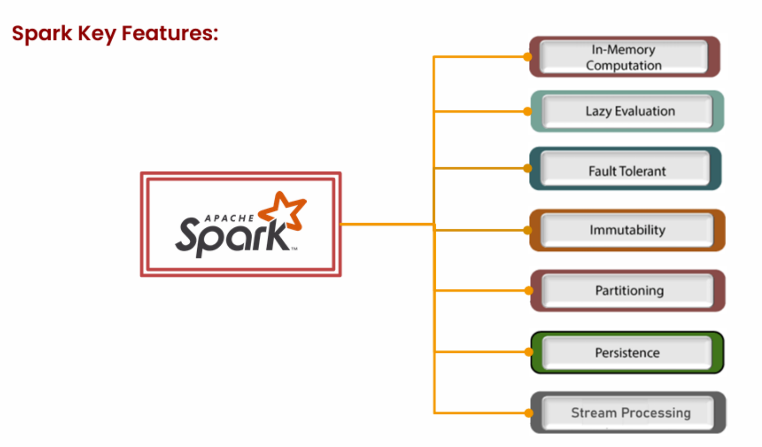
What is spark :

Answer :

Spark is an engine upon which we can perform data oprations related to data engineering , machine learning and other different data domains using single node or multinodes cluster.

Spark Ecosystem contains SQL, Python, Scala, yarn, kubernities

Key Features : Spark do in memory process, lazy Evalution (only on Actions) ,It is fault tolarent due to replications, Data frame are immutable ,partioning can be done , Persistence: persist the result at n$^{th(intermediate\ step)}$ step instead of executing all the commands from the start



It uses spark context/session which is entry point used to create rdd, DatasetApi, Dataframe

What is spark context and what is spark session :

Spark context :

Spark context is entry point and is used to create

SPARK CONNECTION

RDD CREATION

JOB SCHEDULING AND

CONFIGURATION

```
from pyspark import SparkContext, SparkConf

conf = SparkConf().setAppName("MyApp").setMaster("local")

sc = SparkContext(conf=conf)
```

SparkContext is used for RDD operations and basic Spark functions.

Spark session:

Spark session contains Spark, Streaming , sql and hive Context

SparkSession follows a session-based model, meaning you can create multiple SparkSessions within the same Spark application

❖ A single Spark application can have multiple Spark sessions. Starting from Spark 2.0, when the concept of SparkSession was introduced.

❖ It is particularly useful for running isolated tasks on different datasets or with different configurations within the same Spark application.

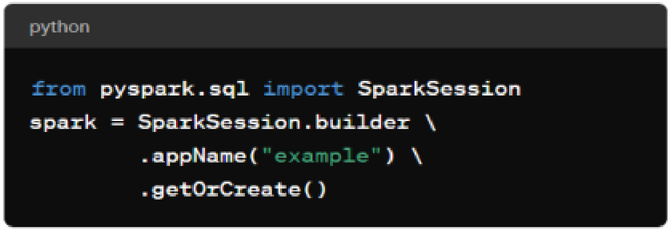
❖ Each SparkSession can have its own configurations, user-defined functions, and temporary views.

❖ While we have multiple SparkSession instances, they share the same underlying resources, such as executors and the core Spark engine.

Spark session is used to create:

- Dataframe api
- Spark sql context
- Integration with hive

SparkSession is used for working with DataFrames, Datasets, SQL, and Hive tables.



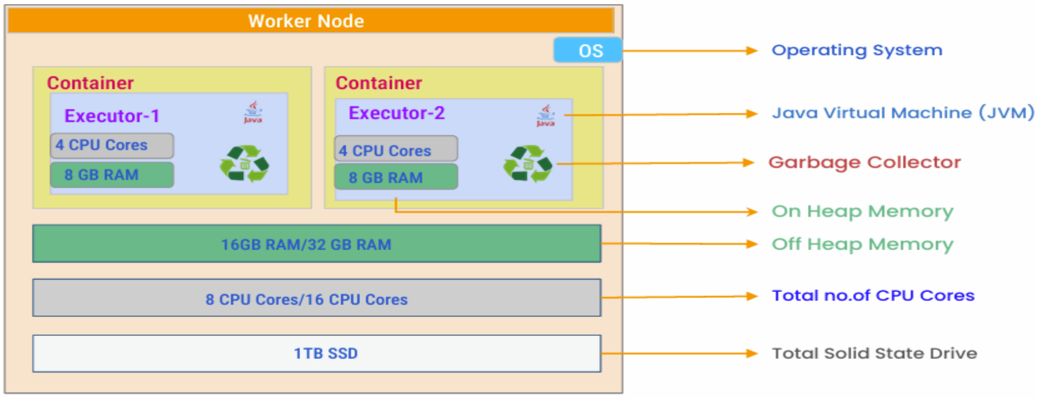
python

```python
from pyspark.sql import SparkSession
spark = SparkSession.builder \
        .appName("example") \
        .getOrCreate()
```

Worker Node/Slave Node characteristics:

1. Worker node contains Container/Executor, Executor contains on heap memory / Of heap memory and Cores and garbage collector
2. Worker node (server, Laptop): Is combination of Core, Ram(memory) ,ssd.
3. Worker node contains multiple containers. We can define config of container based on config of worker node
4. Single-Node clusters do not have worker nodes, and all the Spark jobs run on the driver node. Single-Node cluster mode be used for building and testing small data pipelines

Worker Node: It is a virtual machine part of the distributed computing cluster and is responsible for executing tasks.

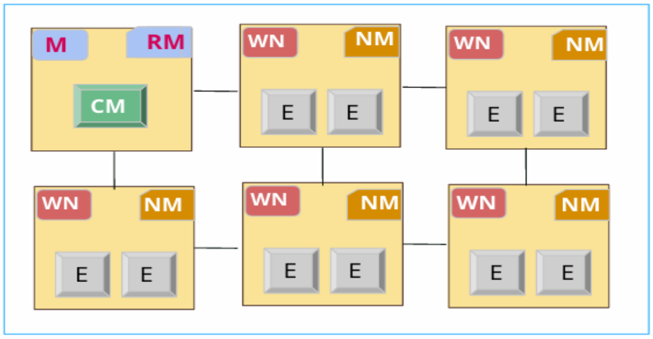


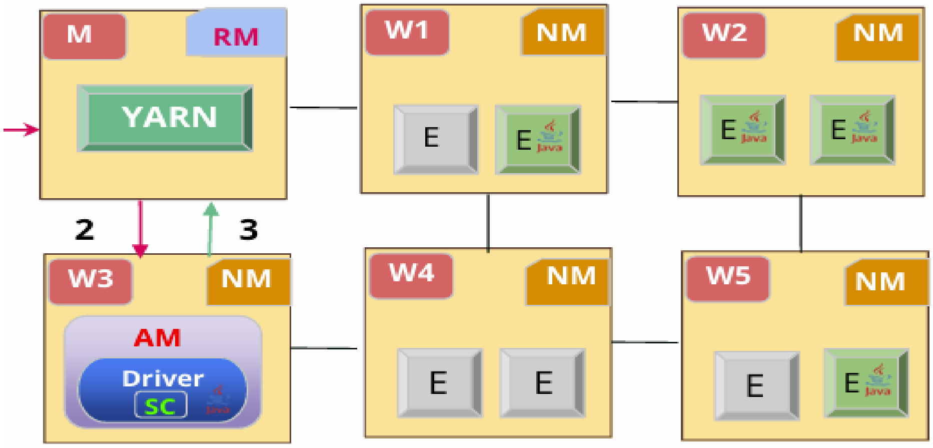
Spark Cluster - Master::Slave

CM - Cluster Manager

- YARN(Yet Another Resource Negotiator)
- Kubernetes
- Mesos
- Standalone

RM - Resource Manager
NM - Node Manager
M - Master Node
WN- Worker/Slave Node
E - Executor





What is container and executor and task

★ Spark Executor is a process that runs on a worker node (inside the container) in a Spark cluster.

Process runs on worker node

★ Executors are responsible for executing the tasks and returning the results back to the driver program.

Executing task

★ Spark can launch multiple executors on a worker node based on the requirement of spark application.

