1. What is a data warehouse, explain ETL,OLAP and OLTP ?

Data Warehouse

A data warehouse is a centralized repository of integrated data from various sources within an organization. It is designed to support decision-making by providing a consistent and reliable platform for analysis. Unlike operational databases (OLTP), data warehouses focus on historical data and are optimized for querying and analysis rather than transaction processing.

Key characteristics:

- Subject-oriented: Data is organized around subjects (e.g., customers, products, sales).
- Integrated: Data from multiple sources is integrated into a consistent format.
- Time-variant: Data includes historical information for trend analysis.
- Non-volatile: Data is read-only, ensuring data integrity.

ETL (Extract, Transform, Load)

ETL is a process used to extract data from various sources, transform it into a suitable format, and load it into a data warehouse. It's a crucial component in building and maintaining a data warehouse.

Key steps:

- **Extract:** Data is retrieved from different sources (databases, files, APIs).
- **Transform:** Data is cleaned, standardized, and transformed to match the data warehouse schema.
- **Load:** Transformed data is loaded into the data warehouse.

OLAP (Online Analytical Processing)

OLAP is a technology that allows users to analyze multidimensional data from different perspectives. It is typically used in conjunction with data warehouses to provide interactive exploration and analysis capabilities.

Key characteristics:

- Multidimensional data: Data is organized in a cube-like structure with multiple dimensions (e.g., time, product, region).
- Slicing and dicing: Users can quickly analyze data by drilling down, roll-up, and pivoting.
- Fast response time: OLAP systems are optimized for complex queries and calculations.

OLTP (Online Transaction Processing)

OLTP systems are designed to handle large numbers of short online transactions efficiently. They are used for day-to-day business operations such as order processing, banking transactions, and inventory management.

Key characteristics:

- High transaction volume: Handles a large number of concurrent transactions.
- Short transaction duration: Transactions are completed quickly.
- Data consistency: Ensures data integrity through ACID properties (Atomicity, Consistency, Isolation, Durability).

- Operational focus: Supports day-to-day business operations.
  Relationship Between Data Warehouse, ETL, OLAP, and OLTP
- **OLTP systems** generate transactional data.
- **ETL processes** extract, transform, and load this data into a **data warehouse**.
- The **data warehouse** stores integrated and historical data.
- **OLAP systems** utilize the data warehouse for analysis and reporting.
  In essence, OLTP systems provide the raw data, ETL prepares the data for analysis, the data warehouse stores the prepared data, and OLAP enables users to explore and analyze the data for decision-making.

2. In Hive, what are the important parameters to configure the data warehouse ?

Configuring Hive effectively is crucial for optimal performance and resource utilization.
Here are some key parameters to consider:

General Configuration

- **hive.exec.reducers:** Determines the number of reducers used for MapReduce jobs. This impacts query performance and resource utilization.
- **hive.exec.parallel:** Enables parallel execution of map and reduce tasks for improved performance.
- **hive.mapred.local.dir:** Specifies the local directory for temporary files. Adjust based on available disk space.
- **hive.metastore.uris:** Defines the location of the Hive Metastore.
- **hive.execution.engine:** Determines the execution engine (Tez, Spark, MR).

Storage and Data Format

- **hive.input.format:** Specifies the input format for data (e.g., TextInputFormat, SequenceFileInputFormat).
- **hive.output.format:** Specifies the output format for data (e.g., TextOutputFormat, SequenceFileOutputFormat).
- **hive.compression.codec:** Enables compression for data stored in HDFS (e.g., Snappy, Gzip).

Query Optimization

- **hive.optimize.rewrite:** Enables query optimization techniques like constant folding, predicate pushdown, and join reordering.
- **hive.vectorized.execution.enabled:** Enables vectorized execution for improved query performance.
- **hive.stats.autogather:** Enables automatic statistics collection for better query optimization.
- **hive.cbo.enable:** Enables cost-based optimizer for making intelligent query execution plans.

Resource Management

- **mapred.job.tracker:** Specifies the location of the JobTracker in a Hadoop cluster.
- **mapred.reduce.tasks:** Sets the default number of reducers for MapReduce jobs.
- **yarn.nodemanager.resource.memory-mb:** Specifies the amount of memory available to each NodeManager in YARN.
- **yarn.app.mapreduce.am.resource.mb:** Specifies the amount of memory for the MapReduce ApplicationMaster.

Other Important Parameters

- **hive.support.concurrency:** Enables concurrent execution of multiple queries.
- **hive.fetch.task.conversion:** Controls the conversion of fetch tasks to map tasks for better performance.
- **hive.exec.dynamic.partition.mode:** Configures dynamic partition insertion behavior.
  Additional Considerations
- **Performance Testing:** Conduct thorough performance testing to fine-tune parameters based on your workload.
- **Monitoring:** Monitor system metrics and query performance to identify bottlenecks and areas for improvement.
- **Best Practices:** Follow Hive best practices for data modeling, partitioning, and indexing.

3. How can you create a DataFrame a) using existing RDD, and b) from a CSV file?

a) From an Existing RDD

To create a DataFrame from an existing RDD, we'll use the toDF() method. This method converts an RDD of tuples or Rows into a DataFrame.

Python

```
from pyspark.sql import SparkSession

# Create a SparkSession
spark = SparkSession.builder.appName("CreateDataFrame").getOrCreate()
# Sample RDD
data = [("Alice", 25), ("Bob", 30), ("Charlie", 35)]
rdd = spark.sparkContext.parallelize(data)

# Convert RDD to DataFrame
df = rdd.toDF(["name", "age"])

# Show the DataFrame
df.show()
```

b) From a CSV File

PySpark provides a convenient way to create DataFrames directly from CSV files using the read.csv() method.
Python

```
from pyspark.sql import SparkSession

# Create a SparkSession
spark = SparkSession.builder.appName("CreateDataFrame").getOrCreate()
# Read CSV file
df = spark.read.csv("path/to/your/file.csv", header=True, inferSchema=True)

# Show the DataFrame
df.show()
```

4. explain HDFS Architecture

HDFS (Hadoop Distributed File System) is a distributed file system designed to store large datasets across multiple commodity hardware. It is highly fault-tolerant and optimized for data access.

