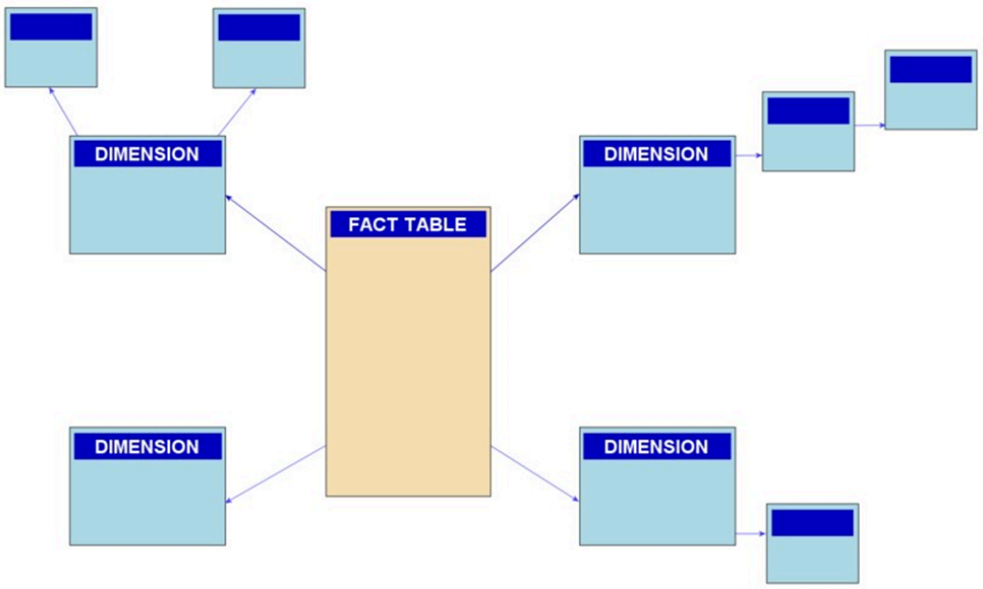
Snowflake Schema: For Enhanced Normalization

If you're considering ways to reduce redundancy and conserve storage space, the **Snowflake Schema** is an excellent option. Similar to the Star Schema, it features a twist: the dimension tables are further divided into related sub-tables, creating a structure that resembles a snowflake.

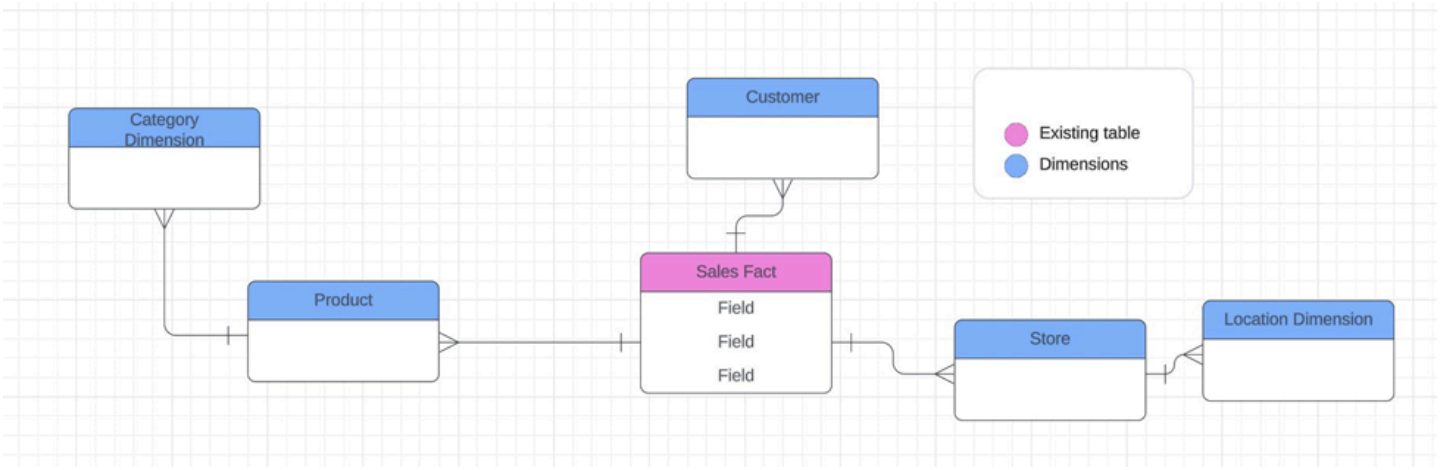
How It Works:

- **Fact Table**: Remains central, containing your primary data.
- **Dimension Tables**: These tables are normalized, meaning they are segmented into multiple related tables to minimize redundancy.

In a visual representation, the snowflake schema maintains the central fact table but has dimension tables that branch out into related sub-tables, resembling the intricate patterns of a snowflake.



Example:
Continuing with the retail store example:



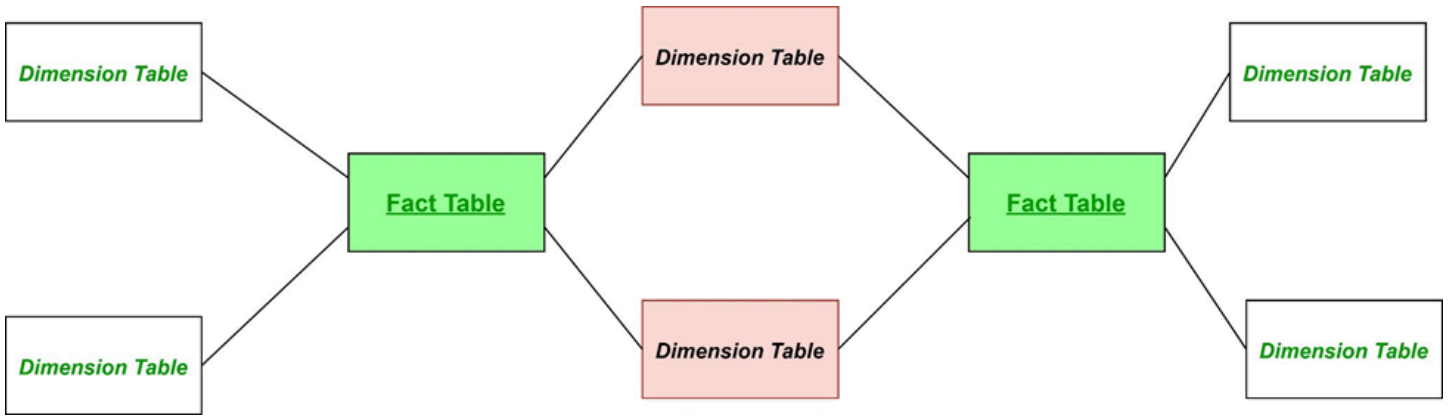
Snowflake Schema: For Enhanced Normalization

If you're considering ways to reduce redundancy and conserve storage space, the **Snowflake Schema** is an excellent option. Similar to the Star Schema, it features a twist: the dimension tables are further divided into related sub-tables, creating a structure that resembles a snowflake.

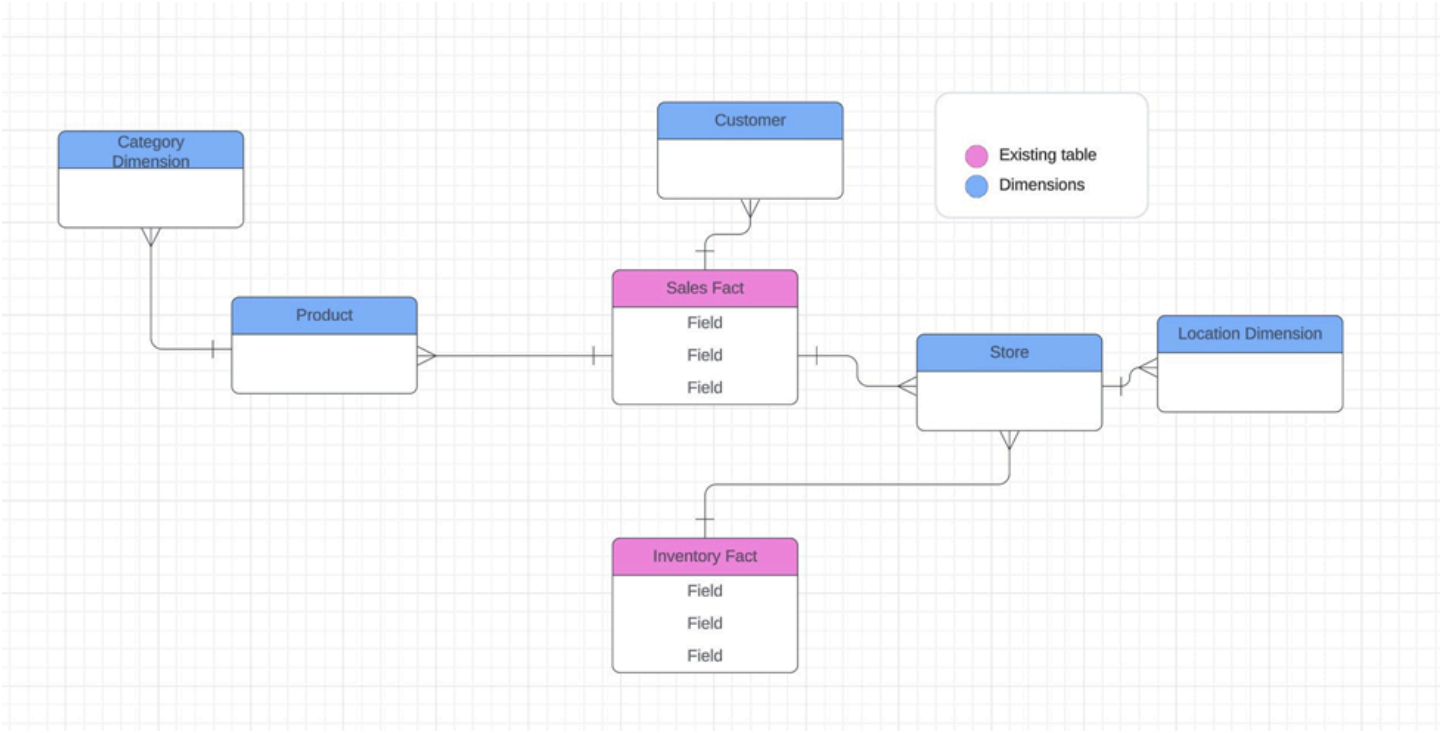
How It Works:

- **Fact Table**: Remains central, containing your primary data.
- **Dimension Tables**: These tables are normalized, meaning they are segmented into multiple related tables to minimize redundancy.

In a visual representation, the snowflake schema maintains the central fact table but has dimension tables that branch out into related sub-tables, resembling the intricate patterns of a snowflake.



Example: Imagine you are managing both sales and inventory for the retail store:



- **SalesFact**: Stores sales data (e.g., SalesAmount, QuantitySold).
- **InventoryFact**: Stores inventory data (e.g., StockLevel, ReorderQuantity).
- **ProductDimension**: Shared between both SalesFact and InventoryFact.
- **StoreDimension**: Also shared between fact tables.

When to Use a Galaxy Schema:

- **Multiple Fact Tables**: Ideal for tracking multiple processes or events, such as sales and inventory.
- **Shared Dimensions**: Ensures consistency across fact tables.
- **Complex Queries**: Supports advanced queries across different types of data.

Slowly Changing Dimensions (SCDs): Handling Changes Over Time

Real-world data changes over time—customers move, products are rebranded, and so on. **Slowly Changing Dimensions (SCDs)** handle these changes, allowing us to capture evolving details in the data warehouse.

SCD Approaches:

- **SCD Type 0 (Do Nothing)**: Leaves data as it was initially inserted.
    - *Example*: A product initially labeled as "Electronics" stays the same even if later reclassified.
- **SCD Type 1 (Overwrite Old Value)**: Updates the old value with the new one, losing history.
    - *Example*: If a customer changes their last name, the old name is overwritten with the new one.
    - *Benefits*: Simple with minimal storage.
    - *Downside*: No historical data.
- **SCD Type 2 (Full History)**: Creates a new record for each change, allowing complete tracking over time.
    - *Example*: When a customer moves, a new row is added with the updated address while the old address remains.
    - *Benefits*: Retains all historical data.
    - *Downside*: More complex and requires additional storage.
- **Surrogate Keys**: Often used with SCD Type 2 to uniquely identify records, especially when multiple entries share the same natural key.
- **SCD Type 3 (Add New Column)**: Adds a new column to capture only the previous value, allowing limited historical tracking.
    - *Example*: A product rebrand would store the old brand in a separate column.
    - *Benefits*: Simple history of recent changes.
    - *Downside*: Limited to tracking only one previous state.
- **SCD Type 4 (Historical Table)**: Uses two tables—one for current data and another for historical records.
    - *Example*: Current table holds the latest customer address, while a separate history table keeps previous addresses.
    - *Benefits*: Clean separation of current and historical data.
    - *Downside*: Adds complexity by requiring two tables.

SCD Type 6: The Hybrid Approach

SCD Type 6 combines features from Types 1, 2, and 3, capturing every change, keeping a full history, and tracking the most recent change within a single table. This approach, often called "Unpredictable Changes with Single-Version Overlay" (a term by Ralph Kimball in *The Data Warehouse Toolkit*), provides comprehensive tracking by maintaining both historical and current states in one place.

Example: When a product is rebranded, SCD Type 6:

- Adds a new record (as in SCD Type 2),
- Overwrites the current value (as in SCD Type 1), and
- Stores the previous value in a separate column (as in SCD Type 3).

Why Use SCD Type 6?

- **Comprehensive Tracking**: Captures all changes, tracks the latest update, and keeps a full history.
- **Complexity**: It's the most complex SCD type but ideal for scenarios where detailed tracking is crucial.

Designing an OLAP Data Model for Uber Rides

Now, let's use these principles to design an OLAP (Online Analytical Processing) data model for Uber rides, focused on deep analysis of ride patterns, driver performance, customer behavior, and more.

Key Entities for Uber Rides:

1. **Rides**: Each trip taken by a passenger.
2. **Drivers**: People who drive the passengers.
3. **Customers (Passengers)**: People who book and take rides.
4. **Time**: When rides occur.
5. **Locations**: Pickup and drop-off points.
6. **Vehicles**: Cars used for rides.

Identifying the Key Facts: In this OLAP model, facts represent the measurable data you want to analyze. For Uber rides, the main fact table might be **Ride Details** with these attributes:

- **RideID** (Primary Key)
- **DriverID** (Foreign Key to Driver dimension)
- **CustomerID** (Foreign Key to Customer dimension)
- **VehicleID** (Foreign Key to Vehicle dimension)
- **PickupLocationID** (Foreign Key to Location dimension)
- **DropoffLocationID** (Foreign Key to Location dimension)
- **TimeID** (Foreign Key to Time dimension)
- **Distance**: Distance covered during the ride.
- **Fare**: Total ride cost.
- **Duration**: Ride time.

Defining the Key Dimensions: Dimensions provide context to facts, offering various ways to view and analyze the data.

1. **Time Dimension**:
    - **Attributes**: TimeID (Primary Key), Date, Day, Month, Quarter, Year, Hour, Minute.
    - **Purpose**: Analyzes ride trends over different time periods, such as peak hours or specific months.
2. **Driver Dimension**:
    - **Attributes**: DriverID (Primary Key), Name, Age, Gender, License Number, Rating, Experience (in years).
    - **Purpose**: Allows analysis based on driver characteristics, like performance by experience level.
3. **Customer Dimension**:
    - **Attributes**: CustomerID (Primary Key), Name, Age, Gender, Account Creation Date, Total Rides Taken.
    - **Purpose**: Enables ride segmentation by customer demographics and behavior.
4. **Location Dimension**:
    - **Attributes**: LocationID (Primary Key), Address, City, State, Zip Code, Latitude, Longitude.
    - **Purpose**: Provides geographical context, allowing analysis of popular pickup and drop-off points.

5. **Vehicle Dimension**:
    - **Attributes**: VehicleID (Primary Key), VehicleType, Make, Model, Year, License Plate Number, Capacity.
    - **Purpose**: Enables analysis by vehicle characteristics, such as ride data by car type or model.

This data model allows Uber to analyze patterns and trends, supporting a wide range of queries, from popular ride times to customer demographics and driver performance, all of which can help Uber optimize its services.

Designing the Schema

In OLAP systems, schema design usually follows a **Star Schema** or **Snowflake Schema** approach. Here's how to structure the Uber rides data model using a **Star Schema**.

Star Schema Design:

- **Central Fact Table**:
    - **Ride Fact Table**: Contains foreign keys linking to Time, Driver, Customer, Location, and Vehicle dimensions.
    - Stores metrics like **Distance**, **Fare**, and **Duration**.
- **Surrounding Dimension Tables**:
    - **Time Dimension Table**: Linked by **TimeID**.
    - **Driver Dimension Table**: Linked by **DriverID**.
    - **Customer Dimension Table**: Linked by **CustomerID**.
    - **Location Dimension Table**: Linked by **PickupLocationID** and **DropoffLocationID**.
    - **Vehicle Dimension Table**: Linked by **VehicleID**.

This design makes it efficient to analyze Uber ride data by centralizing the quantitative data in the fact table, with related details organized in dimension tables for quick access.

Schema Example

Time Dimension

```
+---------+
| TimeID  |
| Date    |
| Hour    |
| Month   |
+---------+
```

Driver Dimension

```
+---------+
| DriverID |
| Name    |
| Rating  |
+---------+
```

Customer Dimension

```
+---------+
| CustID  |
| Name    |
| Gender  |
+---------+
```

Location Dimension

```
+---------+
| LocID   |
| Address |
| City    |
+---------+
```

Vehicle Dimension

```
+---------+
| VehicleID |
| Make      |
| Model     |
+---------+
```

Ride Fact Table

```
+--------------+
| RideID       |
| DriverID     |
| CustID       |
| VehicleID    |
| PickupLocID  |
| DropoffLocID |
| TimeID       |
| Distance     |
| Fare         |
| Duration     |
+--------------+
```

Use Case Scenarios

With this schema, you can perform a variety of analyses:

1. **Ride Trend Analysis**: Track the number of rides over time to identify peak hours, days, or months.
2. **Driver Performance**: Evaluate drivers based on metrics like average fare, distance covered, and customer ratings.
3. **Customer Behavior**: Segment customers by ride frequency, preferred times, or common routes.
4. **Location-Based Insights**: Identify popular pickup and drop-off spots to optimize driver availability.

Advanced Schema Options: Snowflake and Galaxy

Depending on your data complexity and analysis needs, you might consider using:

- **Snowflake Schema**: This schema normalizes dimensions further. For example, you could split the Location dimension into separate tables for City, State, and Country.
- **Galaxy Schema**: Use this schema when multiple fact tables are needed. For instance, a separate fact table for Cancelled Rides could share dimensions with the main Ride Fact Table.

Hive Architecture - Oct 11

- **Data Warehousing**
- **Hive's Architecture**
- **Key Components**
- **Flow of Query Execution**
- **Commonly used HQL commands**

Data Warehousing with Apache Hive

What is a Data Warehouse?

A data warehouse is like a big library where data is stored in an organized way. Unlike regular databases that are optimized for everyday transactions (like taking money from an ATM), a data warehouse is built for analyzing large amounts of data and answering questions about trends and patterns over time.

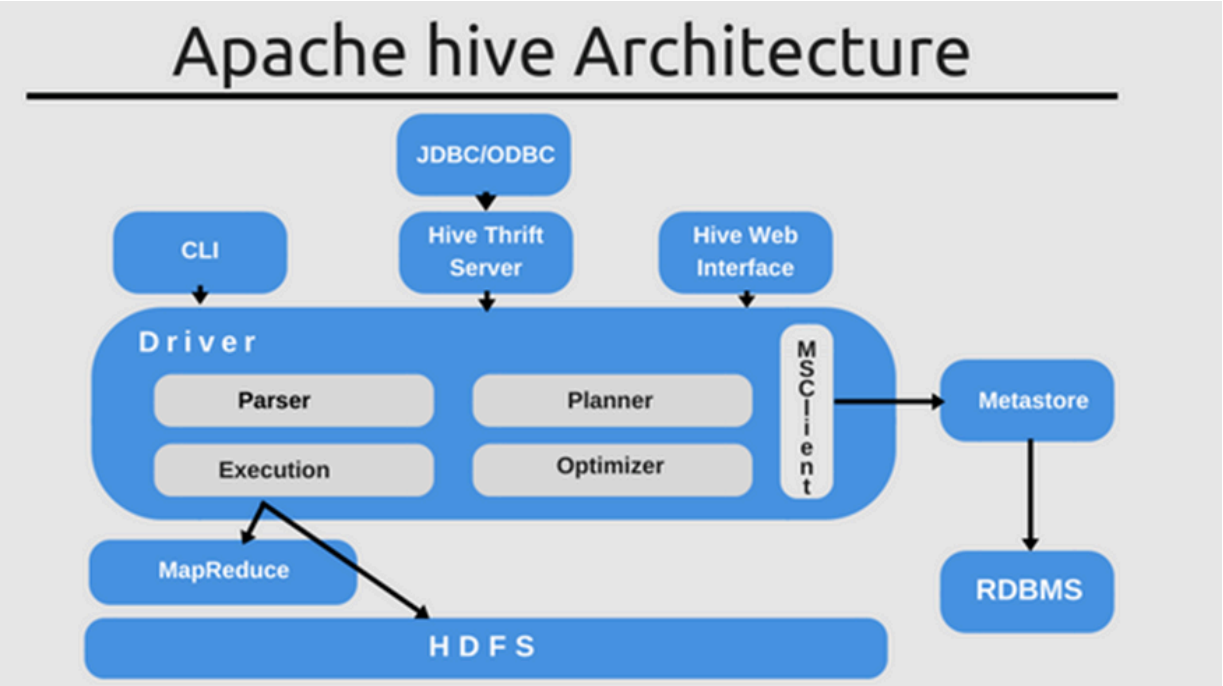
For example, imagine a retail company. Every time a customer buys something, a record of that sale is created. Over years, these records add up, creating a huge amount of data. The company can store all this data in a data warehouse, which lets analysts run queries to find answers to questions like, "What was the best-selling product last holiday season?" or "Which store had the most sales growth?"

Why Use Hive for Data Warehousing?

With big data, traditional databases can have trouble handling the scale. That's where Apache Hive comes in. Hive is built on top of Hadoop, a system that can handle very large amounts of data by spreading the data across many computers (called nodes). Hive allows you to use SQL-like queries to work with this big data, making it easier to interact with and analyze.

Simply put, Apache Hive connects the world of SQL (a language for managing databases) with the world of big data. It lets you query large datasets stored in Hadoop's distributed file system (HDFS) without needing to write complex code, making big data analysis much more accessible.

Hive's Architecture



Apache Hive is a data warehousing and SQL-like query language system built on top of Hadoop. It allows users to manage and query large datasets stored in distributed storage. Here's a simple overview of the Hive architecture:

Key Components of Hive Architecture

1. **Storage**:
    - Hive stores data in HDFS, which allows for distributed storage and processing.
    - Supports various file formats like Text, ORC, Parquet, and Avro.
2. **Hive Metastore**:

    The Metastore is a key part of Hive and A central repository that stores all the metadata (like table names, column names, data types, and partition information) about Hive tables. It doesn't store the actual data; that data is stored in HDFS. The Metastore only keeps information about tables, columns, partitions, and their data type. By default, Hive uses an in-memory database called Derby for the Metastore, but it can also use external databases like MySQL, Oracle, PostgreSQL, or SQLite
3. **Hive Driver**:
    - **Parser**: It checks the query for any mistakes in the syntax (grammar) and ensures the query makes sense.
    - **Planner**: This part creates a plan for how to run the query, breaking it down into steps.
    - **Optimizer**: It improves the query by making changes to run it more efficiently and quickly.
    - **Execution Engine**: Finally, it turns the query into tasks that can be run on Hadoop, usually creating a file (JAR) that runs the job.
4. **Hive Compiler**:
    - Takes the parsed HiveQL and generates an execution plan.
    - It optimizes the query and translates it into a series of MapReduce, Tez, or Spark jobs for execution.
5. **Execution Engine**:
    - Executes the tasks generated by the compiler.
    - Uses MapReduce, Tez, or Spark to process the data in HDFS (Hadoop Distributed File System).
    - Manages job execution, monitoring, and error handling.
    - SQL/HQL queries are converted into MapReduce jobs, which are then compiled into a JAR file (an executable file) for execution.
6. **Hive CLI / Hive Web Interface**:
    - The Hive Command Line Interface (CLI) allows users to interact with Hive and run queries but it is deprecated
    - Now We have to use beeline to get data
    - Hive also provides a web interface for easier access and management.
7. **User Interfaces**:
    - Users can interact with Hive through different interfaces:
        - **Hive CLI**: Command line tool to run queries.
        - **Hive Web Interface**: Web-based GUI for running queries and managing data.
        - **Thrift API**: Allows remote access to Hive from various programming languages.

Understanding Hive's Architecture

When you query data in Hive, there's a lot happening under the hood. Let's break down the main components of Hive's architecture:

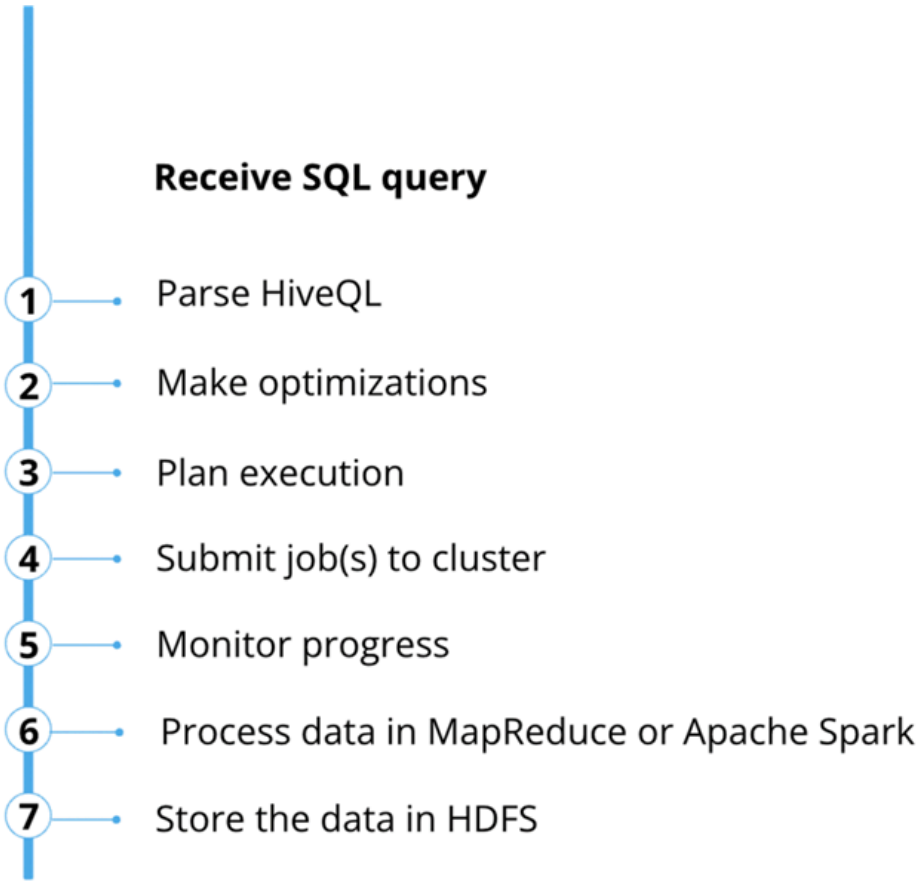
1. HiveCLI and Beeline: Getting Started with Hive

- **HiveCLI**: This is the classic command-line tool for Hive. It lets you run HiveQL (Hive Query Language) queries directly. It's simple and useful for quick tests or basic tasks, but it lacks some advanced features needed for production.
- **Beeline**: This is the updated tool for connecting to Hive. Unlike HiveCLI, it connects to Hive through JDBC, making it more flexible and secure. Beeline is often used in production because it supports remote access and enhanced security, making it ideal if you're connecting from another location or need a more secure setup.

2. The Driver: Managing Query Processing

When you run a query through HiveCLI or Beeline, it goes to the **Driver**, which manages query processing in the following steps:

- **Parsing**: The Driver first checks if the HiveQL query is written correctly and creates an Abstract Syntax Tree (AST), a structured form of your query that Hive can understand.



- **Logical Plan**: The Driver then builds a logical plan, like a high-level recipe that lists what actions need to be done to execute your query, without getting into the technical details yet.

- **Physical Plan**: Next, the logical plan is converted into a physical plan, breaking down the actions into specific jobs—like MapReduce, Tez, or Spark jobs—that will process the data step by step.
- **Query Optimization**: The Driver optimizes the physical plan to make the query run faster, for example by filtering out unnecessary data or reordering joins.
- **Execution**: Finally, Hive executes the optimized plan by running the necessary jobs (MapReduce, Spark, or Tez). Once processing is finished, the results are sent back to you.

3. HDFS: The Storage for Your Data

- **HDFS (Hadoop Distributed File System)** is where Hive stores the actual data for your tables. It's like a distributed hard drive, built to handle large amounts of data across multiple machines. HDFS also provides fault tolerance and fast access, ensuring your data is safe and easily reachable.

4. The Metastore: The Metadata Repository

- **Metadata Repository**: The **Hive Metastore** stores metadata (data about data) for Hive tables, databases, partitions, and more. This metadata is essential because it tells Hive about the structure and layout of your data.
- **Database Storage**: The Metastore uses a relational database (such as MySQL, PostgreSQL, or Derby) to keep the metadata separate and consistent, allowing it to be managed independently from the actual data stored in HDFS.

Flow of Query Execution in Hive

When you run a query in Hive, it goes through several steps before you get the results. Here's a simple breakdown of the process:

1. **Submit the Query**
    - You write and submit a HiveQL query using tools like HiveCLI or Beeline. This is where you start.
2. **Send Query to Driver**
    - The query is sent to the Hive Driver, which manages the execution process.
3. **Parse the Query**
    - **Action**: The Driver parses the query.
    - **Purpose**: It checks for any syntax errors.
    - **Result**: The query is converted into an Abstract Syntax Tree (AST), which is a structured format Hive can understand.
4. **Semantic Analysis**
    - **Action**: The AST goes through semantic analysis.
    - **Purpose**: This step checks if the tables, columns, and other items in the query exist and are used correctly.
    - **Result**: Confirms that the query makes sense based on the schema in the Hive Metastore.

5. **Create Logical Plan**
    - **Action**: The Driver creates a logical plan based on the analysis.
    - **Purpose**: It outlines the steps to execute the query, like a blueprint, without deciding how to perform each step.
6. **Create Physical Plan**
    - **Action**: The logical plan is transformed into a physical plan.
    - **Purpose**: This plan specifies the exact jobs (MapReduce, Tez, or Spark) that will run to process the data.
    - **Result**: A detailed list of actions like data scans, joins, and aggregations.
7. **Optimize the Query**
    - **Action**: The physical plan is optimized to improve efficiency.
    - **Purpose**: Optimizations might include:
        - **Predicate Pushdown**: Applying filters earlier to reduce data.
        - **Join Reordering**: Changing join order to minimize data shuffling.
        - **Column Pruning**: Removing unnecessary columns from processing.
    - **Result**: An optimized plan that executes faster.
8. **Execute the Plan**
    - **Action**: The optimized plan is executed.
    - **Purpose**: The specified jobs (MapReduce, Spark, or Tez) process the data.
    - **Result**: Data is processed, and the results are generated.
9. **Fetch the Results**
    - **Action**: After execution, results are fetched.
    - **Purpose**: Gather the query output to send back to you.
    - **Result**: You receive the final data.
10. **Return Results to User**

- **Action**: The results are sent to you through HiveCLI or Beeline.
- **Purpose**: Lets you view or use the data as needed.

Summary Flow

1. **Submit Query** → 2. **Driver** → 3. **Parse Query** → 4. **Semantic Analysis** → 5. **Logical Plan** → 6. **Physical Plan** → 7. **Optimize Query** → 8. **Execute Plan** → 9. **Fetch Results** → 10. **Return Results**

This process helps Hive handle large queries efficiently, turning your request into a series of steps to retrieve and process data from a distributed system.

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

Detailed Explanation of application.properties

Visit

https://youtu.be/_MmSSunw_gM

Like | Comment | Subscribe

Common HQL Commands

Hive offers several common commands to manage databases, tables, and other objects. Here's an overview of some key Hive Query Language (HQL) commands, especially for Data Definition Language (DDL):

---

1. CREATE

- Used to create databases, tables, views, and indexes.

Create Database

```
CREATE DATABASE IF NOT EXISTS database_name;
```

- **IF NOT EXISTS**: Avoids errors if the database already exists.

Create Table

```
CREATE TABLE IF NOT EXISTS table_name (
  column1_name column1_datatype,
  column2_name column2_datatype,
  ...
)
COMMENT 'Table description'
PARTITIONED BY (partition_column1 datatype, partition_column2 datatype)
STORED AS file_format;
```

- **PARTITIONED BY**: Specifies columns for partitioning data.
- **STORED AS**: Defines the file format (e.g., TEXTFILE, ORC, PARQUET).

---

2. DROP

- Deletes databases or tables.

Drop Database

```
DROP DATABASE IF EXISTS database_name [CASCADE | RESTRICT];
```

- **CASCADE**: Deletes all tables within the database before dropping it.
- **RESTRICT**: Prevents dropping if tables still exist in the database.

Drop Table

```
DROP TABLE IF EXISTS table_name;
```

- **IF EXISTS**: Prevents errors if the table doesn't exist.

---

3. TRUNCATE

- Deletes all rows from a table or specific partition without altering the table schema.

Truncate Table

```
TRUNCATE TABLE table_name [PARTITION (partition_column = value)];
```

- **PARTITION**: Truncates only a specified partition instead of the entire table.

---

4. ALTER

- Modifies the structure of an existing database or table.

Alter Database Properties

```
ALTER DATABASE database_name SET DBPROPERTIES ('property_name' = 'property_value');
```

Alter Table

Rename Table
```
ALTER TABLE old_table_name RENAME TO new_table_name;
```

Add Column
```
ALTER TABLE table_name ADD COLUMNS (new_column_name column_datatype);
```

Drop Column
```
ALTER TABLE table_name REPLACE COLUMNS (column1_name column1_datatype, ...);
```

Change Column Name/Type
```
ALTER TABLE table_name CHANGE old_column_name new_column_name column_datatype;
```

Set Table Properties
```
ALTER TABLE table_name SET TBLPROPERTIES ('property_name' = 'property_value');
```

---

5. SHOW

- Displays metadata about databases, tables, and other objects.

Show Databases
```
SHOW DATABASES;
```

Show Tables
```
SHOW TABLES [IN database_name] ['identifier_with_wildcards'];
```

Show Table Properties
```
SHOW TBLPROPERTIES table_name;
```

Show Partitions
```
SHOW PARTITIONS table_name;
```

Show Functions
```
SHOW FUNCTIONS;
```

Show Index
```
SHOW INDEX ON table_name;
```

6. DESCRIBE

- Provides detailed information about a database, table, or view.

Describe Database

```
DESCRIBE DATABASE database_name;
```

Describe Table

```
DESCRIBE [EXTENDED|FORMATTED] table_name;
```

- **EXTENDED**: Shows extra details, such as storage information.
- **FORMATTED**: Outputs in a readable format.

Describe View

```
DESCRIBE [EXTENDED|FORMATTED] view_name;
```

These commands help manage and control the structure, data, and metadata in Hive, making it easier to organize and access data efficiently.

Hive Data Manipulation Language (DML) Commands

In Apache Hive, Data Manipulation Language (DML) commands are used to manipulate data stored in Hive tables. These commands are essential for loading, inserting, updating, deleting, exporting, and importing data, allowing for effective data management and processing in a Hive environment. Below is a detailed explanation of various DML commands in Hive:

1. INSERT

The `INSERT` command is used to add data into Hive tables. You can insert data by either overwriting existing data or appending new data.

Insert Overwrite: This command replaces existing data in the target table or partition with new data.