One node can have multiple executor

★ Each executor runs multiple tasks concurrently and manages the computation and data storage for those tasks

Can run multiple tasks

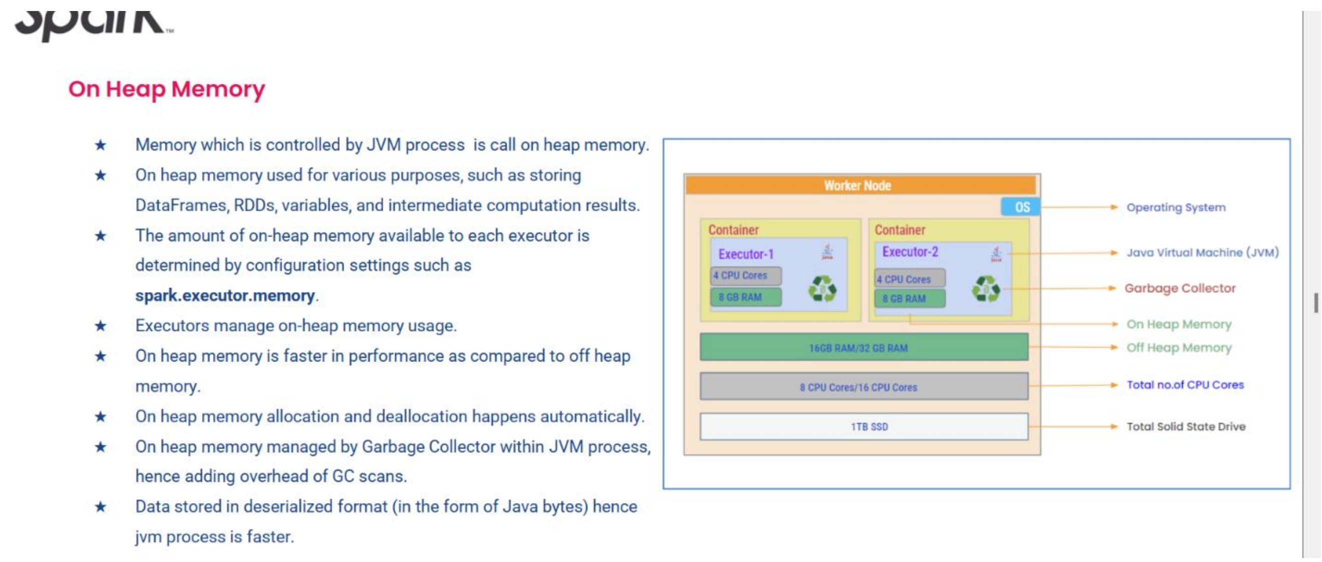
★ Executors manage both on-heap and off-heap memory based on the configuration provided

It manages both on heap and off heap

★ A task is the smallest unit of work in Spark.

★ Tasks are executed by the executors in parallel.

What is On Heap and Off Heap memory and its usages



On Heap Memory

- ★ Memory which is controlled by JVM process is call on heap memory.
- ★ On heap memory used for various purposes, such as storing DataFrames, RDDs, variables, and intermediate computation results.
- ★ The amount of on-heap memory available to each executor is determined by configuration settings such as **spark.executor.memory**.
- ★ Executors manage on-heap memory usage.
- ★ On heap memory is faster in performance as compared to off heap memory.
- ★ On heap memory allocation and deallocation happens automatically.
- ★ On heap memory managed by Garbage Collector within JVM process, hence adding overhead of GC scans.
- ★ Data stored in deserialized format (in the form of Java bytes) hence jvm process is faster.

On Heap Memory - Around 60 to 80 % of total ram

On heap memory characteristics:

Memory that is controlled by jvm used for caching (if on heap memory is not sufficient then it can use offheap memory for caching)

Used for :

- caching RDD and Dataframe ,
- UDF,
- Reserved Memory,
- Storage memory,
- Executor memory,
- (Dynamic shuffling of memory between storage and Executor memory),
- Compute operation (Join, aggregation, shuffle)
- Data stored in deserialized format
- Spark.executor.memory
- On heap memory can be further divided into
  - Reserved ,
  - Spark memory
  - And user memory
    - User memory is used to store

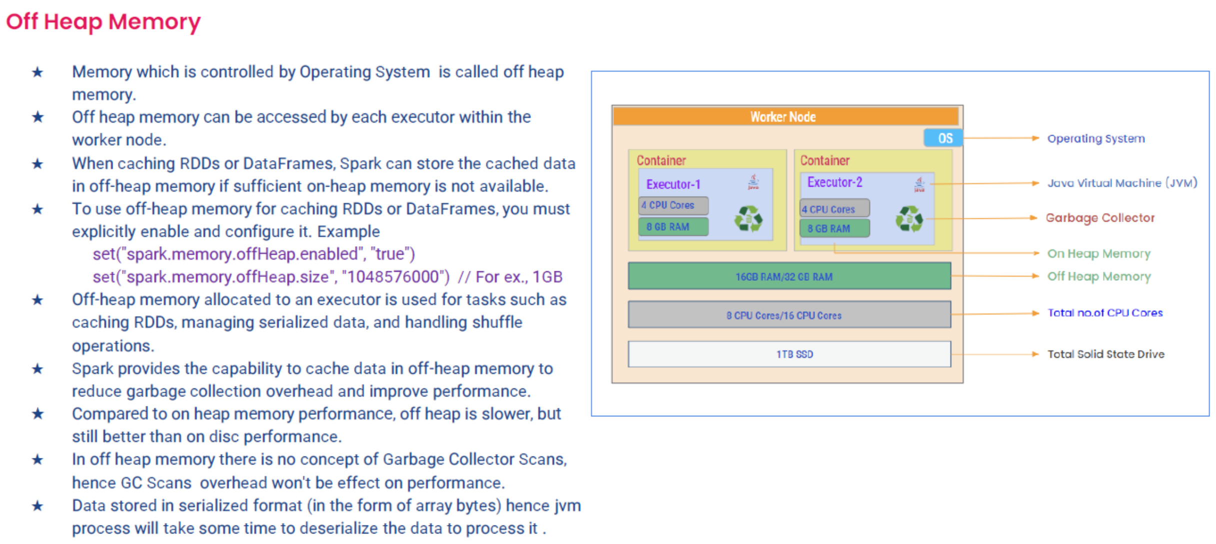
- UDF , broadcast variable, RDD conversion operations

Spark memory:

1. Used for caching and persisting
2. Rdd ,dataframes , and datasets

Off heap memory Char:

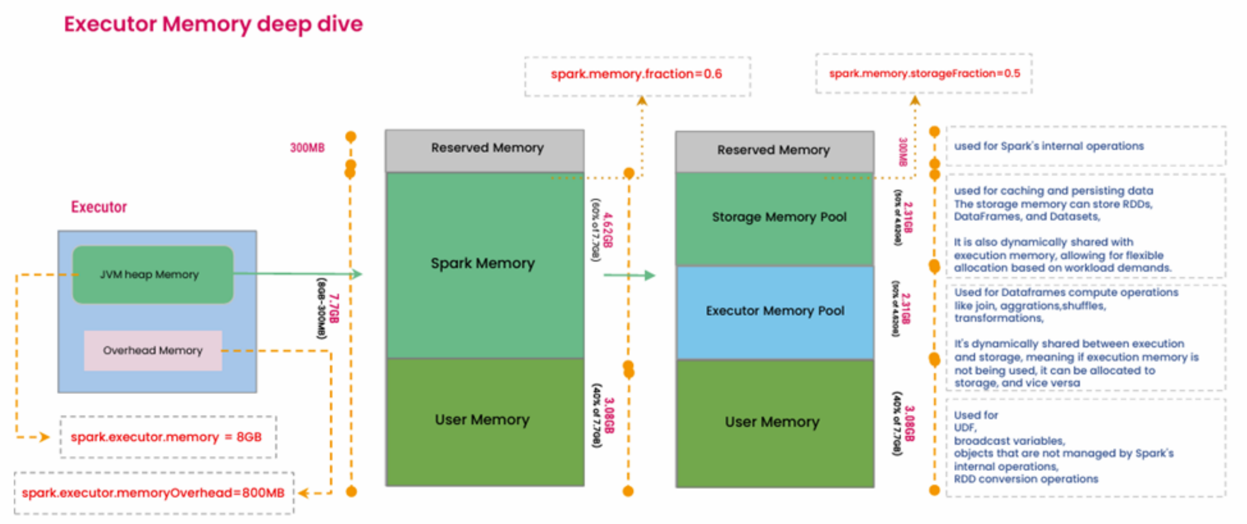
1. When caching RDDs or DataFrames, Spark can store the cached data in off-heap memory if sufficient on-heap memory is not available
2. Data stored in serialized format (in the form of array bytes) hence jvm process will take some time to deserialize the data to process it
3. Storing data in Ofheap memory helps improve performance as GC scans are less
4. This memory can be accessed by each executor/container as and when required



Off Heap Memory

★ Memory which is controlled by Operating System is called off heap memory.
★ Off heap memory can be accessed by each executor within the worker node.
★ When caching RDDs or DataFrames, Spark can store the cached data in off-heap memory if sufficient on-heap memory is not available.
★ To use off-heap memory for caching RDDs or DataFrames, you must explicitly enable and configure it. Example
    set("spark.memory.offHeap.enabled", "true")
    set("spark.memory.offHeap.size", "1048576000") // For ex., 1GB
★ Off-heap memory allocated to an executor is used for tasks such as caching RDDs, managing serialized data, and handling shuffle operations.
★ Spark provides the capability to cache data in off-heap memory to reduce garbage collection overhead and improve performance.
★ Compared to on heap memory performance, off heap is slower, but still better than on disc performance.
★ In off heap memory there is no concept of Garbage Collector Scans, hence GC Scans overhead won't be effect on performance.
★ Data stored in serialized format (in the form of array bytes) hence jvm process will take some time to deserialize the data to process it .

Of Heap Memory - Around 20 to 40 % of Total Ram



Executor Memory deep dive

used for Spark's internal operations

used for caching and persisting data
The storage memory can store RDDs, DataFrames, and Datasets,

It is also dynamically shared with execution memory, allowing for flexible allocation based on workload demands.