Key Components

- **NameNode**:
  - Master node responsible for managing the file system namespace.
  - Maintains metadata about files, blocks, and their locations.
  - Handles client requests for file system operations.
  - Performs namespace operations like creating, deleting, and renaming files.
  - Determines the location of data blocks.
- **DataNode**:
  - Slave nodes responsible for storing data blocks.
  - Reports block information to the NameNode.
  - Executes read and write requests from clients.
  - Handles block replication and deletion.
- **Secondary NameNode**:
  - Periodically creates a checkpoint of the NameNode's namespace.
  - Helps in faster recovery of the NameNode in case of failure.

Data Storage

- **Blocks**: Files are divided into large blocks (default 128MB) to improve data transfer efficiency.
- **Replication**: Each block is replicated across multiple DataNodes for fault tolerance.
- **Rack Awareness**: HDFS is rack-aware, meaning it tries to place replicas of a block on different racks to improve data locality and fault tolerance.

  Data Access
- **Client**: Clients interact with the NameNode to get information about file locations.
- **Read Operations**: The NameNode provides locations of data blocks, and the client reads data directly from DataNodes.

- **Write Operations**: The client writes data to DataNodes, and the NameNode updates metadata.

Fault Tolerance

- **Replication**: Multiple copies of data blocks ensure data availability.
- **NameNode Checkpoint**: Periodic checkpoints help in recovering the NameNode state.
- **Heartbeat Mechanism**: DataNodes periodically send heartbeats to the NameNode to indicate their status.

Key features of HDFS:

- Scalability: Handles petabytes of data across thousands of nodes.
- Fault tolerance: Built-in redundancy for data protection.
- High throughput: Optimized for large data transfers.
- Simple design: Easy to understand and manage.
- Streaming access: Supports continuous data processing.

5. Explain RDD vs Dataframe in PySpark. Which is better ?

RDD (Resilient Distributed Dataset)

- The fundamental data structure in Spark.
- An immutable collection of objects partitioned across multiple nodes.
- Provides low-level API for data manipulation.
- Offers fine-grained control over data processing.
- Suitable for complex data transformations and custom algorithms.

DataFrame

- A higher-level abstraction built on top of RDDs.
- Represents data as a structured collection of rows and columns, similar to a SQL table.
- Provides a domain-specific language for data manipulation.
- Optimized for structured data and SQL-like operations.
- Offers better performance and ease of use compared to RDDs.
  Which is better?
  The choice between RDD and DataFrame depends on the specific use case and the nature of the data.

Use RDD when:

- You need fine-grained control over data processing.
- You're dealing with unstructured or complex data.
- You're building custom algorithms or libraries.

Use DataFrame when:

- You're working with structured data.
- You need to perform SQL-like operations (e.g., filtering, grouping, aggregation).
- You prioritize performance and ease of use.

  In most cases, DataFrame is preferred due to its higher-level abstraction and optimized performance. However, there might be scenarios where RDDs are more suitable.

6. Explain how to use PySpark for Machine Learning

1. Import Necessary Libraries
2. Create a SparkSession
3. Load and Prepare Data
4. Feature Engineering
5. Split Data into Training and Test Sets
6. Create and Train a Model
7. Make Predictions
8. Evaluate the Model

- **DataFrames:** PySpark primarily uses DataFrames for structured data manipulation.
- **Feature Engineering:** This is crucial for model performance. Use transformers like VectorAssembler, Imputer, OneHotEncoder, etc.
- **MLlib:** PySpark's machine learning library provides algorithms for classification, regression, clustering, and more.
- **Pipeline:** For complex workflows, use the Pipeline API to streamline the process.
- **Hyperparameter Tuning:** Explore different hyperparameter values to optimize model performance.
- **Model Evaluation:** Use appropriate metrics like accuracy, precision, recall, F1-score based on the problem.

7. What is Spark Architecture?

Spark employs a master-slave architecture with a driver as the master and executors as slaves. It's designed for in-memory processing, making it significantly faster than MapReduce.

Key Components

- Driver Program:
  - The main application process that defines the transformations and actions on the data.
  - Creates the SparkContext object, which is the entry point to the Spark cluster.
  - Submits jobs to the cluster manager.
  - Coordinates the execution of tasks across executors.
- Cluster Manager:
  - Manages the resources of the cluster.
  - Allocates resources to Spark applications.

    - Examples: Hadoop YARN, Apache Mesos, Standalone.
- Executor:
    - Process launched on a worker node.
    - Runs tasks assigned by the driver.
    - Stores data in memory for efficient processing.
- Worker Node:
    - A machine in the cluster that runs executors.

Core Abstractions

- RDD (Resilient Distributed Dataset):
    - An immutable collection of objects distributed across multiple nodes.
    - Forms the foundation of Spark's data processing.
    - Supports various operations like map, reduce, filter, etc.
- DAG (Directed Acyclic Graph):
    - Represents the execution plan for a Spark job.
    - Optimizes job execution by identifying stages and dependencies.

How it Works

1. The driver program creates an RDD and defines transformations.
2. Spark creates a DAG to represent the computation.
3. The DAG is divided into stages based on shuffle operations.
4. The driver submits stages to the cluster manager.
5. The cluster manager allocates resources and assigns tasks to executors.
6. Executors execute tasks and return results to the driver.

Advantages of Spark

- In-memory computation: Faster than disk-based systems.
- Fault tolerance: RDDs can be reconstructed in case of failures.
- Versatility: Supports batch, streaming, SQL, machine learning, and graph processing.
- Ease of use: High-level API for data manipulation.

8. Explain Spark Session

Spark Session is the unified entry point to all Spark functionalities. It provides a single interface to interact with Spark's different components, including RDDs, DataFrames, and Datasets.

Benefits of Spark Session

- **Unified Interface:** Provides a single point of entry for different Spark components.
- **Simplified Development:** Reduces boilerplate code compared to using separate contexts.
- **Improved Performance:** Offers optimizations for various operations.
- **Configuration:** Can be configured with various options using config() method.

- **Hive Support:** Enables integration with Hive metastore using enableHiveSupport().
- **Streaming Support:** Can be used for streaming data processing using spark.readStream.