Syntax:

```
INSERT OVERWRITE TABLE tablename1 [PARTITION (partcol1=val1, partcol2=val2 ...)
[IF NOT EXISTS]]
SELECT select_statement1 FROM from_statement;
```

Explanation:

- **INSERT OVERWRITE TABLE:** Replaces the contents of the target table or partition with the results of the select statement.
- **PARTITION (partcol1=val1, ...):** Specifies the partition for the data insertion. If omitted, the entire table is overwritten.
- **IF NOT EXISTS:** Optional clause to prevent overwriting if the partition already exists.
- **select_statement1:** The query that selects the data to be inserted.

Example:

```
INSERT OVERWRITE TABLE sales_partitioned PARTITION (year=2024, month=09)
SELECT * FROM sales_temp;
```

This command overwrites the partition of `sales_partitioned` for September 2024 with data from `sales_temp`.

Insert Into: This command appends new data to the target table or partition without affecting existing data.

Syntax:

```
INSERT INTO TABLE tablename1 [PARTITION (partcol1=val1, partcol2=val2 ...)]
SELECT select_statement1 FROM from_statement;
```

Explanation:

- **INSERT INTO TABLE:** Appends data selected by the query to the target table or partition.
- The rest of the syntax is similar to `INSERT OVERWRITE`.

Example:

```
INSERT INTO TABLE sales_partitioned PARTITION (year=2024, month=09)
SELECT * FROM daily_sales WHERE date = '2024-09-01';
```

This command appends data from `daily_sales` for September 1, 2024, to the appropriate partition in `sales_partitioned`.

2. UPDATE

The `UPDATE` command in Hive modifies existing records in a table. It changes the value of one or more columns in rows that meet the specified condition.

Syntax:

```
UPDATE tablename SET column = value [, column = value ...] [WHERE expression];
```

Explanation:

- **SET column = value:** Specifies the column(s) to update and their new value(s).
- **WHERE expression:** A condition to identify the rows to update. If omitted, all rows are updated.

Example:

```
UPDATE employee SET salary = 60000 WHERE employee_id = 101;
```

This command updates the salary of the employee with `employee_id` 101 to 60,000.

3. DELETE

The `DELETE` command removes rows from a table that meet the specified condition.

Syntax:

```
DELETE FROM tablename [WHERE expression];
```

Explanation:

- **DELETE FROM tablename:** Specifies the table from which to delete rows.
- **WHERE expression:** A condition to identify the rows to delete. If omitted, all rows are deleted.

Example:

```
DELETE FROM employee WHERE employee_id = 102;
```

This command deletes the record of the employee with `employee_id` 102 from the employee table.

Working with Hive: A Practical Example

Let's explore how to use Hive in a real-world scenario with a simple example.

1. Creating a Table

Imagine you have a CSV file containing sales data that you want to analyze using Hive. First, you need to create a table to hold this data:

```
CREATE TABLE sales_data (
  transaction_id STRING,
  product_id STRING,
  price FLOAT,
  quantity INT,
  transaction_date STRING
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE;
```

This HiveQL statement creates a table with columns matching the structure of your CSV file. The `ROW FORMAT DELIMITED` and `FIELDS TERMINATED BY ','` clauses specify how to parse the data.

2. Loading Data

Next, load the CSV data into your Hive table:

```
LOAD DATA INPATH '/path/to/your/csvfile' INTO TABLE sales_data;
```

This command instructs Hive to read the data from the specified path in HDFS and load it into the `sales_data` table.

3. Running a Query

With the data loaded, you can run queries on it. For example, to find the total sales for each product, you could use:

```
SELECT product_id, SUM(price * quantity) as total_sales
FROM sales_data
GROUP BY product_id;
```

Hive translates this query into a series of MapReduce jobs, distributes them across the cluster, and returns the results.

Hive Execution Engine

In Apache Hive, the execution engine is crucial for processing and executing queries. Let's look at the different execution engines supported by Hive and their roles.

1. MapReduce

- **Overview:** MapReduce was the original execution engine for Hive, processing queries by breaking them into map and reduce tasks.
- **Characteristics:**
  - **Batch-Oriented:** Designed for batch processing, less efficient for interactive queries.
  - **High Latency:** Typically has higher query latency due to its batch nature.
  - **Fault Tolerance:** Provides strong fault tolerance, essential for large-scale data processing.
- **Limitations:**
  - **Slow Performance:** As Hive's popularity grew, MapReduce's slow performance became evident.
  - **High Resource Usage:** MapReduce jobs tend to use more resources and are less efficient than modern engines.

2. Apache Tez

- **Overview:** Apache Tez is a flexible and powerful execution engine designed to replace MapReduce, offering more efficient data processing.
- **Characteristics:**
  - **DAG-Based Execution:** Allows Directed Acyclic Graph (DAG)-based execution, enabling complex query plans and optimizations.
  - **Low Latency:** Significantly reduces query latency, making it suitable for interactive queries.
  - **Efficient Resource Utilization:** Can reuse containers and avoid unnecessary intermediate writes.
- **Benefits:**
  - **Improved Performance:** Generally offers better performance than MapReduce, especially for complex queries.
  - **Customization:** Allows custom processors and edges for more control over the execution process.

3. Apache Spark

- **Overview:** Apache Spark is another execution engine supported by Hive, known for its in-memory processing capabilities, which enhance query execution speed.
- **Characteristics:**
  - **In-Memory Processing:** Processes data in memory, reducing disk I/O and speeding up query execution.
  - **Real-Time and Batch Processing:** Supports both batch and real-time stream processing, providing versatility.
  - **Rich API:** Offers a wide set of APIs in various languages (Scala, Java, Python, R) for complex data processing tasks.
- **Benefits:**
  - **High Performance:** Often outperforms Tez, especially for iterative algorithms and complex transformations.
  - **Scalability:** Highly scalable and efficient for very large datasets.
  - **Integration with Machine Learning:** Works well with machine learning (MLlib) and graph processing (GraphX) libraries.

When to Use Apache Hive

Hive is a powerful tool, best used in the right contexts. Think of Hive as a data warehousing solution; it's excellent for reporting and analytics but not suited for day-to-day transactional operations, such as those in online shopping carts or banking applications. Hive is most effective when analyzing large amounts of data to uncover trends, generate reports, or run complex queries that aid in business decision-making.

Big Companies Using Hive for Analytics

Here are some examples of major companies that use Hive for their big data needs:

- **TikTok:** Utilizes Hive to store vast amounts of data that feed its recommendation engine, suggesting videos based on user preferences and behavior, which keeps the platform engaging.
- **Walmart:** Integrates Hive into decision-making processes to analyze customer behavior and sales patterns, resulting in a 10% to 15% increase in online sales, translating to an additional $1 billion in revenue.
- **Netflix:** Relies on Hive for analytics, running hundreds of reporting jobs daily to gain insights into viewer behavior and optimize content delivery.
- **Facebook:** Manages over 300 petabytes of data in Hive, processing about 600 terabytes of new data daily to transform raw data into actionable insights for platform optimization.
- **UnitedHealth Group (UHG):** Analyzes customer feedback from marketing channels and social media using Hive, enabling informed decisions and targeted marketing efforts.

Store Data Effieciently in Big Data - Oct 14

- **Hive Sessions**
- **Partitioning**
- **Bucketing**
- **SET Commands in Hive**
- **Hive Storage Formats**

Table Storage Optimizations

In Apache Hive, managing data efficiently and speeding up queries are key for handling large amounts of data across multiple systems. Hive has several ways to make tables and queries faster, including *partitioning*, *bucketing*, and other methods. Understanding these will help you work with data more effectively.

Hive Sessions

Before diving deeper, it's helpful to understand what a Hive session is. A *session* in Hive starts when a user connects to the Hive server and ends when they disconnect or the session is manually closed. During a session, a user can run queries and commands and interact with the Hive database. This concept will make more sense as we move into advanced topics.

Hive Table Partitioning

Partitioning is a way to break a large table into smaller parts based on the values in one or more columns. Partitioning helps speed up queries by reducing the amount of data scanned during a query.

How Partitioning Works

- **Partition Columns**: Data is divided based on specific column values. Each unique value creates a separate partition.
- **Partition Directory Structure**: In HDFS, partitions are stored in subdirectories. For example, if a table is partitioned by *year* and *month*, the directory might look like `/year=2024/month=08/`.

Creating a Partitioned Table

You can create a partitioned table like this:

```
CREATE TABLE sales_data (
  product_id INT,
  sales_amount FLOAT
)
PARTITIONED BY (year INT, month INT);
```

Adding Partitions

To add a partition manually, use the `ALTER TABLE` command:

```
ALTER TABLE sales_data ADD PARTITION (year=2024, month=08) LOCATION
'/user/hive/warehouse/sales_data/year=2024/month=08/';
```

Automatic Partition Management

Hive can automatically manage partitions if you use the `LOAD DATA` command and specify the partition columns:

```
LOAD DATA INPATH '/path/to/data' INTO TABLE sales_data PARTITION (year=2024, month=08);
```

Querying Partitioned Tables

When querying a partitioned table, Hive can skip irrelevant partitions, making queries faster:

```
SELECT * FROM sales_data WHERE year=2024 AND month=08;
```

Where is Partition Information Stored?

Partition details, along with table metadata (like column details), are stored in the *Hive MetaStore*.

Hive Table Bucketing

Bucketing in Hive is a way to improve query speed by dividing a dataset into smaller parts called buckets. Each bucket holds part of the data, making it faster to query, especially for large datasets. Bucketing is useful when you often filter data by a specific column (called the bucketing column) and need quick access to rows based on that column.

How Bucketing Works

- **Bucket Column**: Data is divided into buckets based on a hash of the bucketing column value. This hash value decides which bucket a row will go into.
- **Number of Buckets**: The number of buckets is set when creating the table.

Imagine a table called `CustomerOrders` with millions of rows that includes customer IDs, order details, and transaction amounts. A query like this might take a while if the table is large:

```
SELECT * FROM CustomerOrders WHERE CustomerID = 12345;
```

Using bucketing in this case can make the query faster by focusing on specific buckets instead of the entire dataset.

Creating a Bucketed Table

Here's an example of creating a bucketed table:

```
CREATE TABLE IF NOT EXISTS
CustomerOrders (
  OrderID INT,
  CustomerID INT,
  OrderDate STRING,
  OrderAmount DECIMAL(10, 2),
  Status STRING
```

```
)
CLUSTERED BY (CustomerID) INTO 10 BUCKETS
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
STORED AS TEXTFILE;
```

- `CLUSTERED BY (CustomerID)` specifies `CustomerID` as the bucketing column.
- `INTO 10 BUCKETS` divides the table into 10 buckets based on `CustomerID`.

Loading Data into Buckets

When you load data into this table, Hive places rows in the correct bucket based on the hash of the `CustomerID`:

```
LOAD DATA INPATH '/path/to/customer_data' INTO TABLE CustomerOrders;
```

Advantages of Bucketing

- **Optimized Query Performance**: Queries filtered by `CustomerID` are faster since Hive only scans the relevant bucket(s).
- **Smaller Data Files**: Instead of storing all data in one large file, bucketing creates smaller, manageable files within each partition.
- **Efficient Joins**: When two tables are bucketed on the same column (e.g., `CustomerID`), Hive can join them more efficiently. This is called a *bucket join*, which reduces the need to move data across nodes.

Querying Bucketed Tables

When you query a bucketed table, Hive can quickly find the relevant bucket:

```
SELECT * FROM CustomerOrders WHERE CustomerID = 1001;
```

Since Hive knows which bucket holds `CustomerID = 1001`, it only scans that bucket, speeding up the query.

Bucketed Table Join

If there's another table, `CustomerDetails`, also bucketed by `CustomerID`, you can join `CustomerOrders` and `CustomerDetails` efficiently:

```
SELECT o.OrderID, d.CustomerName
FROM CustomerOrders o
JOIN CustomerDetails d
ON o.CustomerID = d.CustomerID;
```

Since both tables are bucketed by `CustomerID`, Hive can perform a *bucket-by-bucket join*, which skips a full table scan and avoids moving data between nodes.

SET Commands in Hive

SET commands in Hive allow you to adjust settings to optimize performance or control query behavior. These settings apply to the entire Hive session (the time you are connected to HiveServer), not just to individual queries. Once a setting is applied, it affects all queries until you either change the setting or end the session.

Setting and Retrieving Configuration Properties

To set a property:

```
SET <property_name>=<value>;
```

To check the value of a specific property:

```
SET <property_name>;
```

When to Use SET Commands in Hive

1. **Optimizing Query Performance**: Adjust memory, enable/disable certain features like vectorized execution, or set up parallel processing.
2. **Controlling Query Execution**: Modify settings for dynamic partitioning, choose the execution engine, or handle query caching.
3. **Customizing Logging and Debugging**: Set logging levels, enable debug mode, or configure error handling.
4. **Defining and Using Variables**: Create reusable variables that can be used across multiple queries or scripts.

Example of Special Settings: Dynamic Partitioning

Here's an example to illustrate how these settings can affect query execution.

Enable Dynamic Partitioning:

```
SET hive.exec.dynamic.partition=true;
```

- **Purpose**: Allows Hive to automatically create partitions during an `INSERT` operation based on the values in the data.
- **When to Use**: Useful when you don't know partition values beforehand or need to load data with many possible partition values.

Set Partition Mode to Nonstrict:

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

- **Purpose**: Allows the creation of dynamic partitions even if not all partition columns are specified in the `INSERT` statement.
- **When to Use**: Helpful when you want Hive to handle partition creation without requiring you to specify every partition column.

Modes:

- *Strict Mode (default)*: Requires at least one static partition column in the `INSERT` statement to avoid creating too many partitions, which can overload the system.
- *Nonstrict Mode*: Allows Hive to dynamically create all partition values without needing any static columns.

Example Usage

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

INSERT INTO TABLE sales_partitioned
PARTITION (year, month)
SELECT product_id, product_name, sale_amount, sale_date, year(sale_date) as year,
month(sale_date) as month
FROM sales;
```

In this example:

- Hive dynamically creates partitions for `year` and `month` based on `sale_date`.
- Since `nonstrict` mode is enabled, you don't need to specify a static partition value, letting Hive fully manage the partitions.

We'll cover more useful settings in upcoming lessons.

Hive Storage Formats

1. TEXTFILE

- **Description**: Default storage format in Hive, storing data as plain text where each line is a row and fields are separated by a delimiter (e.g., comma or tab).
- **Pros**:
  - Human-readable, easy to inspect.
  - Compatible with most tools.
  - Simple to use.
- **Cons**:
  - Inefficient storage and slower processing due to no compression.
  - Requires more storage space.
- **Use Case**: Simple datasets, prototyping, or when interoperability with other systems is a priority.

Example:

```
CREATE TABLE sales_txt (
  product_id INT,
  product_name STRING,
  sale_date STRING,
  sale_amount FLOAT
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE;
```

2. SequenceFile

- **Description**: Binary file format storing data as key-value pairs, designed for efficient MapReduce processing.
- **Pros**:
  - Splittable, enabling parallel processing.
  - Compressible, saving storage.
  - Suitable for large datasets.
- **Cons**:
  - Not human-readable.
  - Requires complex handling for keys/values.
- **Use Case**: Large datasets requiring efficient storage and processing, especially in MapReduce jobs.

Example:

```
CREATE TABLE sales_seq (
  product_id INT,
  product_name STRING,
  sale_date STRING,
  sale_amount FLOAT
)
STORED AS SEQUENCEFILE;
```

3. ORC (Optimized Row Columnar)

- **Description**: Columnar storage format optimized for Hive, providing fast processing, high compression, and support for ACID operations.
- **Pros**:
  - High compression ratio.
  - Optimized for complex queries.
  - Supports ACID and advanced features like predicate pushdown.
- **Cons**:
  - Slightly slower write performance due to optimization.
  - Limited compatibility outside the Hadoop ecosystem.
- **Use Case**: Large datasets where performance and storage efficiency are critical, such as data warehouses.

Example:

```
CREATE TABLE sales_orc (
  product_id INT,
  product_name STRING,
  sale_date STRING,
  sale_amount FLOAT
)
STORED AS ORC;
```

4. Parquet

- **Description**: Columnar storage format optimized for analytical queries, with broad compatibility beyond Hive.
- **Pros**:
  - High compression and efficient storage.
  - Suitable for queries on specific columns.
  - Widely supported across various tools (e.g., Spark, Impala).
- **Cons**:
  - Slower write times due to columnar format.
  - Less efficient for full row reads.
- **Use Case**: Analytical workloads in environments using multiple tools.

Example:

```
CREATE TABLE sales_parquet (
  product_id INT,
  product_name STRING,
  sale_date STRING,
  sale_amount FLOAT
)
STORED AS PARQUET;
```

5. Avro

- **Description**: Row-based format supporting schema evolution, storing schema definitions with data.
- **Pros**:
  - Supports schema evolution.
  - Compact and efficient.
  - Integrates well with various serialization systems.
- **Cons**:
  - Requires schema management.
  - Less efficient for column-based queries.
- **Use Case**: Datasets with evolving schema, interoperability with systems like Kafka.

Example:

```
CREATE TABLE sales_avro (
  product_id INT,
  product_name STRING,
  sale_date STRING,
  sale_amount FLOAT
)
STORED AS AVRO;
```

==========

==========

6. JSON

- **Description**: Stores data as JSON, a human-readable format supporting nested structures.
- **Pros**:
  - Easy to read and use.
  - Flexible, supports nested data.
  - Widely compatible.
- **Cons**:
  - Inefficient for storage and slower to process.
  - No built-in schema, risking inconsistencies.
- **Use Case**: Logs, web data, or when human readability and nested structures are important.

Example:

```
CREATE TABLE sales_json (
  product_id INT,
  product_name STRING,
  sale_date STRING,
  sale_amount FLOAT
)
ROW FORMAT SERDE 'org.apache.hive.hcatalog.data.JsonSerDe'
STORED AS TEXTFILE;
```

7. CSV (Comma-Separated Values)

- **Description**: Simple text file format with rows separated by commas, commonly used for data exchange.
- **Pros**:
  - Simple and readable.
  - Widely supported.
  - Easy to generate and parse.
- **Cons**:
  - No support for complex/nested data types.
  - Large file size and slower processing compared to binary formats.
- **Use Case**: Simple datasets, data exchange, or when readability is needed.

Example:

```
CREATE TABLE sales_csv (
  product_id INT,
  product_name STRING,
  sale_date STRING,
  sale_amount FLOAT
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
```

- `STORED AS TEXTFILE;`

HiveQL 1: CTE/Temporary table/Views/UDF - Oct 16

- **Views**
- **Materialized View**
- **CTE (Common table expresstions)**
- **UDF**

View:

A **view** is essentially a virtual table in a database. It does not store actual data but rather stores a query. This query is executed every time the view is accessed. So, when you create a view, you are storing the SQL query under an alias or a name, but not the result of that query. Views are useful for simplifying complex queries, improving security by limiting data access, and providing a consistent interface for data retrieval. However, since the view does not store data, each time you query the view, the underlying tables must be accessed to retrieve the latest data.

CREATE VIEW employee_view AS
SELECT employee_id, name, department FROM employees;

-- We want to design the dataset to have one row per date.
-- We do not need to include detailed information about all the customers or all products.

```
CREATE VIEW manager_query_reporting AS
(
   SELECT
      c.customer_id,
      CONCAT(c.customer_first_name, ' ', c.customer_last_name) AS full_name,
      cp.market_date,
      ROUND(SUM(cp.quantity * cp.cost_to_customer_per_qty), 2) AS total_qty,
      v.vendor_id,
      v.vendor_name,
      v.vendor_type
   FROM
      customer_purchases cp
   LEFT JOIN
      customer c ON c.customer_id = cp.customer_id
   LEFT JOIN
      vendor v USING (vendor_id)
   GROUP BY
      cp.market_date,
      c.customer_id,
      c.customer_first_name,
      c.customer_last_name,
      v.vendor_id,
      v.vendor_name,
      v.vendor_type
   ORDER BY
      1, 3
);
```

Materialized View:

A **materialized view**, unlike a regular view, stores both the query and the actual data. This means that when the materialized view is created, the result of the query is physically stored in the database. As a result, accessing a materialized view is typically faster because the data doesn't need to be fetched from the underlying tables each time. However, the data in a materialized view may become stale, and it needs to be refreshed periodically to reflect the latest changes from the underlying tables.

```
CREATE MATERIALIZED VIEW employee_summary AS
SELECT department, COUNT(*) AS total_employees
FROM employees
GROUP BY department;
```

CTE (Common Table Expression):

A **Common Table Expression (CTE)** is a temporary result set that you can reference within a `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement. CTEs make your SQL queries more readable and maintainable by allowing you to break down complex queries into smaller, manageable parts. You can think of a CTE as creating a temporary table or a "query alias" that can be reused within a single query. It is like a variable and It is only valid till the first Select query. If we want to access it in the next Select query it is not available there

A CTE is defined using the `WITH` keyword followed by one or more query aliases, and these aliases can then be used in the final query. It's similar to defining a variable in programming, allowing you to reference the CTE multiple times in your query without repeating the logic.

WITH [query_alias] AS ([query]), [query2_alias] AS ([query2]) SELECT * FROM query_alias;

```
WITH employee_cte AS
(
        SELECT employee_id, name, department_id FROM employees
        ),
department_cte AS
(
        SELECT department_id, department_name FROM departments
        )
SELECT e.name, d.department_name
FROM employee_cte e
JOIN department_cte d ON e.department_id = d.department_id;
-- Get the information from employees.
-- If they have commission values, add that to the salary.
-- If not, give them a 10% salary hike.
```

Using subqueries

When using subqueries, we follow a bottom-to-top approach. The inner (bottom) query is executed first, and its result is then passed to the outer (top) query. This means the outer query depends on the output of the inner query to perform its operation.

```
SELECT *
FROM
(
   SELECT
      employee_id,
      CONCAT(first_name, ' ', last_name) AS full_name,
      salary,
```

```
        commission_pct,
        department_id,
        ROUND(
            CASE
                WHEN commission_pct IS NULL
                THEN salary + salary * 0.10
                ELSE salary + salary * commission_pct
            END, 2
        ) AS new_salary
    FROM
        employees
)t
WHERE t.new_salary > 15000;
```

Using CTE (Common Table Expression):

When using a CTE, we follow a **top-to-bottom** approach. The CTE (top query) is defined first, and then the main query (bottom query) is executed using the result of the CTE.

-- Select only those who are earning more than 15000.

```
WITH amit AS
(
    SELECT
        employee_id,
        CONCAT(first_name, ' ', last_name) AS full_name,
        salary,
        commission_pct,
        department_id,
        ROUND(
            CASE
                WHEN commission_pct IS NULL
                THEN salary + salary * 0.10
                ELSE salary + salary * commission_pct
            END, 2
        ) AS new_salary
    FROM
        employees
),
cte1 AS
(
    SELECT *
    FROM amit
    WHERE commission_pct IS NOT NULL
)
SELECT *
FROM cte1
WHERE new_salary > 15000;
```

```
-- Get the list of employees who earn more than the average salary of their department.
-- 1. Get average salary from department.
-- 2. Find employees earning more than the average salary.

SELECT *
FROM employees e
INNER JOIN
(
   SELECT
      department_id,
      ROUND(AVG(salary), 2) AS avg_salary_dept
   FROM employees
   GROUP BY department_id
) d
USING (department_id)
WHERE e.salary > d.avg_salary_dept;

-- Get all employees from department 90.

SELECT *
FROM employees
WHERE department_id = 90;

-- Using Common Table Expressions (CTE) to find employees earning more than the average salary.

WITH demo AS
(
   SELECT
      department_id,
      ROUND(AVG(salary), 2) AS avg_salary_dept
   FROM employees
   GROUP BY department_id
),
abv_avg AS
(
   SELECT *
   FROM employees e
   JOIN demo d
   USING (department_id)
   WHERE salary > avg_salary_dept
)
SELECT *
FROM abv_avg;
```

A User-Defined Function (UDF)

It is a custom function created by the user to extend the capabilities of a query language, like SQL, in databases such as BigQuery. UDFs allow you to write reusable logic that you can call within your queries, which is especially useful when you need to perform complex or repetitive tasks that are not built into standard SQL.

Optimized queries - Indexes and Partitioning Cont

How to Write Efficient SQL Queries

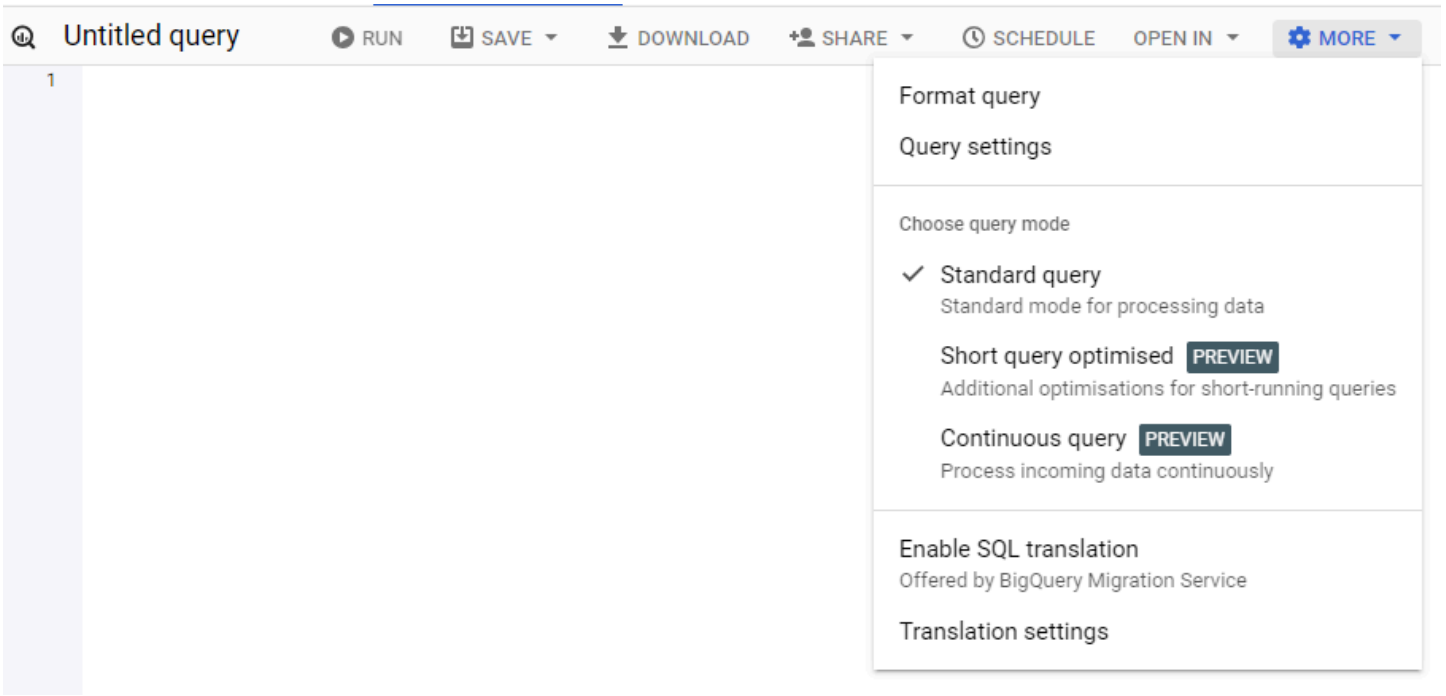
Throughout this lecture, we'll be using the following datasets from **BigQuery's public data repository**:

- **crypto_ethereum dataset**
- **bikeshare tables from san_francisco dataset**

Before You Start Writing Your Queries:

1. **Click on the "More" tab** on your SQL workspace.
2. **Select the "Query settings" option.**
3. **Go to Cache Preference** and
4. **Uncheck "Use cached results."**

Remember that you'll have to **repeat this process every time you open a new query editor.**



Common Terminology Definitions in BigQuery

1. Elapsed Time:

- **Elapsed time**, also known as query execution time, is the total time it takes for a query to run from the moment it is submitted until the results are returned.
- It includes the time spent on processing, reading data, shuffling data (if necessary), and any other operations required to execute the query.

2. Slot Time Consumed:

- **Slot time consumed** refers to the amount of computational resources (slots) that a query consumes during its execution.
- The more slots a query consumes, the faster it is likely to execute. However, higher slot usage also means higher costs, as you pay for the slots used.

3. Bytes Shuffled:

- The redistribution of data across different slots for processing is called **shuffling**, and it involves moving data between different parts of the distributed system.
- **Bytes shuffled** represents the volume of data that is moved or shuffled between different computational nodes during query execution.

4. Bytes Spilled to Disk:

- When a query's intermediate results or data exceed the memory capacity of the allocated slots, some of that data may be written to disk temporarily for storage. This is referred to as **spilling to disk**.

Summary:

Lower elapsed times, efficient slot usage, minimized bytes shuffled, and reduced bytes spilled to disk are all indicative of well-optimized and performant queries.

Optimization tips

1. **COUNT(*) vs COUNT(1):** There's no difference in speed between `COUNT(*)` and `COUNT(1)`. Both take the same time to run.
2. **Select only the columns you need:** Never use `SELECT *`. It's better to choose only the specific columns you need, which makes the query faster.
3. **LIMIT doesn't save cost**:
    - Using `LIMIT` can make a query run faster, but it doesn't lower the cost.
    - The database still reads all the data before applying the limit.
4. **Use EXISTS() instead of COUNT():**
    - If you just want to check if something exists, use `EXISTS()` instead of `COUNT()`. `EXISTS()` stops as soon as it finds what it's looking for, while `COUNT()` checks the entire table.
5. **Use APPROX_COUNT_DISTINCT instead of COUNT(DISTINCT):**
    - `COUNT(DISTINCT)` gives an exact number but is slower.
    - `APPROX_COUNT_DISTINCT` gives an approximate result and is much faster, especially for large datasets.
6. **Use window functions instead of self-joins:** Self-joins have high complexity (n * n), but window functions process each row once (1: n), making them faster. Always prefer window functions over joins when possible.
7. **Filter your data early and often:** When using `GROUP BY`, it's better to filter earlier rather than after the data is processed. For example, if you have two queries (CTEs), and you apply `GROUP BY` after both queries run, it might scan a lot of data (like 100 records). Instead, apply `GROUP BY` in the second query (CTE), which will reduce the data processed, scanning fewer records (like 12 instead of 100). This makes the query much more efficient.

8. **Use MAX() instead of RANK():** It's better to use aggregation functions like `MAX()` with window functions. Aggregations work on the whole table after getting the final result. `RANK()` checks each row individually, which scans more data.
9. **Join big tables first:** When joining tables, start with the big tables and then move to smaller ones. This can help reduce the time it takes to scan the data.
10. **Does the order of WHERE clauses matter?:** Some complex queries with multiple conditions, placing the most selective conditions (those that filter the most rows) first can sometimes improve performance by reducing the amount of data scanned early on.
11. **Should we use ORDER BY at the end?:** Yes, it's best to use `ORDER BY` at the end of your query. Sorting the entire result at once is faster than sorting multiple times during the query.

IF we want optimization then Use Aggregation functions like groupby first then window functions at last slef join

HiveQL 2 : Window functions - Oct 18

In the last lecture, we discussed a few unique features of HiveQL, in this lecture, we will discuss about window functions. Window functions are also used in MySQL. Let's understand window functions with the help of a case study.

Window functions are special types of functions in SQL that allow you to perform calculations across a set of rows related to the current row, without collapsing the result set into a single output. This is useful for calculating running totals, rankings, and other types of analytics.

Key Features of Window Functions:

- **Over a Window**: Unlike regular aggregate functions that group rows into a single result, window functions maintain the detail of each row while still providing summary information.
- **PARTITION BY**: You can divide the data into partitions (groups) to perform calculations separately within each group.
- **ORDER BY**: You can specify the order of rows within each partition to control how the calculations are performed.

Common Window Functions:

1. **ROW_NUMBER()**: Assigns a unique number to each row within a partition, based on the specified order.
    - Example: `ROW_NUMBER() OVER (PARTITION BY vendor_id ORDER BY sales DESC)`.
2. **RANK()**: Similar to ROW_NUMBER(), but if two rows have the same value, they receive the same rank, and the next rank will skip numbers.
    - Example: `RANK() OVER (ORDER BY score DESC)`.
3. **DENSE_RANK()**: Similar to RANK(), but does not skip ranks when there are ties.
    - Example: `DENSE_RANK() OVER (ORDER BY score DESC)`.
4. **SUM()**: Can be used as a window function to calculate a running total.
    - Example: `SUM(sales) OVER (ORDER BY date)`.
5. **AVG()**: Calculates the average value over a specified window.
    - Example: `AVG(price) OVER (PARTITION BY category ORDER BY date)`.
6. **LEAD()**: Accesses data from the next row in the result set.
    - Example: `LEAD(price) OVER (ORDER BY date)` gives the price of the next row.

7. **LAG()**: Accesses data from the previous row.
    - Example: `LAG(price) OVER (ORDER BY date)` gives the price of the previous row.

```
SELECT
    vendor_id,
    product_id,
    original_price,
    ROW_NUMBER() OVER (PARTITION BY vendor_id ORDER BY original_price DESC) AS price_rank
FROM vendor_inventory;
```

In this example, each product is ranked by price within its vendor, without losing the details of each row.

Problem Statement

You are a Data Analyst at the Food Corporation of India (FCI). Your job is to study the Farmer's market, called Mandis.

Dataset: Farmer's Market Database

Create a Table:

```
CREATE TABLE vendor_inventory (
    vendor_id INT,
    market_date DATE,
    product_id INT,
    original_price DECIMAL(10, 2),
    product_name VARCHAR(100)
);
```

Insert Data:

```
INSERT INTO vendor_inventory (vendor_id, market_date, product_id, original_price, product_name) VALUES
(1, '2024-10-15', 101, 150.00, 'Tomatoes'),
(1, '2024-10-15', 102, 250.00, 'Potatoes'),
(1, '2024-10-15', 103, 200.00, 'Carrots'),
(2, '2024-10-15', 201, 100.00, 'Spinach'),
(2, '2024-10-15', 202, 50.00, 'Cabbage'),
(2, '2024-10-15', 203, 120.00, 'Onions'),
(3, '2024-10-15', 301, 180.00, 'Apples'),
(3, '2024-10-15', 302, 220.00, 'Oranges'),
(3, '2024-10-15', 303, 210.00, 'Bananas'),
(4, '2024-10-15', 401, 95.00, 'Peppers'),
(4, '2024-10-15', 402, 180.00, 'Broccoli'),
(4, '2024-10-15', 403, 75.00, 'Lettuce');
```

Use Case 1: Find the Most Expensive Item per Vendor

Goal: Get the highest price for each vendor.

Steps:

1. Group the data by `vendor_id`.
2. Find the maximum price in each group.

SQL Query:

SELECT vendor_id, MAX(original_price) AS highest_price

FROM vendor_inventory

GROUP BY vendor_id;

This query shows the most expensive item for each vendor.

Use Case 2: Rank Products by Price

Goal: Rank products in each vendor's inventory, where higher prices get a lower rank.

Steps:

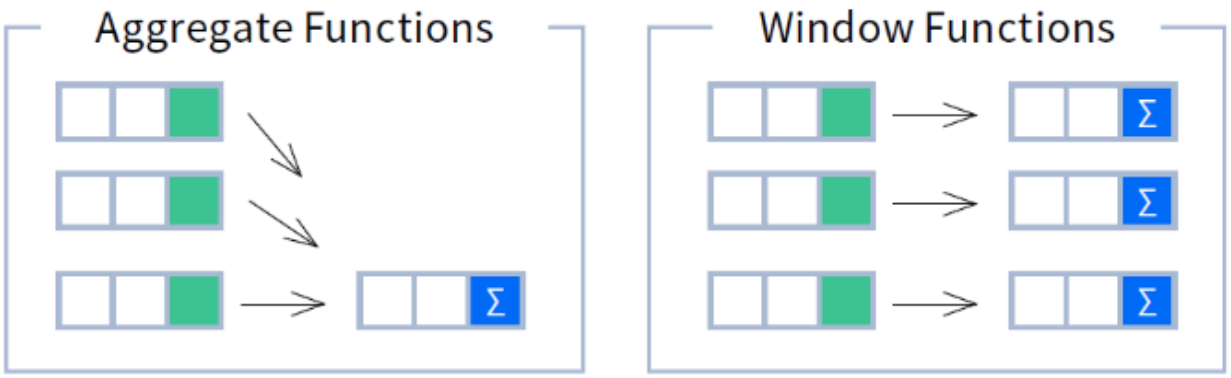
- **Don't group the rows, as we want to rank all products.**
- **Use window functions to keep details for each product while ranking.**

What is a Window Function?

A window function lets you compare a row to other rows in a group without losing individual row details. It helps answer questions like:

- What rank does this row have in the sorted data?
- How does a row's value compare to the previous or next row?
- How does a row's value compare to the average of its group?

Window functions return both group calculations and details for each item.



We need a function to rank rows based on a value—in this case, to rank products by price for each vendor. This function is called ROW_NUMBER().

ROW_NUMBER() OVER (<partition_definition> <order_definition>)

Example Query:

SELECT

```
    vendor_id,
    market_date,
    product_id,
    original_price,
    ROW_NUMBER() OVER (PARTITION BY vendor_id ORDER BY original_price DESC) AS price_rank
FROM farmers_market.vendor_inventory;
```

Explanation of the Syntax:

- The ROW_NUMBER() line means: "Give a number to each product for each vendor, based on price, from highest to lowest."
- OVER() tells the database to apply this ranking function to a group of rows.
- Inside the parentheses:
    - PARTITION BY vendor_id splits the rows into groups by vendor, similar to grouping but without combining them.
    - ORDER BY original_price DESC sorts the products within each vendor group by price, from highest to lowest. The most expensive product gets the rank of 1.

Output Explanation:

- For each vendor, the products will be sorted by price from highest to lowest, and the ranking will be shown in a column called price_rank.
- The numbering restarts for each vendor, so the most expensive item for each vendor will have a price_rank of 1.

Use Case 3: Get All Products with a Price Rank of 1 for Each Vendor

Goal: Find all products for each vendor that are the most expensive.