Used for Dataframes compute operations like join, aggrations,shuffles, transformations,

It's dynamically shared between execution and storage, meaning if execution memory is not being used, it can be allocated to storage, and vice versa

Used for
UDF,
broadcast variables,
objects that are not managed by Spark's internal operations,
RDD conversion operations

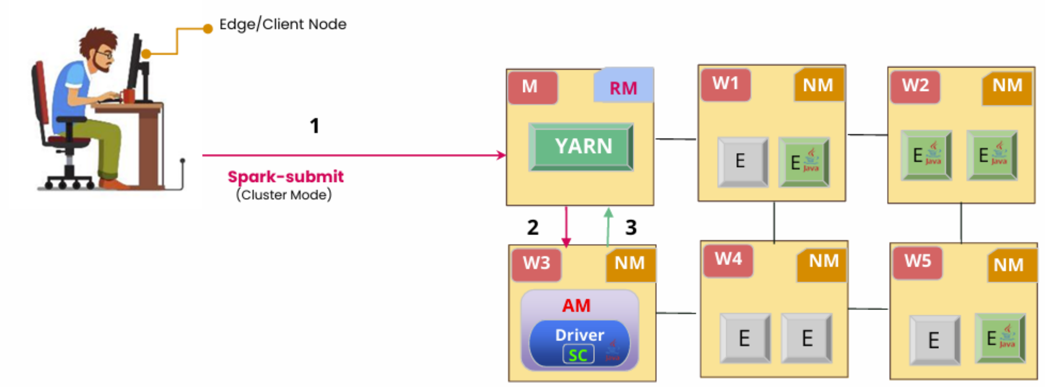
Garbage Collector:

To use GC effectively while spark process is in execution, make sure GC uses its algo which reduces pause time for GC scans (Algo name - (Garbage-First Garbage Collector) or CMS (Concurrent Mark Sweep)

Cluster based Architecture:

1. Prod always uses this Archi
2. Process:
3. Application submission to YARN
4. YARN has RM. RM creates application master in different WN
5. AM creates Driver
6. Driver creates SC
7. SC interacts with RM and Asks for number of executor with config for each container
8. RM creates container is WN
9. Driver assigns tasks to each container / Executor
10. Driver keeps track of all the tasks performed in each executor
11. Once results are ready, Driver takes those results and all the executor gets decouples with the Driver

Spark Runtime Architecture :: Cluster Mode



E Available Executor

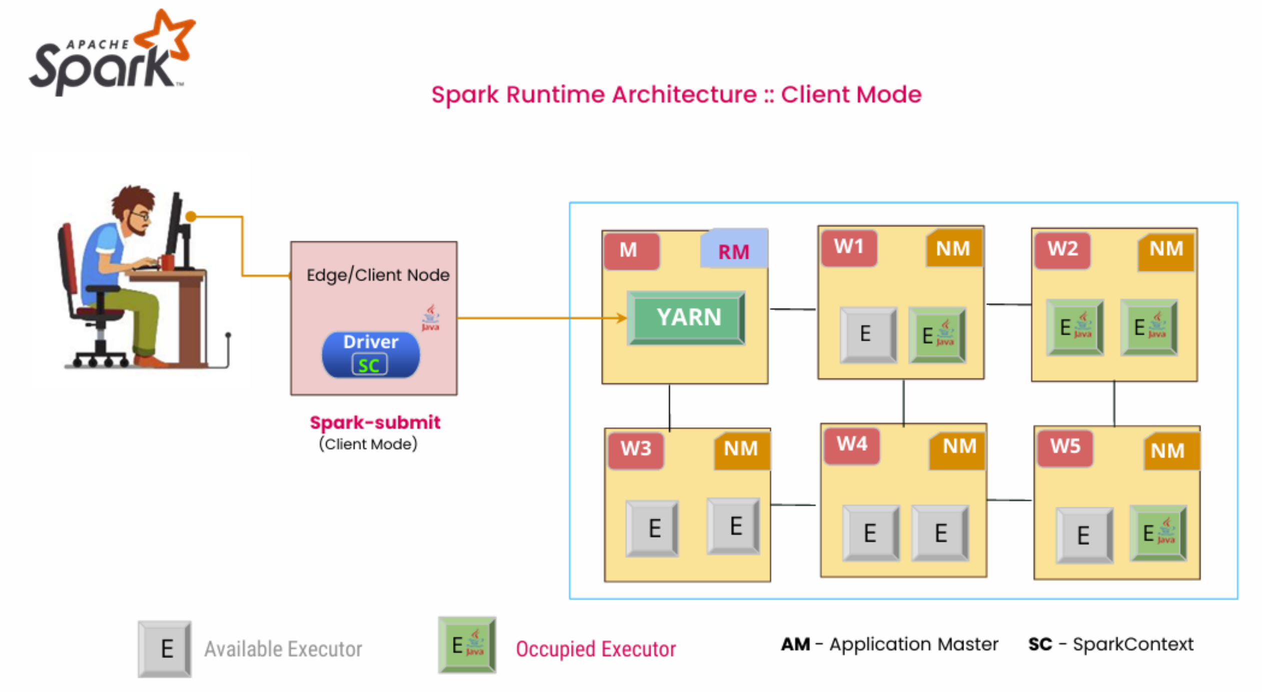
E Occupied Executor

AM - Application Master

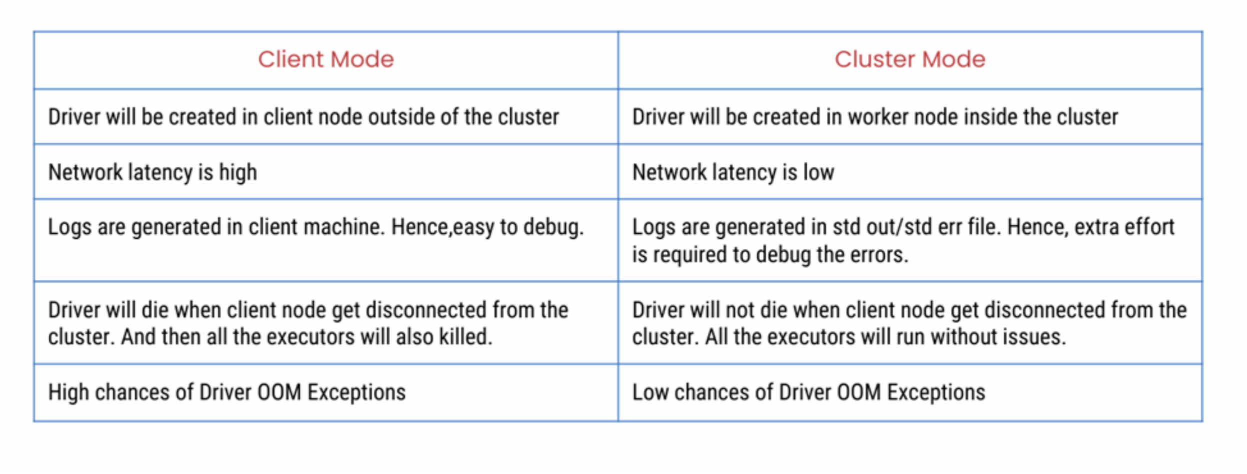
SC - SparkContext

Client based Architecture:

- Always used for Dev
- Driver is created on Client machine itself. Driver creates SC and SC interacts with RM
- Driver Program is responsible for coordinating task execution, collecting results and handling failure within the archi and remain active on client machine
- Issue – Power Failure (Loss of Computed data), **High latency**



Difference:



Client Mode	Cluster Mode
Driver will be created in client node outside of the cluster	Driver will be created in worker node inside the cluster
Network latency is high	Network latency is low
Logs are generated in client machine. Hence,easy to debug.	Logs are generated in std out/std err file. Hence, extra effort is required to debug the errors.
Driver will die when client node get disconnected from the cluster. And then all the executors will also killed.	Driver will not die when client node get disconnected from the cluster. All the executors will run without issues.
High chances of Driver OOM Exceptions	Low chances of Driver OOM Exceptions

Question :

How to do driver memory allocation and executor memory allocation ?

Ans:

Executor Memory Allocation

spark.executor.memory 8G

- Executing tasks assigned by the driver.
- Caching RDDs (Resilient Distributed Datasets) that are persisted by user programs.
- Storing shuffle data between stages.

Typical memory allocation

4 gb for small task and

16 for large task

Driver memory allocation:

- Maintaining information about the Spark application.
- Responding to the user's program or interactive queries.
- Distributing and scheduling tasks across executors.

spark.driver.memory 4G

small task: 1 to 2 gb
large task: 4 to 8 gb

How many executor CPU cores are required to process 25 GB data when all tasks are In parallel ? (Configuration might change)

Number of Partitions =(25*1024)/128
Number of CPU Cores=200
This is the case when there is parallel processing when 200 task are in parallel .If number of core are less than tasks will be in queue

How much each executor memory is required to process 25 GB data?

CPU cores for each executor =4

Memory for each executor =4*(128*4)=2GB

Expected memory for each core = Minimum 4 x (default partition size)

How many executors are required to process 25 GB data?