In essence, SparkSession serves as the bridge between your application and the Spark cluster, providing a streamlined way to work with data and perform computations.

9. What is pyspark queue / job scheduling in pyspark ?

10. What is streaming in pyspark?

Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams. It allows you to process real-time data from various sources, such as Kafka, Kinesis, or TCP sockets, and perform complex computations on it.

- **DStream (Discretized Stream):** A continuous stream of data represented as a sequence of RDDs.
- **Input DStreams:** Create DStreams from input sources like Kafka, Kinesis, etc.
- **Transformations:** Apply operations like map, filter, reduce, join, etc. on DStreams.
- **Output Operations:** Write processed data to sinks like files, databases, or other streams.

11. If their is error in ETL pipeline at 12 pm how will you understand it or get it on email [Airflow] ? Solution to automate it?

Ans: Set Email Alerts in Your DAG

You can define this in the DAG's default_args:

```
from airflow import DAG
from datetime import datetime
from airflow.operators.python import PythonOperator

default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'email': ['your_email@example.com'],
    'email_on_failure': True,
    'email_on_retry': False,
    'retries': 1
}

def my_etl_function():
    # Your ETL logic
    raise Exception("Simulating failure at 12 PM")
```

```
with DAG(
    'etl_pipeline',
    default_args=default_args,
    description='ETL pipeline with alerting',
    schedule_interval='0 12 * * *',  # Runs daily at 12 PM
    start_date=datetime(2023, 1, 1),
    catchup=False
) as dag:

    etl_task = PythonOperator(
        task_id='run_etl',
        python_callable=my_etl_function
    )
```

12.Explain difference repartition vs coalesce

`coalesce` uses existing partitions to minimize the amount of data that's shuffled.

Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset.

`repartition` creates new partitions and does a full shuffle. Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.

`coalesce` results in partitions with different amounts of data (sometimes partitions that have much different sizes) and `repartition` results in roughly equal sized partitions.

Is `coalesce` or `repartition` faster?

`coalesce` may run faster than `repartition`, but unequal sized partitions are generally slower to work with than equal sized partitions. You'll usually need to repartition datasets after filtering a large data set. I've found `repartition` to be faster overall because Spark is built to work with equal sized partitions.

13.Explain difference between rdd,dataframe and dataset ?

Feature/Aspect	RDDs (Resilient Distributed Datasets)	DataFrames	Datasets
Data Representation	Distributed collection of data elements without any schema.	Distributed collection organized into named columns.	Extension of DataFrames with type safety and

			object-oriented interface.
Optimization	No in-built optimization; requires manual code optimization.	Uses Catalyst optimizer for query optimization.	Uses Catalyst optimizer for query optimization.
Schema	Schema must be manually defined.	Automatically infers schema of the dataset.	Automatically infers schema using the SQL Engine.
Aggregation Operations	Slower for simple operations like grouping data.	Provides easy API and performs faster aggregations than RDDs and Datasets.	Faster than RDDs but generally slower than DataFrames.
Type Safety	No compile-time type safety.	No compile-time type safety.	Provides compile-time type safety.
Functional Programming	Supports functional programming constructs.	Supports functional programming but with some limitations compared to RDDs.	Supports rich functional programming constructs.
Fault Tolerance	Inherently fault-tolerant with automatic recovery through lineage information.	Inherently fault-tolerant.	Inherently fault-tolerant.
Ease of Use	Low-level API can be complex and cumbersome.	Higher-level API with SQL-like capabilities; more user-friendly.	Combines high-level API with type safety; more complex to use.
Performance	Generally slower due to lack of built-in optimizations.	Generally faster due to Catalyst optimizer and Tungsten execution engine.	Optimized, but may have slight performance overhead due to type checks and schema enforcement.
Use Cases	Suitable for unstructured/semi-structured data, custom	Ideal for structured/semi-structured data, SQL-like queries,	Best for type-safe data processing, complex

	transformations, iterative algorithms, interactive analysis, and graph processing.	data aggregation, and integration with BI tools.	business logic, and interoperability with Java/Scala.
Language Support	Available in multiple languages, including Python.	Available in multiple languages, including Python.	Available only in Java and Scala.
Interoperability	Flexible but less optimized for interoperability with BI tools.	Highly compatible with BI tools like Tableau and Power BI.	Strong typing makes it suitable for Java/Scala applications.
Complexity	Requires deeper understanding of Spark's core concepts for effective use.	Simplified API reduces complexity for common tasks.	Combines features of RDDs and DataFrames, adding complexity.

14.what is Narrow transformation, wide transformation in spark ?

Narrow Transformation

- A **narrow transformation** is one where **each input partition contributes to exactly one output partition**.
- Data is **not shuffled** across the cluster.
- Tasks depend only on **a single partition** of the parent RDD/DataFrame.
- These transformations are **faster** and **more efficient** because no data movement (shuffle) occurs.
  Examples:

map(),filter(),union() (when no shuffle required),mapPartitions(),sample()

Wide Transformation

- A **wide transformation** is one where **each input partition contributes to multiple output partitions**.
- Requires **shuffle** of data across the network between executors.
- Tasks depend on **multiple partitions** of the parent RDD/DataFrame.

- These transformations are **more expensive** because they involve **data movement and disk I/O**.

Examples:

reduceByKey(),groupByKey(),sortByKey(),join(),distinct()

15. Can we rename column in pyspark? Write syntax for it.
    withColumnRenamed() renamesacolumn in theDataFrame
    test_df_sex = test_df.withColumnRenamed('Gender', 'Sex')

16. Why did you choose pandas over pyspark in your project?

17. What are differences in Pyspark vs pandas

1) Performance and Efficiency:

PySpark, designed to manage big data on distributed systems, excels in large-scale data processing, outperforming Pandas in this regard. PySpark's architecture allows it to distribute computations across a cluster of machines, making it highly efficient for handling voluminous data. It employs resilient distributed datasets (RDDs) to parallelize data processing, enhancing its performance.

On the other hand, Pandas is a robust tool for manipulating and analyzing datasets of moderate size, typically up to a few gigabytes. It provides fast and efficient data manipulation and processing on a single machine, making it ideal for smaller datasets.