```
SELECT * FROM (
    SELECT
        vendor_id,
        market_date,
        product_id,
        original_price,
        ROW_NUMBER() OVER (PARTITION BY vendor_id ORDER BY original_price DESC) AS price_rank
    FROM farmers_market.vendor_inventory
    ORDER BY vendor_id
) x
WHERE x.price_rank = 1;
```

Explanation of the Query:

- This query has a different structure because it includes a **subquery**. A subquery is a query inside another query, also called "querying from a derived table."
- We treat the results of the inner query as if it's a table, naming it **x**. We then select all columns from it and filter to keep only rows where the **price_rank** is 1.
- The **price_rank** column shows the rank of each product based on its price, with the most expensive item having the lowest rank (1).

Why Use a Subquery?

- If we tried to filter for **price_rank = 1** directly in the main query, it would not work because the ranking has not been calculated yet.
- The database processes queries in a strict order. Here's the order:
    1. **FROM** - gets data from the tables.
    2. **WHERE** - filters data based on conditions.
    3. **GROUP BY** - groups the data.
    4. **Aggregate functions** - applies calculations to the groups. (COUNT(), SUM(), AVG(), MIN(), MAX(), GROUP_CONCAT())
    5. **HAVING** - filters the groups.
    6. **Window functions** - calculates rankings and other functions.
    7. **SELECT** - selects the columns to display.
    8. **DISTINCT** - removes duplicate values.
    9. **UNION/INTERSECT/EXCEPT** - combines or compares results.
    10. **ORDER BY** - sorts the results.
    11. **OFFSET** - skips the first rows.
    12. **LIMIT/FETCH/TOP** - selects only a certain number of rows.

Note:

You can also use **ROW_NUMBER** without **PARTITION BY** to rank all records together instead of by vendor.

Use Case 4: Using RANK for Tied Values

The issue with the **ROW_NUMBER()** function is that it gives different ranks to rows with the same values. If you want rows with the same value to have the same rank, use the **RANK()** function instead.

Goal: To return all products with the highest price per vendor, even when multiple products have the same price.

```
SELECT
    vendor_id,
    market_date,
    product_id,
    original_price,
    RANK() OVER (PARTITION BY vendor_id ORDER BY original_price DESC) AS price_rank
FROM farmers_market.vendor_inventory
ORDER BY vendor_id, original_price DESC;
```

Output Explanation:

- In this output, for **vendor_id 1**, the ranks go from 1 to 2 to 4, skipping 3. This happens because two products have the same price, so they share the second rank, resulting in no third place.
- If you want to avoid skipping ranks when there's a tie, use the **DENSE_RANK** function instead.
- If you don't want ties at all, stick with **ROW_NUMBER**.

Practical Use:

Both **ROW_NUMBER()** and **RANK()** can help answer questions like:

- “What are the top 10 items sold at the farmer’s market by price?” You can filter the results to show only rows with ranks of 10 or lower.

Use Case 5: Finding Products Above Average Price

As a farmer, you want to know which of your products are priced higher than the average price for each market date.

Step 1: Calculate the Average Price

We can use the **AVG()** function as a window function to find the average price for each market date and compare it to each product's price.

```
SELECT * FROM (
   SELECT
      vendor_id,
      market_date,
      product_id,
      original_price,
      ROUND(AVG(original_price) OVER (PARTITION BY market_date), 2) AS
average_cost_product_by_market_date
   FROM farmers_market.vendor_inventory
) x
WHERE x.vendor_id = 8
AND x.original_price > x.average_cost_product_by_market_date
ORDER BY x.market_date, x.original_price DESC;
```

Use Case 6: Count Different Products by Vendor and Date

You want to count how many different products each vendor brought to the market on each date and show that count for every product.

```
SELECT
   vendor_id,
   market_date,
   product_id,
   original_price,
   COUNT(product_id) OVER (PARTITION BY market_date, vendor_id) AS
vendor_product_count_per_market_date
FROM vendor_inventory
ORDER BY vendor_id, market_date, original_price DESC;
```

Explanation:

- This query counts the number of products for each vendor on each market date.
- Even if you only look at one row for vendor 7 on July 6, 2019, you can see that it represents one of 4 products that the vendor had that day.

Window Frames

Let's consider a dataset like this

Sales Data

	sale_id	product_id	sale_date	amount
1	1	101	2023-01-01	200
2	2	101	2023-01-02	300
3	3	102	2023-01-03	150
4	4	102	2023-01-04	350
5	5	103	2023-01-05	100
6	6	103	2023-01-06	250
7	7	101	2023-01-07	220
8	8	101	2023-01-08	320
9	9	102	2023-01-09	180
10	10	102	2023-01-10	400

We'll use this to explain **window functions** like **LEAD, LAG, FIRST_VALUE, and NTH_VALUE.**

This dataset includes:

- **sale_id: A unique identifier for each sale.**
- **product_id: The ID of the product sold.**
- **sale_date: The date of the sale.**
- **amount: The sales amount on that day.**

Window frames let you choose a specific group of rows for the window function to use, giving you more control over calculations. While functions like LEAD, LAG, FIRST_VALUE, and NTH_VALUE work on rows defined by a partition and order, window frames can limit these rows even more for each calculation.

A window frame sets a range of rows that moves with each row in the ordered set. This range shows how many rows before or after the current row are included in the calculation. It might sound complex now, but let's look at some examples to make it clearer.

Examples of Window Frames

1. LEAD() Function

What is LEAD?
The LEAD function helps you access data from upcoming rows. This is useful if you want to compare the current row's data with data from future rows. For example, looking at today's sales, you could ask, "What was the sales amount for the next day?"

How it works
The LEAD function's syntax looks like this:

LEAD(column_name, offset, default_value)
OVER (PARTITION BY column_name ORDER BY column_name)

- **column_name**: The column you want to fetch data from.
- **offset**: The number of rows ahead you want to look. For example, 1 means "the next row," 2 means "two rows ahead."
- **default_value**: Optional. This defines what value should be returned if the function goes beyond available rows.

Example
Let's say you want to compare today's sales with the next day's sales. Here's how it would look with the following dataset.

sale_id	product_id	sale_date	amount
1	101	2023-01-01	200
2	101	2023-01-02	300
3	101	2023-01-03	250

Using the LEAD function, you can see each day's sales along with the next day's sales for comparison.

Here's how applying the **LEAD function** to view the next day's sale amount for each product would look:

```
SELECT
   sale_id,
   product_id,
   sale_date,
   amount,
   LEAD(amount, 1) OVER (PARTITION BY product_id ORDER BY sale_date) AS next_day_sales
FROM
   sales_data;
```

This query retrieves each sale along with the sale amount for the next day, helping you easily compare daily sales.

Output:

sale_id	product_id	sale_date	amount	next_day_sales
1	**101**	**2023-01-01**	**200**	**300**
2	**101**	**2023-01-02**	**300**	**NULL**
3	**102**	**2023-01-03**	**150**	**NULL**

Explanation:

- **For the first row (2023-01-01), the next day's sale amount is 300.**

- **For the second row (2023-01-02), there is no sale on the following day, so it returns NULL.**

2. LAG() Function

What is LAG?
The LAG function works in the opposite way of LEAD. It lets you access data from previous rows, which is useful if you want to compare today's value with yesterday's.

How it works
The syntax for LAG is similar to LEAD:

LAG(column_name, offset, default_value) OVER (PARTITION BY column_name ORDER BY column_name)

- column_name: The column you want to pull data from.
- offset: The number of rows back to look (e.g., 1 for the previous row).
- default_value: The value to return if there isn't a previous row.

Example
Using the same dataset, let's see how LAG works if we want to compare today's sale with the previous day's sale:

SELECT sale_id, product_id, sale_date, amount, LAG(amount, 1) OVER (PARTITION BY product_id ORDER BY sale_date) AS prev_day_sales FROM sales_data;

Output:

sale_id	product_id	sale_date	amount	prev_day_sales
1	101	2023-01-01	200	(NULL)
2	101	2023-01-02	300	200
3	102	2023-01-03	150	(NULL)

Explanation:

- For the first row (2023-01-01), there is no previous day, so it returns NULL.
- For the second row (2023-01-02), the previous day's sale amount is 200.

3. FIRST_VALUE() Function

What is FIRST_VALUE?
The `FIRST_VALUE` function returns the first value within a defined window, ordered by a specific column. It's useful when you want to compare the current value to the first value in a dataset or within a group.

How it works
Here's the syntax:

FIRST_VALUE(column_name)
OVER (PARTITION BY column_name ORDER BY column_name)

Example
Suppose you want to find the very first sale amount for each product and compare it with today's sale amount:

SELECT sale_id, product_id, sale_date, amount,
FIRST_VALUE(amount) OVER (PARTITION BY product_id ORDER BY sale_date) AS first_sale_amount
FROM sales_data;

Output:

sale_id	product_id	sale_date	amount	first_sale_amount
1	101	2023-01-01	200	200
2	101	2023-01-02	300	200
3	102	2023-01-03	150	150

Explanation:

- For each row, it shows the first sale amount for that product.
- For product 101, the first sale amount is 200.

What is NTH_VALUE?
The NTH_VALUE function helps you get the Nth value from a series of rows in a specific order. This is useful when you want to see the 2nd or 3rd sale amount over time.

How it works
Here's the syntax:

NTH_VALUE(column_name, N)
OVER (PARTITION BY column_name ORDER BY column_name)

Example:
If you want to find the 2nd sale amount for each product, you can use the following query:

SELECT sale_id, product_id, sale_date, amount,
NTH_VALUE(amount, 2) OVER (PARTITION BY product_id ORDER BY sale_date) AS second_sale_amount
FROM sales_data;

Here's the formatted output:

sale_id	product_id	sale_date	amount	second_sale_amount
1	101	2023-01-01	200	300
2	101	2023-01-02	300	300
3	102	2023-01-03	150	(NULL)

Explanation:

- For product 101, the 2nd sale amount is 300, so every row for this product shows 300.
- For product 102, there is no second sale, so it returns NULL.

HiveQL Case Study - Oct 21

Lecture Overview

In this lecture, we'll design a complete system using a case study on Uber Eats. We'll create a schema, identify important queries for analytics, and improve the queries where necessary.

Case Study

Uber Eats is a worldwide food delivery platform that connects customers with many restaurants and food outlets. As it grows, Uber Eats gathers a large amount of data about orders, customers, restaurants, delivery drivers, and promotions. The goal is to use this data to improve efficiency, increase customer satisfaction, and refine business strategies.

We'll begin by defining our facts and dimensions.

Dimension Tables

1. `dim_date`

```
CREATE TABLE dim_date (
    date_id INT COMMENT 'Date in YYYYMMDD format',
    full_date DATE COMMENT 'Full date',
    day TINYINT COMMENT 'Day of the month',
    month TINYINT COMMENT 'Month',
    year INT COMMENT 'Year',
    quarter TINYINT COMMENT 'Quarter',
    day_of_week TINYINT COMMENT 'Day of the week (1=Monday)',
    is_weekend BOOLEAN COMMENT 'Is weekend (true if Saturday or Sunday)'
)
COMMENT 'Date dimension table'
STORED AS PARQUET;
```

- **Storage Format:** PARQUET for efficient storage and faster reading.
- **Description:** Stores date-related attributes for time-based analysis, making it easier to analyze patterns over time.

---

2. `dim_customer`

```
CREATE TABLE dim_customer (
    customer_id BIGINT COMMENT 'Unique customer identifier',
    first_name STRING COMMENT 'First name of the customer',
    last_name STRING COMMENT 'Last name of the customer',
```

```
    email STRING COMMENT 'Email address',
    phone_number STRING COMMENT 'Phone number',
    gender STRING COMMENT 'Gender',
    date_of_birth DATE COMMENT 'Date of birth',
    registration_date DATE COMMENT 'Date the customer registered',
    city STRING COMMENT 'City of residence',
    loyalty_level STRING COMMENT 'Loyalty program level'
)
COMMENT 'Customer dimension table'
STORED AS PARQUET
CLUSTERED BY (city) INTO 8 BUCKETS;
```

- **Bucketing:** Clustered by `city` into 8 buckets to optimize city-based queries.
- **Storage Format:** PARQUET.
- **Description:** Contains customer profiles and demographics to analyze customer behavior and loyalty levels.

---

3. `dim_restaurant`

```
CREATE TABLE dim_restaurant (
    restaurant_id BIGINT COMMENT 'Unique restaurant identifier',
    restaurant_name STRING COMMENT 'Name of the restaurant',
    cuisine_type STRING COMMENT 'Type of cuisine',
    city STRING COMMENT 'City where the restaurant is located',
    rating FLOAT COMMENT 'Average customer rating',
    average_delivery_time INT COMMENT 'Average delivery time in minutes',
    operational_status STRING COMMENT 'Current operational status'
)
COMMENT 'Restaurant dimension table'
STORED AS PARQUET
CLUSTERED BY (city) INTO 8 BUCKETS;
```

- **Bucketing:** By `city` to optimize city-based queries and joins.
- **Storage Format:** PARQUET.
- **Description:** Stores information about partner restaurants, including ratings, cuisine type, and delivery times.

---

4. `dim_delivery_partner`

```
CREATE TABLE dim_delivery_partner (
    delivery_partner_id BIGINT COMMENT 'Unique delivery partner identifier',
    first_name STRING COMMENT 'First name of the delivery partner',
    last_name STRING COMMENT 'Last name of the delivery partner',
    vehicle_type STRING COMMENT 'Type of vehicle used',
    registration_date DATE COMMENT 'Date the partner registered',
```

```
    city STRING COMMENT 'City of operation',
    average_rating FLOAT COMMENT 'Average customer rating',
    total_deliveries INT COMMENT 'Total number of deliveries completed'
)
COMMENT 'Delivery Partner dimension table'
STORED AS PARQUET
CLUSTERED BY (city) INTO 8 BUCKETS;
```

- **Bucketing:** By `city` for efficient joins with city-based data.
- **Storage Format:** PARQUET.
- **Description:** Stores data on delivery personnel, such as vehicle type, average ratings, and total deliveries completed.

---

5. `dim_promotion`

```
CREATE TABLE dim_promotion (
    promo_code STRING COMMENT 'Promotion code',
    description STRING COMMENT 'Description of the promotion',
    start_date DATE COMMENT 'Promotion start date',
    end_date DATE COMMENT 'Promotion end date',
    discount_type STRING COMMENT 'Type of discount (e.g., percentage, fixed amount)',
    discount_value FLOAT COMMENT 'Value of the discount'
)
COMMENT 'Promotion dimension table'
STORED AS PARQUET;
```

- **Storage Format:** PARQUET.
- **Description:** Contains details about promotions and discounts, including start and end dates, type, and value of discounts.

---

6. `dim_menu_item`

```
CREATE TABLE dim_menu_item (
    menu_item_id BIGINT COMMENT 'Unique menu item identifier',
    restaurant_id BIGINT COMMENT 'Restaurant ID',
    item_name STRING COMMENT 'Name of the menu item',
    item_category STRING COMMENT 'Category of the menu item',
    price FLOAT COMMENT 'Price of the item',
    availability_status STRING COMMENT 'Availability status of the item'
)
COMMENT 'Menu Item dimension table'
STORED AS PARQUET
CLUSTERED BY (restaurant_id) INTO 32 BUCKETS;
```

- **Bucketing:** By `restaurant_id` into 32 buckets to optimize joins with `dim_restaurant`.

- **Storage Format:** PARQUET.
- **Description:** Contains details about menu items offered by restaurants, such as item category, price, and availability.

Summary

The dimension tables provide foundational details for analyzing time-based metrics, customer demographics, restaurant data, delivery partners, promotions, and menu items. These tables support efficient querying and join operations with fact tables, facilitating deeper insights into various aspects of the Uber Eats platform.

Fact Tables

Fact tables store measurable, quantitative data for analysis and are typically linked to dimension tables. In the case of Uber Eats, the following fact tables can capture core data about orders, payments, deliveries, and promotions.

1. fact_order_items

```
CREATE TABLE fact_orders (
    order_id BIGINT COMMENT 'Unique order identifier',
    customer_id BIGINT COMMENT 'Customer ID',
    restaurant_id BIGINT COMMENT 'Restaurant ID',
    delivery_partner_id BIGINT COMMENT 'Delivery Partner ID',
    promo_code STRING COMMENT 'Promotion code used',
    order_amount DECIMAL(10, 2) COMMENT 'Total amount of the order',
    delivery_fee DECIMAL(10, 2) COMMENT 'Delivery fee',
    tip_amount DECIMAL(10, 2) COMMENT 'Tip amount given',
    delivery_time INT COMMENT 'Delivery time in minutes',
    order_status STRING COMMENT 'Status of the order',
    total_items INT COMMENT 'Total number of distinct items in the order',
    total_quantity INT COMMENT 'Total quantity of all items in the order'
)
PARTITIONED BY (date_id INT COMMENT 'Date ID in YYYYMMDD format')
COMMENT 'Fact table for orders'
STORED AS PARQUET;
```

- **Partitioning**: By date_id to improve query performance on date-based queries.
- **Bucketing**: By order_id into 64 buckets to optimize joins on order_id.
- **Storage Format**: PARQUET
- **Description**: Records transactional data for each order.

---

2. fact_order_items

```
CREATE TABLE fact_order_items (
    order_id BIGINT COMMENT 'Order ID',
    menu_item_id BIGINT COMMENT 'Menu Item ID',
    quantity INT COMMENT 'Quantity ordered',
    item_price DECIMAL(10, 2) COMMENT 'Price of the item at time of order',
    special_instructions STRING COMMENT 'Special instructions for the item'
```

```
)
CLUSTERED BY (order_id) INTO 64 BUCKETS
COMMENT 'Fact table for order items'
STORED AS PARQUET;
```

- **Bucketing:** Clustered by `order_id` into 64 buckets to optimize joins with `fact_orders`.
- **Storage Format:** PARQUET for efficient storage and faster query performance.
- **Description:** Contains detailed information about each item in an order.

---

3. fact_customer_feedback

```
CREATE TABLE fact_customer_feedback (
    feedback_id BIGINT COMMENT 'Unique feedback identifier',
    order_id BIGINT COMMENT 'Order ID',
    customer_id BIGINT COMMENT 'Customer ID',
    rating INT COMMENT 'Rating given by the customer',
    comments STRING COMMENT 'Feedback comments',
    feedback_date DATE COMMENT 'Date of feedback'
)
PARTITIONED BY (feedback_date)
CLUSTERED BY (customer_id) INTO 32 BUCKETS
COMMENT 'Customer Feedback fact table'
STORED AS PARQUET;
```

- **Partitioning:** By `feedback_date` to facilitate time-based queries.
- **Bucketing:** Clustered by `customer_id` into 32 buckets to optimize joins and aggregations on customer data.
- **Storage Format:** PARQUET.
- **Description:** Stores customer feedback on orders.

---

4. fact_restaurant_feedback

```
CREATE TABLE fact_restaurant_feedback (
    feedback_id BIGINT COMMENT 'Unique feedback identifier',
    order_id BIGINT COMMENT 'Order ID',
    restaurant_id BIGINT COMMENT 'Restaurant ID',
    comments STRING COMMENT 'Feedback comments from the restaurant',
    resolution_status STRING COMMENT 'Status of feedback resolution',
    feedback_date DATE COMMENT 'Date of feedback'
)
PARTITIONED BY (feedback_date)
CLUSTERED BY (restaurant_id) INTO 32 BUCKETS
COMMENT 'Restaurant Feedback fact table'
STORED AS PARQUET;
```

- **Partitioning:** By `feedback_date` to support time-based analysis.

- **Bucketing:** Clustered by `restaurant_id` into 32 buckets to optimize queries involving restaurants.
- **Storage Format:** PARQUET.
- **Description:** Stores feedback from restaurants regarding orders.

Key Points about Fact Tables

- **Purpose:** Fact tables store large volumes of quantitative data related to business processes, such as orders, payments, deliveries, and feedback.
- **Indexes and Clustering:** Proper indexing and clustering (e.g., by `order_id`, `customer_id`, `restaurant_id`) enhance query performance, especially for frequent join operations.
- **Storage Format:** Using PARQUET format ensures efficient storage and faster data retrieval.
- **Partitioning and Bucketing:** These techniques improve query performance by organizing data logically and physically, making it easier to filter and join tables.

Summary

The fact tables designed for the Uber Eats case study capture essential transactional data, enabling comprehensive analytics and reporting. By leveraging partitioning, bucketing, and efficient storage formats, these tables ensure optimal performance and scalability for large-scale data processing.

Insert Data

The provided `INSERT` statements populate various tables with sample data for a database schema focused on restaurant orders, customers, promotions, and feedback. Here's a breakdown of the different parts:

1. Populate `dim_date` Table

```
INSERT INTO dim_date VALUES
(20220101, '2022-01-01', 1, 1, 2022, 1, 6, true),
(20220102, '2022-01-02', 2, 1, 2022, 1, 7, true),
(20220103, '2022-01-03', 3, 1, 2022, 1, 1, false),
(20220104, '2022-01-04', 4, 1, 2022, 1, 2, false),
(20220105, '2022-01-05', 5, 1, 2022, 1, 3, false),
(20220106, '2022-01-06', 6, 1, 2022, 1, 4, false),
(20220107, '2022-01-07', 7, 1, 2022, 1, 5, false),
(20220108, '2022-01-08', 8, 1, 2022, 1, 6, true),
(20220109, '2022-01-09', 9, 1, 2022, 1, 7, true);
```

Explanation:

- Provides date entries for the first nine days of January 2022.
- Covers weekdays and weekends to support time-based queries.

---

2. Populate `dim_customer` Table

```
INSERT INTO dim_customer VALUES
(1001, 'John', 'Doe', 'john.doe@example.com', '123-456-7890', 'Male', '1990-05-15', '2021-01-10', 'New York', 'Gold'),
(1002, 'Jane', 'Smith', 'jane.smith@example.com', '234-567-8901', 'Female', '1985-08-22', '2021-02-20', 'Los Angeles', 'Silver'),
(1003, 'Mike', 'Johnson', 'mike.johnson@example.com', '345-678-9012', 'Male', '1992-11-30', '2021-03-15', 'Chicago', 'Bronze'),
(1004, 'Emily', 'Davis', 'emily.davis@example.com', '456-789-0123', 'Female', '1988-07-07', '2021-04-25', 'Houston', 'Gold'),
(1005, 'David', 'Wilson', 'david.wilson@example.com', '567-890-1234', 'Male', '1995-02-28', '2021-05-30', 'Phoenix', 'Silver');
```

Explanation:

- Adds five customers from different cities with varying loyalty levels.
- Supports customer segmentation and loyalty program analysis.

---

3. Populate `dim_restaurant` Table

```
INSERT INTO dim_restaurant VALUES
(2001, 'Pizza Palace', 'Italian', 'New York', 4.5, 30, 'Open'),
(2002, 'Sushi World', 'Japanese', 'Los Angeles', 4.7, 25, 'Open'),
(2003, 'Burger Barn', 'American', 'Chicago', 4.2, 20, 'Open'),
(2004, 'Curry House', 'Indian', 'Houston', 4.6, 35, 'Open'),
(2005, 'Taco Town', 'Mexican', 'Phoenix', 4.3, 15, 'Open');
```

Explanation:

- Introduces five restaurants with different cuisines and locations.
- Enables analysis of restaurant performance and customer preferences.

---

4. Populate `dim_delivery_partner` Table

```
INSERT INTO dim_delivery_partner VALUES
(3001, 'Alex', 'Brown', 'Bike', '2020-06-15', 'New York', 4.8, 500),
(3002, 'Sam', 'Green', 'Car', '2020-07-20', 'Los Angeles', 4.6, 450),
(3003, 'Chris', 'White', 'Scooter', '2020-08-25', 'Chicago', 4.7, 400),
(3004, 'Pat', 'Black', 'Bike', '2020-09-30', 'Houston', 4.5, 550),
(3005, 'Taylor', 'Gray', 'Car', '2020-10-05', 'Phoenix', 4.9, 600);
```

Explanation:

- Provides delivery partners with various vehicle types and cities.
- Allows for performance analysis and comparisons.

---

5. Populate `dim_promotion` Table

```
INSERT INTO dim_promotion VALUES
('PROMO2021', 'New Year Promotion', '2022-01-01', '2022-01-10', 'Percentage', 10),
('SAVE5', 'Save $5 on orders over $50', '2022-01-05', '2022-01-15', 'Fixed', 5),
('FREEDelivery', 'Free Delivery Promo', '2022-01-01', '2022-01-31', 'Fixed', 0),
('WELCOME10', 'Welcome Discount', '2022-01-01', '2022-12-31', 'Percentage', 10),
('SUMMER20', 'Summer Special 20% Off', '2022-06-01', '2022-08-31', 'Percentage', 20);
```

Explanation:

- Adds various promotions with different discount types and durations.
- Supports analysis of promotion effectiveness.

---

6. Populate `dim_menu_item` Table

```
INSERT INTO dim_menu_item VALUES
(4001, 2001, 'Margherita Pizza', 'Pizza', 12.99, 'Available'),
(4002, 2001, 'Pepperoni Pizza', 'Pizza', 14.99, 'Available'),
(4003, 2002, 'California Roll', 'Sushi', 8.99, 'Available'),
(4004, 2002, 'Spicy Tuna Roll', 'Sushi', 9.99, 'Available'),
(4005, 2003, 'Classic Burger', 'Burger', 10.99, 'Available'),
(4006, 2003, 'Cheeseburger', 'Burger', 11.99, 'Available'),
(4007, 2004, 'Chicken Curry', 'Curry', 13.99, 'Available'),
(4008, 2004, 'Vegetable Curry', 'Curry', 11.99, 'Available'),
(4009, 2005, 'Beef Taco', 'Taco', 3.99, 'Available'),
(4010, 2005, 'Chicken Taco', 'Taco', 3.49, 'Available');
```

Explanation:

- Lists menu items for each restaurant.
- Enables detailed order item analysis.

---

7. Populate `fact_orders` Table

```
INSERT INTO fact_orders PARTITION (date_id) VALUES
-- Orders on 2022-01-01
(5001, 1001, 2001, 3001, 'PROMO2021', 25.98, 2.99, 3.00, 30, 'Completed', 2, 2, 20220101),
(5002, 1002, 2002, 3002, 'WELCOME10', 18.98, 2.99, 2.00, 25, 'Completed', 2, 2, 20220101),
-- Orders on 2022-01-02
(5003, 1003, 2003, 3003, NULL, 10.99, 2.99, 1.50, 20, 'Completed', 1, 1, 20220102),
(5004, 1004, 2004, 3004, 'FREEDelivery', 13.99, 0.00, 2.00, 35, 'Completed', 1, 1, 20220102),
-- Orders on 2022-01-03
(5005, 1005, 2005, 3005, 'SAVE5', 50.00, 2.99, 5.00, 15, 'Completed', 10, 10, 20220103),
(5006, 1001, 2001, 3001, 'PROMO2021', 25.98, 2.99, 3.00, 28, 'Completed', 2, 2, 20220103),
-- Orders on 2022-01-04
(5007, 1002, 2002, 3002, 'SUMMER20', 37.96, 2.99, 4.00, 24, 'Completed', 4, 4, 20220104),
(5008, 1003, 2003, 3003, NULL, 21.98, 2.99, 2.00, 22, 'Completed', 2, 2, 20220104),
-- Orders on 2022-01-05
```

(5009, 1004, 2004, 3004, 'PROMO2021', 19.98, 2.99, 1.50, 33, 'Completed', 2, 2, 20220105),
(5010, 1005, 2005, 3005, 'WELCOME10', 35.00, 2.99, 2.00, 27, 'Completed', 8, 8, 20220105);

Explanation:

- Inserts order records with sample promotions, delivery fees, and restaurant partnerships.
- Supports analysis of order patterns, promotions, and delivery performance.

Use Case 1: Calculate Monthly Average Delivery Time per City Using Window Functions

Business Problem:
Uber Eats wants to identify cities with delivery delays by calculating the average delivery time per city for each month.

HiveQL Solution:

```
SELECT
    city,
    month,
    AVG(delivery_time) OVER (PARTITION BY city, month) AS avg_delivery_time
FROM (
    SELECT
        r.city,
        MONTH(TO_DATE(CAST(f.date_id AS STRING), 'yyyyMMdd')) AS month,
        f.delivery_time
    FROM fact_orders f
    JOIN dim_restaurant r ON f.restaurant_id = r.restaurant_id
) sub;
```

Explanation:

- **Subquery:** Joins `fact_orders` and `dim_restaurant` to get city and delivery times.
- **MONTH Function:** Extracts the month from `date_id`.
- **Window Function:** `AVG(delivery_time)` calculates the average delivery time partitioned by city and month.

Use Case 2: Identify Top 10 Customers Using a UDF to Calculate Loyalty Score

Business Problem:
Classify customers into loyalty tiers based on a custom loyalty score considering total spend and order frequency.

HiveQL Solution:

Step 1: Create a UDF
Assuming we have a UDF `calculate_loyalty_score` defined in Java or Python.