Avg CPU cores for each executor =4

Total number of executor =200/4=50

What is the total memory required to process 25 GB data?

Total number of executor =50

Memory for each executor =2GB

Total Memory for all the executor =50*2=100GB

Task and Job and Wide transformations

- Number of job = Number of action
- Number of task = Number of partions
  but (maximum number of task executing currently will be equal to number of cores)
- wide transformation is responsible to create **stages**
- Number of Stages = WT + 1
- how to decide number of executor - It is not defined by number of partitions ...
  but defined by number of core in each executor and availability of number of core
- job contains multiple task
- Number of task = number of executor --> No (false because number of partion depends on file size and partion size but number of executer depends upon number of core available and number of core in each executer)

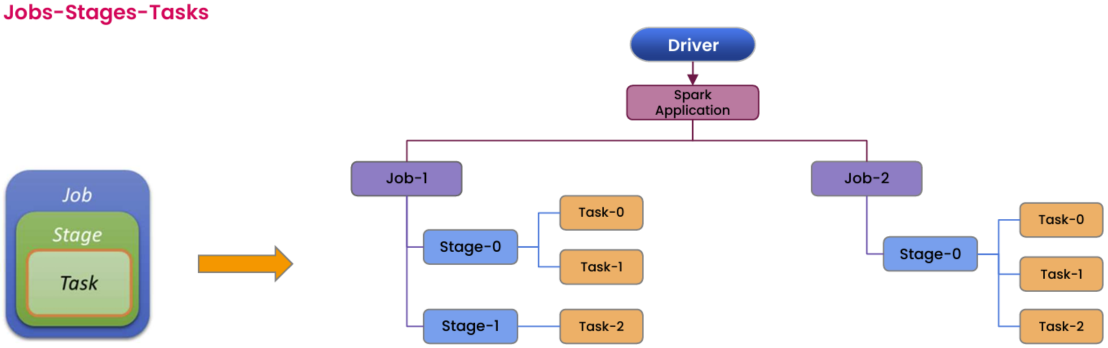
What is Spark DAG

In Apache Spark, DAG stands for Directed Acyclic Graph. It is a fundamental concept that represents the logical execution plan of a Spark application. The DAG describes the sequence of transformations and actions applied to the distributed datasets (RDDs or DataFrames) to produce thefinal results.

Note: Whole-Stage Code Generation (WSCG) is a performance optimization technique used in Apache Spark to improve the execution speed of DataFrame and Dataset operations. It is a key component of Spark's query optimization framework and plays a significant role in accelerating Spark applications

Seassion 3

Spark Jobs-Stages-Tasks



In Apache Spark, Driver doesn't perform any data processing. However it divides a Spark application into multiple jobs, and each job is further divided into stages, which in turn consist of individual tasks.

- **Spark Application:** The Spark application is the entire program that you submit to the Spark cluster for execution. It includes all the code, configurations, and dependencies required to perform data processing tasks using Apache Spark.
- **Jobs:** A Spark application is typically divided into multiple jobs. Each job represents a logical unit of work that consists of a sequence of transformations and actions on RDDs (Resilient Distributed Datasets) or DataFrames/Datasets.
- **Stages:** Each job is further divided into one or more stages. A stage represents a set of transformations that can be executed together in parallel. There are two types of stages: shuffle stages and non-shuffle stages. Shuffle stages involve data shuffling across the cluster, while non-shuffle stages do not.
- **Tasks:** Each stage is divided into multiple tasks. A task is the smallest unit of work in Spark and represents the execution of a single operation on a partition of the data.

Job: Number of jobs is equal to number of Actions

Stages: number of wide shuffling +1

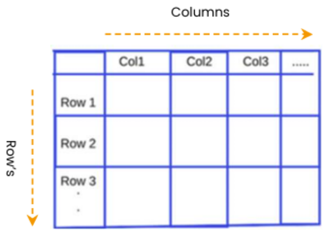
Special Case :
If the partition is 1, here Job will be 2, Stage will be 1 and task will be 1

Transformations and actions are two types of operations that we can perform on RDDs (Resilient Distributed Datasets) or DataFrames/Datasets in spark.

1. Transformations
   a. Transformations are lazy in nature, meaning they do not compute their results immediately. Instead, they create a new RDD or DataFrame, when an action is invoked
2. Actions
   a. Actions are operations that trigger the execution of the transformations and return a result to the driver program or write data to external storage

Dataframe :

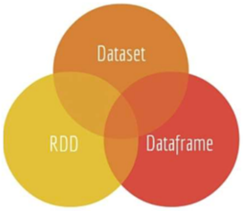
DataFrame



- A Data Frame is a distributed collection of data organized into named columns, conceptually equivalent to a table in a relational database.
- Apache Spark DataFrames are an abstraction built on top of Resilient Distributed Datasets (RDDs).
- DataFrames enable processing of large datasets across multiple nodes in a Spark cluster, offering both high-level operations (like selecting, filtering, aggregating) and the ability to execute SQL queries.
- Users can interact with DataFrames using domain-specific language (DSL) operations available in Scala, Java, Python, and R, making them more accessible than the lower-level RDDs (Resilient Distributed Datasets) for most data processing tasks.
- Create DataFrames from an existing RDD, from a Hive table, Structured Data files, external databases or Spark data sources.

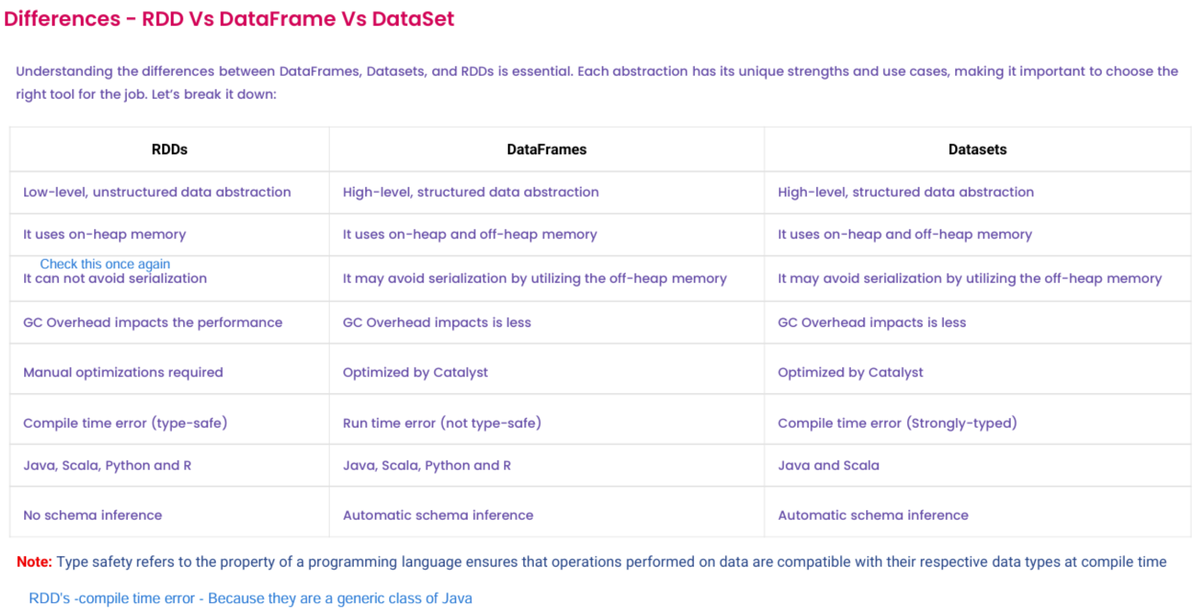
Dataset :

DataSet



- DataSet is a distributed collection of data that provides the benefits of both RDDs (Resilient Distributed Datasets) and DataFrames.
- Datasets provide a strongly-typed interface for working with distributed data in Spark. This allows for compile-time type safety, ensuring that operations are performed on data of the correct type.
- The Dataset API is available in Scala and Java. Python also has support for Datasets, although it is not as strongly-typed due to limitations in the Python language.
- Datasets can offer better performance than DataFrames for certain types of operations, particularly those that require more complex data transformations or involve user-defined functions.
- Datasets are not typically used for processing unstructured data, such as free-form text or binary data, as these types of data do not have a predefined schema.
- However, Datasets can handle semi-structured data to some extent. Semi-structured data refers to data that does not conform to a strict schema but has some organizational structure, such as JSON or XML data.

Difference



Differences - RDD Vs DataFrame Vs DataSet

Understanding the differences between DataFrames, Datasets, and RDDs is essential. Each abstraction has its unique strengths and use cases, making it important to choose the right tool for the job. Let's break it down:

RDDs	DataFrames	Datasets
Low-level, unstructured data abstraction	High-level, structured data abstraction	High-level, structured data abstraction
It uses on-heap memory	It uses on-heap and off-heap memory	It uses on-heap and off-heap memory
Check this once again It can not avoid serialization	It may avoid serialization by utilizing the off-heap memory	It may avoid serialization by utilizing the off-heap memory
GC Overhead impacts the performance	GC Overhead impacts is less	GC Overhead impacts is less
Manual optimizations required	Optimized by Catalyst	Optimized by Catalyst
Compile time error (type-safe)	Run time error (not type-safe)	Compile time error (Strongly-typed)
Java, Scala, Python and R	Java, Scala, Python and R	Java and Scala
No schema inference	Automatic schema inference	Automatic schema inference

Note: Type safety refers to the property of a programming language ensures that operations performed on data are compatible with their respective data types at compile time

RDD's -compile time error - Because they are a generic class of Java

RDD store everything in form of string

Session 4

Slowly Changing Dimensions (SCD's)

Type 0 – Fixed Dimension :

No changes allowed, dimension never changes
Our table remains the same. This means our existing data will continue to show the same figures

Type 1 – No History: Only the current state is stored. When an attribute value changes, it overwrites the old value, and no history is preserved.