2) Processing Speed:

When it comes to processing speed, PySpark has a significant advantage over Pandas for large datasets. PySpark's ability to perform parallel computation on distributed systems and its use of in-memory caching contribute to its superior speed. In contrast, Pandas, while providing fast performance for small to medium-sized datasets, may not match PySpark's speed for larger datasets due to its lack of parallel processing capabilities.

3) Memory Consumption:

In terms of memory usage, PySpark is more efficient than Pandas. PySpark employs lazy evaluation, retrieving data from the disk only when necessary, reducing memory consumption. Conversely, Pandas keeps all data in memory, leading to higher memory usage, particularly for large datasets.

4) Ease of Use and Flexibility:

Pandas shines in terms of ease of use and flexibility. Its API is straightforward, and its syntax resembles SQL and Excel, making it accessible for analysts and data scientists. It provides an interactive environment for data exploration and analysis through Jupyter notebooks, which allows for easy visualization and experimentation. Furthermore, Pandas can handle a wide variety of data sources, including CSV, Excel, SQL databases, and more. It also integrates seamlessly with other Python libraries, such as NumPy, Matplotlib, and Scikit-learn.

In contrast, PySpark requires a deeper understanding of distributed computing concepts, which may present a steeper learning curve. However, it offers robust integration with big data tools and technologies, including Hadoop, Hive, Cassandra, and HBase.

5) Scalability and Distributed Computing:

PySpark is designed for scalability and distributed computing. It can handle large-scale datasets by distributing computations across a cluster of machines. This scalability makes PySpark an excellent choice for big data processing tasks that exceed the memory capacity of a single machine.

On the other hand, Pandas is limited by the memory of a single machine, making it less suitable for processing very large datasets.

6) Real-Time Data Processing:

PySpark offers streaming data processing capabilities, allowing users to process real-time data streams using Spark's distributed computing capabilities. This feature is absent in Pandas, making PySpark a better choice for real-time data processing tasks.

7) Community Support:

Both PySpark and Pandas have active and vibrant communities, providing extensive documentation, tutorials, and forums for user support.

18.Explain Difference between task and jobs

Job	Task
A **Job** is a complete unit of work submitted by the user to the cluster. It represents the entire processing operation (e.g., running a MapReduce program or a Spark application).	A **Task** is a smaller unit of work that is part of a Job. Jobs are divided into multiple tasks that run in parallel on different cluster nodes.
Runs on the cluster's resource manager (e.g., YARN or Spark's driver).	Executed by worker nodes (NodeManagers in YARN, executors in Spark).
Usually one per user request (e.g., one MapReduce job).	Multiple tasks per job — can be hundreds or thousands depending on data and cluster size.
To complete an entire data processing workflow.	To process a subset (partition) of data within the job.
Starts when a user submits a job and ends when all tasks complete successfully.	Starts when scheduled by the job tracker or resource manager and ends after processing its assigned data.

Running a MapReduce job to process 1TB of logs.	Individual map tasks processing 64MB splits of the input data.

19. What does withcolumn do in pyspark

Returns a new DataFrame by adding a column or replacing the existing column that has the same name.
The column expression must be an expression over this DataFrame; attempting to add a column from some other DataFrame will raise an error.
Parameters

- **colName** – string, name of the new column.
- **col** – a Column expression for the new column.

df.withColumn('age2', df.age + 2).collect()
[Row(age=2, name='Alice', age2=4), Row(age=5, name='Bob', age2=7)]

20. What is AWS Glue dataframe and spark dataframe conversion in AWS glue?

A `DynamicFrame` is similar to a `DataFrame`, except that each record is self-describing, so no schema is required initially. Instead, AWS Glue computes a schema on-the-fly when required, and explicitly encodes schema inconsistencies using a choice (or union) type. You can resolve these inconsistencies to make your datasets compatible with data stores that require a fixed schema.

Similarly, a `DynamicRecord` represents a logical record within a `DynamicFrame`. It is like a row in a Spark `DataFrame`, except that it is self-describing and can be used for data that does not conform to a fixed schema. When using AWS Glue with PySpark, you do not typically manipulate independent `DynamicRecords`. Rather, you will transform the dataset together through its `DynamicFrame`.

You can convert `DynamicFrames` to and from `DataFrames` after you resolve any schema inconsistencies.

toDF(options)
Converts a DynamicFrame to an Apache Spark DataFrame by converting DynamicRecords into DataFrame fields. Returns the new DataFrame.
A DynamicRecord represents a logical record in a DynamicFrame. It is similar to a row in a Spark DataFrame, except that it is self-describing and can be used for data that does not conform to a fixed schema.

- options – A list of options. Allows you to specify additional options for the conversion process. Some valid options that you can use with the `options` parameter:
    - format – specifies the format of the data, such as json, csv, parquet).
    - separater or sep – for CSV files, specifies the delimiter.
    - header – for CSV files, indicates whether the first row is a header (true/false).

- inferSchema – directs Spark to infer the schema automatically (true/false).

21. Explain Difference between internal and external table in hive

Internal Table:

When a user creates a table in Hive it is by default an internal table created in the ***/user/hive/warehouse*** directory in HDFS which is its default storage location. The data present in the internal table will be stored in this directory and is fully managed by Hive and thus an internal table is also referred to as a managed table.

Loading the Data

To load the data, we use the following command:

LOAD DATA LOCAL INPATH '/home/Hadoop/student.txt'
OVERWRITE INTO TABLE student_internal

Here we are importing the data from a file present in the local HDFS path- **'/home/Hadoop/student.txt'** and overwriting or we can say loading it into the **student_internal** table we created above.

Storage

An internal table is stored on HDFS in the ***/user/hive/warehouse*** directory which is its default storage location. This location can be changed by updating the path in the configuration file present in the config file – **hive.metastore.warehouse.dir.**

We can also alter the location of the table by providing a new path present in HDFS using **the SET LOCATION** clause.

ALTER TABLE student_internal
SET LOCATION
'hdfs://localhost:8020/user/tables/student';

Usage

We may use an internal table if:

1. Data is temporary and doesn't affect businesses in real-time.
2. If we want the hive to manage the data and the tables.