```
-- Register the UDF
CREATE TEMPORARY FUNCTION calculate_loyalty_score AS
'com.example.hive.udf.CalculateLoyaltyScore';
```

Step 2: Use the UDF in a Query

```
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    calculate_loyalty_score(SUM(f.order_amount), COUNT(f.order_id)) AS
loyalty_score
FROM fact_orders f
JOIN dim_customer c ON f.customer_id = c.customer_id
GROUP BY c.customer_id, c.first_name, c.last_name
ORDER BY loyalty_score DESC
LIMIT 10;
```

Explanation:

- **UDF Usage:** Applies `calculate_loyalty_score` to total spend and order count per customer.
- **Aggregation:** Groups by customer to compute total spend and order count.
- **Ordering:** Ranks customers by loyalty score to identify the top 10.

Use Case 3: Analyze Promotion Effectiveness Using Temporary Tables

Business Problem:
Measure the impact of a specific promotion code on order volume and revenue.

HiveQL Solution:

```
-- Create a temporary table for orders with the specific promo code
CREATE TEMPORARY TABLE temp_promo_orders AS
SELECT *
FROM fact_orders
WHERE promo_code = 'PROMO2021';

-- Total orders and revenue with the promotion
SELECT
```

```
    COUNT(*) AS total_promo_orders,
    SUM(order_amount) AS total_promo_revenue
FROM temp_promo_orders;

-- Total orders and revenue without the promotion
SELECT
    COUNT(*) AS total_non_promo_orders,
    SUM(order_amount) AS total_non_promo_revenue
FROM fact_orders
WHERE promo_code != 'PROMO2021' OR promo_code IS NULL;
```

Explanation:

- **Temporary Table:** `temp_promo_orders` holds orders using `'PROMO2021'`.
- **Comparison:** Calculates and compares metrics for promo and non-promo orders.

Use Case 4: Create a View for Frequent Customers

Business Problem:
Create a view of customers who placed more than 10 orders in the last three months for targeted marketing.

HiveQL Solution:

```
-- Calculate the date three months ago
SET three_months_ago = DATE_SUB(CURRENT_DATE, 90);

-- Create the view
CREATE VIEW frequent_customers AS
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    COUNT(f.order_id) AS total_orders
FROM fact_orders f
JOIN dim_customer c ON f.customer_id = c.customer_id
WHERE TO_DATE(CAST(f.date_id AS STRING), 'yyyyMMdd') >=
'${hiveconf:three_months_ago}'
GROUP BY c.customer_id, c.first_name, c.last_name
HAVING COUNT(f.order_id) > 10;
```

Explanation:

- **Variable:** Uses `SET` to define a date variable for three months ago.
- **View:** `frequent_customers` lists customers exceeding 10 orders since that date.
- **HAVING Clause:** Filters customers based on order count

Use Case 5: Compute Rolling Average Order Amount Using Window Frames

Business Problem: Calculate the 7-day rolling average of order amounts for each city to identify short-term trends.

HiveQL Solution:

```
SELECT
    city,
    order_date,
    AVG(order_amount) OVER (
        PARTITION BY city
        ORDER BY order_date
        RANGE BETWEEN 6 PRECEDING AND CURRENT ROW
    ) AS rolling_avg_order_amount
FROM (
    SELECT
        r.city,
        TO_DATE(CAST(f.date_id AS STRING), 'yyyyMMdd') AS order_date,
        f.order_amount
    FROM fact_orders f
    JOIN dim_restaurant r ON f.restaurant_id = r.restaurant_id
) sub;
```

Explanation:

- **Window Frame:** Computes the average over current and previous 6 days.
- **Partitioning:** By city to get city-specific trends.

Use Case 6: Use SET Configurations to Optimize Query Execution

Business Problem: Optimize a resource-intensive query to prevent exceeding cluster resources.

HiveQL Solution:

```
SET hive.exec.reducers.max = 50;
SET hive.execution.engine = tez;

SELECT
    r.city,
    SUM(f.order_amount) AS total_revenue
FROM fact_orders f
JOIN dim_restaurant r ON f.restaurant_id = r.restaurant_id
GROUP BY r.city;
```

Explanation:

- **Reducers Limit:** Prevents overconsumption of cluster resources.
- **Execution Engine:** Switches to Tez for better performance.

Use Case 7: Sample Data Using Bucketing and TABLESAMPLE

Business Problem: Analyze a sample of orders to test a new analytical model without processing the entire dataset.

HiveQL Solution:

```
SELECT * FROM fact_orders TABLESAMPLE(BUCKET 1 OUT OF 10 ON order_id);
```

Explanation:

- **TABLESAMPLE:** Efficiently samples data using the bucketing of `order_id`.
- **Purpose:** Reduces data volume for testing.

Use Case 8: Identify Delivery Partners with Above-Average Ratings Using CTEs

Business Problem: List delivery partners in each city whose average rating is above the city average.

HiveQL Solution:

```
WITH city_avg_ratings AS (
    SELECT
        city,
        AVG(average_rating) AS city_avg_rating
    FROM dim_delivery_partner
    GROUP BY city
)
SELECT
    d.delivery_partner_id,
    d.first_name,
    d.last_name,
    d.city,
    d.average_rating
FROM dim_delivery_partner d
JOIN city_avg_ratings c ON d.city = c.city
WHERE d.average_rating > c.city_avg_rating;
```

Explanation:

- **CTE:** `city_avg_ratings` calculates the average rating per city.
- **Comparison:** Filters delivery partners exceeding city averages.

Use Case 9: Calculate Customer Lifetime Value Using a View and Window Functions

Business Problem: Calculate the lifetime value (LTV) of customers for strategic decision-making.

HiveQL Solution:

Create a View:

```
CREATE VIEW customer_orders AS
SELECT
    customer_id,
    SUM(order_amount) AS total_spent,
    COUNT(order_id) AS total_orders,
    MIN(TO_DATE(CAST(date_id AS STRING), 'yyyyMMdd')) AS first_order_date,
    MAX(TO_DATE(CAST(date_id AS STRING), 'yyyyMMdd')) AS last_order_date
FROM fact_orders
GROUP BY customer_id;
```

1. **Calculate LTV:**

```
SELECT
c.customer_id,
c.first_name,
c.last_name,
co.total_spent,
DATEDIFF(co.last_order_date, co.first_order_date) AS customer_lifetime_days,
co.total_spent / co.total_orders AS average_order_value
FROM customer_orders co
JOIN dim_customer c ON co.customer_id = c.customer_id;
```

Explanation:

- **View:** Aggregates key metrics per customer.
- **Calculations:** Computes lifetime, average order value, and total spend.

Use Case 10: Use Window Functions to Rank Restaurants by Revenue

Business Problem: Rank restaurants within each city based on total revenue to identify top performers.

HiveQL Solution:

```
SELECT
    city,
    restaurant_name,
    total_revenue,
    RANK() OVER (PARTITION BY city ORDER BY total_revenue DESC) AS revenue_rank
FROM (
    SELECT
        r.city,
        r.restaurant_name,
        SUM(f.order_amount) AS total_revenue
```

```
    FROM fact_orders f
    JOIN dim_restaurant r ON f.restaurant_id = r.restaurant_id
    GROUP BY r.city, r.restaurant_name
) sub
ORDER BY city, revenue_rank;
```

Explanation:

- **Subquery:** Calculates total revenue per restaurant.
- **Window Function:** Ranks restaurants within each city.

Use Case 11: Analyze Order Patterns Using Bucketing and Partitioning

Business Problem: Retrieve orders for a specific date and customer efficiently to analyze order patterns.

HiveQL Solution:

```
SELECT * FROM fact_orders WHERE date_id = 20220101 AND customer_id = 12345;
```

Explanation:

- **Partition Pruning:** Filters by `date_id` using partitioning.
- **Bucketing:** Optimizes query by bucketing on `order_id`.

Use Case 12: Use a Temporary Table to Stage Data for Analysis

Business Problem: Stage recent data to run quick tests without impacting the primary table.

HiveQL Solution:

```
CREATE TEMPORARY TABLE recent_orders AS
SELECT * FROM fact_orders WHERE date_id >= 20230101;
```

Explanation:

- **Temporary Table:** Holds recent data for quick analysis.
- **Purpose:** Avoids altering main table for testing.

Map Reduce Framework - Oct 23

Now we have built a good understanding of Hive, let's deep dive into the real brain i.e. execution engines that are helping to make Hive relevant for Big data ecosystems.

Hive mainly uses three execution engines

1. MR
2. Tez
3. Spark

You can set the execution engine which needs to be used by using this statement

SET hive.execution.engine= <mr/tez/spark>;

Lets first start by discussing MR i.e MapReduce

Map Reduce is the framework that has revolutionized the whole Big data ecosystem. It was among the first execution engines to be used by Hive. Even though in today's market it's not widely used, other execution engines that are a lot faster than MapReduce are built on fundamentals of MapReduce Framework only. So it's very necessary to understand the essence of MapReduce Framework.

When you run a HiveQL query, behind the scenes, Hive translates these SQL-like queries into MapReduce jobs. So, while you're using Hive to interact with your data in a high-level, SQL-like way, MapReduce does the heavy lifting. The job of MapReduce is to execute these data transformations and aggregations in a distributed way, across a large cluster of machines.

So ok, we now know MR is important, but what does MR do that make it so special?
To understand that, let's start with an example.

Scenario:

You are a data engineer working for a large retail company that operates across several regions in India. The company stores its sales data in Hadoop's HDFS. Each record in the dataset represents a sale, including details like the product sold, the region, the sales amount, and the date of sale. The company has been collecting this data for over 5 years, resulting in a dataset that now contains 10 billion records (several terabytes of data).

You are tasked with finding the total sales per region for the last year, which requires scanning the entire dataset and aggregating the sales figures based on the region.

Challenge:

Handling such large datasets can't be done effectively on a single machine due to resource constraints (memory, CPU, storage). Additionally, the data is stored in HDFS, distributed across hundreds of nodes in the Hadoop cluster.

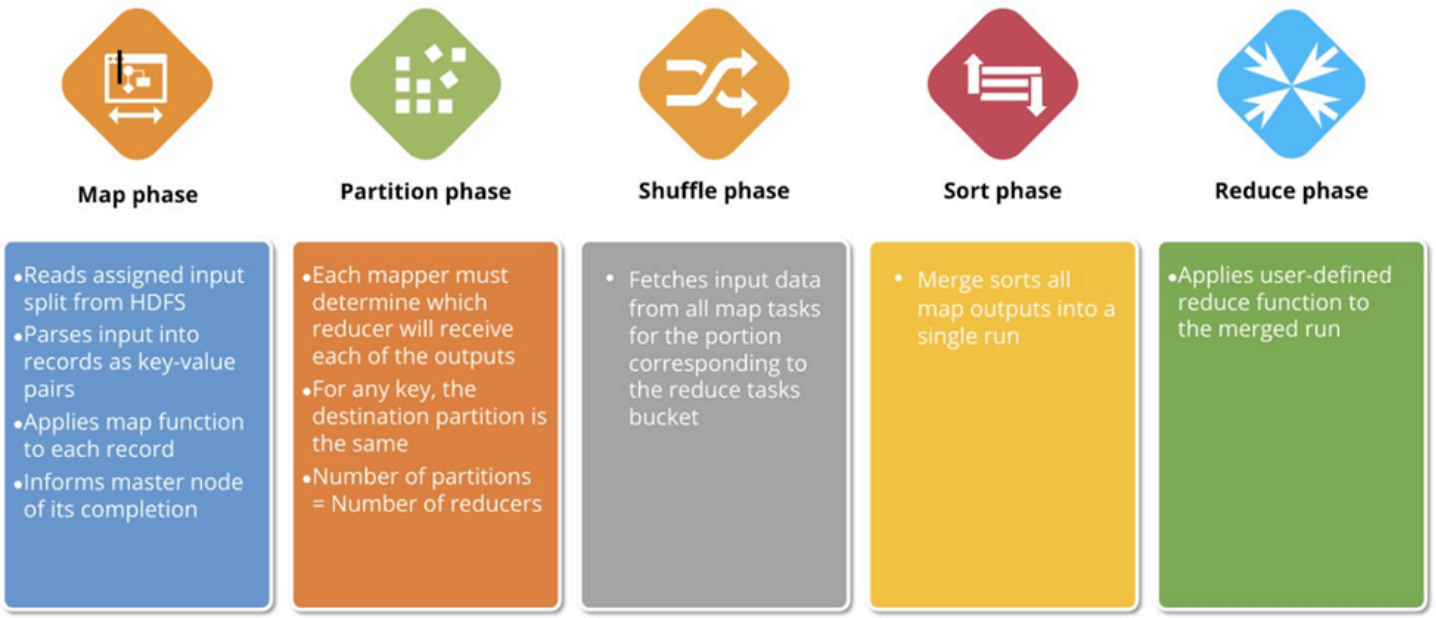
Without MapReduce: A Sequential Approach

Suppose you decide to query this data using Hive without MapReduce or any other distributed execution engine. Here's what would happen:

1. Data Fetching: The entire dataset (10 billion records) would need to be fetched from HDFS and brought to a single node for processing.
2. Memory Limitations: The data is too large to fit in the memory of the single node. Processing might spill over to disk, causing severe performance degradation, or the job may fail entirely if memory is exhausted.
3. Manual Data Distribution: To distribute the load, you would have to manually divide the dataset across multiple machines and implement custom code for parallel processing, which would be extremely complex and error-prone.
4. No Fault Tolerance: If the node processing the data crashes or encounters an error, the entire job would fail. You'd need to restart the process manually and figure out how to recover the partial results.
5. Network Bottlenecks: Since the data is stored in HDFS, you'd need to move large chunks of data across the network to the node doing the processing, adding significant overhead due to data transfer.

This approach is inefficient, and in most cases, simply not feasible for handling large datasets. So lets now see, how a MR framework will solve this problem

With MapReduce:
A typical MR job is divided in to 5 stages, lets see how each stage contributes in computing the result for the given query



1. Map Phase – "Dividing the Work"

Imagine you have millions of sales records in a huge warehouse. Each record contains two important details:

- Region: The part of the country where the sale happened (like North, South, East, West).
- Sales Amount: How much money was made in that sale.

You can’t count all the sales by yourself, so you hire 100 friends to help. The first thing you do is divide the massive pile of records into small piles and give each person one pile to work on. Each friend will look through their small pile of sales records and count how much money was made in each region.

This is exactly what happens in the Map Phase of MapReduce:

- The data (sales records) is split into small parts, and each part is given to a different computer (just like giving each friend a pile of records).
- Each computer looks at its part of the data and creates key-value pairs:
  ○ The key is the region (e.g., North, South). ○ The value is the sales amount.

Example:

Let’s say one computer is processing these three sales records:

- Record 1: {Region: North, Sales: ₹500}
- Record 2: {Region: South, Sales: ₹300}
- Record 3: {Region: North, Sales: ₹200}

That computer will turn the records into simple pairs like this:

- (North, ₹500)
- (South, ₹300)
- (North, ₹200)

These pairs tell us how much money was made in each region for that small chunk of data. Every computer does this for its part of the data.

2. Partition Phase – "Grouping Work for Later"

Now that each friend (or computer) has processed their pile of records and counted sales by region, the next step is to group all the data by region.

For example, we want all the sales data for the North region to end up in one place, all the sales data for the South region in another place, and so on. This way, when it’s time to add everything up, we know that all the data for each region is together.

In the Partition Phase, the system decides which computer will handle which region, and groups all the sales data by region.

Example:

Let’s say two different computers produced these results:

- Computer 1: (North, ₹500), (South, ₹300)
- Computer 2: (North, ₹200), (South, ₹100)

Now, in the Partition Phase, the data is grouped so that:

- All the sales amounts for North are put together: (North, ₹500), (North, ₹200) ● All the sales amounts for South are put together: (South, ₹300), (South, ₹100)

3. Shuffle Phase – "Sending Data to the Right Places"

Once the data is grouped by region, it needs to be sent to the right computers that will add up the sales totals for each region. This is called the Shuffle Phase.

During this phase, the system moves the data around to make sure that all the information about a specific region (like `North`) is sent to the same computer.

This is like collecting all the piles of sales data from different friends, then bringing them to one person who is in charge of counting the sales for the North region, another person for the South region, and so on.

Example:

- Computer 1 might have found some sales for the North and South regions. ● Computer 2 also found sales for the North and South regions.

During the shuffle phase, all the sales data for `North` is sent to one place (a computer that will handle `North`), and all the sales data for `South` is sent to another place (a computer that will handle `South`).

After this phase:

- One computer might have all the pairs for North: (North, ₹500), (North, ₹200).
- Another computer has all the pairs for South: (South, ₹300), (South, ₹100).

4. Sort Phase – "Organizing the Data"

Before the next step (where we add everything up), the system sorts the data so that it's neatly organized by region. This makes it easier and faster to process.

This is like making sure all the sales records for North are grouped together in one stack, all the records for South in another stack, and so on.

Example: For the North region, the data might look like this after sorting:

- (North, ₹200) ● (North, ₹500)

The same happens for the other regions.

5. Reduce Phase – "Adding It All Up"

Now comes the final step, the Reduce Phase. This is where the computers take all the sales numbers for each region and add them up to get the total sales for that region.

- The computer that has all the data for the North region will add up all the sales amounts.
- The computer that has all the data for the South region will do the same for South.

Example:

For the North region:

- (North, ₹200) ● (North, ₹500)

The computer handling `North` will add up the sales: ₹200 + ₹500 = ₹700.

For the South region:

- (South, ₹300)
- (South, ₹100)

The computer handling `South` will add up the sales: ₹300 + ₹100 = ₹400.

Final Result: After all the computers finish adding up the sales for each region, the system combines the results and gives you the final total:

Region	Total Sales
North	₹700
South	₹400

This means:

- The North region made ₹700 in sales.
- The South region made ₹400 in sales.

Now lets see , how it actually looks like when Hive and HDFS come into play

1. Data Stored in Parquet Format in HDFS

Your sales data is stored in HDFS using the Parquet format. This columnar format organizes the data by columns rather than rows, which makes reading specific columns (like `region` and `sales_amount`) faster and more efficient.

For example, your Parquet file might be located here:

```
/hdfs/sales_data/2023_sales.parquet
```

Unlike a row-based format (like CSV), Parquet stores each column separately, which means it reads only the columns needed for your query, skipping over the others.
This reduces the amount of data transferred and processed during query execution.

Let's assume this file contains data like this:

```
Region     Sales_Amount Date
North      500          2023-01-01
South      300          2023-01-02
North      200          2023-01-03
...
```

2. Hive Query Execution

When you submit a Hive query, such as calculating the total sales per region, Hive automatically optimizes the query to take advantage of the Parquet format's efficiency.

Example Hive Query:

```
SELECT region, SUM(sales_amount)
FROM sales_data
WHERE date BETWEEN '2023-01-01' AND '2023-12-31' GROUP BY
region;
```

In this query, Hive will instruct MapReduce to read only the `region` and `sales_amount` columns, avoiding unnecessary reads of other columns like `date`. This selective column reading is one of the key benefits of using Parquet.

3. Input Format in MapReduce (Parquet)

When the MapReduce job begins, the system needs to know how to read the Parquet file. This is where ParquetInputFormat comes into play.

- ParquetInputFormat is designed to read Parquet files and ensures that only the required columns (in this case, `region` and `sales_amount`) are read from HDFS, skipping the `date` column or any other irrelevant data.

Since the Parquet file is stored in blocks in HDFS, the input format will divide the data into splits. Each split corresponds to a subset of the file, typically aligned with the HDFS block size. For instance, if the file is 1 GB and the HDFS block size is 128 MB, the file will be split into multiple chunks, each handled by a different Mapper.

4. Map Phase (Parquet)

Once each Mapper is assigned a split of the Parquet file, it begins processing the data. Thanks to ParquetInputFormat, the Mapper will read only the relevant columns (`region` and `sales_amount`) from its assigned split.

During the Map Phase, the mapper extracts the relevant key-value pairs. The key represents the `region`, and the value represents the `sales_amount`.

Example Mapper Input:

Let's say a Mapper processes part of the Parquet file with the following data:

```
Region     Sales_Amount
North      500
South      300
North      200
```

The Mapper will produce key-value pairs like this:

- Key: `North`, Value: `500`
- Key: `South`, Value: `300`
- Key: `North`, Value: `200`

This is how the Mapper prepares data for further processing, focusing only on the columns needed for the query.

5. Partition Phase (Parquet)

After the Map Phase, the system must ensure that all records related to the same key (`region`) are grouped together for further processing. This is done during the Partition Phase.

Each Mapper's output is split into partitions based on the key (region). All the records for the `North` region are directed to one partition, while records for the `South` region are directed to another. This ensures that when the reducers take over, they receive all the data related to a specific region.

Example:

- Partition 1 will contain all records for the `North` region.
- Partition 2 will contain all records for the `South` region.

The partitions prepare the data for the next steps, ensuring efficient processing.

6. Shuffle and Sort Phase (Parquet)

Once the data is partitioned, it needs to be transferred to the appropriate reducers, where the actual aggregation will take place. This happens during the Shuffle Phase.

- Shuffle: During this stage, the key-value pairs generated by the Mappers are moved across the network to the appropriate Reducer. For example, all the data for `North` is sent to one reducer, and all the data for `South` is sent to another.
- Sort: While transferring the data, MapReduce also sorts the key-value pairs by the key (`region`). Sorting ensures that when the Reducer processes the data, it can handle it in an ordered manner.

Example:

- Reducer 1 will receive all records for the `North` region, sorted and grouped together.
- Reducer 2 will receive all records for the `South` region.

7. Reduce Phase (Parquet)

During the Reduce Phase, each Reducer processes the key-value pairs assigned to it. Since all the key-value pairs for a particular region are grouped together, the Reducer can now aggregate the data.

In this case, the aggregation task is to calculate the total sales for each region. The Reducer sums the `sales_amount` values for each region.

Example:

For the `North` region, Reducer 1 will receive:

- Key: `North`, Values: [500, 200] The Reducer will sum these values to get:
- Total for `North`: 500 + 200 = 700.

Similarly, Reducer 2 will sum the values for the `South` region:

- Total for `South`: 300 + 100 = 400.

8. Output Format (Parquet)

Once the Reduce Phase is complete, the results need to be written back to HDFS. Since you're working with Parquet data, it's beneficial to write the output back in Parquet format for future use. Hive uses ParquetOutputFormat to handle this.

- ParquetOutputFormat ensures that the results are written back to HDFS in Parquet format, stored column by column. This keeps the storage efficient and ready for further queries.

Example Output:

The output will be written to a Parquet file in HDFS, like:

```
/hdfs/sales_output/2023_sales_totals.parquet
```

This file will contain the final aggregated results:

```
Region     | Total_Sales
-----------|--------------
North      | 700
South      | 400
```

9. Hive and the Final Result

After the MapReduce job finishes, Hive gathers the results from the Parquet output file and presents them according to your HiveQL query. The final results can be stored in a Hive table or output file, making them available for further analysis.

Now, we are not going to deep dive into how to write MapReduce code as its not relevant in today's world, learning it is same as learning languages like COBOL. That being said, its very necessary to understand how MR works to understand future execution engines as all other engines are built on top of MR core concepts.

Map reduce applications is not just limited to HiveQL, but is way beyond that. Lets take a look at various applications of MapReduce.

Key Use Cases of MapReduce Beyond HiveQL

Use Case	Description	Example
ETL Processing	Extract, transform, and load large datasets across systems.	Data cleaning and transformation in big data workflows.
Log Analysis	Parse and analyze logs for insights, patterns, or error detection.	Analyzing web server logs for user behavior.
Indexing and Searching	Build inverted indexes for search engines.	Creating a search index for documents or web pages.

Recommendation Systems	Build large-scale recommendation engines using collaborative filtering algorithms.	Netflix movie recommendations.

Data Mining/ML	Implement machine learning algorithms at scale.	Running K-means clustering on large datasets.
Graph Processing	Process large-scale graph data for algorithms like PageRank.	Google's PageRank algorithm.
Distributed Sorting	Sort massive datasets in parallel across multiple nodes.	Sorting customer transaction records.
Bioinformatics	Process large biological datasets for genome sequencing or DNA alignment.	Aligning DNA sequences.
Natural Language Processing	Perform feature extraction, word counts, and sentiment analysis on large text datasets.	Counting words across millions of documents.
Join Operations	Perform distributed joins between large datasets, similar to SQL joins.	Joining customer and transaction datasets.
Data Aggregation	Summarize and aggregate large datasets.	Summing total sales by region.
Fraud Detection	Detect anomalies or patterns in large transaction datasets for fraud detection.	Identifying suspicious credit card transactions.

Fault Tolerance in MapReduce

Fault tolerance in MapReduce ensures that the system can recover from failures without losing data or compromising the results of the job. Here's how MapReduce handles different types of failures:

1. Task Failure:

- **Detection:** The JobTracker monitors the progress of tasks running on different nodes. If a task fails (e.g., due to hardware failure or software issues), the JobTracker detects this failure based on heartbeat signals or timeouts.
- **Retry Mechanism:** The JobTracker will reassign the failed task to another available node in the cluster. This is done based on the same input splits, ensuring that the task is executed from scratch.

- **Speculative Execution:** To further improve reliability, MapReduce can perform speculative execution. If a task is running slower than expected, a duplicate task is scheduled to run on another node. The results of the first task to complete are used, mitigating the impact of slow or faulty tasks.

2. Node Failure:

- **Detection:** The JobTracker continuously monitors the status of all nodes. If a node fails, the JobTracker will detect the failure and reassign tasks from the failed node to other nodes in the cluster.
- **Data Replication:** HDFS, the underlying storage system used by MapReduce, replicates data across multiple nodes. If a node fails, the data stored on that node is still available from other replicas. This ensures that the MapReduce job can continue processing without data loss.

3. Data Corruption:

- **Detection:** During the Map Phase, data is read from HDFS blocks. If data corruption is detected, such as reading malformed data or encountering errors, the system will identify the issue through checksums and error reports.
- **Recovery:** The corrupted data can be read from a replica of the HDFS block, ensuring that the processing continues smoothly. The system will use the replica to retry the failed reads.

4. Network Issues:

- **Handling Network Partitions:** If there are network partitions or connectivity issues between nodes, MapReduce jobs can handle these by retrying communications and redistributing tasks. The JobTracker will ensure that tasks are reassigned if necessary.
- **Retry Mechanism:** Communication failures between nodes are retried automatically by the system. Data transfer and task execution retries help ensure that network issues do not impact the overall job execution.

One of the most important concepts we need to understand in MapReduce (MR) or any other engine that we are going to learn is DAG. Let's try to understand what a DAG is.

Introduction to DAGs

In big data processing, such as MapReduce or Tez, understanding how tasks are organized and executed is crucial. One key concept in this organization is the Directed Acyclic Graph (DAG). Let's break down what a DAG is and how it relates to MapReduce in a way that's easy to grasp.

What is a DAG?

- Directed: This means the graph shows a specific direction from one node (or task) to another. Arrows indicate this direction, showing how tasks depend on each other.
- Acyclic: This means the graph doesn't contain any loops or cycles. You can't start at one task and follow arrows to come back to the same task.

In simple terms, a DAG is like a flowchart that shows how different steps in a process are connected and in what order they should be executed. Each step is represented by a node, and the arrows represent the flow of data or dependencies between these steps.

DAG in MapReduce

In MapReduce, the job execution process can be visualized using a DAG. Here's how it fits into the MapReduce framework:

1. Map Phase:
   - Nodes: Represent the individual map tasks that process chunks of data. ○ Arrows: Indicate the flow of data from the input to these map tasks.
2. Shuffle and Sort Phases:
   - Nodes: Represent the shuffle and sort operations that organize the output from the map tasks.
   - Arrows: Show how the output from the map tasks is transferred and organized for the next phase.
3. Reduce Phase:
   - Nodes: Represent the reduce tasks that aggregate and process the organized data.
   - Arrows: Indicate the flow of data from the shuffle phase to the reduce tasks.

Example Scenario

Task: Counting the Number of Words in a Large Set of Documents (Represented as a DAG)

1. Map Phase:

- **Node 1:** Process Document 1 → Emit (word, count) pairs.
- **Node 2:** Process Document 2 → Emit (word, count) pairs.
- **Arrows:** Show data flowing from documents to map tasks.

2. Shuffle and Sort Phase:

- **Node 3:** Aggregate (word, count) pairs from different map tasks.
- **Arrows:** Show the data being shuffled and sorted by word.

3. Reduce Phase:

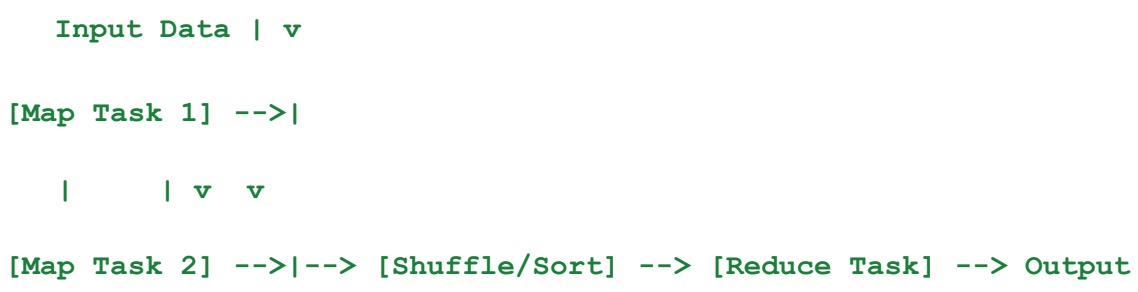
- **Node 4:** Sum counts for each word to get the final word count.

Why DAGs are Important

- Dependency Management: DAGs help visualize and manage dependencies between tasks. For example, the reduce tasks depend on the output of the map tasks. A DAG clearly shows this dependency, ensuring tasks are executed in the correct order.
- Error Handling: If a task fails, the DAG helps identify which dependent tasks need to be re-executed. This ensures that errors are handled without affecting the entire process.
- Optimization: DAGs provide a clear view of the execution plan, which helps in optimizing the process. For example, tasks can be scheduled in parallel where possible, reducing overall processing time.

Visualizing a DAG

Here's a simple visualization of a DAG for a Sample job (not necessarily a Map Reduce one):



```
    |     | v   |

  [Map Task 3] -->|
```

In this DAG:

- The Input Data flows into the Map Tasks.
- The output of Map Tasks is shuffled and sorted.
- The sorted data is then fed into the Reduce Task.
- The final Output is generated after the reduce phase

Now Map Reduce doesn't follow the complex DAG structure, but wouldn't it be great if we could create a DAG like this for any complex job and execute the jobs that can be executed in parallel to execute them in parallel to speed up the execution? That is what other execution engines do better than MR.

Tez and Hive on Spark - Oct 25

Apache Tez and Hive on Spark

In the last lecture, we learned about **MapReduce**. While MapReduce is powerful, other execution engines are often more efficient and widely used today. In this class, we'll explore these improved engines.

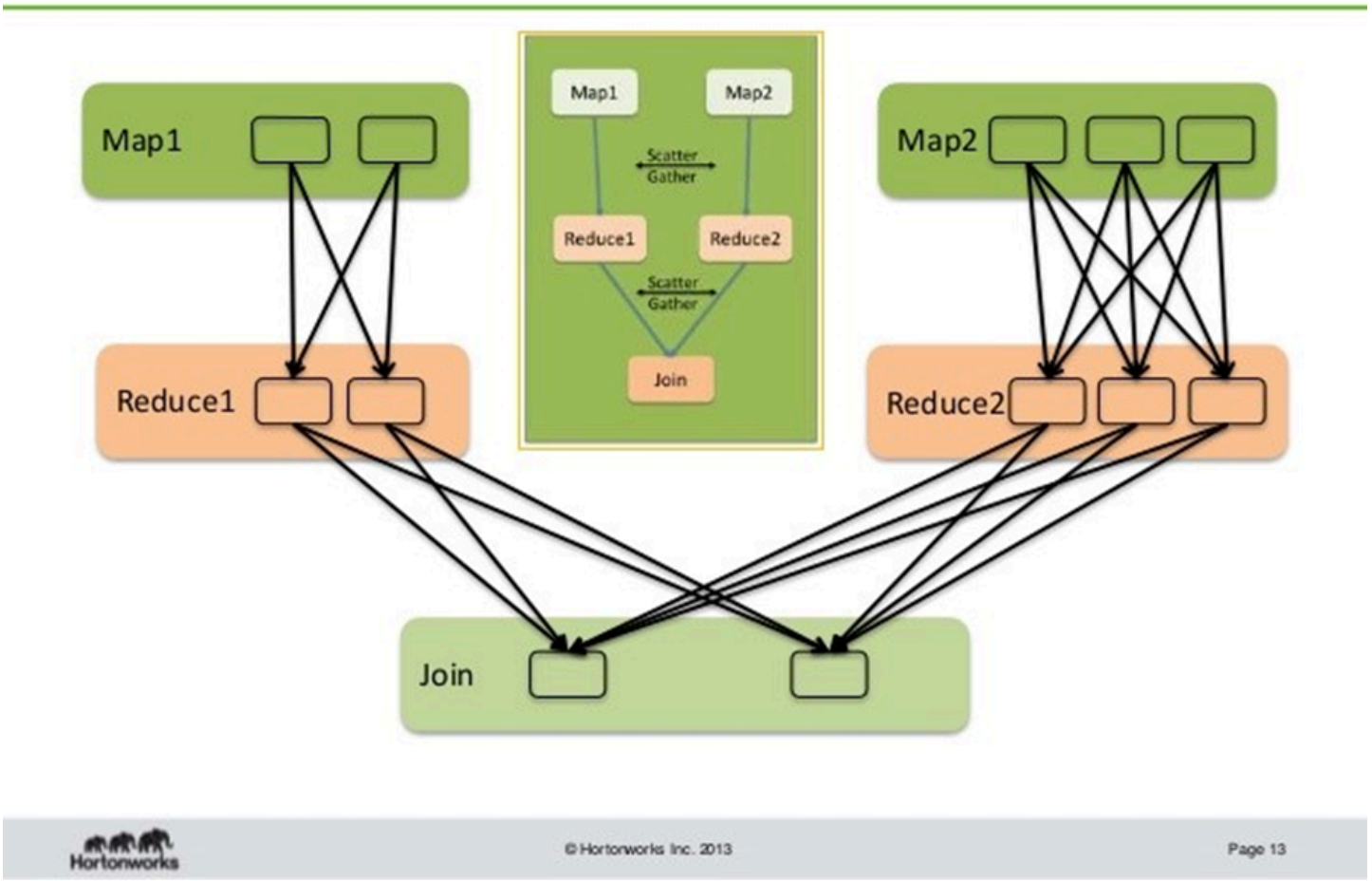
Apache Tez

MapReduce, despite its strengths, has some limitations:

- **High Startup Latency**: MapReduce follows a fixed two-stage model, always requiring a map stage followed by a reduce stage. This creates delays, especially when tasks don't need both stages.
- **Excessive Disk I/O**: Data between stages in MapReduce is written to disk, even for intermediate results that are only temporarily needed. This extra disk input/output (I/O) slows down the entire process.

Apache Tez was developed to address these challenges, providing a more efficient execution engine. Let's dive into **Tez's architecture**, **features**, and how it improves data processing.

Architecture Components



A. **Directed Acyclic Graph (DAG)**

- **Tez**: In Tez, jobs are set up as a DAG, where each node (called a vertex) represents a task, like a map or reduce task. The lines connecting nodes (edges) show data flow, allowing some tasks to run at the same time if their data is ready. The DAG Manager organizes the flow of these tasks, handles scheduling, and manages dependencies.
- **MapReduce**: Jobs run in a fixed sequence: map tasks go first, then reduce tasks. This limits parallel processing, making it less efficient for complex tasks.

B. **Vertex**

- **Tez**: Each vertex in Tez is a specific task that can run alongside others to speed things up. The Vertex Manager schedules these tasks and manages resources within each vertex.
- **MapReduce**: MapReduce jobs follow a set structure, with map and reduce phases separated. Each phase must finish before the next starts, which limits the flexibility for custom processing within a single job.

C. **Edges**

- **Tez**: Edges (connections between vertices) can either hold tasks back or let them start early as soon as data is ready. This flexibility improves processing speed.
- **MapReduce**: Data is saved to disk between map and reduce stages, adding time and resource load. Reduce tasks must wait until all map tasks finish before starting.

D. **Tez Application Master**

- **Tez**: The Tez Application Master supervise the job's execution, coordinating the DAG Manager, Vertex Managers, Task Executors, and YARN (resource manager).
    - **Resource Management**: Requests resources from YARN to run the DAG.
    - **Job Monitoring**: Tracks the progress of the DAG, vertices, and tasks.
    - **Failure Handling**: If tasks fail, it can restart or reschedule them.
- **MapReduce**: The JobTracker manages all map and reduce tasks' resources, but in large jobs, it can become a bottleneck.

E. **Execution Engine**

- **Tez**: Uses YARN to allocate resources for each stage, which helps make better use of resources across job stages.
- **MapReduce**: Also uses YARN, but resources are assigned for the entire job upfront, which can result in unused resources during multi-stage jobs.

Data Flow in Tez

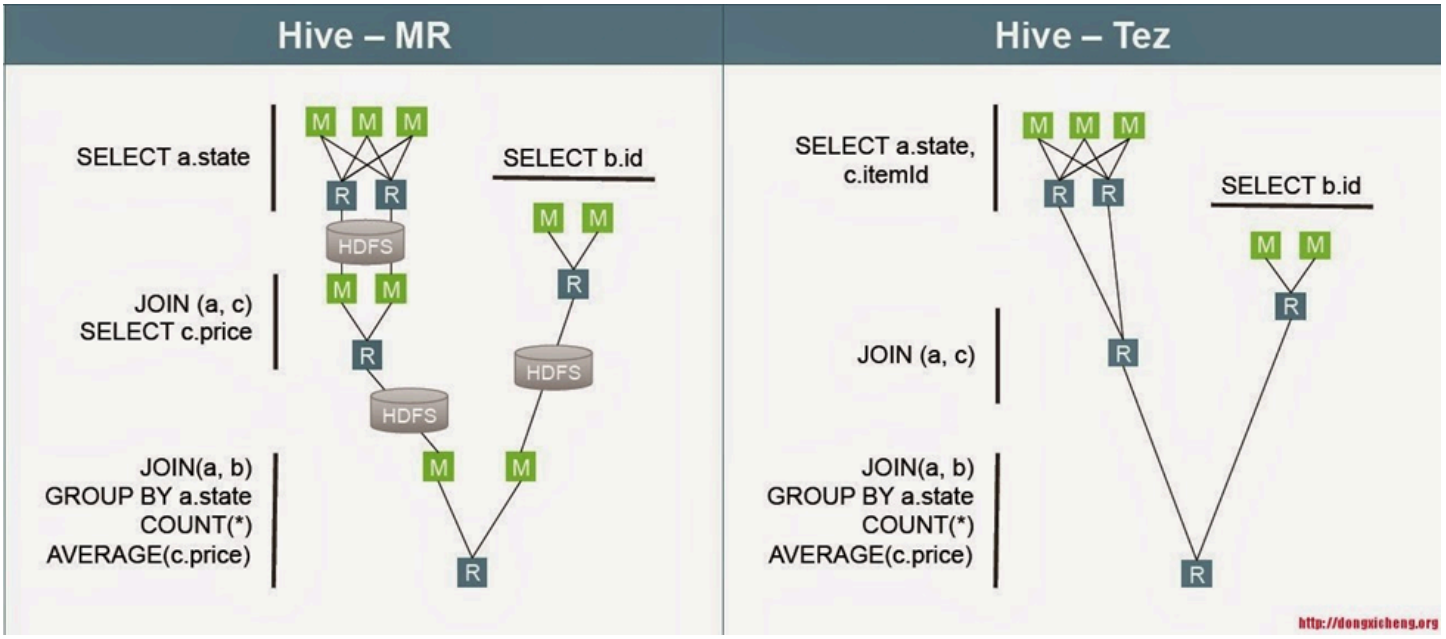
A. Input and Output Formats

- **Tez**: Supports a wide range of input/output formats, making it adaptable to various data sources. Custom formats can be defined easily.
- **MapReduce**: Uses predefined formats, making customization more complex and requiring additional coding.

B. Data Serialization

- **Tez**: Utilizes efficient serialization (binaries) methods (e.g., Apache Avro, Protocol Buffers) to reduce data transfer overhead.
- **MapReduce**: Although it supports serialization, MapReduce's reliance on disk writes for intermediates increases overhead and latency.

C. Buffer Management



- **Tez**: Employs in-memory buffers for passing data between tasks, significantly reducing disk I/O needs.
- **MapReduce**: Heavily relies on disk storage for intermediate data, resulting in slower performance due to frequent read/write operations.

Execution Model

A. Job Submission

- **Tez**: Users submit a DAG-defined job, offering greater workflow flexibility.
- **MapReduce**: Jobs follow a fixed map-reduce task sequence, limiting workflow complexity.

B. Resource Allocation

- **Tez**: The Tez Application Master dynamically allocates resources per vertex, optimizing resource use based on real-time needs.
- **MapReduce**: Resources are allocated at the job level, often leading to resource under-utilization or over-provisioning.

C. DAG Execution

- **Tez**: Vertices are scheduled for concurrent execution as dependencies are resolved, minimizing idle time and maximizing resource use.
- **MapReduce**: Execution is sequential; no reduce tasks start until all map tasks finish, often leading to wasted resources during idle periods.

D. Handling Failures

- **Tez**: Robust failure recovery mechanisms allow retrying or rescheduling failed vertices based on configurations, increasing resilience.
- **MapReduce**: Supports retries, but its fixed structure can lead to longer recovery times, especially in complex jobs.

5. Performance Optimization

Category	**MR**	**Tez**
Latency	Reliance on intermediate disk storage leads to increased I/O overhead and slower execution times, especially for complex data processing.	By allowing direct communication between vertices and reducing disk writes, Tez enhances data transfer efficiency and reduces latency.
Pipeline Execution	MR's linear execution model does not support pipelining, often leading	Tez allows for pipelining, where multiple vertices can execute concurrently,
	to increased latencies between job phases.	improving throughput and reducing overall execution time.

Dynamic Scaling	MR's static resource allocation can lead to inefficiencies, particularly in varying workloads.	Tez can dynamically adjust the resources allocated to tasks based on runtime metrics, optimizing performance based on workload demands

Hive on Spark

Spark is another Big data processing engine that came in 2013 and disrupted the Big data ecosystem. It does both what Hive used to do and what MapReduce (MR) can do and even more. Hive, rather than directly taking it as competition, decided to adopt the execution optimizations provided by Spark and introduced Hive on Spark.

Key Features of Spark:

- **In-Memory Computing**: Spark processes data in-memory, significantly improving performance compared to disk-based processing (like traditional MapReduce).
- **Speed**: Spark can perform data processing tasks much faster than Hadoop MapReduce due to its DAG (Directed Acyclic Graph) execution model and in-memory capabilities.
- **Unified Framework**: Supports various data processing paradigms, including batch processing, streaming, machine learning, and graph processing.

Why Use Hive on Spark?

- **Performance**: Spark's in-memory computing capabilities allow Hive queries to execute much faster than with traditional MapReduce.
- **Resource Efficiency**: Spark optimizes resource usage, making it more efficient for executing complex queries.
- **Compatibility**: Users can continue using Hive's SQL-like syntax while benefiting from Spark's performance advantages.

Key Architectural Components

- Query Compilation: When a HiveQL query is submitted, it is parsed and compiled into an execution plan. This involves creating a logical plan and transforming it into a physical execution plan.
- DAG Execution: The physical execution plan is translated into a Directed Acyclic Graph (DAG), which represents the sequence of operations needed to execute the query. This allows Spark to optimize the execution of the query by scheduling tasks in parallel where possible.
- Vertex Manager: This component manages individual vertices within the DAG, ensuring that tasks are executed in the correct order and that dependencies are respected.
- Task Executors: The actual execution of tasks happens in Spark's task executors, which perform the computations on the data.
- Spark Application Master: When a Hive query is executed, a Spark Application Master is started to manage the execution of the Spark job. It negotiates resources with YARN and coordinates the execution of tasks.

Workflow of Hive on Spark

1. Query Submission: The user submits a HiveQL query via the Hive CLI or HiveServer2.
2. Query Compilation: HiveServer2 compiles the query, creating a logical plan and transforming it into a physical plan optimized for execution on Spark.

3. DAG Creation: The physical plan is converted into a DAG that details how the query will be executed. Each stage in the DAG represents a transformation or action on the data.
4. Resource Allocation: The Spark Application Master requests the necessary resources from YARN, which allocates containers for the tasks.
5. Task Execution: Spark's task executors execute the tasks as per the DAG, performing computations on the data stored in HDFS.
6. Result Retrieval: After all tasks are completed, the results are collected and sent back to HiveServer2, which returns them to the user.

Example Scenario Let's walk through an example of how a Hive query is converted into a Directed Acyclic Graph (DAG) and the subsequent execution involving jobs, stages, and tasks in Hive on Spark.

Suppose we have a Hive table called `sales` with the following schema:

```
CREATE TABLE sales (
  id INT,
  product STRING,
  amount DOUBLE,
  sale_date DATE
);
```

If we want to find the total sales amount for each product in the year 2023, we can run the following HiveQL query:

```
SELECT product, SUM(amount) AS total_sales
FROM sales
WHERE YEAR(sale_date) = 2023
GROUP BY product;
```

Conversion of the Hive Query to DAG

1. **Query Submission**
    - The user submits the above HiveQL query through the Hive CLI or HiveServer2.
2. **Query Parsing**
    - The Hive parser processes the query:
        - Lexical Analysis: Validates the syntax.
        - Semantic Analysis: Checks the existence of the sales table and its columns.
3. **Query Optimization**
    - Logical Plan Generation: A logical plan is created, representing the intent of the query (select `product` and `SUM(amount)` from `sales` with a filter for the year 2023 and grouped by product).
    - Optimization: The logical plan is optimized:
        - Predicate Pushdown: The filter `YEAR(sale_date) = 2023` is applied as early as possible to minimize the data processed in later stages.
4. **Physical Plan Generation**
    - The optimized logical plan is transformed into a physical execution plan:
        - Scan the sales table.
        - Filter the results where `YEAR(sale_date) = 2023`.

        - Group by product and compute `SUM(amount)`.
5. **DAG Creation**
    - The physical plan is converted into a DAG. For our query, the DAG would consist of the following nodes:
        - Node 1: Scan the sales table.
        - Node 2: Filter by year 2023.
        - Node 3: Group by product and calculate `SUM(amount)`.

Execution of the DAG in Hive on Spark

1. **Job**
    - Definition: The entire execution of the query is represented as a single job.
    - Job Submission: The job is submitted to the Spark execution engine.
2. **Stages**
    - Stage 1:
        - Tasks: Scan the sales table and apply the filter.
        - Example Execution: This stage may have multiple tasks if the sales table is large and distributed across multiple nodes.
    - Stage 2:
        - Tasks: Group the filtered results by product and calculate `SUM(amount)`.
        - Dependency: This stage cannot start until Stage 1 completes, as it requires the output of the filtering process.
3. **Tasks**
    - Stage 1 Tasks:
        - Each task in this stage reads a partition of the sales table from HDFS, applies the filter condition, and produces intermediate results.
        - For example, if the sales table has four partitions, there will be four tasks executing in parallel, each processing a different partition.
    - Stage 2 Tasks:
        - After Stage 1 completes, the results are shuffled based on the product key, which may involve data transfer across nodes.
        - Tasks for this stage will take the filtered data and perform the group-by operation, calculating the total sales for each product.

Execution Workflow

1. User submits the query: The HiveQL is sent for execution.
2. Parsing and Optimization: The query is parsed, optimized, and transformed into a DAG.
3. Job Submission: The entire query execution is submitted as a job to Spark.
4. Stage Execution:
    - Stage 1 executes tasks to scan and filter data.
    - Stage 2 executes tasks to group data and compute totals.
5. Results: Once all tasks are completed, the final results are collected, aggregated, and returned to the user.

Getting Started with Hive on Spark

- Installation and Setup:
    - Ensure you have a Hadoop cluster set up with HDFS.
    - Install Apache Hive and configure it to use Spark as the execution engine. Set the following property in `hive-site.xml`:

```
<property>
```

```
  <name>hive.execution.engine</name>
  <value>spark</value>
</property>
```

-
- Writing HiveQL Queries:
    - Use HiveQL to create tables, load data, and execute queries. Here’s an example:

```
CREATE TABLE IF NOT EXISTS sales (
  id INT,
  product STRING,
  amount DOUBLE
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE;

LOAD DATA INPATH 'hdfs://path/to/sales_data.csv' INTO TABLE sales;

SELECT product, SUM(amount) AS total_sales
FROM sales
```

- GROUP BY product;

Yarn Architecture - Oct 28

In past lessons, we learned that Hive sends queries to execution engines like MapReduce (MR), Tez, or Hive on Spark. These engines need resources to run the queries, so a system is needed to manage and provide those resources. This system acts like a "bank" that execution engines can ask for resources, and it decides what type and amount of resources are needed and for how long.

The main purpose of YARN is to separate resource management and data processing into different parts. This way, a central resource manager can help different data processing applications (like MapReduce, Spark, Storm, and Tez) to run on the same cluster.

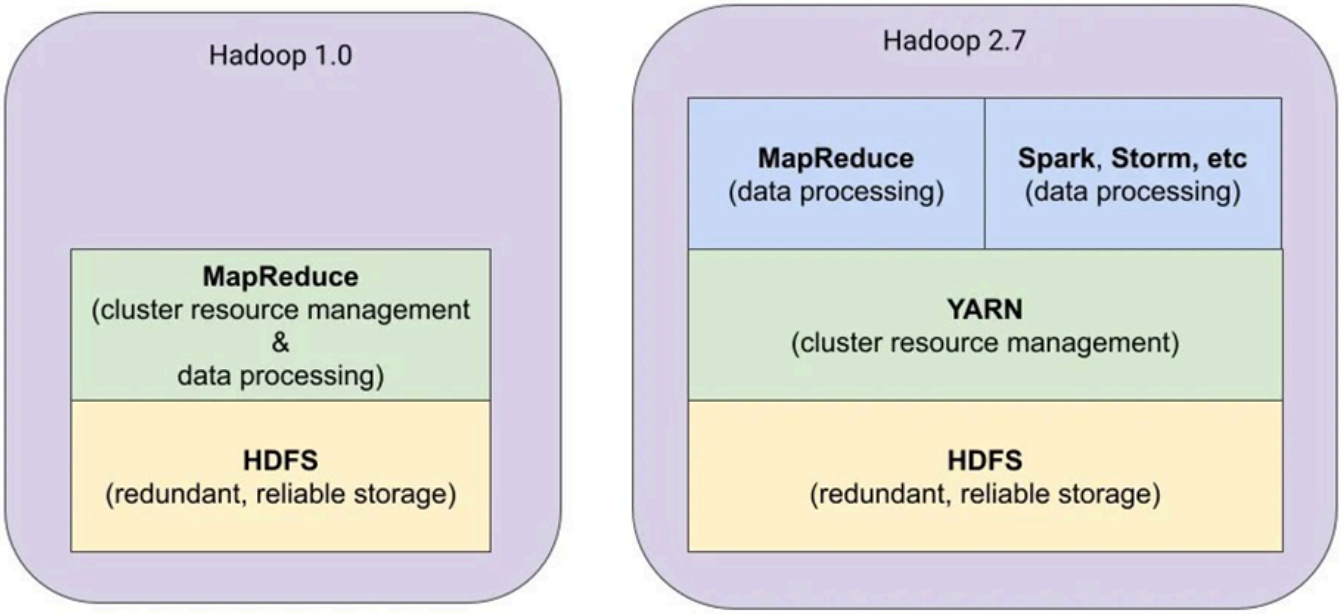
Before YARN, Hadoop used MapReduce as both a resource manager and processing engine, which was connected directly with Hadoop's file system (HDFS) and could only handle MapReduce jobs. This setup made it hard to run other applications on a Hadoop cluster.

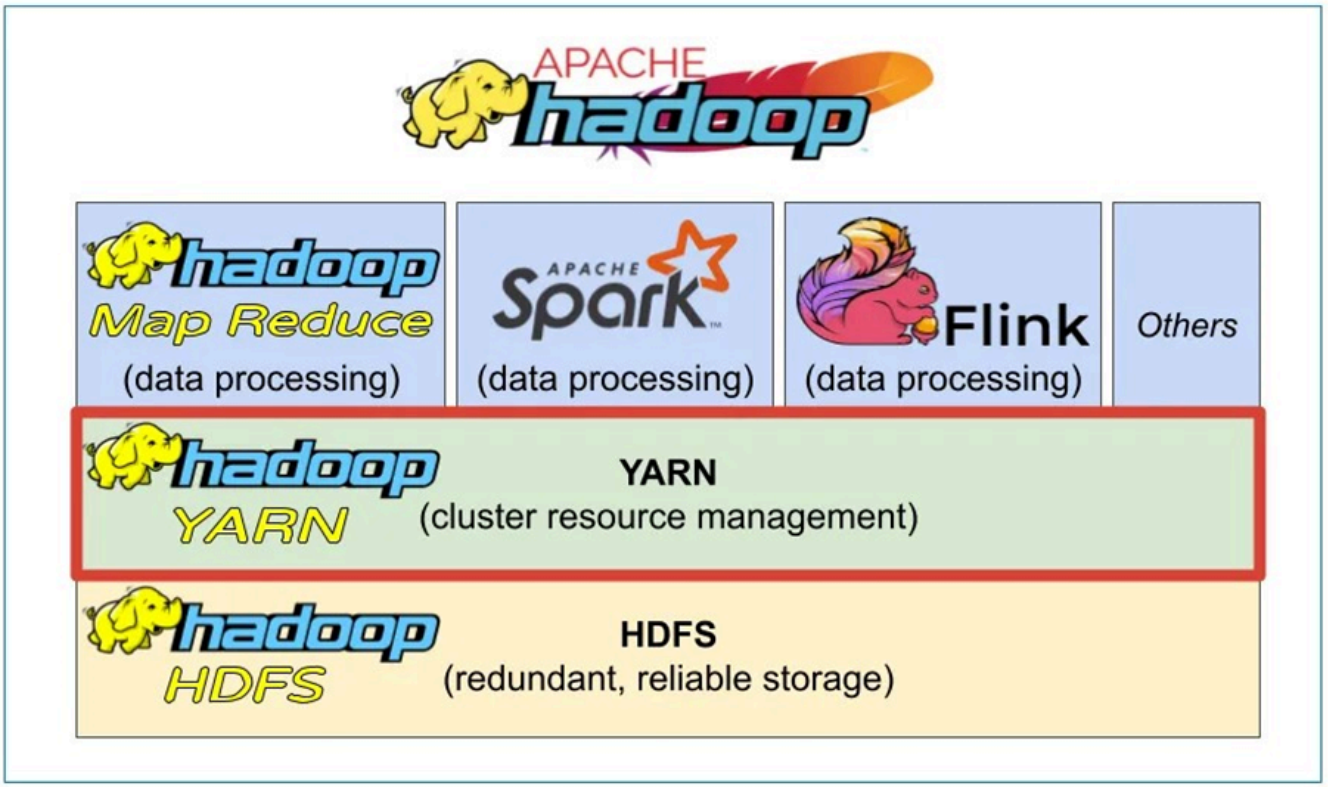
To solve these issues, YARN was created. YARN splits the processing and management roles of MapReduce, allowing multiple processing engines and applications to run on the same cluster.

Do you have any specific processing engines you’re interested in using with YARN, like Spark or Tez?

Before 2012
Users could write MapReduce programs using scripting languages

Since 2012
Users could work on multiple processing models in addition to MapReduce



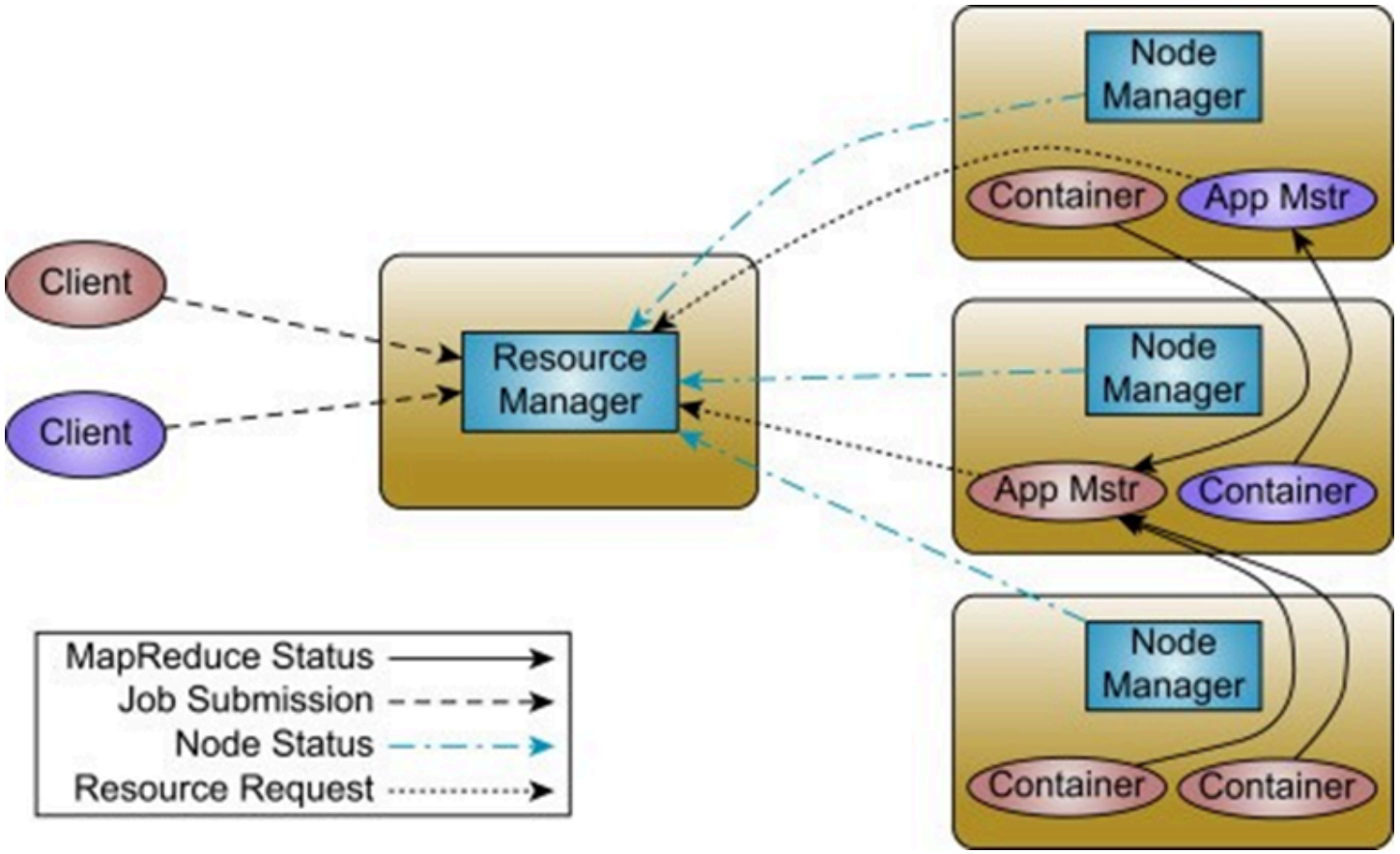


Benefits of YARN

- **Centralized Resource Management:** YARN offers a central system for managing resources, which can allocate and adjust resources for different applications running on a Hadoop cluster.
- **Flexibility:** YARN separates scheduling and resource management from data processing. This makes it possible to run various types of data processing applications on the same cluster.
- **Better Cluster Utilization:** YARN's dynamic resource allocation ensures each application gets what it needs to run without disrupting others. Resources not used by one application can be used by another.
- **Cost-Effective:** With YARN, a single Hadoop cluster can support many types of workloads and applications, making it more efficient and reducing costs for big data processing.
- **Reduced Data Movement:** Since data doesn't need to move between Hadoop, YARN, and other systems on different computer clusters, data movement is minimized.

YARN Architecture

YARN acts like the operating system for a Hadoop cluster. A cluster is a group of connected computers that work together as one system, sharing resources like CPU, memory, disk space, and network speed. YARN's job is to manage these resources and distribute them to different tasks running on the cluster. Just like an operating system manages a computer's resources and gives them to different programs, YARN makes sure resources in the cluster are shared fairly and efficiently among different tasks, boosting performance and resource use.



It has two major responsibilities:

- Management of cluster resources such as compute, network, and memory
- Scheduling and monitoring of jobs

YARN's architecture comprises three primary components which helps it to fulfil its responsibilities

- **Resource Manager (RM)**
- **Node Manager (NM)**
- **Application Master (AM)**

Resource Manager

The Resource Manager (RM) is the main part of YARN, managing resources across the whole Hadoop cluster. It runs on a master node and serves as a central authority, assigning resources to different applications based on their needs.

The RM has two important parts:

1. **Scheduler**
2. **Application Manager**

Scheduler

The YARN Scheduler is a key part of the Resource Manager. It's responsible for deciding how resources (like CPU, memory, etc.) are shared across applications in the cluster. The Scheduler itself doesn't run or monitor applications; it simply allocates resources (called "containers") based on available resources and job priority.

The Scheduler balances several factors:

- **Job Requirements:** Different jobs need different resources (e.g., some need more memory, others need more CPU).
- **Resource Availability:** Resources are allocated based on what's available in the cluster.
- **Fairness and Prioritization:** The Scheduler ensures that resources are shared fairly, so no application takes over all resources, and that job priorities are respected.

Scheduler: Pluggable Architecture

YARN's Scheduler uses a flexible, "pluggable" system, allowing different scheduling algorithms based on the cluster's needs:

a) FIFO Scheduler (First In, First Out)

- **Overview:** This is the simplest approach. Jobs are processed in the order they arrive, one at a time.
- **How It Works:**
    1. Jobs are placed in a queue.
    2. The first job in line gets all needed resources.
    3. Once it finishes, the next job is given resources.
    4. This repeats until all jobs are done.

b) Capacity Scheduler

- **Overview:** Designed for shared clusters, it lets multiple teams share resources while ensuring each group has a fixed portion of the cluster.
- **How It Works:**

1. Resources are divided into queues (groups) with each queue given a certain percentage of resources.
    - E.g., 40% of resources go to data science, 30% to ETL jobs, and 30% to real-time analytics.
2. Each queue can run multiple jobs and may lend unused resources to other queues temporarily.
3. Jobs in a queue are scheduled by FIFO or Fair Scheduling.
4. If a queue uses all its resources, its jobs will wait until resources are available.

c) Fair Scheduler

- **Overview:** Ensures that all jobs share resources equally over time. Both short and long jobs receive fair resources.
- **How It Works:**
    1. Jobs are placed in pools, each pool getting an equal share of resources.
        - E.g., if there are two pools, each pool receives 50% of resources, regardless of job submission order.
    2. Jobs in each pool are prioritized and scheduled based on their needs.
    3. Resources are adjusted dynamically; underutilized jobs may get more resources.
    4. **Preemption:** If one job waits too long while another is overusing resources, the Scheduler can temporarily stop the overusing job to reallocate resources.
    5.

d) Dominant Resource Fairness (DRF)

- **Overview:** DRF extends Fair Scheduling by considering multiple resource types (like CPU and memory) at once, ensuring no job monopolizes any single resource type.
- **How It Works:**
    1. DRF identifies the "dominant resource" for each job (the resource it needs most, e.g., CPU or memory).
    2. It ensures each job gets a fair share of its dominant resource.
    3. By balancing CPU and memory usage, DRF prevents jobs from overusing one resource (e.g., memory) while underutilizing another (e.g., CPU), promoting efficient resource use.

Yarn Resource Allocations and Optimizations - Nov 4

- **Memory Management in Yarn**
- **CPU Allocation in Yarn**
- **How Dynamic Allocation Works**
- **Job Scheduling in Yarn: Capacity vs. Fair Scheduler**
- **Fair Scheduler**
- **High Availability in YARN**
- **Disk and Storage Management in YARN**

In the last lecture, we learned about Yarn, its purpose, its design, and more. Some settings help Yarn work better and fix issues that often happen in production.

Just like *hive-site.xml* had settings for Hive, *yarn-site.xml* lets you adjust Yarn settings for better performance.

Memory Management in Yarn

Memory is very important in distributed computing. If jobs run out of memory, they might fail or run slowly. Yarn has several options for allocating and managing memory across containers.

- **yarn.nodemanager.resource.memory-mb**: This setting controls the total memory on a node that Yarn can allocate to containers. It helps avoid overloading a node with too many memory-heavy containers.

```
<property>
  <name>yarn.nodemanager.resource.memory-mb</name>
  <value>[Total Memory per Node]</value>
</property>
```

Example: If a node has 64 GB of RAM, you might set 60 GB for Yarn containers, leaving some memory for other system processes.

- **yarn.scheduler.maximum-allocation-mb**: This setting limits the maximum memory that any single container can request. It helps prevent one job from taking all the resources, which could block other jobs.

```
<property>
  <name>yarn.scheduler.maximum-allocation-mb</name>
  <value>[Maximum Memory per Container]</value>
</property>
```

Example: To limit any container to 16 GB, set this value to 16,000 MB.

Yarn also allows memory tuning for specific jobs. For instance, in a MapReduce job, the *Application Master (AM)* may need extra memory. This can be set in *mapred-site.xml*:

```
<property>
  <name>yarn.app.mapreduce.am.resource.mb</name>
  <value>[Memory for AM]</value>
</property>
```

For Spark jobs, you can configure the AM memory with:

spark.yarn.am.memory=4g

This helps manage cases where the AM, which oversees the job's execution, needs more memory to work well.

CPU Allocation in Yarn

Besides memory, managing CPU allocation is also important, especially for jobs that need a lot of processing power. Yarn lets you control the number of CPU cores a container can use.

- **yarn.nodemanager.resource.cpu-vcores**: This setting defines the total number of virtual CPU cores available on a node.

```
<property>
  <name>yarn.nodemanager.resource.cpu-vcores</name>
  <value>[Total CPU Cores per Node]</value>
</property>
```

Example: If a node has 16 physical cores, you can set this property to 16.

- **yarn.scheduler.maximum-allocation-vcores**: This setting limits the maximum number of vCores (CPU cores) that a single container can request.

```
<property>
  <name>yarn.scheduler.maximum-allocation-vcores</name>
  <value>[Max CPU Cores per Container]</value>
</property>
```

For MapReduce jobs, you can specify the CPU cores for each map or reduce task:

```
<property>
  <name>mapreduce.map.cpu.vcores</name>
  <value>[vCores per Map Task]</value>
</property>
<property>
  <name>mapreduce.reduce.cpu.vcores</name>
  <value>[vCores per Reduce Task]</value>
</property>
```

This tuning helps ensure each task gets enough CPU resources to avoid slowdowns, especially in jobs that are heavy on processing.

Stopping or Restarting Jobs with Dynamic Resource Allocation

Stopping or restarting jobs can be useful, especially for frameworks like Spark that adjust resource use based on the job's needs.

In Spark, dynamic resource allocation is managed by these settings in *spark-defaults.conf*:

```
spark.dynamicAllocation.enabled=true
spark.dynamicAllocation.minExecutors=2
spark.dynamicAllocation.maxExecutors=50
spark.dynamicAllocation.initialExecutors=10
```

With this setup, Spark starts with a specific number of executors (e.g., 10), can scale up to 50 executors if needed, and scale down to 2 executors if fewer resources are required. Dynamic allocation helps make the best use of cluster resources, especially when workloads vary in shared environments.

How Dynamic Allocation Works:

- **Idle Executors**: If executors are idle for a certain time, Yarn automatically scales them down to free up resources.
- **Resource Needs**: If a job suddenly needs more resources (like more mappers or reducers in Spark), Yarn adds more containers without restarting the job.

Job Scheduling in Yarn: Capacity vs. Fair Scheduler

In Yarn clusters used by multiple users, scheduling policies make sure resources are shared fairly among users or applications. Yarn has two main scheduling options: the Capacity Scheduler and the Fair Scheduler.

Capacity Scheduler

The Capacity Scheduler lets you split resources into multiple queues, where each queue has a guaranteed share but can also use extra resources if other queues are idle.

- **yarn.scheduler.capacity.root.default.capacity**: This setting defines the default percentage of resources for a specific queue.

```
<property>
  <name>yarn.scheduler.capacity.root.default.capacity</name>
  <value>[Capacity Percentage]</value>
</property>
```

Example: If you have two main user groups, one for data analytics (using Hive or Spark) and one for ETL (using MapReduce), you might allocate 60% of resources to analytics and 40% to ETL.

- **yarn.scheduler.capacity.maximum-am-resource-percent**: This limits the amount of resources any single *Application Master (AM)* can use, keeping one job from using too many resources at once.

Fair Scheduler

The Fair Scheduler ensures that all jobs receive a fair share of resources over time. Unlike the Capacity Scheduler, which assigns fixed resources to each queue, the Fair Scheduler dynamically adjusts resources based on each job's need.

Preemption: The Fair Scheduler can take resources from long-running, low-priority jobs and give them to high-priority, short-running jobs. Enable preemption with the following:

```
<property>
  <name>fairScheduler.preemption</name>
  <value>true</value>
</property>
```

Pool Configuration: Jobs can be assigned to pools, with each pool having a guaranteed minimum share of resources. This makes sure high-priority jobs get resources even if there are many ongoing long jobs.

```
<pool name="highPriority">
  <minShare>4</minShare> <!-- Minimum of 4 vCores -->
  <weight>2</weight> <!-- High priority jobs get more resources -->
</pool>
<pool name="lowPriority">
  <minShare>2</minShare>
  <weight>1</weight>
</pool>
```

With pool settings and preemption, the Fair Scheduler prevents large jobs from blocking smaller, urgent jobs, ensuring resources are shared efficiently.

High Availability in YARN

Problem: If the Resource Manager (RM) fails, the whole cluster's resource allocation stops, causing job failures.

Solution: YARN supports High Availability (HA) for the Resource Manager, using Zookeeper for failover. This setup includes an active and a standby RM. If the active RM fails, Zookeeper automatically promotes the standby RM, keeping services uninterrupted.

To enable HA in *yarn-site.xml*, add:

```
<property>
  <name>yarn.resourcemanager.ha.enabled</name>
  <value>true</value>
</property>
<property>
  <name>yarn.resourcemanager.ha.rm-ids</name>
  <value>rm1,rm2</value>
</property>
<property>
  <name>yarn.resourcemanager.hostname.rm1</name>
  <value>[Host1]</value>
</property>
<property>
  <name>yarn.resourcemanager.hostname.rm2</name>
  <value>[Host2]</value>
</property>
<property>
  <name>yarn.resourcemanager.zk-address</name>
  <value>[Zookeeper Quorum Address]</value>
</property>
```

This configuration ensures jobs continue to run even if the primary RM fails.

Disk and Storage Management in YARN

Problem: Containers need memory, CPU, and disk space for temporary data and logs. Without disk space management, jobs may fail due to storage shortages.

Solution: YARN lets you set up local directories for temporary storage and logs to manage disk space efficiently:

- **yarn.nodemanager.local-dirs**: Sets directories where YARN stores temporary container data.

```
<property>
  <name>yarn.nodemanager.local-dirs</name>
  <value>/path/to/local/dir1,/path/to/local/dir2</value>
</property>
```

yarn.nodemanager.log-dirs: Defines where container logs are saved.

```
<property>
  <name>yarn.nodemanager.log-dirs</name>
  <value>/path/to/log/dir1,/path/to/log/dir2</value>
</property>
```

This configuration helps manage disk space, as YARN can clear these directories after jobs finish.

Table 1: Memory Management Settings

Parameter	Description	Default Value	Framework
`yarn.nodemanager.resource.memory-mb`	Memory available per node for YARN containers.	40960 (40GB)	YARN
`yarn.scheduler.maximum-allocation-mb`	Maximum memory allocation per container.	16384 (16GB)	YARN
`yarn.app.mapreduce.am.resource.mb`	Memory allocation for MapReduce Application Master.	1024 (1GB)	YARN (MapReduce)
`spark.yarn.am.memory`	Memory allocated to Spark Application Master.	1024m (1GB)	Spark
`hive.tez.container.size`	Memory allocation for each Tez container in Hive.	4096 (4GB)	Hive (Tez)
`spark.executor.memory`	Memory allocated to each Spark executor.	8g (8GB)	Spark
`spark.driver.memory`	Memory allocated to the Spark driver.	4g (4GB)	Spark
`spark.yarn.executor.memoryOverhead`	Additional memory overhead for executors.	512m (512MB)	Spark
`spark.yarn.driver.memoryOverhead`	Additional memory overhead for the driver.	512m (512MB)	Spark

Table 2: CPU and Core Allocation Settings

Parameter	Description	Example Value	Component
`yarn.nodemanager.resource.cpu-vcores`	Number of CPU virtual cores per node.	16	YARN
`yarn.scheduler.maximum-allocation-vcores`	Maximum virtual cores per container.	8	YARN
`spark.executor.cores`	Number of cores allocated to each Spark executor.	4	Spark
`spark.driver.cores`	Number of cores allocated to the Spark driver.	2	Spark

`spark.task.cpus`	Number of CPU cores per Spark task.	1	Spark
`hive.tez.cpu.vcores`	Number of vcores allocated per Tez container.	2	Hive (Tez)

Table 3: Executor and Driver Settings

Parameter	Description	Example Value
`spark.executor.instances`	Number of executor instances for Spark.	10
`spark.dynamicAllocation.enabled`	Enables dynamic scaling of executors.	true

Table 4: Queue and Network Configurations

Parameter	Description	Example Value	Component
`spark.yarn.queue`	YARN queue for Spark production jobs.	production jobs	Spark
`hive.server2.tez.queue.name`	YARN queue for Hive Tez high-priority jobs.	high_priority	Hive (Tez)
`spark.network.timeout`	Timeout for network connections in Spark.	600s	Spark
`spark.yarn.maxAppAttempts`	Maximum retries for a failed Spark job.	3	Spark

Table 5: Performance and Optimization Settings

Parameter	Description	Example Value	Component
`spark.speculation`	Enables speculative execution to handle slow tasks.	`true`	Spark
`spark.executor.extraJavaOptions`	JVM options for executor memory management (e.g., GC tuning).	`-XX:+UseG1GC`	Spark
`spark.driver.extraJavaOptions`	JVM options for driver memory management (e.g., GC tuning).	`-XX:+UseG1GC`	Spark

Writing Hive queries effectively- Make Hive queries 2x faster - Nov 8

- **Inefficient Joins Due to Large Data Skew**
- **Small Files Problem in HDFS**
- **Broadcast Join**
- **Overloaded Reducers Causing Bottlenecks in Hive**

While working as a Data Engineer, you will encounter a lot of issues regarding slow Hive performance, and it's necessary for you to be ware of these issues and how to solve them. I have collated the most common issues that have been encountered and are bound to happen in the Data engineering world. Let's go over them one by one.

Inefficient Joins Due to Large Data Skew

Data skew means that data is not spread out evenly across different parts (partitions or nodes) of a system. In Hive, when you run a join operation, the system splits the data into smaller parts called *partitions* and assigns these parts to different nodes (or reducers) to process at the same time. If the data isn't balanced, some reducers will get much more data to handle than others. This can slow down the whole query because the overloaded reducers become bottlenecks.

Data skew often happens when joining tables with certain *skewed keys* — values that appear much more often than others. For example, let's say you have a table of customer orders, and 90% of the data is for just one customer. If you join on customer ID, the reducer handling that customer's data will have to process most of the data, while other reducers are left with very little work. This reduces the benefits of parallel processing.

Example Scenario of Data Skew

Consider two tables:

- An *orders* table with millions of records, each linked to a customer_id.
- A *customers* table with customer information, where a join is done on the customer_id.

If one customer (e.g., customer ID = 12345) has made a very large number of orders compared to others, a join on customer_id will send most of this customer's records to one reducer. The other reducers will get much less data, resulting in uneven work distribution.