Type 2 – Row Versioning: Keeps full history by adding a new record with the updated values, often with start and end dates to indicate the validity period of a particular record. This allows for maintaining a complete history of value changes

Type 3 – Previous Value Column: Maintains the current value and the previous value in separate columns within the same record. This allows tracking of limited history.

Suitable when only the most recent change is relevant for historical analysis

Here only two values wll be stored (Present and Previous)

Type 4 – History Table: Uses a separate history table to track changes, keeping the current data in the original table and detailed history in the secondary table.

Beneficial for maintaining a clear separation between current and historical data, optimizing performance for queries against current state data

Type 6 – Hybrid (1+2+3): Combines techniques from Types 1, 2, and 3 to track current and previous data and full history.

Uses when the requirements demand the benefits of Types 1, 2, and 3 simultaneously, offering a comprehensive approach to data change management

Magic commands

Magic commands in Databricks are special commands that begin with % or %% and provide shortcuts for performing various tasks within Databricks notebooks. Here's a breakdown of commonly used magic commands

Databricks Utilities

Databricks Utilities offer a range of functionalities to simplify and enhance your work within Databricks notebooks. These utilities are accessible through the dbutils module. Below are some commonly used utilities:

- **File System Utilities**
- **Notebook Utilities**
- **Widgets Utilities**

DBFS

When you create DB workspace, it create its own storage which is called dbfs and can be access using below command

```
% fs
ls dbfs /:
```

Dbfs stores data such as when you upload notebooks or anything you upload in workspace

Data of managed table is stored in dbfs where as

data of external table is stored in ADLS which we created and established link between DBX and ADLS using SPN

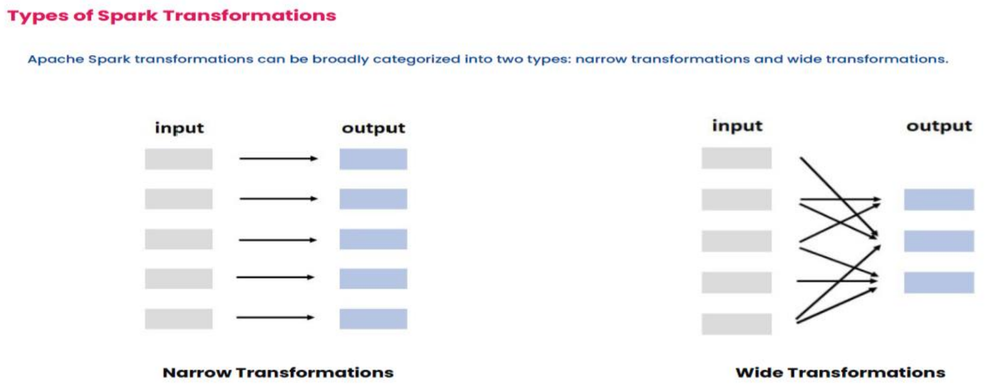
in case of managed table all the data is stored in dbfs .. that is internal storage ..that is created along with the databricks itself .

When we create DBX, it created its own Datastorage account that it uses as an HDFS storage. Now, when we need to create hive tables, be it managed or external we need to read data from file which can be either present in any location (dbfs, S3, ADLS, GCP). When we need to create hive table we create a dataframe first. After creating dataframe, we can save it as a managed table using (df.write.format("parquet").saveAsTable(permanent_table_name)) or external table using (df.write.format("parquet").mode("overwrite").save(external_location))

Table deletion case in catelog : if you delete table from catelog which is managed table, then the unerslying data will be delete from “/user/hive/warehouse/” but whe you delete table from catelog which has underlying location as external. Only table gets deleted and not the actual file stored in ADLS (parquet file)

Narrow Transformation:

- ❖ narrow transformations are operations that do not require shuffling of data across partitions.
- ❖ These transformations can be performed on individual partitions independently, without the need to exchange data between partitions.
- ❖ Examples of narrow transformations include map, filter, flatMap, mapPartitions, union, intersection, distinct, sample, etc.
- ❖ Narrow transformations are typically more efficient because they operate on each partition separately and can be executed in parallel across partitions.



Types of Spark Transformations

Apache Spark transformations can be broadly categorized into two types: narrow transformations and wide transformations.

Wide Transformations :

❖ Wide transformations are operations that require shuffling of data across partitions.

❖ These transformations involve data exchange between partitions, often resulting in data movement and redistribution across the cluster.

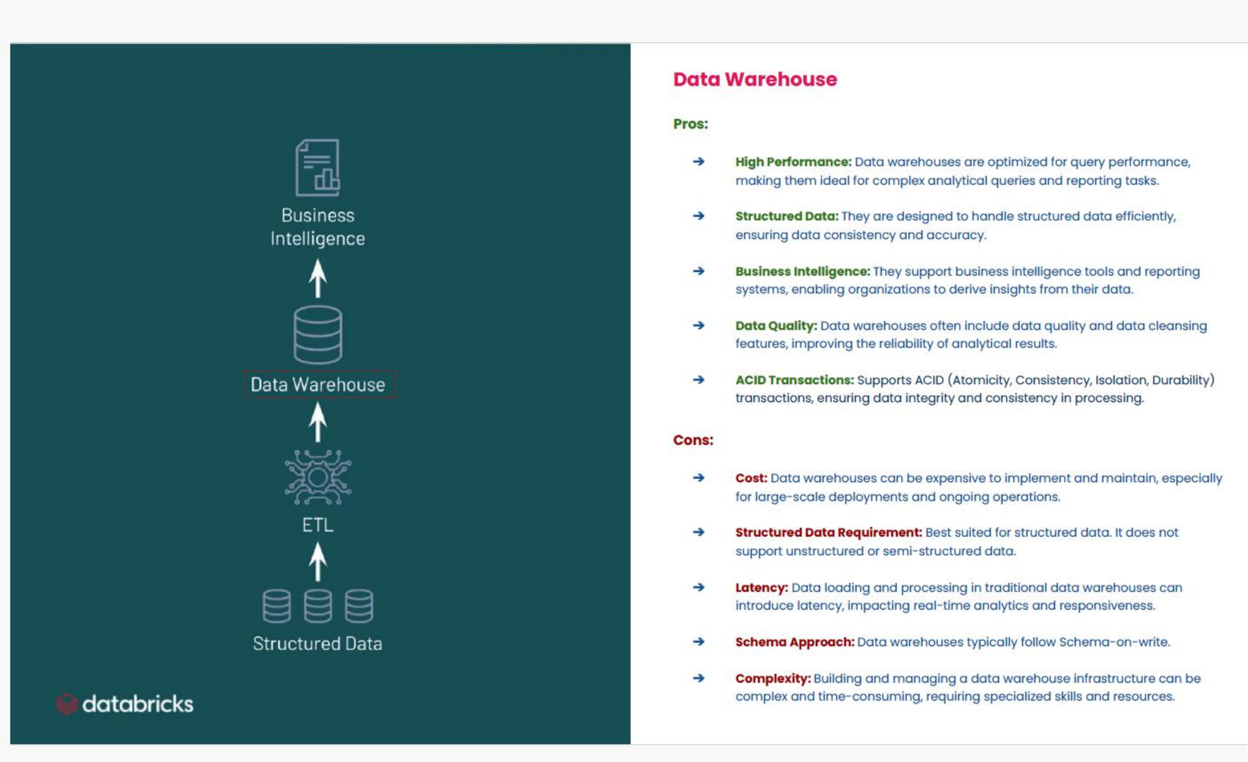
❖ Examples of wide transformations include groupBy, reduceByKey, aggregateByKey, join, sortByKey, cogroup, combineByKey, etc.

❖ Wide transformations may require significant network communication and can be more computationally expensive compared to narrow transformations.

❖ They typically involve a shuffle stage, where data is redistributed and sorted across partitions based on a specified key.

Session 5 :

Data warehouse :



Data Warehouse

Pros:

- **High Performance:** Data warehouses are optimized for query performance, making them ideal for complex analytical queries and reporting tasks.
- **Structured Data:** They are designed to handle structured data efficiently, ensuring data consistency and accuracy.
- **Business Intelligence:** They support business intelligence tools and reporting systems, enabling organizations to derive insights from their data.
- **Data Quality:** Data warehouses often include data quality and data cleansing features, improving the reliability of analytical results.
- **ACID Transactions:** Supports ACID (Atomicity, Consistency, Isolation, Durability) transactions, ensuring data integrity and consistency in processing.