External Table

When a user creates a table in Hive specifying the external keyword, then an external table is created. The data present in the external table will be fully managed by HDFS contrary to an internal table.

External table creation query has the same intuition and syntax as the internal table. Note that while creating an external table we specify the keyword **EXTERNAL** to inform Hive to create an external table for us.

Now that we have learned to create an external table let's see how we can load the data and populate it.

Storage

An external table is stored on HDFS or any storage compatible with HDFS, because we want to use the data outside of Hive. Thus, Hive is not responsible for managing the

storage of the external table. Tables can be stored on an external location for instance on a cloud platform like google cloud or AWS.

Usage

We may use an external table if:

1. We want to use data outside HIVE for performing a different operations such as loading and merging.
2. The data is of production quality.

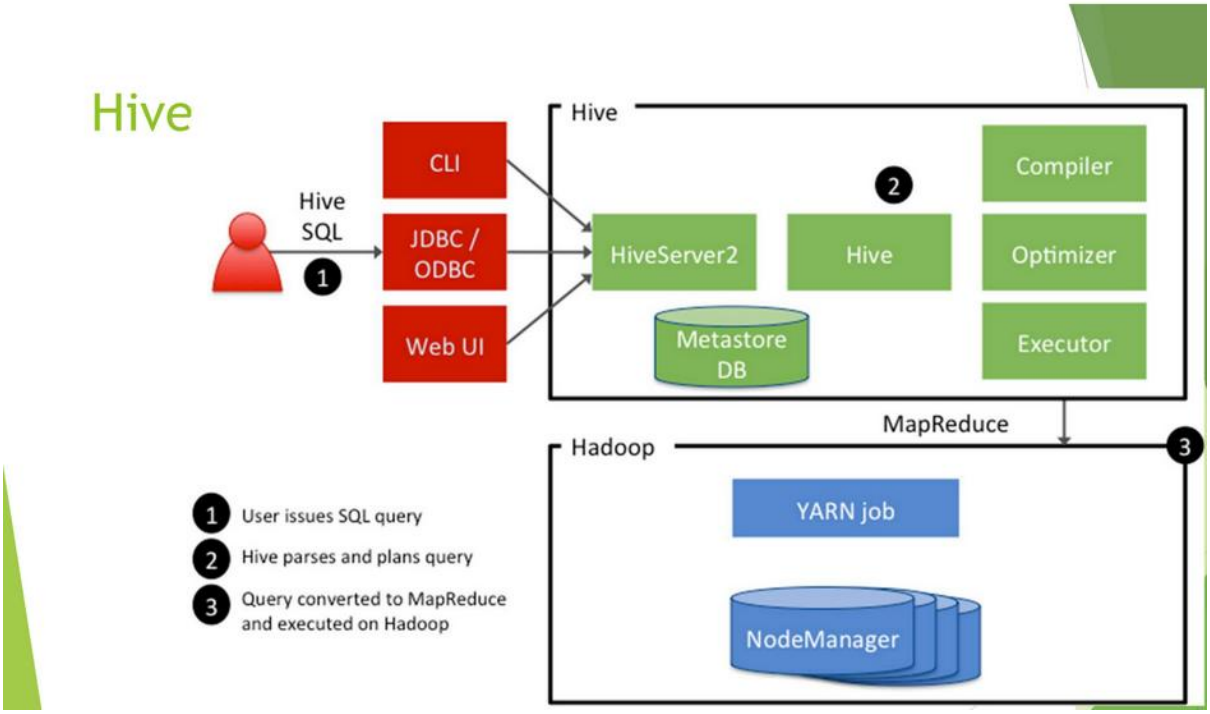
22. What is hive

It is a data warehouse system for Hadoop

• It maintains metadata information about your big data stored on HDFS

• It treats your big data as tables

• It performs SQL-like operations on the data using a scripting language called HiveQL



23. What is AWS Glue bookmark ?

AWS Glue tracks data that has already been processed during a previous run of an ETL job by persisting state information from the job run. This persisted state information is called a *job bookmark*. Job bookmarks help AWS Glue maintain state information and prevent the reprocessing of old data. With job bookmarks, you can process new data when rerunning on a scheduled interval. A job bookmark is composed of the states for various elements of jobs, such as sources, transformations, and targets. For example, your ETL job might read new partitions in an Amazon S3 file. AWS Glue tracks which partitions the job has processed successfully to prevent duplicate processing and duplicate data in the job's target data store.

Job bookmarks are implemented for JDBC data sources, the Relationalize transform, and some Amazon Simple Storage Service (Amazon S3) sources. The following table lists the Amazon S3 source formats that AWS Glue supports for job bookmarks.

AWS Glue version	Amazon S3 source formats
Version 0.9	JSON, CSV, Apache Avro, XML
Version 1.0 and later	JSON, CSV, Apache Avro, XML, Parquet, ORC

24. Explain AWS Redshift optimization techniques

- Materialized views can significantly boost query performance for repeated and predictable analytical workloads such as dash-boarding, queries from BI tools, and extract, load, transform (ELT) data processing. Data engineers can easily create and maintain efficient data-processing pipelines with materialized views while seamlessly extending the performance benefits to data analysts and BI tools.

  Materialized views are especially useful for queries that are predictable and repeated over and over. Instead of performing resource-intensive queries on large tables, applications can query the pre-computed data stored in the materialized view.