```
SELECT o.order_id, c.customer_name
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id;
```

In this query, if customer_id = 12345 has 90% of the total orders, one reducer will be overloaded, delaying the entire query as the other reducers will finish their work early.

Symptoms of Data Skew in Joins

- **Long query execution times**: Some reducers take much longer to finish compared to others.
- **Inefficient resource usage**: Some reducers might be idle while others are overloaded with data.
- **Memory errors**: Reducers handling large amounts of skewed data can run out of memory (OOM errors).

How to Detect Data Skew

1. **Query Execution Plans**: You can check the execution plan of your query to see how data is divided across reducers. If one or a few reducers are handling a lot more data than others, this indicates data skew.
2. **Monitoring Logs**: Logs from Hadoop or YARN (if you are using it as the execution engine) can show if some reducers are handling much heavier loads.
3. **Skewed Key Analysis**: Analyze the data distribution before the join to find keys (values) that appear very frequently and may cause skew.

How to Fix Data Skew in Hive Joins

1. **Skew Join Optimization**
   Hive has a setting called *Skew Join Optimization* that helps manage data skew automatically in joins. When this is turned on, Hive detects skewed keys and breaks them into smaller chunks to distribute the processing more evenly across reducers.

   How it works: If a certain key (e.g., a customer ID) has many records, Hive splits these records into smaller parts and assigns them to different reducers, preventing any single reducer from being overloaded.

   Steps to enable Skew Join Optimization:

   SET hive.optimize.skewjoin=true;

   When this setting is enabled, Hive watches for skewed data in the *map phase*. If it finds skew, it splits the join into two phases:

- **Phase 1**: Processes the skewed key separately.
- **Phase 2**: Processes the rest of the data normally.

2. Sampling Skewed Data
Another way to handle skew is by *sampling* the data to find skewed keys. You can run a quick query to identify keys with a large number of records and then decide how to handle these keys (like running special queries or adjusting the query strategy).

Example:

```
SELECT customer_id, COUNT(*)
FROM orders
GROUP BY customer_id
ORDER BY COUNT(*) DESC
LIMIT 10;
```

This query finds the top 10 customers with the most orders. Once you identify these keys, you can treat them differently to balance the data more evenly across reducers.

Small Files Problem in HDFS

What is the Small Files Problem?

In Hadoop's Distributed File System (HDFS), the *small files problem* occurs when there are too many small files (smaller than the HDFS block size, typically 128MB or 256MB). HDFS is designed for large data blocks, so handling many small files is inefficient because HDFS is not optimized for small files.

Why Are Small Files a Problem in HDFS?

1. **NameNode Memory Overload**: The NameNode (which tracks where blocks are stored) keeps metadata for every file in memory. If there are millions of small files, the NameNode's memory becomes overloaded, which can slow down the system or even cause out-of-memory errors.
2. **Increased I/O Overhead**: HDFS works best with large files split into multiple blocks. When there are many small files, I/O operations (for reading and processing files) are less efficient. Each small file acts as a separate block, so accessing millions of files leads to high disk activity and slower processing.
3. **Inefficient MapReduce Jobs**: In MapReduce, each mapper processes one input split. Small files cause inefficiency because each mapper handles very little data, leading to a large number of tasks with minimal work, creating extra overhead.
4. **Network Congestion**: Small files create network congestion as each file requires a separate read operation across the network. This increases network traffic and processing delays.

Symptoms of the Small Files Problem

- Longer query execution times due to inefficient file access
- Overloaded NameNode memory, leading to memory issues or slower performance
- Increased task overhead in MapReduce, with many mappers doing minimal work
- High disk I/O and network use, with frequent file switching and network requests

Common Causes of Small Files

1. **Data Ingestion from External Sources**: Data from sources like sensors or logs often come in as small files. For example, if logs are generated every minute, this leads to a large number of small files.
2. **Improper File Splitting**: Data may be split into small chunks without being merged. This happens if data isn't batched well or if the storage format isn't efficient.
3. **Frequent Data Dumping**: Systems that save data every second or minute create many small files instead of combining them into larger files.

Solutions to the Small Files Problem

1. **Combine Small Files into Larger Files**
   The best solution is to combine many small files into larger ones. This reduces the number of files HDFS has to manage, making processing more efficient.

   Using Hive's File Merging Capabilities

   Hive has built-in settings to combine small files during query execution. You can enable these to merge small files at different stages, like in the map-reduce phase or the output phase.

   SET hive.merge.mapfiles=true; -- Merge small files in the map stage
   SET hive.merge.mapredfiles=true; -- Merge small files in the reduce stage
   SET hive.merge.size.per.task=256000000; -- Set merged file size (e.g., 256MB)
   SET hive.merge.smallfiles.avgsize=128000000; -- Set average file size threshold

Explanation of These Settings:

- `hive.merge.mapfiles=true`: Ensures small files created during the map phase are merged.
- `hive.merge.mapredfiles=true`: Merges small files at the reduce phase.
- `hive.merge.size.per.task`: Controls the size of each merged file.
- `hive.merge.smallfiles.avgsize`: Triggers the merge when the average file size is below the specified limit.

2. Use Efficient File Formats (ORC/Parquet)

Using columnar formats like ORC (Optimized Row Columnar) or Parquet helps reduce the number of small files because these formats are highly compressed and optimized for large data processing. Unlike text formats like CSV or JSON, columnar formats store data in a compact way, making it easier to combine files and lowering the number of files generated during data processing.

```
CREATE TABLE my_table (
  id INT,
  name STRING,
  sales DOUBLE
) STORED AS ORC;
```

Benefits of ORC and Parquet Tables:
These formats improve query performance with features like predicate pushdown, compression, and splitting, which make them effective for handling large datasets efficiently.

Broadcast Join

In a distributed computing environment like Hadoop or Spark, a join operation combines two datasets based on a common key. In Hive or Spark, joins can be resource-heavy because they need to move data across nodes so matching keys can be processed together.

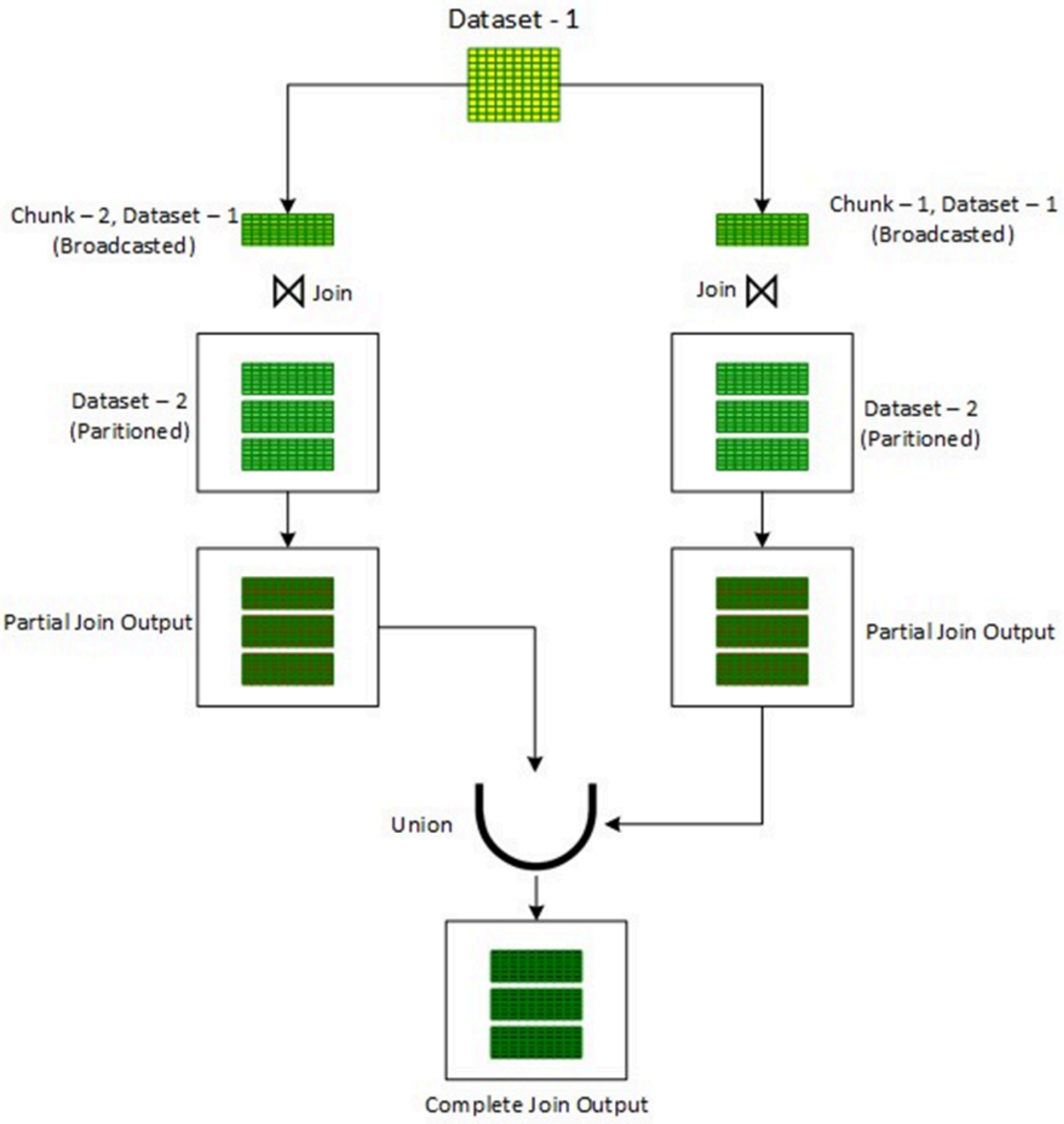
There are two main types of joins:

1. **Shuffle Join**: Both datasets are split across nodes, and data with matching keys is moved (or "shuffled") between nodes to align matching records.
2. **Broadcast Join**: A more efficient option if one dataset is small enough to fit in memory. The smaller dataset is broadcasted (copied) to all nodes, allowing each node to perform the join locally without shuffling data.

What is a Broadcast Join?

A broadcast join (also called a map-side join in Hive) is a join method where a smaller dataset is sent (or "broadcast") to all worker nodes, avoiding the expensive shuffle operation in traditional joins. Instead of transferring large data chunks between nodes (as in a shuffle join), the smaller dataset is copied to every node in the cluster. Each node then joins it with its portion of the larger dataset locally.

This process happens entirely on the mapper side, avoiding the reduce phase (hence the name "map-side join"). This method is highly efficient when the smaller dataset can fit into memory, as it reduces network traffic and eliminates the need to shuffle large amounts of data.



When to Use a Broadcast Join

Broadcast joins are most useful when:

1. **One table is much smaller than the other**: The smaller table can fit into memory and be sent to all worker nodes.
2. **Reducing network traffic is important**: Broadcasting the smaller table avoids the need to shuffle large data between nodes.
3. **Map-side join optimizations are available**: Systems like Hive, Spark, or Presto can optimize query performance by broadcasting smaller tables in joins when suitable.

How a Broadcast Join Works

In a broadcast join, the smaller dataset is replicated across all nodes, and each node performs the join locally with its partition of the larger dataset. Here's how it works:

1. **Small Dataset Identification**: The system identifies the smaller dataset, which must be small enough to fit into the memory of the worker nodes.
2. **Broadcasting**: The smaller dataset is sent (or broadcast) to all worker nodes across the cluster.
3. **Local Join Execution**: Each worker node holds a partition of the larger dataset. With the smaller dataset in memory, each node performs the join locally with its partition.
4. **Result Gathering**: The results from each node are collected without any additional data shuffling or movement, avoiding the overhead of a traditional join.

Example in Hive

Consider two tables:

- **customers** (small table): Contains customer information with a few thousand records.
- **orders** (large table): Contains millions of orders, each associated with a customer ID.

Normally, joining these tables would require data to be shuffled between nodes:

```
SELECT o.order_id, c.customer_name
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id;
```

If the **customers** table is small, Hive can broadcast it to all nodes for the join. By enabling map-side joins, Hive automatically broadcasts the smaller table.

Enabling a Broadcast Join in Hive:

```
SET hive.auto.convert.join=true;
```

With this setting, Hive detects if the **customers** table is smaller than a certain threshold and will broadcast it, performing a map-side join instead of a reduce-side shuffle join.

Configuration for Broadcast Join in Hive:
`hive.auto.convert.join.noconditionaltask.size`: This controls the size threshold (in bytes) for Hive to automatically broadcast the smaller table. If the smaller table is below this size, Hive will use a broadcast join.

SET hive.auto.convert.join.noconditionaltask.size=10000000; -- 10MB

In this case, if the **customers** table is smaller than 10MB, Hive will broadcast it for the join.

Overloaded Reducers Causing Bottlenecks in Hive

What Happens When Reducers Are Overloaded?
When a reducer becomes overloaded, it has to process more data than it can handle, leading to several issues:

1. **Memory Overflows (OOM Errors):** An overloaded reducer may use more memory than it's allocated, which can result in Out of Memory (OOM) errors. These errors can cause the entire job or query to fail.

2. **Prolonged Execution Times:** If certain reducers receive more data than others, they take longer to process, creating a bottleneck. The job cannot finish until all reducers complete their tasks, so even reducers that finish early will remain idle while waiting for the overloaded ones.
3. **Imbalanced Resource Utilization:** When some reducers are overloaded while others are underutilized, it reduces the efficiency of the system's parallel processing capabilities. This leads to wasted resources, such as CPU, memory, and disk I/O.
4. **High Disk I/O and Network Traffic:** An overloaded reducer may spill data to disk, causing high disk I/O. Additionally, large amounts of data may need to be shuffled across the network to the reducers, resulting in network congestion.

How to Detect Overloaded Reducers?

1. **Logs and Metrics:**
   Review Hive and Hadoop logs for indicators of data skew, such as Out of Memory (OOM) errors or unusually long reduce task durations. Monitoring tools like YARN Resource Manager UI, Hadoop JobTracker, or platforms like Ganglia and Prometheus can offer insights into how tasks are distributed among reducers. If certain reducers take significantly longer to complete than others, it may point to data skew or overloaded reducers.
2. **Execution Plans:**
   Use the `EXPLAIN` command to analyze the query's execution plan. This will help identify if the workload is unevenly distributed across reducers. A small number of reducers handling large datasets could suggest a configuration problem.
3. **Monitor Shuffle Data Size:**
   If a query generates a large volume of shuffle data (data transferred between mappers and reducers), it can cause reducers to become overloaded. Tracking the shuffle data size per reducer can help pinpoint where the issue lies.

Solutions to Overloaded Reducers

1. **Adjust the Number of Reducers**
   The number of reducers in Hive is determined by the amount of data being processed and the `hive.exec.reducers.bytes.per.reducer` setting. This setting controls how much data each reducer processes. If the value is too high, there will be fewer reducers, leading to overload.
   How to adjust the number of reducers:

   SET hive.exec.reducers.bytes.per.reducer=67108864; -- 64MB per reducer

   This ensures each reducer processes approximately 64MB of data. For large datasets, this will create more reducers and help balance the load. You can also explicitly set the number of reducers:

   SET mapreduce.job.reduces=100; -- Set to 100 reducers

   However, setting the number of reducers manually should be done carefully. Too few reducers can cause overload, while too many may introduce overhead without improving performance.

2. Enable Data Skew Handling in Joins
If data skew is the cause of overloaded reducers, enable skew join optimization in Hive. This automatically detects when certain keys are disproportionately large and processes them separately, avoiding overload on reducers.

How to enable skew join optimization:

SET hive.optimize.skewjoin=true;

With this setting, Hive splits the processing of skewed keys into smaller tasks, distributing them across multiple reducers to avoid overloading any single reducer.

3. Use Partitioning and Bucketing
Partitioning and bucketing help distribute data more evenly across reducers, reducing the likelihood of overload.

Partitioning breaks large datasets into smaller pieces based on specific columns. For example, partition by date:

```
CREATE TABLE partitioned_orders (
  order_id INT,
  customer_id INT,
  amount DOUBLE
) PARTITIONED BY (order_date STRING)
STORED AS PARQUET;
```

Bucketing splits data into a fixed number of "buckets" based on a specific column, ensuring each reducer handles a similar amount of data. Example:

```
CREATE TABLE bucketed_orders (
  order_id INT,
  customer_id INT,
  amount DOUBLE
) CLUSTERED BY (customer_id) INTO 32 BUCKETS
STORED AS ORC;
```

4. Increase Memory for Reducers
If reducers are overloaded due to insufficient memory, increasing the memory allocation for each task can help. This allows reducers to process larger amounts of data without spilling to disk or causing memory errors.

SET mapreduce.reduce.memory.mb=8192; -- 8GB memory for reducers SET **mapreduce.reduce.java.opts=-Xmx6144m;** -- 6GB heap space for reducer JVM

For Tez:

```
SET tez.task.resource.memory.mb=8192;  -- 8GB memory for Tez tasks
SET tez.task.java.opts=-Xmx6144m;  -- 6GB heap space for Tez task JVM
```

Increasing memory reduces the chance of reducers failing due to memory overflows and helps improve performance for memory-intensive tasks.

Limitations of Hive Engine and strategies to overcome them - Nov 11

In the last lecture, we learned about some ways to make Hive faster. In this lecture, we'll look deeper into a few more of these ways and understand why, despite these optimizations, other technologies are slowly replacing Hive.

Partition Pruning

What is Partition Pruning?
Partition pruning is a way Hive avoids reading unnecessary data by skipping irrelevant partitions in a query. When a table is divided into partitions (like by date or region), Hive reads only the partitions that match the query condition. This means Hive can process data faster because it reads less.

Detailed Example:
Suppose we have a table called `sales`, which is split (or partitioned) by `region` and `order_date`:

```
CREATE TABLE sales (
   order_id INT,
   amount DOUBLE
)
PARTITIONED BY (region STRING, order_date STRING);
```

If you run a query on this table with conditions for `region` or `order_date`, Hive will skip partitions that don't match. For example:

```
SELECT order_id, amount
FROM sales
WHERE region = 'US' AND order_date = '2024-01-01';
```

Without partition pruning: Hive would read all partitions (for all regions and dates), even the ones that don't match.
With partition pruning: Hive will only look at the data for `region = 'US'` and `order_date = '2024-01-01'`, ignoring all other partitions. This makes the query run faster.

Another Example:
Imagine you have a year of sales data divided by `order_date`. If you run this query:

```
SELECT COUNT(*)
FROM sales
WHERE order_date = '2023-11-01';
```

Hive will read only the partition for `2023-11-01` and skip the rest of the year's data, making it much faster.

Best Practices

- Always use partition columns in the `WHERE` clause to benefit from partition pruning.
- Partition tables by columns you commonly filter by, such as `date` or `region`.

This way, Hive can work faster by skipping unnecessary data.

Predicate Pushdown

What is Predicate Pushdown?
Predicate pushdown is a way for Hive to filter data directly at the storage level (like in ORC or Parquet formats) before bringing it into the query engine. This means only relevant data is read, which reduces input/output (I/O) and speeds up the query.

Detailed Example:
Imagine we have a table called `orders` stored in ORC format:

```
CREATE TABLE orders (
    order_id INT,
    customer_id INT,
    amount DOUBLE,
    status STRING
) STORED AS ORC;
```

If we run this query:

```
SELECT order_id, amount
FROM orders
WHERE status = 'completed';
```

- **Without predicate pushdown:** Hive would read all rows from the ORC file and then filter for `status = 'completed'` after loading the data into memory.
- **With predicate pushdown:** The filter `status = 'completed'` is applied at the storage layer. Only rows with `status = 'completed'` are read from disk, so less data is transferred and processed.

More Examples:

- **Numeric Filters:**

```
SELECT order_id, amount
FROM orders
WHERE amount > 1000;
```

With predicate pushdown, Hive will only read rows where `amount > 1000`, reducing data read from disk.
Date Filters:

```
SELECT order_id
FROM orders
WHERE order_date = '2023-01-15';
```

- Hive will only read rows with `order_date = '2023-01-15'`, skipping unnecessary data.

Best Practices

- Use columnar storage formats like ORC or Parquet to enable predicate pushdown.
- Always specify conditions in `WHERE` clauses to take advantage of this optimization.

This way, Hive processes less data and queries run faster.

Dynamic Partitioning

What is Dynamic Partitioning?
Dynamic partitioning lets Hive automatically decide partition values for new data based on the data itself. This is helpful when you're adding data from a table without partitions to a table with partitions.

Detailed Example:
Imagine you have an unpartitioned table called `unpartitioned_sales` and a partitioned table called `sales_partitioned`:

```
CREATE TABLE sales_partitioned (
   order_id INT,
   amount DOUBLE
) PARTITIONED BY (region STRING, order_date STRING);
```

If you want to insert data into the partitioned table based on `region` and `order_date`, dynamic partitioning can handle this for you:

```
INSERT INTO TABLE sales_partitioned PARTITION (region, order_date)
SELECT order_id, amount, region, order_date
FROM unpartitioned_sales;
```

- **Without dynamic partitioning:** You would have to manually enter partition values for each insert, which would be difficult with large data.
- **With dynamic partitioning:** Hive automatically uses the values in `region` and `order_date` to create and insert into the correct partitions.

More Examples:

- **Dynamic Partitioning with Dates:**

```
INSERT INTO TABLE sales_partitioned PARTITION (region, order_date)
SELECT order_id, amount, region, order_date
FROM daily_sales;
```

Hive will create partitions dynamically based on `order_date`.

Dynamic Partitioning with Regions:

```
INSERT INTO TABLE sales_partitioned PARTITION (region, order_date)
SELECT order_id, amount, region, order_date
FROM region_sales;
```

Here, Hive partitions the data by `region` and `order_date` automatically.

Best Practices

- Use dynamic partitioning when loading data from unpartitioned sources.
- Make sure the partition columns are included in the `SELECT` statement when inserting into a partitioned table.

Dynamic partitioning can save time and simplify loading data into partitioned tables.

Compression Techniques

What is Compression?
In Hive, compression reduces the size of data during processing and in the final output, which helps save storage space and reduces input/output (I/O) when running queries.

Detailed Example:
To turn on compression for data processed during queries (called intermediate data) and for final output files, use these settings:

For intermediate data (shuffle):
SET hive.exec.compress.intermediate=true; -- Compress intermediate data
SET mapreduce.map.output.compress=true; -- Enable compression for MapReduce intermediate output
SET mapreduce.map.output.compress.codec=org.apache.hadoop.io.compress.SnappyCodec; -- Use Snappy compression

For final output files:
SET hive.exec.compress.output=true; -- Compress final output data
SET mapreduce.output.fileoutputformat.compress=true; -- Enable compression for output files
SET mapreduce.output.fileoutputformat.compress.codec=org.apache.hadoop.io.compress.SnappyCodec; -- Use Snappy compression

More Examples:

- **MapReduce with Compression:**
  When running a MapReduce query that generates a lot of intermediate (shuffle) data, enabling compression reduces the data transferred between mappers and reducers:

SET hive.exec.compress.intermediate=true;
SET mapreduce.map.output.compress=true;

Output Compression:
For queries that produce large outputs saved to HDFS:

SET hive.exec.compress.output=true;
SET mapreduce.output.fileoutputformat.compress=true;

Compressed output files will be smaller, which saves disk space and speeds up future queries that read this data.

Best Practices

- Use compression for large datasets, especially when there's heavy data transfer or large outputs.
- **Snappy** is often a good choice as it balances compression ratio and speed well.

Compression helps manage big data more efficiently by reducing data size and speeding up processing.

Auto Reduce Parallelism

What is Auto Reduce Parallelism?
Auto reduce parallelism in Hive automatically adjusts the number of reducers based on the size of the data. Hive uses a setting to estimate how much data each reducer should process and then sets the number of reducers to match.

Detailed Example:
Suppose you have a large dataset, and Hive assigns too few reducers, making some of them handle too much data. You can improve this with these settings:

SET hive.exec.reducers.bytes.per.reducer=67108864; -- Set 64 MB per reducer
SET hive.exec.reducers.max=1000; -- Set a maximum of 1000 reducers

With these settings, Hive calculates the optimal number of reducers for the data size. For example, if your dataset is 1 GB and each reducer handles 64 MB, Hive will use around 16 reducers.

More Examples:

- **Optimizing Large Joins:**
  For a query with a large join:

  ```
  SELECT o.order_id, c.customer_name
  FROM orders o
  JOIN customers c ON o.customer_id = c.customer_id;
  ```

- By adjusting `hive.exec.reducers.bytes.per.reducer`, Hive can assign more reducers for this join, speeding up the query.
- **Handling Skewed Data:**
  For uneven data (skewed data), adjusting the reducer size helps balance the load, so all reducers finish at similar times.

Best Practices

- Start with a manageable setting for `hive.exec.reducers.bytes.per.reducer`, like 64 MB or 128 MB, and adjust based on the data size.
- Set a maximum number of reducers to prevent too much parallelism, which can increase overhead.

Auto reduce parallelism helps Hive balance workload and process large datasets efficiently.

Avoid Cartesian Joins

What is a Cartesian Join?
A Cartesian join (or cross join) happens when there is no specific join condition between tables. This causes all rows from one table to pair with all rows from the other, creating a massive dataset that can drastically slow down performance.

Detailed Example:
Suppose you have two tables: `customers` and `orders`. Running a query without a join condition, like this:

SELECT * FROM customers, orders;

will produce a Cartesian product of all rows in `customers` and `orders`. For large datasets, this can lead to a huge, unmanageable result.

How to Prevent Cartesian Joins:
To avoid Cartesian joins, always use a clear join condition. For example:

```
SELECT c.customer_name, o.order_id
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id;
```

You can also prevent accidental Cartesian joins by setting Hive to strict mode:
SET hive.mapred.mode=nonstrict; -- Default mode allows Cartesian joins
SET hive.mapred.mode=strict; -- Prevents queries without a join condition

Additional Examples:

Correct Join:
SELECT c.customer_name, o.amount
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id;

This correctly joins only matching rows.

Accidental Cartesian Join:
SELECT * FROM orders o, customers c;

This creates a Cartesian product and should be avoided, especially with large datasets.

Best Practices:

- Always use explicit join conditions.
- Enable strict mode to prevent accidental Cartesian joins.

As we move forward, we'll explore one of the most popular batch data technologies: Spark. It's essential to understand why the industry is shifting to Spark and the areas where Spark outperforms Hive.

Hive vs. Spark: Why Spark is Replacing Hive

To understand why Spark has become more popular than Hive, let's look at their history, architecture, performance, and functionality.

1. Historical Context: How Hive and Spark Evolved

Apache Hive:
Apache Hive was created by Facebook to manage large data stored in the Hadoop Distributed File System (HDFS). Hive's main purpose was to make it easy to query big data using SQL-like language, HiveQL. Before Hive, Hadoop processing was mostly done using MapReduce, a complex programming model that required users to write custom Java code. Hive simplified this by allowing users to run SQL-like queries instead of writing MapReduce code. Originally, Hive used MapReduce as its main engine, but it later added support for Tez and Spark. Hive became a popular choice for running long batch-processing queries on large datasets.

Apache Spark:
Spark was developed at the University of California, Berkeley, as a response to MapReduce's limitations. Spark introduced a faster, in-memory computing model, which made it more efficient than MapReduce. Spark was created to handle not just batch processing but also real-time and streaming data. With libraries like Spark SQL, Spark Streaming, MLlib, and GraphX, Spark quickly became a unified engine for both batch and real-time processing, making it much faster and more versatile than the older MapReduce model used by Hadoop.

2. Hive's MapReduce Bottleneck

Hive's dependency on MapReduce for query execution leads to several limitations:

a. Disk I/O Overhead

MapReduce relies on writing intermediate data to disk between its map and reduce stages. This disk-based processing causes heavy I/O overhead, especially with complex queries that involve multiple stages, such as joins and aggregations. For example, running a query like:

SELECT customer_id, SUM(amount) FROM sales GROUP BY customer_id;

in Hive with MapReduce would require several phases where intermediate results are written to disk in HDFS, creating a performance bottleneck due to constant disk reads and writes. This makes MapReduce slow for iterative tasks that require repeated data access and transformations.

b. High Latency

MapReduce's dependence on disk operations results in high latency, making Hive unsuitable for real-time or interactive queries. Even simple analytical queries can have long wait times, especially with large datasets. This high latency is a significant drawback for data pipelines that need quick insights or near-instant results.

c. Multiple Job Initiation Overhead

MapReduce initiates a new job for each stage in a query (e.g., a join followed by an aggregation), which involves starting a new Java Virtual Machine (JVM) each time. The overhead from repeatedly starting and stopping JVMs further slows down the query execution and reduces overall efficiency.

3. Spark's In-Memory Computing Advantage

Spark was designed to overcome the limitations of MapReduce, and its biggest advantage is its in-memory computing capability.

a. In-Memory Processing

Spark minimizes disk I/O by keeping data in memory whenever possible. Once data is loaded into memory, Spark can apply multiple transformations without needing to write intermediate results to disk, which greatly speeds up processing. This is particularly beneficial for iterative tasks, such as machine learning algorithms or graph processing, where the same data needs to be accessed multiple times. For instance, a query like:

SELECT customer_id, SUM(amount) FROM sales GROUP BY customer_id;

in Spark would load the data into memory, perform all required transformations, and only write the final result to disk, drastically reducing execution time.

b. Directed Acyclic Graph (DAG) Execution

Instead of the traditional MapReduce approach, Spark uses a Directed Acyclic Graph (DAG) execution model, which allows for optimized query plans where tasks can be processed in parallel or pipelined. This reduces the need for multiple disk operations and shuffling stages. Spark schedules tasks based on dependencies within the DAG, achieving faster, more efficient execution.

c. Unified Engine for Batch and Real-Time Processing

Hive was originally built for batch processing, with real-time capabilities added later using engines like Tez and Spark. In contrast, Spark was designed from the start to handle both batch and real-time data processing. With Structured Streaming, Spark provides high-level APIs for real-time data processing, allowing users to process streaming data with the same DataFrame and SQL APIs they use for batch queries. This versatility makes Spark a stronger fit for modern data pipelines, which often require both real-time and batch processing capabilities.

4. Performance Comparisons

The performance differences between Hive and Spark become especially clear with large datasets and complex queries:

a. Query Speed

Spark can be 10 to 100 times faster than Hive for specific types of queries, especially those that involve iterative processes or real-time data. Spark's in-memory processing and DAG-based execution allow it to complete these tasks far quicker than Hive, which relies on MapReduce and writes data to disk after each step. For example, machine learning tasks like k-means clustering or linear regression are much slower in Hive because MapReduce continuously writes intermediate data to disk, while Spark keeps this data in memory, enabling much faster execution.

b. Resource Utilization

Spark uses system resources more efficiently by reducing unnecessary disk I/O. This leads to lower CPU and memory usage compared to Hive for the same workload. In environments where resources are limited, Spark's ability to finish tasks faster with fewer resources offers a key advantage.

c. Concurrency

Spark manages concurrent tasks better than Hive. Since Hive relies on the slower MapReduce system to launch and manage each job, it struggles with multiple simultaneous tasks, particularly in multi-user settings. Spark, however, completes jobs faster and has better resource management, allowing it to handle a higher degree of parallelism, which makes it more suitable for shared clusters.

5. Advanced APIs and Ecosystem Integration

While Hive mainly serves as a SQL query engine, Spark provides a broad ecosystem of APIs for diverse data processing tasks:

a. Spark SQL

Spark SQL allows users to run SQL queries on Spark's distributed data, similar to HiveQL, but with the advantage of in-memory execution. It integrates seamlessly with Spark's other libraries, making it easy to combine SQL queries with machine learning or graph processing in the same workflow.

b. MLlib for Machine Learning

Spark includes MLlib, a built-in machine learning library offering algorithms for classification, clustering, recommendation, and more. Hive lacks this built-in machine learning capability, so Spark becomes a more versatile tool for data scientists and engineers working on big data and AI projects.

c. GraphX for Graph Processing

Spark also includes GraphX, a library for graph computations, which Hive doesn't support. GraphX allows Spark to perform complex graph analytics, such as PageRank and shortest-path algorithms, making it ideal for tasks like social network analysis, recommendation engines, and other graph-based applications.

6. Conclusion: Why Spark Replaces Hive

The transition from Hive to Spark is driven by Spark's superior performance, flexibility, and scalability. Hive, once a powerful tool for querying large datasets, is limited by its reliance on MapReduce, leading to high latency, disk I/O bottlenecks, and slow query performance. Spark overcomes these challenges with in-memory processing, a DAG execution model, and a unified engine that supports batch, real-time, and iterative computations. Additionally, Spark's ability to handle both SQL queries and advanced analytics like machine learning and graph processing makes it more versatile for modern data engineering needs. Overall, Spark's faster performance, rich ecosystem, and real-time capabilities have made it the preferred choice over Hive for big data processing in today's data-driven world.

In Summary

Limitation	Description	Strategy to Overcome
Partition Pruning	Scans only relevant partitions based on query conditions, avoiding full dataset scans.	Use partition columns in `WHERE` clauses; partition by frequently queried columns like date or region.
Predicate Pushdown	Filters data at the storage layer (e.g., ORC, Parquet) before reading, reducing I/O and speeding up execution.	Use columnar storage formats; specify filter conditions in `WHERE` clauses.
Dynamic Partitioning	Automatically determines partition values during data insertion, useful for loading data into partitioned tables.	Enable dynamic partitioning; include partition columns in the `SELECT` clause for partitioned inserts.
Compression Techniques	Reduces intermediate and output data size to save storage and minimize I/O.	Use compression settings like Snappy for shuffle and output data compression in MapReduce queries.
Auto Reduce Parallelism	Dynamically adjusts the number of reducers based on data size, preventing overloaded reducers and balancing the workload.	Set appropriate values for `hive.exec.reducers.bytes.per.reducer` and a maximum reducer count to balance performance.
Avoid Cartesian Joins	Prevents unintended cross joins, which can create a massive data explosion and slow performance.	Use explicit join conditions; enable strict mode to avoid accidental Cartesian joins.
MapReduce Disk I/O Overhead	MapReduce-based execution writes intermediate data to disk, causing significant I/O and slowing down complex queries.	Use Tez or Spark execution engines for Hive; or migrate to Spark for in-memory processing.
High Latency and Job Overheads	High startup latency and multiple job initiations add to execution time, especially in large, iterative queries.	Leverage Spark's DAG-based execution model to minimize multiple stages and reduce startup time.
Limited Real-Time and Advanced Processing	Hive is batch-oriented and lacks support for real-time, iterative computations common in data science and machine learning.	Use Spark for in-memory and streaming capabilities, supporting both batch and real-time processing needs.

Understanding Spark Architecture - Nov 13

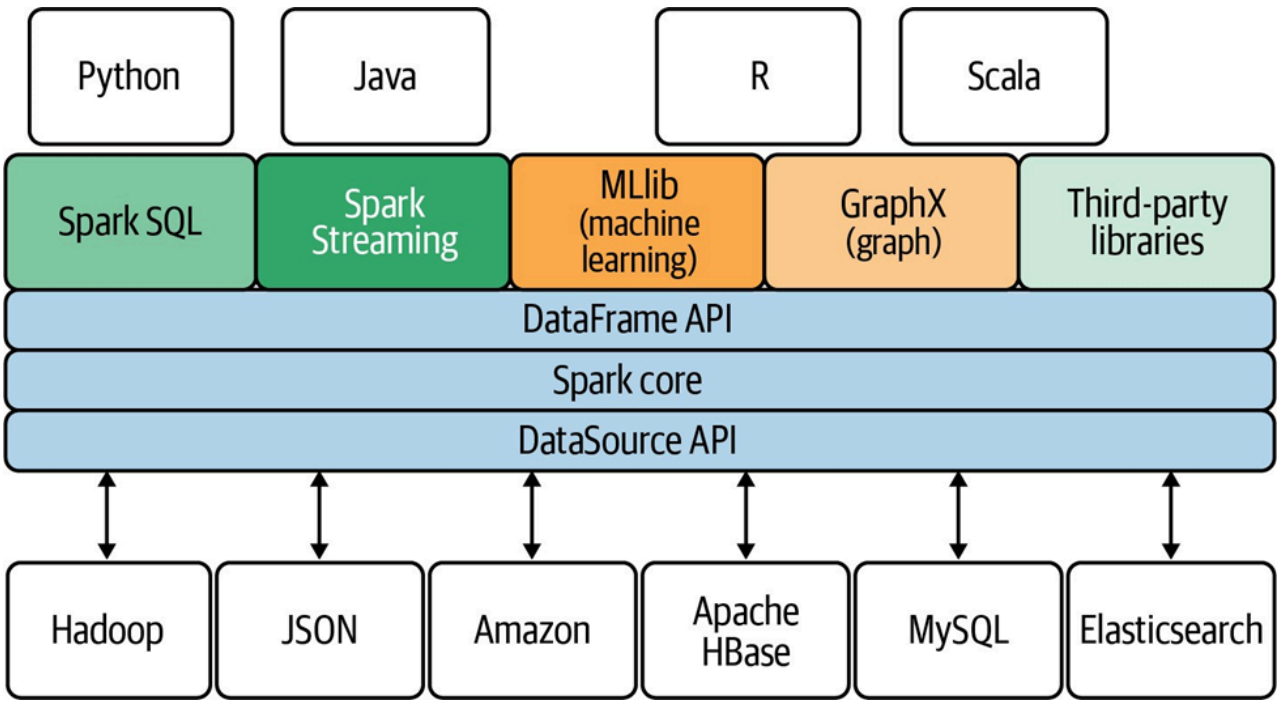
- **Key Features of Spark**
- **Batch Processing**
- **Stream Processing**
- **Machine Learning (ML)**
- **Graph Processing**
- **Support for Unified APIs**
- **Clarification: Spark vs. Hive**
- **Spark Architecture**
    - **Driver Program**
    - **Cluster Manager**
    - **Executors**
- **Catalyst Optimizer**

Apache Spark is a unified analytics engine for large-scale data processing. Unlike Hive, which relies on disk-based MapReduce, Spark is designed to perform **in-memory computation**, making it significantly faster for data processing tasks. Spark supports a variety of workloads, including batch processing, interactive queries, real-time stream processing, machine learning, and graph processing.

Key Features of Spark:

- **In-Memory Processing**: Spark performs most operations in memory, which allows for faster execution of iterative algorithms and reduces I/O overhead.
- **Unified Analytics**: Spark can handle both batch and streaming data using a single engine.
- **Scalability**: Spark can scale from a single machine to thousands of nodes in a cluster.
- **Advanced APIs**: Spark provides high-level APIs in languages like Java, Scala, Python, and R.

Spark is designed to handle various use cases like machine learning (ML), graph processing, streaming, and batch processing within a single platform. Its design philosophy centers around providing a unified solution for multiple data processing needs. Here's how Spark addresses each use case:



Batch Processing

Spark uses a core feature called **Resilient Distributed Dataset (RDD)** for batch processing. RDDs make it easier to process large amounts of data by splitting it into smaller parts and spreading it across a cluster. This is perfect for tasks like **ETL (Extract, Transform, Load)**, analytics, and other big data operations.

How Spark solves this:

- Spark uses **lazy evaluation** for RDD transformations, which means it waits until an action (like collecting results) is triggered to run the tasks. This reduces unnecessary data movement (called **data shuffling**) across the cluster.
- Spark performs **in-memory computation**, which is much faster than older methods like **MapReduce** that rely on disk-based processing.

Example:

Using Spark to process large logs or sales data in a batch to create daily reports.

Stream Processing

Spark offers **Structured Streaming**, which allows real-time data processing. Unlike traditional batch processing, where data is processed in large chunks, stream processing handles a continuous flow of data, such as real-time logs, events, or transactions.

How Spark solves this:

- **Structured Streaming** treats data streams as unbounded tables and processes them incrementally.
- It ensures **exactly-once processing**, meaning no data is missed or processed multiple times.
- It integrates well with streaming sources like **Kafka** and file systems.

Example: Analyzing real-time sensor data from IoT devices to detect anomalies instantly.

Machine Learning (ML)

Spark has **MLlib**, a library for scalable machine learning tasks like classification, regression, clustering, and recommendation systems.

How Spark solves this:

- **MLlib** is built on Spark's distributed architecture, making it scalable for large datasets.
- It uses **in-memory processing**, speeding up iterative algorithms.
- Training and evaluation happen in parallel across nodes, enabling faster experimentation and deployment.

Example: Building a recommendation system using collaborative filtering on millions of user-product interactions.

Graph Processing

Spark includes **GraphX**, a framework for processing graphs and running algorithms like **PageRank**, **connected components**, and **triangle counting**.

How Spark solves this:

- **GraphX** integrates graph processing with Spark's **RDDs**, allowing data reuse for tasks like batch analytics and machine learning.
- It distributes large graphs across the cluster and optimizes storage with techniques like **vertex-cut partitioning**.

Example: Using PageRank on a social network graph to rank users by their influence.

Support for Unified APIs

Spark offers high-level APIs in several languages, including **Scala**, **Java**, **Python**, and **R**. It uses a single API framework, the **Spark SQL API**, to handle different data processing tasks like batch, streaming, or graph processing. This means developers don't need to learn separate frameworks for each type of workload.

Why Spark's Unified Model Matters

By unifying **batch**, **streaming**, **machine learning (ML)**, and **graph processing** within one framework, Spark simplifies the data pipeline. Using the same execution engine for all tasks reduces the need to switch between different tools, which could otherwise add complexity and overhead. For example, a data scientist working on a batch job can use the same Spark cluster to deploy real-time machine learning models, avoiding the need to manage multiple execution environments.

Clarification: Spark vs. Hive

In **Hive**, we used **Beeline** to write queries and get results. In **Spark**, we write **application code** using programming languages like **Python**, **Scala**, or **Java**. For this lecture, we'll use **Python** to write Spark code.

Example Walkthrough

We'll write a Spark application that reads a dataset, performs some transformations, and outputs the results. We'll use a **customer purchases dataset** to calculate the total amount spent by each customer.

1. Setting Up the Application

Spark Session:
Every Spark application begins by initializing a **Spark session**, which serves as the main entry point for all Spark functionalities.