Cons:

- **Cost:** Data warehouses can be expensive to implement and maintain, especially for large-scale deployments and ongoing operations.
- **Structured Data Requirement:** Best suited for structured data. It does not support unstructured or semi-structured data.
- **Latency:** Data loading and processing in traditional data warehouses can introduce latency, impacting real-time analytics and responsiveness.
- **Schema Approach:** Data warehouses typically follow Schema-on-write.
- **Complexity:** Building and managing a data warehouse infrastructure can be complex and time-consuming, requiring specialized skills and resources.

Data Warehouse – Walmart – All objects will be placed in a systematic manner

Vertical Scalable – not possible unlimited (You have a machine, you can increase RAM or Storage or etc to some extent)
Limited data (Petabytes)
Structured Data – Data Stored in Structured Format
Connect BI tools directly
ETL -> Good quality data
Support ACID

Cons:

Cost
Structured Data Only
Latency
Schema Approach – Schema on-Write (When you are writing something and it is throwing an error is called schema on write)
(When you load a string value into an integer datatype column it will throw (Data Mismatch type) error)
Complexity

Data Lake – Structured Data/ Semistructured Data / Unstructured Data

Pros :

Flexibility – Diverse Data type Storage
Horizontally Scalability – Unlimited (here Machines can be placed parallel)
Cost Effective (Different types of S3 Storage and different types of payment methods)
Schema on read – Even if the datatype does not match, it won't throw an error and it will replace the null value (When you load a string value into an integer datatype column it won't throw an error and that string value is replaced by null)
Data Retention policies available

Cons :

Data Retrival is chalenging
Chalenging Data Quality (Raw and Unprocessed)
We cannot implement ACID

ADLS Gen 2 – create a container with no hierarchy– Data Lake
Delta Lake – create a container with hierarchy – Delta Lake

Delta lake:

Extension of Apache Parquet file format
Supports ACID properties
Schema Enforcement in write
Flexible with schema i.e columns can be added or deleted / Data type can be changed / Columns can be renamed
Time Travel (Metadata is stored in JSON format in the backend)
Various Optimization techniques
Supports streaming and batch processing

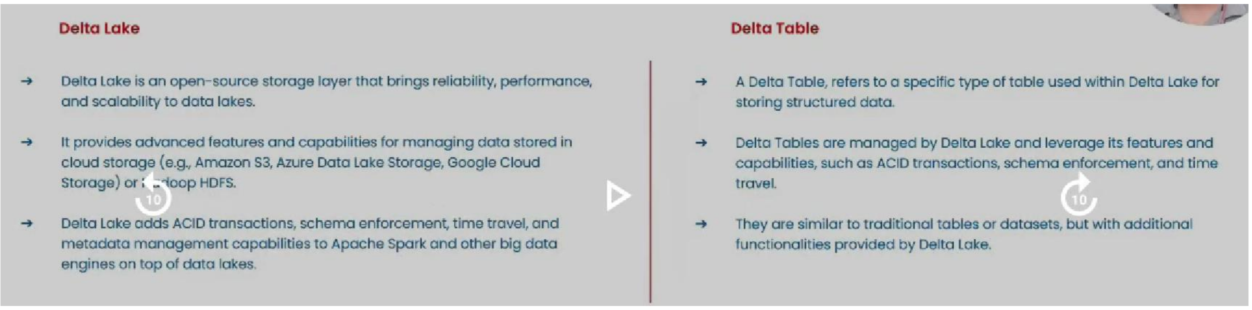
Ingredients for Lakehouse Architecture

- DW
- Data Lake
- Delta Lake

Delta Lake provides ACID transactions, Scalable metadata handling, and unfies streaming and batch data processing on top of existing data lakes, such as S3, HADFS, ADFLS Gen2 , GCS

Delta Lake is open source and Databricks is using this Delta Lake concept

Delta Lake uses Delta Table for storing **structured data** only
Delta Lake just uses the feature of the Delta table
By default, any table created in Databricks is a delta table



Delta Lake

→ Delta Lake is an open-source storage layer that brings reliability, performance, and scalability to data lakes.

→ It provides advanced features and capabilities for managing data stored in cloud storage (e.g., Amazon S3, Azure Data Lake Storage, Google Cloud Storage) or Hadoop HDFS.

→ Delta Lake adds ACID transactions, schema enforcement, time travel, and metadata management capabilities to Apache Spark and other big data engines on top of data lakes.

Delta Table

→ A Delta Table, refers to a specific type of table used within Delta Lake for storing structured data.

→ Delta Tables are managed by Delta Lake and leverage its features and capabilities, such as ACID transactions, schema enforcement, and time travel.

→ They are similar to traditional tables or datasets, but with additional functionalities provided by Delta Lake.

Delta Lakehouse architecture combines the best features of data lakes, data warehouses, delta lake

Delta Lakehouse supports structured, unstructured, and semi-structured data

How SCDs are used for the below cases

1. Transactional and Dimensional Tables

Transactional Tables:

- **Nature**: Store individual transactions, often with details about events, such as sales, purchases, or log entries.
- **SCD Usage**: Typically, transactional tables do not employ SCDs directly because they record events as they occur without needing to maintain historical versions of those events.

Dimensional Tables:

- **Nature**: Store attributes about business entities such as customers, products, or locations. These attributes change slowly over time.
- **SCD Usage**:
    - **Type 1**: Overwrite the old data with new data. There is no history of previous data.
        - *Use Case*: Correcting an error in the data (e.g., fixing a customer's address).
    - **Type 2**: Create a new record with a new surrogate key whenever a change occurs, preserving the history.
        - *Use Case*: Tracking changes in customer information over time (e.g., change in address or marital status).
    - **Type 3**: Add new columns to store previous values for certain attributes.
        - *Use Case*: Keeping track of limited historical data, such as previous and current values (e.g., storing both previous and current sales region).

2. Streaming and Batch Processing

Streaming Processing:

- **Nature**: Continuously processes data in real-time as it flows into the system.
- **SCD Usage**:
    - **Type 1**: Apply changes immediately as they arrive, overwriting old values.
    - **Type 2**: Insert new records for each change event, tagging them with effective and expiry dates.
    - *Example*: A real-time customer profile update stream where changes are continuously applied to the customer dimension table.

Batch Processing:

- **Nature**: Processes data in large chunks or batches at scheduled intervals.
- **SCD Usage**:
  - **Type 1**: Overwrite the existing records in each batch update.
  - **Type 2**: Append new records for changes detected in each batch, preserving historical records.
  - *Example*: A nightly batch job that updates the product dimension table with changes from the day.

3. Incremental and Full Load

Incremental Load:

- **Nature**: Only loads new or changed data since the last update.
- **SCD Usage**:
  - **Type 1**: Update existing records with changes detected in the incremental load.
  - **Type 2**: Insert new records for changes detected in the incremental load, marking the previous records as expired.
  - *Example*: Loading only new sales data or changes in product attributes from the last load into the data warehouse.

Full Load:

- **Nature**: Reloads the entire dataset, often used for initial loads or complete refreshes.
- **SCD Usage**:
  - **Type 1**: Replace the entire table with the new data.
  - **Type 2**: Depending on the business requirements, you might need to merge the new full load with existing data, creating new records for changes and preserving historical data.
  - *Example*: Reloading the customer dimension table entirely after a significant schema change or data correction.

Confirm the below

Data lake - Schema on reading
DW - Schema on write
Delta Lake – Schema on reading
Delta Table – Schema on write

By default, any table created in Databricks is a delta table

Lets say we have 1k records and we delete 500 records. After that we used update functions for few times on remaining records. Now we want to retrive those 500 deleted records using time travel feature of delta table/ Databricks. From where will it get those 500 records as we have previously dfeleted those records?

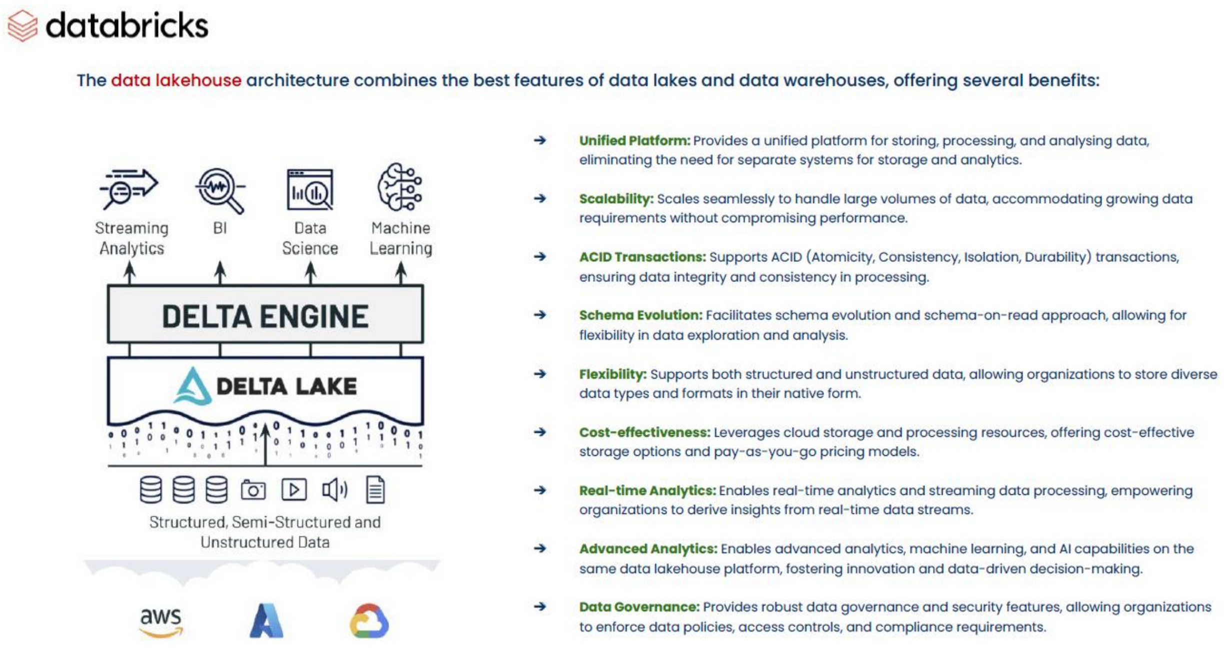
Explanation

When we delete 500 records, those records are deleted logically and not the physical underlying file. Those 500 records can be retrived using time travel function provide by delta table