- **Using the Amazon Redshift Advisor to minimize administrative work**

  Advisor bases its recommendations on observations regarding performance statistics or operations data. Advisor develops observations by running tests on your clusters to determine if a test value is within a specified range. If the test result is outside of that range, Advisor generates an observation for your cluster. At the same time, Advisor creates a recommendation about how to bring the observed value back into the best-practice range. Advisor only displays recommendations that can have a significant impact on performance and operations. When Advisor determines that a recommendation has been addressed, it removes it from your recommendation list.

  examples of Advisor recommendations:

  Distribution key recommendation, Sort key recommendation, Table compression recommendation, Table statistics recommendation

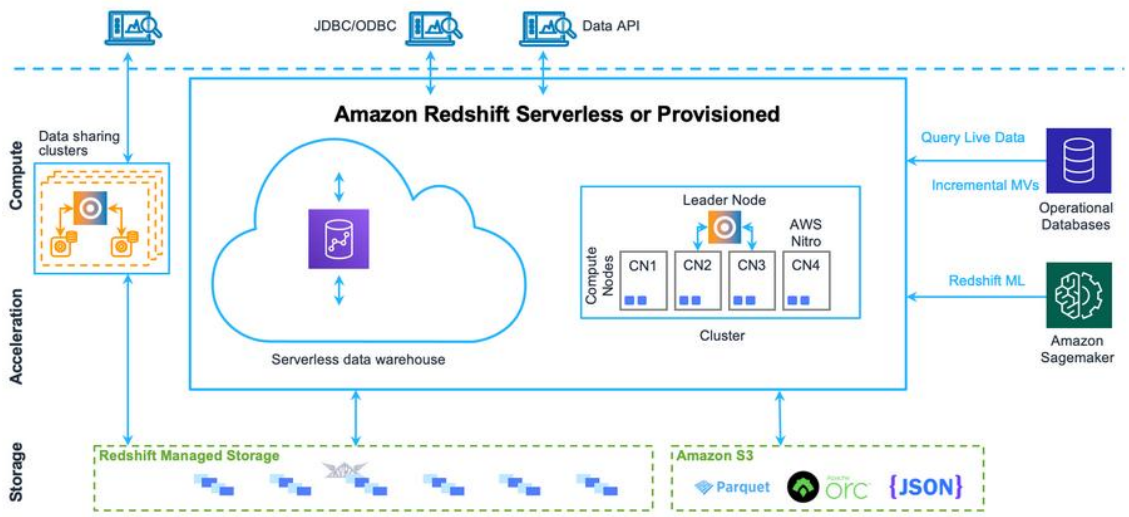
- **Taking advantage of Amazon Redshift data lake integration**

Amazon Redshift is tightly integrated with other AWS-native services such as Amazon S3 which let's the Amazon Redshift cluster interact with the data lake in several useful ways.

Amazon Redshift Spectrum lets you query data directly from files on Amazon S3 through an independent, elastically sized compute layer. Use these patterns independently or apply them together to offload work to the Amazon Redshift Spectrum compute layer, quickly create a transformed or aggregated dataset, or eliminate entire steps in a traditional ETL process.

https://aws.amazon.com/blogs/big-data/top-10-performance-tuning-techniques-for-amazon-redshift/

25. Explain AWS Redshift, with its architecture in details



Client applications

Amazon Redshift integrates with various data loading and ETL (extract, transform, and load) tools and business intelligence (BI) reporting, data mining, and analytics tools. Amazon Redshift is based on open standard PostgreSQL, so most existing SQL client applications will work with only minimal changes

Clusters

A cluster is composed of one or more *compute nodes*. If a cluster is provisioned with two or more compute nodes, an additional *leader node* coordinates the compute nodes and handles external communication. Your client application interacts directly only with the leader node. The compute nodes are transparent to external applications.

Leader node

The leader node manages communications with client programs and all communication with compute nodes. It parses and develops execution plans to carry out database operations

Compute nodes

The leader node compiles code for individual elements of the execution plan and assigns the code to individual compute nodes. The compute nodes run the compiled code and send intermediate results back to the leader node for final aggregation.

Redshift Managed Storage

Data warehouse data is stored in a separate storage tier Redshift Managed Storage (RMS). RMS provides the ability to scale your storage to petabytes using Amazon S3 storage. RMS lets you scale and pay for computing and storage independently, so that you can size your cluster based only on your computing needs. It automatically uses high-performance SSD-based local storage as tier-1 cache. It also takes advantage of optimizations, such as data block temperature, data block age, and workload patterns to deliver high performance while scaling storage automatically to Amazon S3 when needed without requiring any action.

Node slices

A compute node is partitioned into slices. Each slice is allocated a portion of the node's memory and disk space, where it processes a portion of the workload assigned to the node. The leader node manages distributing data to the slices and apportions the workload for any queries or other database operations to the slices. The slices then work in parallel to complete the

operation.The number of slices per node is determined by the node size of the cluster.

Internal network

Amazon Redshift takes advantage of high-bandwidth connections, close proximity, and custom communication protocols to provide private, very high-speed network communication between the leader node and compute nodes. The compute nodes run on a separate, isolated network that client applications never access directly.

Databases

A cluster contains one or more databases. User data is stored on the compute nodes. Your SQL client communicates with the leader node, which in turn coordinates query run with the compute nodes.

Amazon Redshift is a relational database management system (RDBMS), so it is compatible with other RDBMS applications. Although it provides the same functionality as a typical RDBMS, including online transaction processing (OLTP) functions such as inserting and deleting data, Amazon Redshift is optimized for high-performance analysis and reporting of very large datasets.

https://docs.aws.amazon.com/redshift/latest/dg/c_high_level_system_architecture.html

26. How to automate the glue job when the new data is inserted into s3 bucket?

    To **automate an AWS Glue job** when **new data is inserted into an S3 bucket**, you can use **Amazon S3 Event Notifications** in combination with **AWS Lambda**, **Amazon EventBridge**

    S3 (New file upload)
    ↓
    S3 Event Notification
    ↓
    AWS Lambda (triggered)
    ↓
    Start AWS Glue Job

1. Create a Glue Job
2. Create a Lambda Function to Trigger the Glue Job
3. Add an S3 Event Notification

27. Explain Apache Spark - broadcast variable, broadcast join,lineage graph, types of table

Broadcast Variables

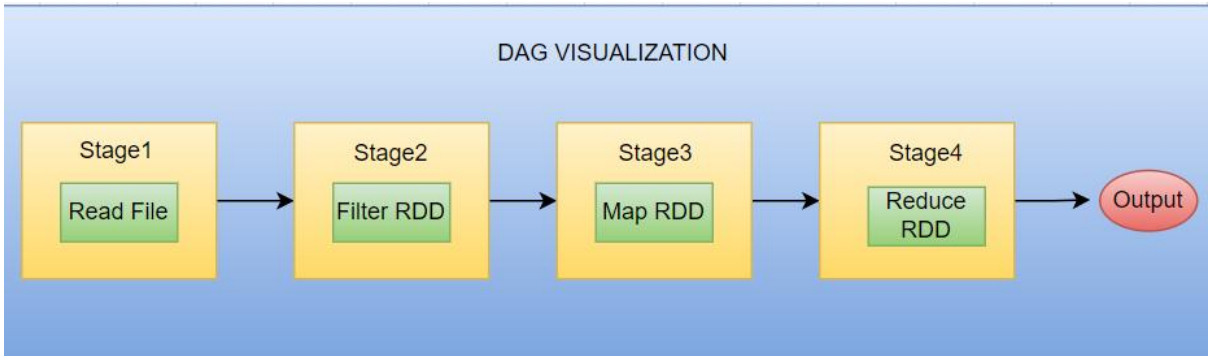
Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks. They can be used, for example, to give every node a copy of a large input dataset in an efficient manner. Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost.