```
from pyspark.sql import SparkSession

# Initialize a Spark session
spark = SparkSession.builder \
    .appName("CustomerPurchaseAnalysis") \
    .getOrCreate()
```

JAVA —----------

```
import org.apache.spark.sql.SparkSession;

public class CustomerPurchaseAnalysis {
    public static void main(String[] args) {
        // Initialize a Spark session
        SparkSession spark = SparkSession.builder()
                .appName("CustomerPurchaseAnalysis")
                .getOrCreate();

        // Your Spark code here...

        // Stop the Spark session when done
        spark.stop();
    }
}
```

2. Loading Data

Dataset:
We assume the existence of a CSV file `purchases.csv` with the following columns: `customer_id`, `item`, `amount`.

Sample Data:

```
customer_id, item, amount
101, Book, 15.99
102, Pencil, 1.99
101, Notebook, 4.99
103, Backpack, 49.99
102, Eraser, 0.99
```

```
# Load data into a DataFrame
df = spark.read.csv("purchases.csv", header=True, inferSchema=True)
```

```
// Read the CSV file into a DataFrame using java
        Dataset<Row> df = spark.read()
                .option("header", "true") // Specify that the CSV file has a header
                .option("inferSchema", "true") // Automatically infer schema
                .csv("purchases.csv");
```

3. Exploring the Data

We can check the **schema** and **some sample records** to make sure the data is loaded correctly.

```
# Display the schema and sample records (java, pythen)
df.printSchema()
df.show()
```

Output:

```
root
 |-- customer_id: integer (nullable = true)
 |-- item: string (nullable = true)
 |-- amount: double (nullable = true)

+-----------------+---------------+-------------+
|customer_id   | item            |amount|
+-----------------+---------------+-------------+
|101              | Book          | 15.99|
|102              | Pencil        | 1.99 |
|101              | Notebook      | 4.99 |
|103              | Backpack      | 49.99|
|102              | Eraser        | 0.99 |
+-----------------+---------------+-------------+
```

4. Transforming the Data

Now, let's calculate the **total amount spent** by each customer.

Code:

```
# Group by customer_id and sum the amount for each customer
total_spent = df.groupBy("customer_id").sum("amount").withColumnRenamed("sum(amount)", "total_spent")

// Java
      Dataset<Row> total_spent  = df.groupBy("customer_id")
        .sum("amount")
            .withColumnRenamed("sum(amount)", "total_spent");
```

5. Displaying the Results

Check the output of the transformation to see the total spent by each customer.

Code:

```
# Show the results
total_spent.show()
```

Output:

```
+-----------+-----------+
|customer_id|total_spent|
+-----------+-----------+
|101        | 20.98     |
|102        | 2.98      |
|103        | 49.99     |
+-----------+-----------+
```

6. Writing the Results to Disk

After calculating the total amount spent, we save the results to a file for future analysis or reporting.

Code:

```
# Save the result as a CSV file
total_spent.write.csv("output/customer_totals.csv", header=True)
```

using java
totalSpent.write() .option("header", "true") // Ensure the header is included in the output
.csv("output/customer_totals.csv");

7. Stopping the Spark Session

At the end of a Spark application, it's important to stop the **Spark session**. This releases the resources (CPU, memory, and any allocated nodes) used by the session, preventing resource leaks and ensuring that resources are available for other applications.

Code: java/python
# Stop the Spark session
spark.stop()

Explanation of Each Step

1. **Initialization:** We start by creating a Spark session to access Spark's distributed computing power.
2. **Loading Data:** We load a CSV file as a **DataFrame**, which is Spark's structured data abstraction that makes querying and transformations easier.
3. **Data Exploration:** Using `.printSchema()` and `.show()`, we inspect the data to understand its structure and contents.
4. **Transformation:** We group the data by `customer_id` and sum the `amount` to calculate the total spent by each customer.
5. **Result Display:** Using `.show()`, we verify the output before saving it.
6. **Saving Results:** We save the final aggregated data as a CSV file for future use.
7. **Shutdown:** Finally, we stop the Spark session to release the resources used by the application.

Spark Architecture

At a high level, Spark's architecture consists of three main components:

1. **Driver Program:**
   The main application process that orchestrates the execution of the entire Spark job.
2. **Cluster Manager:**
   Allocates resources and manages the cluster on which Spark applications run.
3. **Executors:**
   Worker processes that execute tasks assigned by the driver and handle data storage in memory.

These components work together to enable efficient distributed data processing and optimize workload execution.

1. Driver Program

The **Driver Program** is the entry point of any Spark application and plays a central role in its execution. It is responsible for:

- **SparkSession Creation:** When a Spark application starts, the driver initializes a **SparkSession**, which is the main entry point for accessing Spark's functionalities.
- **Task Coordination:** The driver splits the application code into tasks based on the data transformations applied and then coordinates their execution across the cluster.

- **SparkContext Management:** The **SparkContext** is part of the driver and acts as a connection between the driver and the cluster manager. It manages resources, submits jobs, and coordinates with the executors.

Example:
When you create an **RDD** or **DataFrame** and perform an action (e.g., `collect()` or `count()`), the driver breaks down the action into tasks and submits them to the executors for execution.

---

2. Cluster Manager

The **Cluster Manager** is responsible for resource allocation and scheduling of tasks. Spark can use different types of cluster managers:

- **Standalone Mode:** Spark's built-in cluster manager, suitable for simple or development environments.
- **YARN:** Commonly used in Hadoop ecosystems, YARN allows Spark to integrate with an existing Hadoop cluster.
- **Mesos:** Apache Mesos provides a resource management layer for large-scale data centers.
- **Kubernetes:** Enables Spark applications to run in containerized environments, offering scalability and flexibility.

The **Cluster Manager** assigns resources (such as CPU and memory) to the driver and executor processes based on the application's requirements.

Cluster Manager Comparison:

Cluster Manager	Pros	Cons
Standalone Mode	Built into Spark, easy setup, ideal for small or development environments.	Limited scalability and features compared to other managers. Lacks advanced resource sharing.
YARN	Seamlessly integrates with Hadoop ecosystems. Allows Spark to share resources with other Hadoop applications.	Requires an existing Hadoop setup. More complex to configure and manage than Standalone.
Mesos	Suitable for large-scale deployments. Supports multi-tenancy and fine-grained resource allocation.	Complex setup and configuration. Less commonly used, may have less community support.
Kubernetes	Enables containerized Spark deployments. Offers excellent scalability and flexibility in cloud-native environments.	Kubernetes configuration can be complex for beginners. Potential performance overhead of containerization.

---

3. Executors

Executors are worker processes running on the cluster nodes that execute tasks given by the driver. Each application gets its own set of executors, which remain active for the entire duration of the application. Executors have two main responsibilities:

- **Task Execution:** Executors perform the tasks assigned by the driver on a subset of the data.

- **Data Storage:** Executors store data in memory or on disk, depending on Spark's caching mechanism. If an **RDD** or **DataFrame** is cached, it is stored in the executor's memory for faster access.

Executor Lifetime:
Executors are created when a Spark application starts and continue to run until the application finishes. This ensures efficient reuse of resources, particularly for iterative tasks like those used in machine learning.

Key Concepts in Spark Architecture

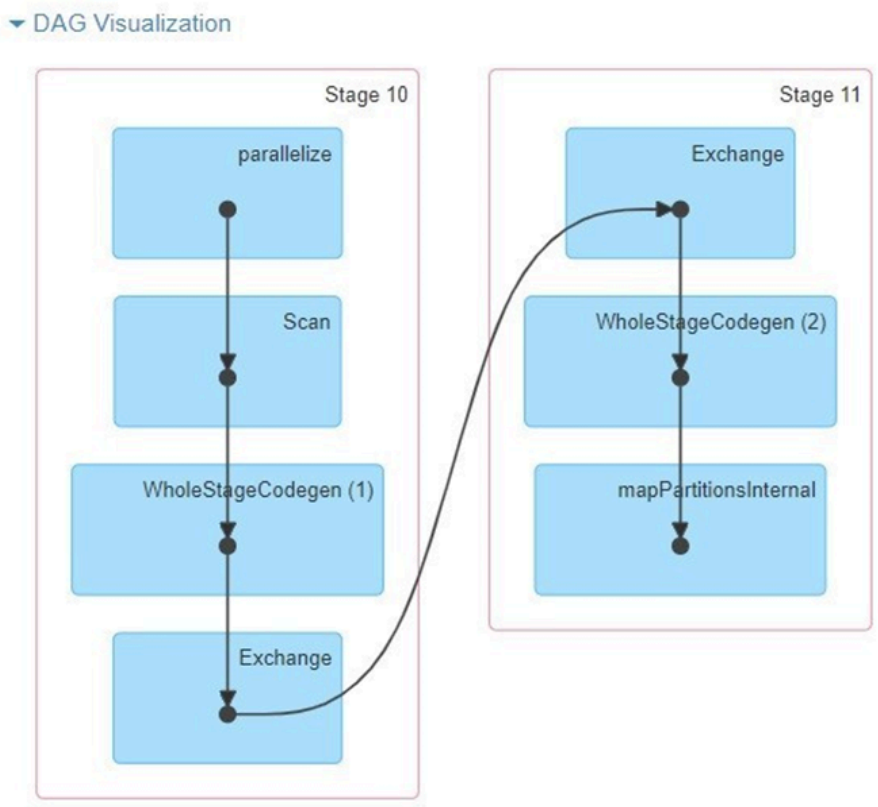
Let's explore some essential concepts that drive how Spark processes and executes applications.

Directed Acyclic Graph (DAG)

A **Directed Acyclic Graph (DAG)** is a crucial concept in Spark's execution model. It represents the sequence of operations (transformations) to be performed on the data, where each operation is a vertex in the graph, and the edges represent the data flow between these operations.

- **Directed:** The flow of data is one-way, from one operation to the next.
- **Acyclic:** The graph has no loops, meaning no operation depends on itself.
- **Graph:** It's a collection of vertices (operations) and edges (data flow).

The DAG is used by Spark to break down a job into smaller stages that can be executed in parallel, optimizing data processing and reducing unnecessary data shuffling. Spark builds the DAG for each job and submits tasks to executors based on this graph, ensuring efficient and fault-tolerant execution.



When you define transformations on RDDs or DataFrames in Spark, the system does not execute them immediately. Instead, it creates a Directed Acyclic Graph (DAG), which outlines all the required transformations in a logical plan.

- **Stage Division:** The DAG scheduler divides the DAG into stages based on shuffle boundaries, which occur when data needs to be rearranged across nodes.

- **Execution Plan:** After creating the DAG, Spark's scheduler converts it into stages and tasks that can run in parallel on different nodes. This allows Spark to optimize the execution plan before starting, ensuring efficient processing.

Job, Stages, and Tasks

In Spark, a **job** is triggered when an action, like `collect()` or `count()`, is called. Each job is divided into **stages**, and each stage is split into **tasks**.

- **Job:** Represents a high-level action, such as reading data from a file or performing a calculation.
- **Stage:** A set of tasks that can run in parallel. Stages are created based on shuffle boundaries.
- **Task:** The smallest unit of work in Spark, which processes a partition of the data.

Example: In an ETL (Extract, Transform, Load) application, reading data from HDFS might be one job, transforming the data another, and saving it back to HDFS as a third job. Each of these jobs is divided into stages and tasks based on the DAG.

In-Memory Processing and Fault Tolerance

Spark's architecture is optimized for **in-memory processing**, which significantly speeds up iterative and interactive workloads.

- **In-Memory Caching:** Spark stores data in memory across multiple iterations, reducing the need to read and write from disk repeatedly.
- **Lineage-Based Fault Tolerance:** Each RDD (Resilient Distributed Dataset) tracks the transformations that created it, known as its lineage. If a partition of an RDD is lost, Spark can recompute it from the original transformations, avoiding the need for data replication.

Catalyst Optimizer

For structured data, such as DataFrames and SQL queries, Spark uses the **Catalyst Optimizer** to enhance query performance. Catalyst is a powerful query optimizer that converts logical plans into optimized physical execution plans.

- **Logical Plan:** Initially, Spark converts the user's query into a logical plan.
- **Optimized Physical Plan:** Catalyst applies various optimization rules, such as predicate pushdown and join reordering, to create an efficient execution plan.

Catalyst enables Spark SQL and DataFrame operations to be highly efficient, often outperforming traditional SQL engines.

Execution Modes in Spark

Spark supports multiple **execution modes**, providing flexibility depending on the deployment environment:

1. **Local Mode:** Ideal for development and testing, it runs Spark on a single machine.
2. **Standalone Mode:** Runs Spark's built-in cluster manager, suitable for small or simple clusters.
3. **Cluster Mode (e.g., YARN, Kubernetes):** Spark runs on a cluster, and resources are managed by a cluster manager, such as YARN in Hadoop environments.

Example: In a production environment running on YARN, Spark applications operate in "cluster mode," where the driver can run on any node within the cluster, and YARN manages resources dynamically.

Spark Execution Flow: Putting It All Together

Let's walk through the execution flow of a Spark application with a simple example: **Word Count**.

Example Application: Word Count

1. **SparkSession Initialization:**
   The driver creates a `SparkSession`, which is the entry point for Spark operations.
2. **Data Loading:**
   The driver loads a text file into an RDD (Resilient Distributed Dataset).
3. **DAG Creation:**
   The driver builds a Directed Acyclic Graph (DAG) based on the transformations applied to the data:
   - Splitting each line into words.
   - Mapping each word to a `(word, 1)` pair.
   - Reducing by key to count the occurrences of each word.
4. **Stage and Task Division:**
   The DAG scheduler divides the job into stages and tasks, creating tasks for each partition of data.
5. **Task Execution:**
   - The cluster manager allocates resources and launches executors on worker nodes.
   - Tasks are sent to executors, which process the data in parallel.
6. **Result Collection:**
   After all tasks are completed, the driver aggregates the results.
7. **Data Persistence:**
   If needed, the results can be saved to a storage system like HDFS or a database.

This workflow illustrates how Spark processes large datasets efficiently using DAGs for optimized task execution and in-memory storage for faster computation.

Key Takeaways:

- **Driver Program:** The coordinator responsible for job submission and task coordination.
- **Executors:** Worker processes that perform tasks and store data in memory.
- **DAG Scheduler:** Optimizes execution by dividing jobs into parallelizable stages.
- **Cluster Manager:** Manages resources and allocates them to the driver and executors.

This architecture allows Spark to process data faster than traditional systems like MapReduce by keeping data in memory, optimizing execution plans, and minimizing disk I/O.

Setup Spark Environment, Thrift Server, Beeline, SHS - Nov 15

- **Spark Shells**
- **Spark Thrift Server**
- **Beeline CLI (Client for Spark Thrift Server)**
- **Jupyter Notebooks**
- **Setting Up Spark Locally (Mac / Linux)**
- **Understanding the Spark History Server (SHS)**
- **Event Logs: The Backbone of the Spark History Server**
- **Uses of Event Logs**

Unlike Hive, Spark offers several ways to write applications and interact with its system. The most common interfaces for interacting with Spark are as follows:

1. Spark Shells (Scala and Python)

- **Description:** Spark provides interactive shells for both Scala (`spark-shell`) and Python (`pyspark`). These shells allow users to execute Spark commands interactively, making them ideal for quick experiments, data exploration, and learning.
- **Primary Use Case:** Data exploration, testing, data science, and machine learning.
- **Usage:**
    - Start the Scala shell: `spark-shell`
    - Start the Python shell: `pyspark`
- **Features:** The shells come preconfigured with a `SparkContext` (`sc`) and `SparkSession` (`spark`), enabling users to start working with Spark immediately.

2. Spark Thrift Server

- **Description:** The Spark Thrift Server is a HiveServer2-compatible interface for Spark SQL. It allows Spark to accept SQL queries over JDBC and ODBC, which is useful for integrating Spark with Business Intelligence (BI) tools and other SQL-based applications.
- **Primary Use Case:** BI tool integration, SQL workloads.
- **Usage:**
    - Start the Thrift Server: `$SPARK_HOME/sbin/start-thriftserver.sh`
    - Connect to the Thrift Server using tools like Beeline or any BI software that supports JDBC/ODBC.
- **Features:** Supports multi-user Spark SQL workloads and is commonly used in data warehousing contexts for SQL-based querying.

3. Beeline CLI (Client for Spark Thrift Server)

- **Description:** Beeline is a command-line interface (CLI) that connects to the Spark Thrift Server, enabling users to run Spark SQL queries directly from the command line.
- **Primary Use Case:** Running SQL queries from the command line.
- **Usage:**
    - Connect to the Thrift Server using Beeline:
      `beeline -u "jdbc:hive2://<thrift_server_host>:<port>/default"`
    - Execute SQL queries:
      `SELECT * FROM table_name LIMIT 10;`
- **Features:** Ideal for ad-hoc SQL querying and connecting to Spark through JDBC.

4. Jupyter Notebooks

- **Description:** Jupyter Notebooks provide an interactive, web-based environment for writing Spark code in Python (via PySpark). These notebooks are often used for data analysis, machine learning, and prototyping.
- **Primary Use Case:** Data science, machine learning, and prototyping.
- **Usage:**
  - Configure PySpark with Jupyter by setting the environment variables:
    ```
    export PYSPARK_DRIVER_PYTHON=jupyter
    export PYSPARK_DRIVER_PYTHON_OPTS='notebook'
    ```
  - Start PySpark with Jupyter: `pyspark`
- **Features:** Jupyter offers a user-friendly interface with support for markdown, code, and inline visualizations, making it excellent for data analysis and presentation.

These interfaces allow you to interact with Spark in various ways, whether you're running ad-hoc queries, developing data science models, or integrating Spark into a larger system with SQL-based applications.

Setting Up Spark Locally

Follow these steps to set up Apache Spark on your local machine:

Step 1: Install Java

1. **Check if Java is installed:**
   Run: `java -version`
   - Spark requires **Java 8 or later**.
2. **Install Java (if not installed):**

   On Mac:
   Use Homebrew:
   ```
   brew install openjdk@11
   ```

   On Linux (Ubuntu/Debian-based):
   ```
   sudo apt update
   sudo apt install openjdk-11-jdk
   ```

3. **Set `JAVA_HOME`:**
   - Confirm the installation directory:
     - **Mac:** `usr/libexec/java_home -v 11`
     - **Linux:** `which java`
   - Add `JAVA_HOME` to your shell profile (`~/.bashrc` or `~/.zshrc`):

Mac:
```
export JAVA_HOME=$(/usr/libexec/java_home)
```
Linux:
```
export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64
```
Add this to the `PATH`:
```
export PATH=$JAVA_HOME/bin:$PATH
```
Apply the changes:
```
source ~/.bashrc  # or ~/.zshrc
```

Step 2: Download and Install Apache Spark

Download Spark:
Use `wget` to download the latest Spark version:
```
wget https://dlcdn.apache.org/spark/spark-3.4.4/spark-3.4.4-bin-hadoop3.tgz
```
Extract Files:
Use `tar` to extract the downloaded file:
```
tar -xvf spark-3.4.4-bin-hadoop3.tgz
```
Move the extracted folder to `/opt`:
```
sudo mv spark-3.4.4-bin-hadoop3 /opt/spark
```
Set Environment Variables:
Open your shell profile (`~/.bashrc` or `~/.zshrc`) and add the following:
```
export SPARK_HOME=/opt/spark
export PATH=$SPARK_HOME/bin:$SPARK_HOME/sbin:$PATH
```
Apply the changes:
```
source ~/.bashrc  # or ~/.zshrc
```

Step 3: Install Python and PySpark

1. **Check Python Installation:**
   Run: `python3 --version`
   - If Python 3 is not installed:

   On Mac:
   ```
   brew install python
   ```
   On Linux:
   ```
   sudo apt install python3
   ```

Install PySpark:
Use `pip` to install PySpark globally:
```
pip3 install pyspark
```

(Optional) Set Python Version for PySpark:
If multiple Python versions are installed, specify the version for PySpark in your shell profile:
```
export PYSPARK_PYTHON=python3
```

Step 4: Verify Installation and Start PySpark

Verify Spark Installation:
Run:
```
spark-shell
```

1. If the Scala Spark shell opens, Spark is installed correctly.

   Start PySpark in Interactive Mode:
   Run: `pyspark`

2. This opens an interactive PySpark session.

Run a Simple PySpark Test:
Test PySpark with the following command in the interactive session:

```
data = spark.sparkContext.parallelize([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
data.count()
```

3. If it returns `10`, your setup is complete.

Understanding the Spark History Server (SHS)

The **Spark History Server (SHS)** is a crucial tool for debugging and analyzing Spark applications. Its primary purpose is to help users understand execution metrics, diagnose issues, and optimize performance. With a user-friendly web interface, the SHS provides a historical view of completed applications, enabling users to:

- Analyze job performance.
- Identify inefficiencies and bottlenecks.
- Debug issues effectively.

As a **Spark application user**, familiarizing yourself with the SHS is essential for monitoring and improving your Spark jobs.

Event Logs: The Backbone of the Spark History Server

Event logs are critical for the functioning of SHS. These logs capture detailed information about the lifecycle of Spark jobs, such as:

- Stages and tasks.
- Memory usage.
- Errors and exceptions.

By leveraging event logs, users can:

1. **Reconstruct job execution** after completion.
2. **Understand performance characteristics** of applications.
3. **Troubleshoot issues** effectively.
4. **Optimize resource allocation** for better efficiency.

How Event Logs Work

1. **Generation:**
   Spark generates event logs during job execution and writes them to a specified directory, often in a distributed file system like HDFS or a local directory.
2. **Structure:**
   Event logs are JSON-formatted and contain detailed records of events, including:
   - Job and stage lifecycle.
   - Executor metrics (e.g., memory and CPU usage).
   - Task-level details, such as execution time and errors.
3. **Usage with SHS:**
   The Spark History Server reads these logs to display a visual representation of completed applications, providing insights into performance and potential areas for improvement.

The Spark History Server, powered by event logs, is an indispensable tool for monitoring and improving the performance of Spark applications. Understanding how to use SHS and interpret event logs can significantly enhance your ability to debug and optimize Spark jobs.

What Are Event Logs?

Event logs in Spark are JSON-based records that document key events during a Spark job's execution. These logs are generated by Spark's event logging system and provide detailed traces of an application's runtime behavior, enabling users to analyze job execution even after it has completed.

Event logs capture a wide range of information, including:

- **Jobs:** Start and completion times, job status, and errors (if any).
- **Stages:** Details such as dependencies, task counts, and shuffle operations.
- **Tasks:** Task duration, status (success or failure), resource usage, and errors.
- **Executors:** Metrics like memory usage, garbage collection (GC) time, and task distribution across executors.

How Event Logs Are Generated

To enable and configure event logging in Spark, the following steps are typically performed:

1. Configure Spark Properties

Set up the event logging properties in `spark-defaults.conf` or directly in your application code. The key properties are:

- **Enable Event Logging:**
  Set `spark.eventLog.enabled` to `true` to activate event logging.
- **Specify Log Directory:**
  Use the `spark.eventLog.dir` property to specify where event logs are stored. The directory can be on local storage, HDFS, S3, or any Hadoop-compatible file system.
- **Optional Compression:**
  Set `spark.eventLog.compress` to `true` to compress log files and save storage.

Example Configuration in `spark-defaults.conf`:

```
spark.eventLog.enabled true
spark.eventLog.dir hdfs://path/to/event/logs
spark.eventLog.compress true
```

How Event Logs Are Created

1. **Activation:**
   When `spark.eventLog.enabled` is `true`, Spark starts recording events during the application's runtime.
2. **Incremental Writing:**
   The driver continuously writes event details to log files in the specified directory as the application runs. This ensures that event data is available even if the application fails mid-execution.
3. **Compression (Optional):**
   If `spark.eventLog.compress` is enabled, the log files are compressed to minimize storage requirements.

Why Event Logs Are Useful

Event logs are crucial for:

- **Post-Execution Analysis:** Understand job behavior after execution.
- **Debugging:** Identify errors and inefficiencies.
- **Resource Optimization:** Analyze memory and CPU usage to improve performance.

Event logs, combined with tools like the **Spark History Server**, provide a comprehensive view of Spark job execution, making them indispensable for debugging and performance tuning.

Structure of Event Logs

Event logs in Spark are stored in JSON format, where each entry represents a specific event during a job's execution. The structure of these entries varies depending on the event type. Below are common event types and what they capture:

Common Event Types

- **`SparkListenerJobStart`**: Records job start details, including job ID, description, and associated stage IDs.
- **`SparkListenerJobEnd`**: Logs job completion information, such as job ID, end time, and completion status (success or failure).
- **`SparkListenerStageSubmitted`** **and** **`SparkListenerStageCompleted`**: Document stage start and completion details.
- **`SparkListenerTaskStart`** **and** **`SparkListenerTaskEnd`**: Capture the start and end of each task, with details like executor ID, task duration, and metrics.
- **`SparkListenerExecutorAdded`** **and** **`SparkListenerExecutorRemoved`**: Log the lifecycle of executors, such as when they are added to or removed from the cluster.

Sample JSON Event (Task Completion)

```
{
  "Event": "SparkListenerTaskEnd",
  "Stage ID": 1,
  "Stage Attempt ID": 0,
  "Task Type": "ResultTask",
  "Task Info": {
    "Task ID": 123,
    "Index": 1,
    "Attempt Number": 1,
    "Launch Time": 1625637893000,
```