Command : **select * from Table_name where Version as of X**

Also if we want to restore the previous version we can do that by using this command

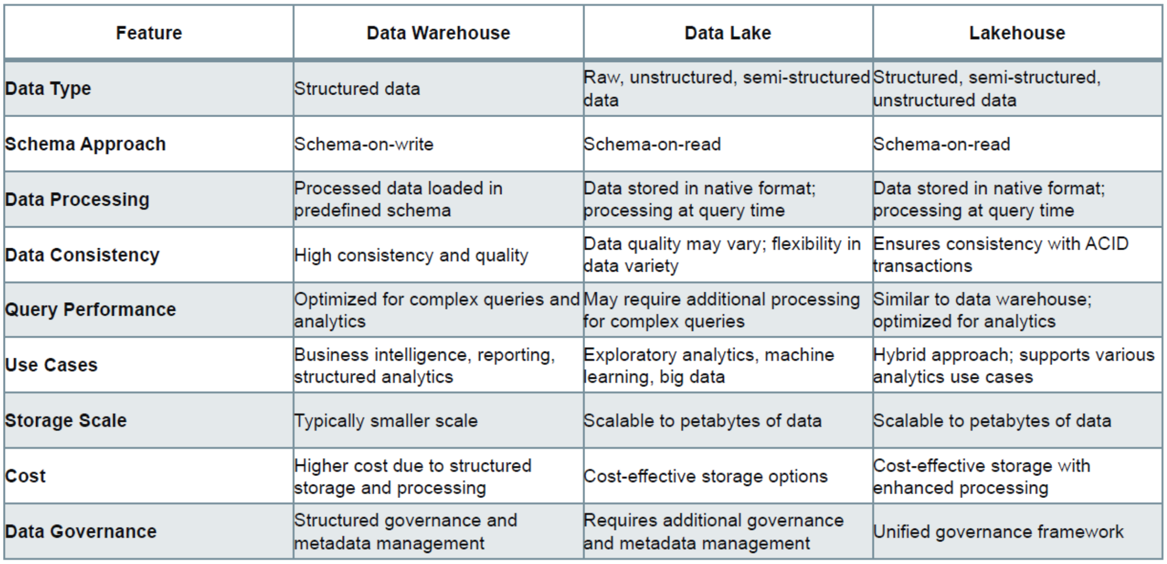
Restore Table Table_name to version as of X



The data lakehouse architecture combines the best features of data lakes and data warehouses, offering several benefits:

- **Unified Platform:** Provides a unified platform for storing, processing, and analysing data, eliminating the need for separate systems for storage and analytics.
- **Scalability:** Scales seamlessly to handle large volumes of data, accommodating growing data requirements without compromising performance.
- **ACID Transactions:** Supports ACID (Atomicity, Consistency, Isolation, Durability) transactions, ensuring data integrity and consistency in processing.
- **Schema Evolution:** Facilitates schema evolution and schema-on-read approach, allowing for flexibility in data exploration and analysis.
- **Flexibility:** Supports both structured and unstructured data, allowing organizations to store diverse data types and formats in their native form.
- **Cost-effectiveness:** Leverages cloud storage and processing resources, offering cost-effective storage options and pay-as-you-go pricing models.
- **Real-time Analytics:** Enables real-time analytics and streaming data processing, empowering organizations to derive insights from real-time data streams.
- **Advanced Analytics:** Enables advanced analytics, machine learning, and AI capabilities on the same data lakehouse platform, fostering innovation and data-driven decision-making.
- **Data Governance:** Provides robust data governance and security features, allowing organizations to enforce data policies, access controls, and compliance requirements.

Data Warehouse Vs Data Lake Vs Data Lakehouse



Feature	Data Warehouse	Data Lake	Lakehouse
Data Type	Structured data	Raw, unstructured, semi-structured data	Structured, semi-structured, unstructured data
Schema Approach	Schema-on-write	Schema-on-read	Schema-on-read
Data Processing	Processed data loaded in predefined schema	Data stored in native format; processing at query time	Data stored in native format; processing at query time
Data Consistency	High consistency and quality	Data quality may vary; flexibility in data variety	Ensures consistency with ACID transactions
Query Performance	Optimized for complex queries and analytics	May require additional processing for complex queries	Similar to data warehouse; optimized for analytics
Use Cases	Business intelligence, reporting, structured analytics	Exploratory analytics, machine learning, big data	Hybrid approach; supports various analytics use cases
Storage Scale	Typically smaller scale	Scalable to petabytes of data	Scalable to petabytes of data
Cost	Higher cost due to structured storage and processing	Cost-effective storage options	Cost-effective storage with enhanced processing
Data Governance	Structured governance and metadata management	Requires additional governance and metadata management	Unified governance framework

Session 7:

Incremental Is common in Batch and streaming

Till now we know that incremental load can be done using watermark Column

Spark Structured streaming

- Manual Schema evolution
- Automatic Schema evolution

To understand Autoloader, below concept needs to be understood

- Incremental laod (watermarks)
- Manual schema evollution
- Automatic Schema evolution

Streaming

- RDD – using destream
- Df – using structired streaming

Since autoloader is already in existance, we can ignore spark structured streaming. But to unserstand autoloader we need to understand spark structured streaming

Spark structured streaming

- Manual Schema Evolution
- Automatic Schema Evolution

Manual Schema Evolution :

General steps in case of manual schema evolution

1. Read Stream : it refers to continous and sequential read of data from data source
2. Write Stream : it refers to continous and sequential writing of data to the destination

In case of manual there was inferschema in readstream

New incomming data will be loaded in raw table smoothly until there is change in schema. From T1 to T19, loading will be smooth but in case of T20, there is change in schema. Here we will have to manually change schema in raw table. Once there changes are done, things will run smoothly

Spark structure streaming with manual schema evolution:

Source s:5 Target table T:5

Readstream is used

We have to create schema for manual schema

S:4 T:5

Null will be there in the table

S:6 T:5

When we have columns more then table column it will give us exception

We have to alter the table in case of manual schema evolution

trigger() Method in Spark Structured Streaming

Overview:

- The trigger() method determines how often the streaming query processes new data.

Triggers in case of manual Spark Structured Streaming:

12. **AvailableNow Trigger**:
    - Immediately triggers the streaming query to process available data without delay.
    - Beneficial for near real-time processing scenarios.

13. **Other Available Triggers**:
    - **once()**: Runs the query once and stops.
    - **continuous(interval)**: Generates micro-batches at specified intervals, offering lower latency.
    - **processingTime(interval)**: Schedules periodic query runs based on a time interval.
    - **default()**: Uses the default trigger, usually equivalent to processingTime(0 seconds).

Explanation:

- **once()**: Useful for batch processing or testing.
- **continuous(interval)**: Suitable for low-latency, continuous data processing.
- **processingTime(interval)**: Appropriate for periodic processing based on time intervals.
- **default()**: Typically processes data as soon as it's available, similar to availableNow.
-

If we don’t want to update schema manually, we can use Automatic Schema evolution

Scriot for Manul Spark Structured Streaming :

Step 1 -> Readstream (InferSchema)

Step 2 -> Inferschema
merge Schema

Check point will act as watermark column

- Metadatastorage
- it will store off set volume (Last prices related information)

Spark structure streaming with automatic schema evolution:
Enable schema inference
Source s:5 Target table T:5
Readstream is used
Automatically inferred schema
S:4 T:5
Null will be there in the table
Infer schema true
Merge schema true
S:6 T:5
Same code we will run no changes in this case
It will smoothly and will insert null for rest of the records
When we have columns more then table column it will give us exception
We dont need to rerun or anyting
Same code can be used at all the locations

Summary of Schema Inference and Management in Spark Autoloader

14. **Initial Read and Schema Inference**:
    - When Spark Autoloader or an automated stream reads a file for the first time, it infers the schema from the file.
    - The inferred schema is then stored in the offset metadata, which tracks the schema information along with the data processing offset.
15. **Schema Comparison and Data Processing**:
    - On subsequent reads, Spark Autoloader infers the schema of the incoming data again and compares it with the stored schema in the offset metadata.
    - If the schema has **not changed**:

        - Spark uses the existing schema from the offset metadata.
        - It reads the data directly and processes it using the writeStream operation without any schema updates.
    - If the schema **has changed**:
        - Spark infers the new schema and updates the schema information in the offset metadata.
        - The new schema is then used to read and process the data, ensuring that the data conforms to the updated schema before writing.