Spark actions are executed through a set of stages, separated by distributed “shuffle” operations. Spark automatically broadcasts the common data needed by tasks within each stage. The data broadcasted this way is cached in serialized form and deserialized before running each task. This means that explicitly creating broadcast variables is only useful when tasks across multiple stages need the same data or when caching the data in deserialized form is important.

Broadcast Join:

A broadcast join is an optimization strategy in Spark, primarily used when one of the tables involved in a join operation is significantly smaller than the other and can fit entirely in the memory of the executors. Instead of shuffling the larger dataset, the smaller dataset is broadcasted to all executor nodes. This allows each executor to perform the join locally with its partition of the larger dataset and the broadcasted smaller dataset, avoiding the costly shuffle operation and improving performance.

DAG (Directed Acyclic Graph) in Spark/PySpark is a fundamental concept that plays a crucial role in the Spark execution model. The DAG is “directed” because the operations are executed in a specific order, and “acyclic” because there are no loops or cycles in the execution plan. This means that each stage depends on the completion of the previous stage, and each task within a stage can run independently of the other.

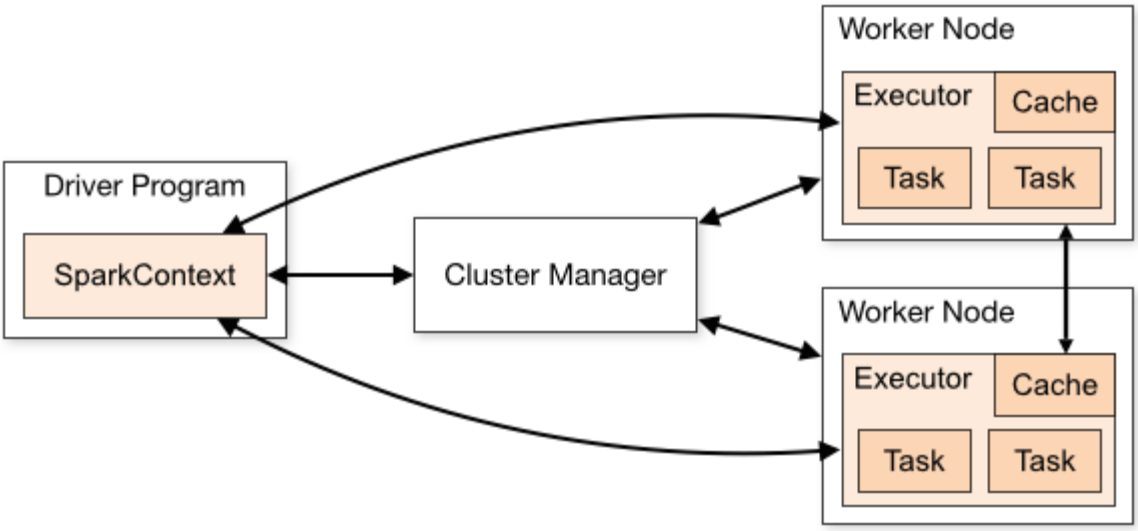


28. Explain Architecture of pyspark, hadoop and what is difference between spark and hadoop?

Spark Architecture:

Spark applications run as independent sets of processes on a cluster, coordinated by the SparkContext object in your main program (called the *driver program*).

Specifically, to run on a cluster, the SparkContext can connect to several types of *cluster managers* (either Spark's own standalone cluster manager, YARN or Kubernetes), which allocate resources across applications. Once connected, Spark acquires *executors* on nodes in the cluster, which are processes that run computations and store data for your application. Next, it sends your application code (defined by JAR or Python files passed to SparkContext) to the executors. Finally, SparkContext sends *tasks* to the executors to run.



There are several useful things to note about this architecture:

1. Each application gets its own executor processes, which stay up for the duration of the whole application and run tasks in multiple threads. This has the benefit of isolating applications from each other, on both the scheduling side (each driver schedules its own tasks) and executor side (tasks from different applications run in different JVMs). However, it also means that data cannot be shared across different Spark applications (instances of SparkContext) without writing it to an external storage system.
2. Spark is agnostic to the underlying cluster manager. As long as it can acquire executor processes, and these communicate with each other, it is relatively easy to run it even on a cluster manager that also supports other applications (e.g. YARN/Kubernetes).
3. The driver program must listen for and accept incoming connections from its executors throughout its lifetime. As such, the driver program must be network addressable from the worker nodes.
4. Because the driver schedules tasks on the cluster, it should be run close to the worker nodes, preferably on the same local area network. If you'd like to send requests to the cluster remotely, it's better to open an RPC to the driver and have it submit operations from nearby than to run a driver far away from the worker nodes.