```
    "Finish Time": 1625637894000,
    "Duration": 1000
  },
  "Task Metrics": {
    "Executor Run Time": 900,
    "JVM GC Time": 50,
    "Shuffle Read Bytes": 2048,
    "Shuffle Write Bytes": 4096,
    "Memory Bytes Spilled": 0
  }
}
```

Each log entry provides detailed information such as task duration, shuffle operations, and memory usage.

Uses of Event Logs

A. Performance Analysis and Optimization

Event logs enable users to analyze and optimize job execution by identifying performance bottlenecks.

- **Straggler Tasks**: Logs highlight tasks that take longer than others. These can indicate data skew or insufficient parallelism, which users can address by adjusting configurations.
- **Shuffle and I/O Bottlenecks**: Shuffle metrics help identify high shuffle or I/O costs. For example, tuning properties like `spark.sql.shuffle.partitions` or `spark.network.timeout` can improve performance.
- **Memory and CPU Usage**: Logs capture memory utilization, garbage collection times, and CPU usage, helping optimize executor configurations and resource allocation.

B. Debugging and Troubleshooting

Event logs are invaluable for identifying the root causes of failures.

- **Task Failures and Retries**: Logs show failed tasks, retry counts, and associated error messages.
- **Executor and Stage Failures**: If executors fail or stages are retried, logs provide insights into potential issues like resource constraints or network instability.
- **Error Messages**: Detailed stack traces are captured for quick debugging.

C. Post-Execution Analysis with Spark History Server

The **Spark History Server** processes event logs to provide a user-friendly interface for analyzing completed jobs. It allows users to:

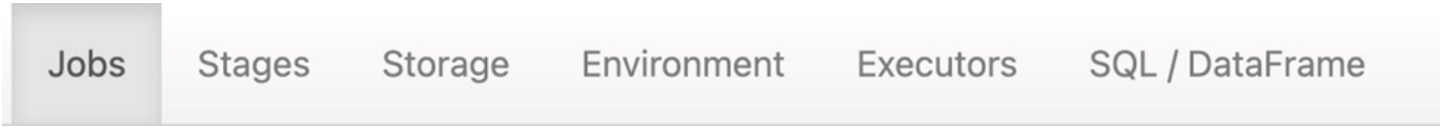
- **View Completed Jobs**: Drill into job details, including stages, tasks, and executor metrics.
- **Diagnose Performance Issues**: Analyze factors affecting performance, such as task durations and resource bottlenecks.
- **Compare Runs**: Assess execution times and resource usage across multiple runs to identify improvements or regressions.

D. Integration with Monitoring Tools

Event logs can be exported to monitoring platforms like **Grafana**, **Prometheus**, **Azure Monitor**, or **Databricks Job Monitor**.

- These tools provide centralized dashboards for job performance, cluster health, and resource trends, offering a comprehensive view of system behavior over time.

Understanding SHS UI



Jobs Tab Overview

The **Jobs Tab** provides a high-level summary of all jobs executed during a Spark application's runtime. It serves as the starting point for analyzing performance, troubleshooting, and understanding application behavior.

39	count at NativeMethodAccessorImpl.java:0 count at NativeMethodAccessorImpl.java:0	2024/11/10 06:10:44	1 s	2/2	2001/2001
38	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2024/11/10 06:10:43	61 ms	1/1	1/1
37	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2024/11/10 06:10:43	75 ms	1/1	1/1
36	localCheckpoint at NativeMethodAccessorImpl.java:0 localCheckpoint at NativeMethodAccessorImpl.java:0	2024/11/10 06:10:25	19 s	12/12	24000/24000
35	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2024/11/10 06:10:22	26 ms	1/1	4/4
34	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2024/11/10 06:10:22	21 ms	1/1	1/1
33	localCheckpoint at NativeMethodAccessorImpl.java:0 localCheckpoint at NativeMethodAccessorImpl.java:0	2024/11/10 06:10:21	1 s	1/1	2000/2000
32	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2024/11/10 06:10:20	57 ms	1/1	4/4
31	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2024/11/10 06:10:20	54 ms	1/1	1/1
30	localCheckpoint at NativeMethodAccessorImpl.java:0 localCheckpoint at NativeMethodAccessorImpl.java:0	2024/11/10 06:10:20	0.7 s	1/1	2000/2000
29	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2024/11/10 06:10:19	26 ms	1/1	4/4
28	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2024/11/10 06:10:19	29 ms	1/1	1/1
27	localCheckpoint at NativeMethodAccessorImpl.java:0 localCheckpoint at NativeMethodAccessorImpl.java:0	2024/11/10 06:10:19	0.7 s	1/1	2000/2000

Key Information

- **Job ID**: A unique identifier for each job in the application.
- **Description**: A brief summary of the job, often describing the triggered action (e.g., `count`, `collect`, `saveAsTable`).
- **Submission and Completion Times**: The timestamps when the job started and finished, useful for analyzing job duration.
- **Status**: Indicates the job's outcome—success, failure, or cancellation.
- **Stages**: Lists the stages associated with the job, with clickable links to explore stage-level details.
- **Duration**: The total time taken for the job to execute.

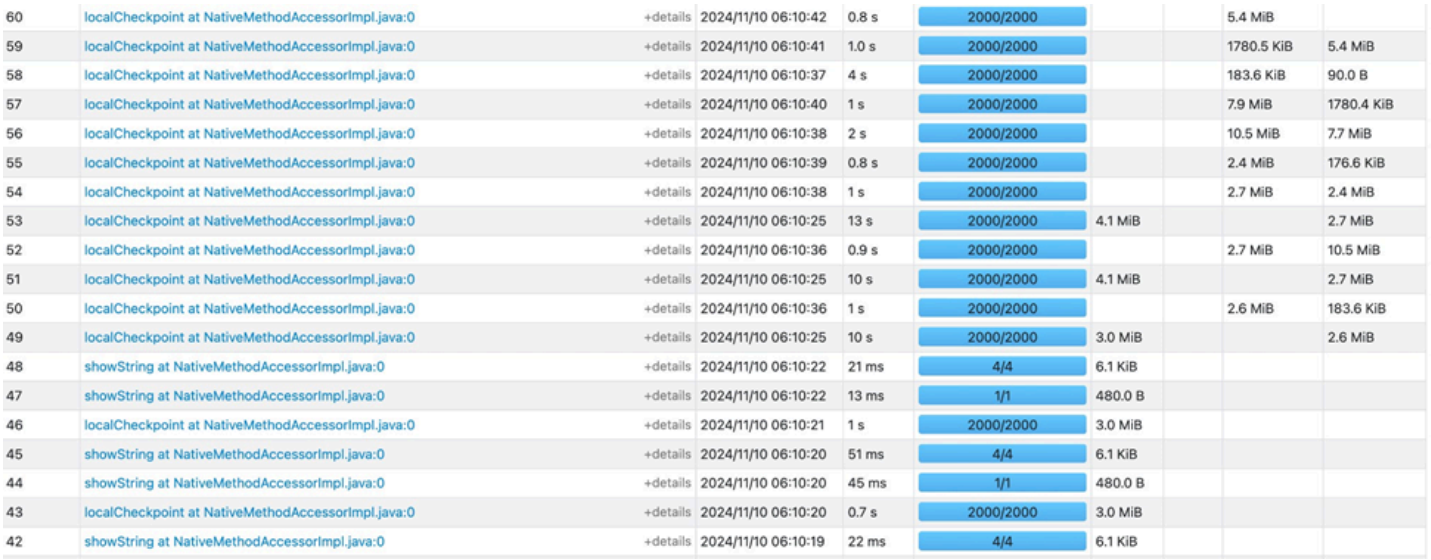
Use Cases

1. **Identifying Slow Jobs**
    - Sort jobs by duration to pinpoint the ones that took the longest.
    - Focus on these for performance tuning and optimization.
2. **Troubleshooting Failed Jobs**
    - Review error messages and stack traces for failed jobs.
    - Use this information to identify and resolve root causes.

The **Jobs Tab** is an essential tool for understanding the overall performance and diagnosing issues in Spark applications.

Stages Tab Overview

The **Stages Tab** provides a detailed view of a Spark application's execution plan by breaking it into stages. Each stage represents a group of parallel tasks that process data partitions, with new stages triggered by data shuffles.



60	localCheckpoint at NativeMethodAccessorImpl.java:0	+details	2024/11/10 06:10:42	0.8 s	2000/2000			5.4 MiB	
59	localCheckpoint at NativeMethodAccessorImpl.java:0	+details	2024/11/10 06:10:41	1.0 s	2000/2000			1780.5 KiB	5.4 MiB
58	localCheckpoint at NativeMethodAccessorImpl.java:0	+details	2024/11/10 06:10:37	4 s	2000/2000			183.6 KiB	90.0 B
57	localCheckpoint at NativeMethodAccessorImpl.java:0	+details	2024/11/10 06:10:40	1 s	2000/2000			7.9 MiB	1780.4 KiB
56	localCheckpoint at NativeMethodAccessorImpl.java:0	+details	2024/11/10 06:10:38	2 s	2000/2000			10.5 MiB	7.7 MiB
55	localCheckpoint at NativeMethodAccessorImpl.java:0	+details	2024/11/10 06:10:39	0.8 s	2000/2000			2.4 MiB	176.6 KiB
54	localCheckpoint at NativeMethodAccessorImpl.java:0	+details	2024/11/10 06:10:38	1 s	2000/2000			2.7 MiB	2.4 MiB
53	localCheckpoint at NativeMethodAccessorImpl.java:0	+details	2024/11/10 06:10:25	13 s	2000/2000	4.1 MiB			2.7 MiB
52	localCheckpoint at NativeMethodAccessorImpl.java:0	+details	2024/11/10 06:10:36	0.9 s	2000/2000			2.7 MiB	10.5 MiB
51	localCheckpoint at NativeMethodAccessorImpl.java:0	+details	2024/11/10 06:10:25	10 s	2000/2000	4.1 MiB			2.7 MiB
50	localCheckpoint at NativeMethodAccessorImpl.java:0	+details	2024/11/10 06:10:36	1 s	2000/2000			2.6 MiB	183.6 KiB
49	localCheckpoint at NativeMethodAccessorImpl.java:0	+details	2024/11/10 06:10:25	10 s	2000/2000	3.0 MiB			2.6 MiB
48	showString at NativeMethodAccessorImpl.java:0	+details	2024/11/10 06:10:22	21 ms	4/4	6.1 KiB			
47	showString at NativeMethodAccessorImpl.java:0	+details	2024/11/10 06:10:22	13 ms	1/1	480.0 B			
46	localCheckpoint at NativeMethodAccessorImpl.java:0	+details	2024/11/10 06:10:21	1 s	2000/2000	3.0 MiB			
45	showString at NativeMethodAccessorImpl.java:0	+details	2024/11/10 06:10:20	51 ms	4/4	6.1 KiB			
44	showString at NativeMethodAccessorImpl.java:0	+details	2024/11/10 06:10:20	45 ms	1/1	480.0 B			
43	localCheckpoint at NativeMethodAccessorImpl.java:0	+details	2024/11/10 06:10:20	0.7 s	2000/2000	3.0 MiB			
42	showString at NativeMethodAccessorImpl.java:0	+details	2024/11/10 06:10:19	22 ms	4/4	6.1 KiB			

Key Information

- **Stage ID**: A unique identifier for each stage.
- **Description**: Summary of transformations within the stage (e.g., `map`, `reduceByKey`).
- **Number of Tasks**: The total tasks in the stage, indicating the degree of parallelism.
- **Status**: Shows whether the stage succeeded, failed, or completed.
- **Task Metrics**: Includes metrics like input size, shuffle data read/write, and task duration.
- **Shuffle Data**: Highlights the volume of data read and written during shuffles—key for assessing shuffle performance.
- **GC Time**: Tracks garbage collection time, offering insights into memory efficiency.

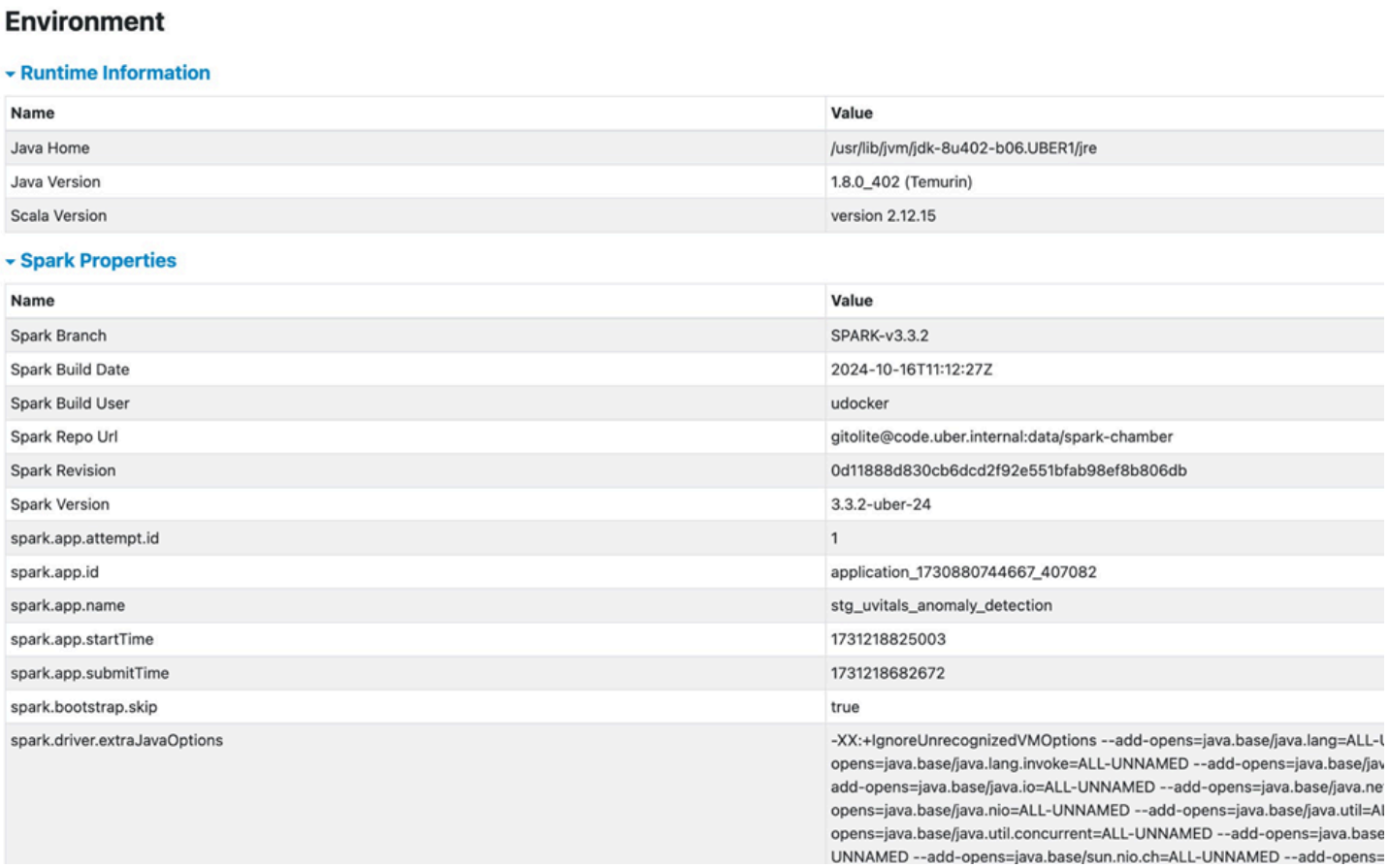
Use Cases

1. **Optimizing Shuffle Performance**
    - Analyze shuffle read/write metrics to detect excessive shuffling.
    - Adjust partitioning strategies to minimize data movement.
2. **Spotting Task Stragglers**
    - Identify tasks that take significantly longer than others.
    - Investigate data distribution and resource contention issues.
3. **Troubleshooting Stage Failures**
    - Drill into failed stages to review error messages.
    - Use this information to debug and resolve the root causes.

The **Stages Tab** is essential for fine-tuning performance and diagnosing stage-specific issues in Spark applications.

Environment Tab Overview

The **Environment Tab** provides a comprehensive view of the Spark application's configuration and runtime environment. It displays key properties, settings, and libraries that influence the application's execution.



Environment

▾ Runtime Information

Name	Value
Java Home	/usr/lib/jvm/jdk-8u402-b06.UBER1/jre
Java Version	1.8.0_402 (Temurin)
Scala Version	version 2.12.15

▾ Spark Properties

Name	Value
Spark Branch	SPARK-v3.3.2
Spark Build Date	2024-10-16T11:12:27Z
Spark Build User	udocker
Spark Repo Url	gitolite@code.uber.internal:data/spark-chamber
Spark Revision	0d11888d830cb6dcd2f92e551bfab98ef8b806db
Spark Version	3.3.2-uber-24
spark.app.attempt.id	1
spark.app.id	application_1730880744667_407082
spark.app.name	stg_uvitals_anomaly_detection
spark.app.startTime	1731218825003
spark.app.submitTime	1731218682672
spark.bootstrap.skip	true
spark.driver.extraJavaOptions	-XX:+IgnoreUnrecognizedVMOptions --add-opens=java.base/java.lang=ALL-U opens=java.base/java.lang.invoke=ALL-UNNAMED --add-opens=java.base/jav add-opens=java.base/java.io=ALL-UNNAMED --add-opens=java.base/java.ne opens=java.base/java.nio=ALL-UNNAMED --add-opens=java.base/java.util=AL opens=java.base/java.util.concurrent=ALL-UNNAMED --add-opens=java.base UNNAMED --add-opens=java.base/sun.nio.ch=ALL-UNNAMED --add-opens=

Key Sections

1. **Spark Properties -** Shows Spark-specific configurations such as memory settings, shuffle settings, and event logging configurations.
2. **Hadoop Properties -** Lists Hadoop-related configurations when Spark runs on a Hadoop cluster.
3. **System Properties -** Displays Java system properties set at runtime.
4. **Classpath Entries -** Lists all libraries and dependencies available in the application’s classpath, aiding in debugging dependency issues.

5. **Runtime Libraries -** Provides a list of JAR files loaded during the application's execution, useful for identifying version mismatches or conflicts.

Use Cases

1. **Configuration Troubleshooting -** Verify critical settings like `spark.executor.memory`, `spark.sql.shuffle.partitions`, and `spark.dynamicAllocation.enabled` to ensure proper configuration.
2. **Dependency Management -** Confirm required libraries are loaded correctly and resolve runtime errors caused by missing or incompatible dependencies.
3. **Verifying Tuning Parameters -** Check memory and parallelism settings to confirm alignment with performance optimization strategies.

The **Environment Tab** is essential for ensuring that the Spark application is running in the expected environment and for diagnosing configuration or dependency-related issues.

Executors Tab Overview

The **Executors Tab** provides detailed metrics for each executor, offering insights into performance, resource utilization, and task execution. Executors are responsible for running tasks and managing data storage in memory or on disk.

Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Logs
Active	0	0.0 B / 6.2 GiB	0.0 B	0	0	0	0	0	3.7 min (0.0 ms)	0.0 B	0.0 B	0.0 B	stdout stderr
Active	0	0.0 B / 4.1 GiB	0.0 B	1	0	0	767	767	2.0 min (2 s)	2.5 MiB	300.7 KiB	793 KiB	stdout stderr
Active	0	0.0 B / 4.1 GiB	0.0 B	1	0	0	742	742	2.0 min (1 s)	2.3 MiB	372.9 KiB	691 KiB	stdout stderr
Active	0	0.0 B / 4.1 GiB	0.0 B	1	0	0	1207	1207	1.8 min (0.6 s)	2.4 MiB	576.8 KiB	767.2 KiB	stdout stderr
Active	0	0.0 B / 4.1 GiB	0.0 B	1	0	0	981	981	1.9 min (1 s)	2.5 MiB	428.1 KiB	801.6 KiB	stdout stderr
Active	0	0.0 B / 4.1 GiB	0.0 B	1	0	0	687	687	1.9 min (2 s)	956 KiB	299.1 KiB	601.2 KiB	stdout stderr
Active	0	0.0 B / 4.1 GiB	0.0 B	1	0	0	1026	1026	1.9 min (0.5 s)	2.4 MiB	378.5 KiB	693.1 KiB	stdout stderr
Active	0	0.0 B / 4.1 GiB	0.0 B	1	0	0	1082	1082	1.8 min (0.8 s)	2.6 MiB	434.2 KiB	742.2 KiB	stdout stderr
Active	0	0.0 B / 4.1 GiB	0.0 B	1	0	0	1110	1110	1.9 min (0.8 s)	2.9 MiB	604.6 KiB	818.3 KiB	stdout stderr
Active	0	0.0 B / 4.1	0.0 B	1	0	0	988	988	1.8 min	1.3 MiB	454.7 KiB	826.6 KiB	stdout

Key Information

1. **Executor ID -** A unique identifier for each executor in the cluster.
2. **Host -** The machine where the executor is running.
3. **Total Tasks -** The number of tasks assigned to the executor throughout the job.
4. **Task Distribution -** Metrics on task execution, including task count, duration, and success rate.
5. **Storage Memory -** Memory used for storing data, such as cached RDDs, helping pinpoint memory-related bottlenecks.
6. **Disk Spilled -** The amount of data written to disk when memory is insufficient, which can slow job execution.
7. **JVM Garbage Collection (GC) Time -** Time spent in garbage collection, indicating memory management efficiency.

8. **Shuffle Read/Write Metrics -** Amount of shuffle data processed, helping identify performance impacts due to data shuffling.

Use Cases

1. **Balancing Resource Usage -** Analyze task distribution and memory utilization to detect under- or over-utilized executors and adjust configurations accordingly.
2. **Identifying Memory Issues -** High disk spill or excessive GC time indicates insufficient memory allocation. Solutions may include increasing `spark.executor.memory` or optimizing data partitioning.
3. **Debugging Executor Failures -** Use metrics to investigate executor failures, such as resource contention, excessive data processing, or hardware issues.

The **Executors Tab** is essential for monitoring resource usage, diagnosing performance issues, and optimizing executor configurations for efficient Spark job execution.

SQL/DataFrames Tab Overview

The **SQL/DataFrames Tab** focuses on metrics related to Spark SQL and DataFrame operations, offering insights into query performance, execution plans, and data movement. This is particularly useful for analyzing complex SQL queries or DataFrame operations in Spark's distributed environment.

8	count at NativeMethodAccessorImpl.java:0 +details	2024/11/10 06:09:46	16 s	[18]
7	isEmpty at NativeMethodAccessorImpl.java:0 +details	2024/11/10 06:09:46	0.2 s	[16][17]
6	showString at NativeMethodAccessorImpl.java:0 +details	2024/11/10 06:09:45	0.5 s	[12][13][14][15]
5	localCheckpoint at NativeMethodAccessorImpl.java:0 +details	2024/11/10 06:09:40	6 s	[11]
4	showString at NativeMethodAccessorImpl.java:0 +details	2024/11/10 06:09:38	1 s	[8][9][10]
3	localCheckpoint at NativeMethodAccessorImpl.java:0 +details	2024/11/10 06:08:16	1.4 min	[7]
2	showString at NativeMethodAccessorImpl.java:0 +details	2024/11/10 06:08:15	1.0 s	[4][5][6]
1	showString at NativeMethodAccessorImpl.java:0 +details	2024/11/10 06:08:02	13 s	[1][2][3]

Key Information

1. **Query**
   - The SQL query string or DataFrame operation executed by the application.
2. **Execution Time**
   - The total time taken to execute the query, giving a high-level view of its performance.
3. **Job and Stage Links**
   - Direct links to jobs and stages associated with the query for deeper analysis of execution details.
4. **Physical Plan**
   - The execution plan detailing transformations such as `Filter`, `Project`, and `Aggregate` that Spark performs.
5. **Logical Plan**
   - The initial representation of the query before optimization, useful for understanding query intent and transformations.
6. **Optimized Plan**
   - The improved query plan after Spark's Catalyst optimizer applies rules to enhance performance.
7. **Execution Plan Metrics**
   - Metrics for individual operators (e.g., joins, aggregates) with details like shuffle read/write sizes, row counts, and processing times.

Use Cases

1. **Query Optimization**
    - Analyze the physical and optimized plans to identify costly operations (e.g., wide shuffles).
    - Adjust query structures or configurations, such as `spark.sql.shuffle.partitions`, to enhance performance.
2. **Identifying SQL Bottlenecks**
    - Pinpoint slow joins, filters, or aggregations that may require optimization or tuning.
3. **Debugging Query Failures**
    - Investigate query plans and error messages to diagnose issues with complex joins, aggregations, or transformations.

The **SQL/DataFrames Tab** is an essential tool for users aiming to refine query performance, resolve inefficiencies, and debug SQL operations within Spark.

Data Processing With Spark: Dataframe 1 - Nov 18

- **Data Sources**
- **Data Ingestion**
- **Data Storage**

In earlier lectures, we saw a small example of how a Spark application is written. In Hive, we used to write queries to get the data we needed. However, queries have limitations. Spark solves these by allowing not only query execution but also features like machine learning (MLlib), connecting to multiple data sources at the same time, and more. These things aren't possible with query-based tools alone.

In Spark, this is done using **DataFrames** and **RDDs**. So, let's first understand what DataFrames and RDDs are.

A **DataFrame** in Spark is a distributed and unchangeable collection of data, organized into named columns, similar to a table in a database or an Excel sheet. It's one of the main tools in Spark's SQL module. DataFrames provide a simpler and more efficient way to handle structured or semi-structured data than RDDs.

Key Features of a DataFrame:

1. **Tabular Structure:**
   DataFrames have a defined structure with named columns and specific data types for each column. This makes it easier to work with and query the data.
2. **Distributed Data:**
   DataFrames are split into parts (called partitions) and stored across multiple nodes in a Spark cluster. This allows Spark to process the data in parallel automatically.
3. **Immutable and Lazy Evaluation:**
    - Once you create a DataFrame, you cannot change it (immutable). If you perform an operation, it creates a new DataFrame.
    - Spark uses **lazy evaluation**, which means it waits to execute operations until you perform an action (like `show()` or `count()`).
4. **Optimizations:**
   DataFrames are optimized using the **Catalyst Optimizer**, which creates the best plan for executing transformations. They also use **Tungsten** for in-memory computations, making them faster and more efficient than RDDs.

Creating DataFrames

You can create DataFrames in Spark from different data sources like:

1. **External Data Sources:**
   Spark supports formats such as JSON, CSV, Parquet, Avro, and ORC. You can also create DataFrames from databases like MySQL and PostgreSQL using JDBC.

   From a JSON file:
   ```
   df = spark.read.json("path/to/file.json")
   ```
   From a CSV file:
   ```
   df = spark.read.csv("path/to/file.csv", header=True, inferSchema=True)
   ```
   From a Parquet file:
   ```
   df = spark.read.parquet("path/to/file.parquet")
   ```

2. **From RDDs:**
   An existing RDD (Resilient Distributed Dataset) can be converted into a DataFrame by providing a schema or letting Spark infer it automatically.

   Example:
   ```
   rdd = spark.sparkContext.parallelize([(1, "Alice"), (2, "Bob")])
   df = rdd.toDF(["id", "name"])
   ```

3. **Using a Structured Schema:**
   You can define a schema using `StructType` and `StructField`, then use it to create a DataFrame.

   Example:
   ```
   from pyspark.sql.types import StructType, StructField, IntegerType,
   StringType

   schema = StructType([
       StructField("id", IntegerType(), True),
       StructField("name", StringType(), True)
   ])

   df = spark.createDataFrame([(1, "Alice"), (2, "Bob")], schema)
   ```

These methods allow flexibility when working with data in Spark. Choose the one that best fits your data source!

Few DataFrame APIs

DataFrames provide many functions for working with and analyzing data. Here are some of the key operations:
Selecting Columns: You can select specific columns using the `.select()` method or by referencing them directly.

```
# Select specific columns
df.select("id", "name").show()
```

Filtering Rows:
Use `.filter()` or `.where()` to filter rows based on conditions.

```
# Filter rows where 'id' is greater than 1
df.filter(df.id > 1).show()
```

Aggregations and Grouping:
Perform aggregations and group data like SQL's `GROUP BY` with functions such as `SUM` and `COUNT`.

```
# Group by 'name' and count occurrences
df.groupBy("name").count().show()
```

Joining DataFrames:
DataFrames support joins like inner, left, and right joins.

```
# Join two DataFrames on the 'id' column
df1.join(df2, df1.id == df2.id, "inner").show()
```

Sorting and Ordering:
Sort data by specific columns in ascending or descending order.

```
# Sort by 'id' in descending order
df.orderBy(df.id.desc()).show()
```

These APIs make it easy to perform operations on DataFrames, similar to SQL queries. You can use them for advanced data manipulation and analysis.

SQL API in DataFrames

While DataFrames have powerful APIs to perform operations similar to SQL, Spark also provides SQL APIs to make it easier for SQL users to transition to Spark. This simplifies the process and allows seamless integration with SQL workflows.

Understanding Spark SQL

- **Overview:**
  Spark SQL is a module in Spark that lets you query structured data using SQL. It allows smooth interaction with data from sources like Hive, JSON, Parquet, and more. Spark SQL helps manage and query DataFrames, which are collections of data organized into named columns and distributed across a cluster.
- **Key Benefit:**
  Spark SQL combines the simplicity of SQL with the scalability of Spark. This makes it ideal for large-scale data processing while maintaining SQL's declarative approach.

DataFrame API vs. SQL API

- **DataFrame API:**
  DataFrames are similar to tables in relational databases but optimized for distributed processing. You can use methods like `.select()`, `.filter()`, `.groupBy()`, and `.join()` for operations.
- **SQL API:**
  The SQL API lets you run SQL queries directly on DataFrames using standard SQL syntax. This is helpful for those familiar with SQL or for handling complex SQL-based analytics.

By offering both APIs, Spark gives flexibility to users, enabling them to choose the approach that fits their needs.

Registering a DataFrame as a Temporary Table

To run SQL queries on a DataFrame, you need to register it as a temporary view or table. This allows Spark SQL to treat the DataFrame like a table in a database.

Temporary Views

- **What They Are:**
  Temporary views exist only during the Spark session and are discarded when the session ends.

  Example:
  ```
  # Load data into a DataFrame
  df = spark.read.json("path/to/file.json")

  # Register the DataFrame as a temporary view
  df.createOrReplaceTempView("people")

  # Query the DataFrame using SQL
  spark.sql("SELECT name, age FROM people WHERE age > 21").show()
  ```

Global Temporary Views

- **What They Are:**
  Global temporary views are similar to temporary views but can be accessed across multiple Spark sessions. They are tied to the Spark application and use the `global_temp` database for referencing.

  Example:
  ```
  # Register the DataFrame as a global temporary view
  df.createOrReplaceGlobalTempView("global_people")

  # Query the global temporary view
  spark.sql("SELECT name, age FROM global_temp.global_people WHERE age >
  21").show()
  ```

Running SQL Queries with `spark.sql()`

The `spark.sql()` function lets you directly execute SQL queries on DataFrames. This is particularly useful for complex operations or when combining multiple SQL functions in a single query.

Example:
```
# Run a SQL query to calculate the average age by city
result_df = spark.sql("SELECT city, AVG(age) AS avg_age FROM people GROUP BY
city")

# Show the results
result_df.show()
```

This approach bridges the gap between SQL users and Spark's powerful data processing capabilities.

Querying External Data Sources

Explanation:
Spark SQL allows you to query data from different sources like JSON, Parquet, ORC, Avro, and Hive tables. It hides the complexity of the underlying data format, enabling you to use SQL queries seamlessly.

Example:

python
Copy code

```
# Read a Parquet file into a DataFrame
parquet_df = spark.read.parquet("path/to/parquet_file")

# Register the DataFrame as a temporary view
parquet_df.createOrReplaceTempView("parquet_table")

# Query using SQL
spark.sql("SELECT * FROM parquet_table WHERE column1 > 50").show()
```

SQL UDFs (User-Defined Functions)

Explanation:
Spark SQL supports custom functions called User-Defined Functions (UDFs) that you can write in Python, Scala, or Java. UDFs are helpful for applying custom logic not available in built-in SQL functions.

Example:

```
from pyspark.sql.functions import udf
from pyspark.sql.types import IntegerType

# Define a Python function
def add_one(value):
    return value + 1

# Register the function as a UDF
spark.udf.register("addOne", add_one, IntegerType())

# Use the UDF in a SQL query
spark.sql("SELECT name, addOne(age) AS age_plus_one FROM people").show()
```

Detailed Steps:

Import UDF Functionality:

```
from pyspark.sql.functions import udf
from pyspark.sql.types import IntegerType
```

- **`udf`:** Converts a Python function into a Spark function for DataFrames and SQL queries.
- **`IntegerType`:** Defines the return type of the UDF to ensure Spark processes data efficiently.

Define the Python Function:

```
def add_one(value):
    return value + 1
```

- **`def add_one(value):`** A simple function that takes an integer as input.

- **`return value + 1`:** Adds 1 to the input value and returns the result.

Register the UDF with Spark:

```
spark.udf.register("addOne", add_one, IntegerType())
```

- **`"addOne":** The name used to call the function in SQL queries.
- **`add_one`:** The actual Python function being registered.
- **`IntegerType`:** Specifies the function's return type (integer in this case).

Use the UDF in SQL Queries:
Once registered, the UDF can be used like any other SQL function:

```
SELECT name, addOne(age) AS age_plus_one FROM people
```

This makes it possible to apply custom transformations directly within SQL queries.

Aggregations and Window Functions

Overview:
Spark SQL provides powerful aggregation functions (e.g., `SUM`, `AVG`, `MIN`, `MAX`) for summarizing data, as well as advanced window functions (`ROW_NUMBER`, `RANK`, `DENSE_RANK`, `LEAD`, `LAG`) for analytical queries.

Examples:

Aggregation Example:
Use `GROUP BY` to summarize data:

```
spark.sql("SELECT city, COUNT(*) AS count FROM people GROUP BY city").show()
```

Window Function Example:
Window functions allow row-wise calculations based on a specific partition of the data:

```
window_query = """
SELECT
    name,
    age,
    city,
    ROW_NUMBER() OVER (PARTITION BY city ORDER BY age DESC) AS rank
FROM people
"""
spark.sql(window_query).show()
```

- **`PARTITION BY city`:** Groups rows by city.
- **`ORDER BY age DESC`:** Orders rows within each city group by age (descending).
- **`ROW_NUMBER()`:** Assigns a unique rank to each row within its partition.

Integration with Hive

Overview:
Spark SQL can read and write Hive tables directly, enabling seamless interaction with existing Hive-based data pipelines. This is especially useful for teams heavily using Hive as their storage layer.

Steps:

Enable Hive Support in Spark:
Add Hive integration during Spark session creation:

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

Query a Hive Table:
Use SQL queries to access Hive data:

```
spark.sql("SELECT * FROM hive_table WHERE column1 > 10").show()
```

- `enableHiveSupport`**:** Ensures Spark can interact with Hive.
- **SQL Queries:** Allow the same syntax as Hive for data manipulation.

By combining these features, Spark SQL supports both traditional aggregations and sophisticated analytics while integrating smoothly with Hive.

Optimizations with Catalyst and Tungsten

1. **Catalyst Optimizer:**
   Spark SQL uses **Catalyst**, an advanced query optimizer that enhances query performance by:
   - **Predicate Pushdown:** Filters data early to minimize data processing.
   - **Column Pruning:** Reads only necessary columns to reduce memory usage.
   - **Join Reordering:** Arranges joins to optimize execution.
2. **Tungsten Execution Engine:**
   - Works alongside Catalyst to optimize **memory** and **CPU usage**.
   - Improves Spark SQL performance with efficient data serialization and in-memory computation.

Note: These optimizations happen automatically; no additional code is required.

Working with JDBC and External Databases

Spark SQL can connect to relational databases (e.g., MySQL, PostgreSQL, Oracle) using **JDBC**. This allows querying external databases and leveraging Spark's distributed processing for analytics.

Steps to Connect
Initialize the JDBC DataFrame Reader:
1. `jdbc_df = spark.read.format("jdbc")`

Set the Database Connection URL:
2. `.option("url", "jdbc:mysql://localhost:3306/database")`
URL Format:

```
jdbc:mysql://<hostname>:<port>/<database>
```

- `localhost`**:** Hostname of the database. Replace with the actual server's IP or domain if needed.
- `3306`**:** Default MySQL port. Update if your database uses a different port.
- `database`**:** Name of the target database.

Specify the Table or Query:
3. `.option("dbtable", "table_name")`

- **Table Reference:** Loads a specific table (`table_name`).

Custom SQL Query:
```
.option("dbtable", "(SELECT * FROM table_name WHERE column1 > 100) AS alias_name")
```

2. **Provide Credentials:**

   Username:
   `.option("user", "username")`
   Password:
   `.option("password", "password")`

3. **Security Tip:** Use environment variables or secure secrets managers to avoid hardcoding sensitive information.

Load the Data:
6. `.load()`

4. This initializes the connection but only fetches data when an action (e.g., `.show()` or `.count()`) is performed.

Complete Example
```
jdbc_df = spark.read.format("jdbc") \
    .option("url", "jdbc:mysql://localhost:3306/test") \
    .option("dbtable", "my_table") \
    .option("user", "root") \
    .option("password", "password") \
    .load()

# Display the data
jdbc_df.show()
```

With this setup, Spark seamlessly integrates with external databases, enabling efficient data querying and processing.

Registering a DataFrame as a Temporary View

Spark SQL allows you to register a DataFrame as a **temporary view**, enabling SQL queries directly on the DataFrame.

Key Method: `createOrReplaceTempView`
```
jdbc_df.createOrReplaceTempView("jdbc_table")
```

1. **Functionality:**
   - Registers the `jdbc_df` DataFrame as a temporary view named `"jdbc_table"`.
   - Lets you query the DataFrame using SQL, similar to a table in a database.
2. **Scope:**
   - **Session-scoped:** The temporary view is valid only for the current Spark session. It is removed once the session ends.

3. **Overwriting Views:**
    - If a view with the same name exists, `createOrReplaceTempView` replaces it automatically.

Running SQL Queries on the Temporary View

You can run SQL queries on the registered view using `spark.sql()`.

Example Query:

```
spark.sql("SELECT * FROM jdbc_table WHERE column1 > 100").show()
```

1. **Query Breakdown:**
    - `SELECT *`: Retrieves all columns.
    - `FROM jdbc_table`: Specifies the temporary view to query.
    - `WHERE column1 > 100`: Filters rows where `column1` values exceed 100.
2. **Execution:**
    - `spark.sql()`: Executes the SQL query on the view.
    - `.show()`: Displays the query result in the console and triggers the actual execution.

Complete Example

Here's how to load data from a database, register it as a view, and query it using Spark SQL:

```
from pyspark.sql import SparkSession

# Initialize Spark session
spark = SparkSession.builder.appName("JDBC Example").getOrCreate()

# Define the JDBC DataFrame
jdbc_df = spark.read.format("jdbc") \
    .option("url", "jdbc:mysql://localhost:3306/my_database") \
    .option("dbtable", "my_table") \
    .option("user", "my_user") \
    .option("password", "my_password") \
    .load()


# Register as a temporary view
jdbc_df.createOrReplaceTempView("jdbc_table")

# Run SQL query
result_df = spark.sql("SELECT * FROM jdbc_table WHERE column1 > 100")

# Show the result
result_df.show()
```

Explanation of the Workflow

1. **Load Data:**
    - Reads data from a MySQL table into a Spark DataFrame using JDBC.
2. **Register View:**
    - The DataFrame is registered as a temporary view named `"jdbc_table"`.
3. **Run Query:**
    - A SQL query is executed on the temporary view to filter and process data.
4. **Display Results:**
    - Results are displayed using `.show()`, which triggers execution and fetches data.

This workflow efficiently integrates external databases with Spark for SQL-based analytics.

Notes and Best Practices for Spark SQL with JDBC

1. Connection Configuration

- **URL Parameters:**
    - Include parameters like connection timeouts, character encodings, and SSL settings as needed by your database.

Example:

```
.option("url",
"jdbc:mysql://localhost:3306/database?useSSL=true&connectTimeout=10000")
```

- **Driver Dependency:**
    - Ensure the JDBC driver is available. For MySQL, use the MySQL JDBC Connector.

Add the `.jar` file to your Spark session using:

```
spark-submit --jars /path/to/mysql-connector-java.jar ...
```

---

2. Performance Considerations

- **Partitioning:**

By default, Spark reads the table as a single partition. For large datasets, enable parallelism:

```
.option("numPartitions", "10") \
.option("partitionColumn", "id") \
.option("lowerBound", "1") \
.option("upperBound", "1000")
```

    - **Parameters Explained:**
        - `partitionColumn`: Column used for dividing data.
        - `lowerBound`/`upperBound`: Range of values in the partition column.
        - `numPartitions`: Number of partitions to create.

---

3. Security

- **Credentials:**
    - Avoid hardcoding sensitive information like usernames and passwords.

Use environment variables or a secrets manager. Example:

```
.option("user", os.getenv("DB_USER")) \
.option("password", os.getenv("DB_PASSWORD"))
```

---

4. Querying Large Tables

- **Select Specific Columns:**
  - Avoid `SELECT *`. Fetch only the required columns to minimize data transfer and improve performance.

Example:

```
.option("dbtable", "(SELECT column1, column2 FROM table_name WHERE column1 > 100) AS filtered_table")
```

---

5. Loading Data Incrementally

- **Incremental Loading:**
  - For dynamic tables, load data incrementally using filters like timestamps.

Example:

```
.option("dbtable", "(SELECT * FROM table_name WHERE updated_at > '2024-01-01') AS incremental_data")
```

Summary

Connecting Spark with external databases via JDBC enables seamless integration of Spark's distributed processing with traditional databases. Key benefits:

- Scalable ETL (Extract, Transform, Load).
- Enhanced analytics workflows.

By following best practices, such as secure credential storage, partitioning, and incremental data loading, you can ensure efficient and secure operations while working with large datasets.

==========

Thanks You

==========

Contact Me

If you have any questions or suggestions regarding the notes, please feel free to reach out to me on WhatsApp: **Sanjay Yadav Phone: 8310206130**

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

https://www.linkedin.com/in/yadav-sanjay/ || https://instagram.com/@geekysanjay