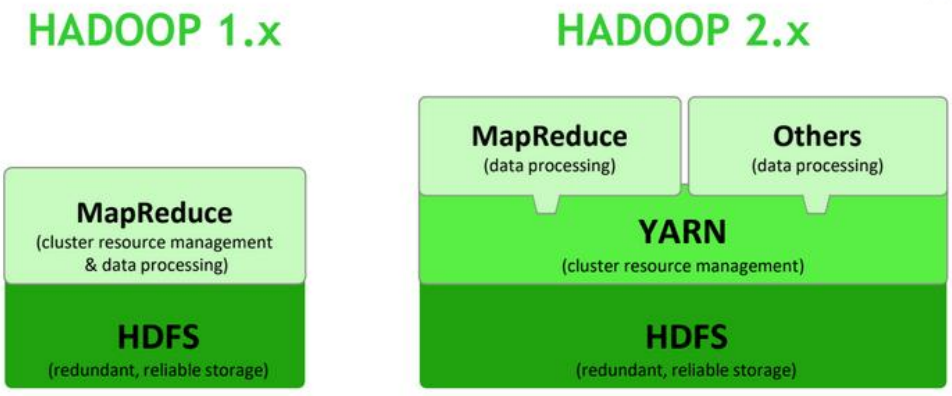
Hadoop Architecture:

The Apache Hadoop 2.x project consists of the following modules:

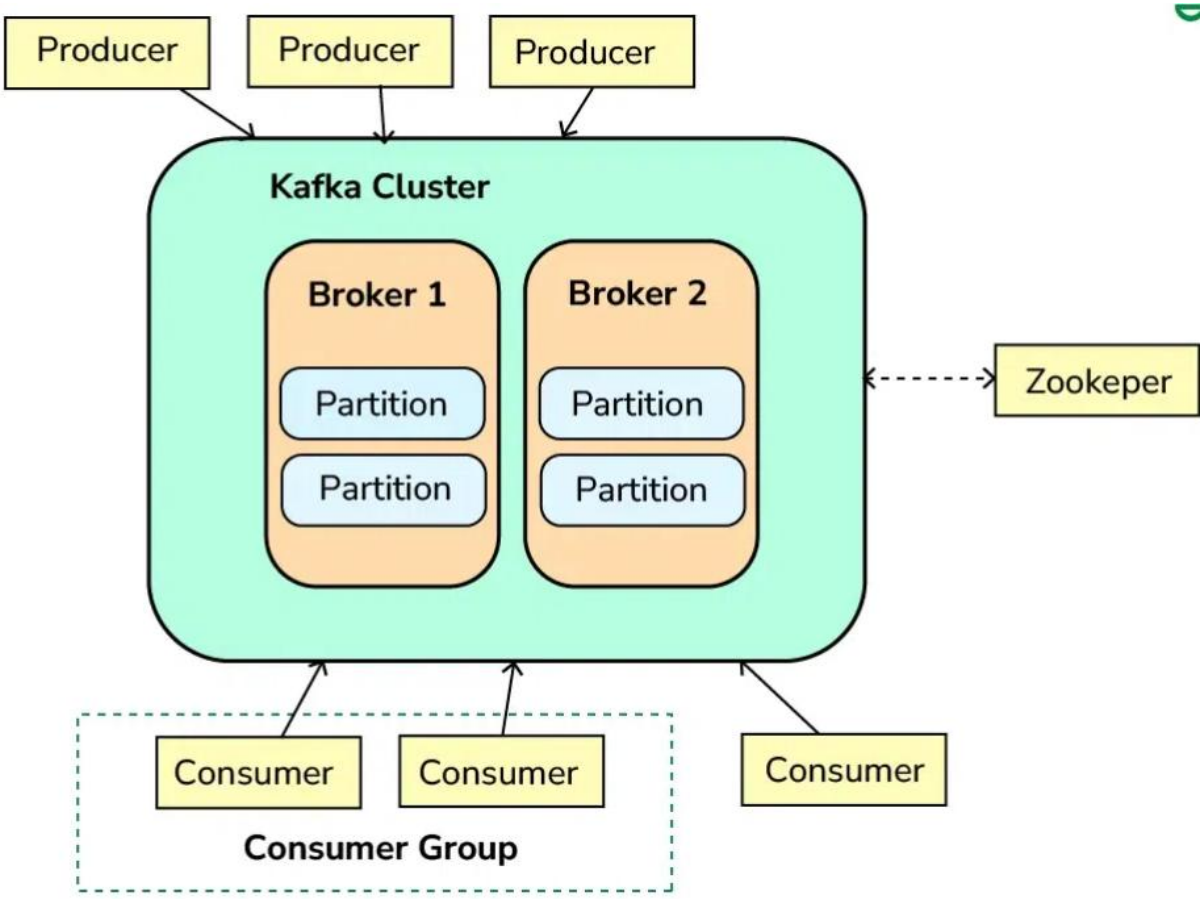
- **Hadoop Common**: the utilities that provide support for the other Hadoop modules
- **HDFS**: the Hadoop Distributed File System
- **YARN**: a framework for job scheduling and cluster resource management
- **MapReduce**: for processing large data sets in a scalable and parallel fashion

New in Hadoop 2.x

YARN is a re-architecture of Hadoop that allows multiple applications to run on the same platform



Q29. Kafka Architecture:



Core Components of Kafka Architecture

1. **Kafka Cluster**: A Kafka cluster is a distributed system composed of multiple Kafka brokers working together to handle the storage and processing of real-time streaming data. It provides fault tolerance, scalability, and high availability for efficient data streaming and messaging in large-scale applications.
2. **Broker**: Brokers are the servers that form the Kafka cluster. Each broker is responsible for receiving, storing, and serving data. They handle the read and write operations from producers and consumers. Brokers also manage the replication of data to ensure fault tolerance.
3. **Topics and Partitions**: Data in Kafka is organized into topics, which are logical channels to which producers send data and from which consumers read data. Each topic is divided into partitions, which are the basic unit of parallelism in Kafka. Partitions allow Kafka to scale horizontally by distributing data across multiple brokers.

4. **Producers**: Producers are client applications that publish (write) data to Kafka topics. They send records to the appropriate topic and partition based on the partitioning strategy, which can be key-based or round-robin.
5. **Consumer**: Consumers are client applications that subscribe to Kafka topics and process the data. They read records from the topics and can be part of a consumer group, which allows for load balancing and fault tolerance. Each consumer in a group reads data from a unique set of partitions.
6. **Zookeeper**: ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. In Kafka, ZooKeeper is used to manage and coordinate the Kafka brokers. ZooKeeper is shown as a separate component interacting with the Kafka cluster.
7. **Offsets**: Offsets are unique identifiers assigned to each message in a partition. Consumers will use these offsets to track their progress in consuming messages from a topic.