16. **Handling Schema Evolution**:
    - Spark Autoloader supports schema evolution, which means it can handle changes to the schema over time.
    - When a schema change is detected, the offset metadata is updated to reflect the new schema.
    - This ensures that any subsequent data reads and writes use the latest schema, maintaining consistency in the data processing pipeline.

Key Points to Note:

- **Schema Inference**: The schema is inferred from the data on initial and subsequent reads.
- **Offset Metadata**: The inferred schema is stored in offset metadata, along with the processing offset information.
- **Schema Comparison**: On subsequent reads, the inferred schema is compared with the stored schema in the offset metadata.
- **Schema Consistency**: If the schema has not changed, the stored schema is used. If the schema has changed, the offset metadata is updated with the new schema.
- **Schema Evolution**: Spark handles schema changes dynamically, ensuring that the data processing pipeline adapts to schema updates seamlessly.

Example Workflow:

17. **First Read**:
    - Infer schema from the file.
    - Store the inferred schema in the offset metadata.
    - Read data and process it.
18. **Subsequent Reads** (No Schema Change):
    - Infer schema from the new file.
    - Compare inferred schema with the stored schema.
    - If schemas match, use the stored schema.
    - Read data and process it.
19. **Subsequent Reads** (With Schema Change):
    - Infer schema from the new file.

- Compare inferred schema with the stored schema.
- If schemas do not match, update the schema in the offset metadata.
- Use the updated schema to read and process data.

Note : The above steps apply are not only for auto loader but also automatics schema evolution

My Notes :

Just give path of offset volumn to autoloader or spark structured stream. It will compare that offset metadata with landing conatiner and accordingly read the latest file based on last process runtime.

In further run it will create schema. In next run, it will compare schema of 2nd file with the schema of 1st file which is stored in metadata format as an offset volume in DB checkpointing. If there is any change in new schema while comparing with previous schema, it will automatically update the schema of raw table

Autoloader Concept

If Automatic Spark streaming is doing everything, why do we need Autoloader?
Because autoloader has few merits over automatic Spark Structured Strem

There are few drawback of using automatic Spark Structured Strem

Drawback 1 :

If our client said, change the datatype of product_id which is int to string. Here there will be datatype missmatch error when spark automatic strem will try to puch the data in raw table

Drawback 2 :

As lakehouse architecture us schema on read. Therefore even if there is bad data, you won't stop appending. Only Null will be displayed in case of bad data

Downstream team might face issue because of null value while performing aggregation. Also tracing back as to from which file this bad data came is difficult

Autoloader address this above issue

Addressing Drawback 1 :

Here we use datatype hit

When we create autoloader syntax, there is a option for us to put datatype hint

Ex : “Col 1 String”

Though we infer schema, since we explicitly write this command, it will create string column only

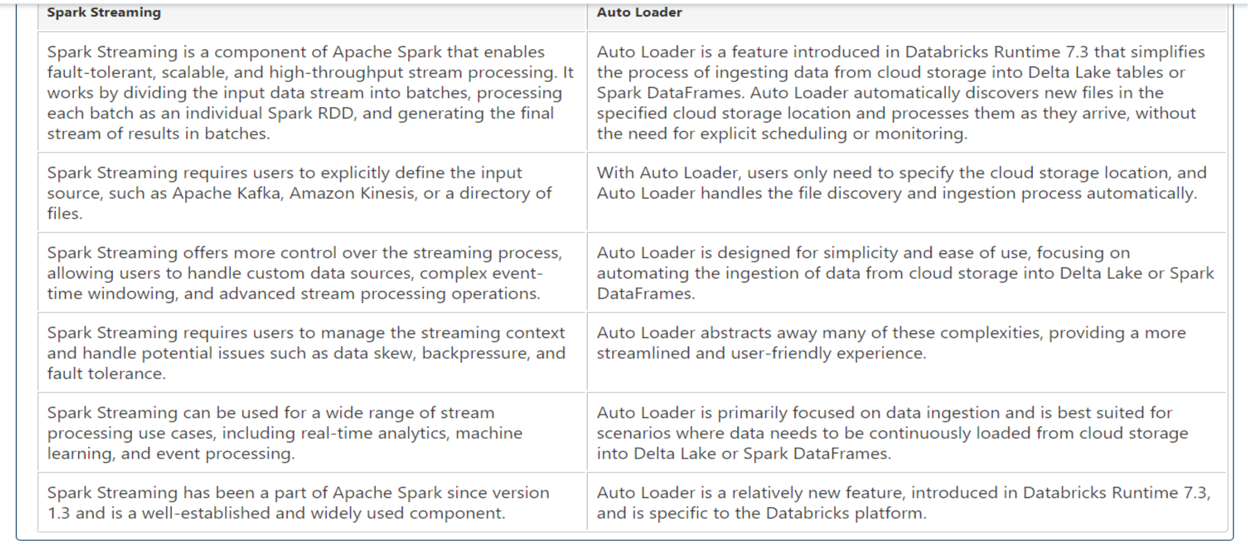
Realtime use case :

Lets say we have provided employee data to out client witht particular schema in project phase 1.0 Now, in our phase 2.0 we decided to some anlytics on our employee data which basically chngaed the datatype of the original column, but our client will need same datatype and format of the data. In this case, we can use auto loader while writing data from internediate to curated layer where we will specifically mention HINT to change the datatype of those column.

Autoloader use cloudfiles

Cloudfiles – Prebuilt function inside databricks that help us to achieve autoloader concepts

Cludfile k thorough auto k concept use hote hai



Spark Streaming	Auto Loader
Spark Streaming is a component of Apache Spark that enables fault-tolerant, scalable, and high-throughput stream processing. It works by dividing the input data stream into batches, processing each batch as an individual Spark RDD, and generating the final stream of results in batches.	Auto Loader is a feature introduced in Databricks Runtime 7.3 that simplifies the process of ingesting data from cloud storage into Delta Lake tables or Spark DataFrames. Auto Loader automatically discovers new files in the specified cloud storage location and processes them as they arrive, without the need for explicit scheduling or monitoring.
Spark Streaming requires users to explicitly define the input source, such as Apache Kafka, Amazon Kinesis, or a directory of files.	With Auto Loader, users only need to specify the cloud storage location, and Auto Loader handles the file discovery and ingestion process automatically.
Spark Streaming offers more control over the streaming process, allowing users to handle custom data sources, complex event-time windowing, and advanced stream processing operations.	Auto Loader is designed for simplicity and ease of use, focusing on automating the ingestion of data from cloud storage into Delta Lake or Spark DataFrames.
Spark Streaming requires users to manage the streaming context and handle potential issues such as data skew, backpressure, and fault tolerance.	Auto Loader abstracts away many of these complexities, providing a more streamlined and user-friendly experience.
Spark Streaming can be used for a wide range of stream processing use cases, including real-time analytics, machine learning, and event processing.	Auto Loader is primarily focused on data ingestion and is best suited for scenarios where data needs to be continuously loaded from cloud storage into Delta Lake or Spark DataFrames.
Spark Streaming has been a part of Apache Spark since version 1.3 and is a well-established and widely used component.	Auto Loader is a relatively new feature, introduced in Databricks Runtime 7.3, and is specific to the Databricks platform.

Autoloader Connector with CloudFiles

- **CloudFiles:**
  - Refers to files stored in any cloud storage (e.g., Amazon S3, Azure Blob Storage).
- **Usage in Databricks:**
  - When using .format('CloudFiles') in Databricks, it signifies the use of the Autoloader connector.
- **Autoloader Connector:**
  - Facilitates data ingestion from cloud storage into Databricks Delta tables.
  - Automates the process of loading various file formats (CSV, Parquet, JSON, etc.) from cloud storage into Delta tables.

- **Functionality:**
  - Connects to and processes data stored in cloud storage within the Databricks environment.
  - Simplifies the loading and analysis of cloud-based data in Databricks.
- **Benefits:**
  - Automates data loading from cloud storage into Delta tables.
  - Enables seamless processing and analysis of cloud-based data within Databricks.

inferSchema Option and Auto Loader in Spark Structured Streaming

- **inferSchema Option:**
  - Used in Spark Structured Streaming for inferring schema when reading static data sources like files or databases.
  - Not available for use with the Auto Loader in Databricks.
- **Auto Loader and Schema Inference:**
  - In Auto Loader, schema inference is automatically enabled by setting cloudFiles.inferColumnTypes option to true.
  - This built-in mechanism in Auto Loader handles schema inference without the need for explicitly setting inferSchema option.
- **Usage Explanation:**
  - Auto Loader is designed for streaming data from cloud storage (e.g., Azure Blob Storage, Amazon S3, Google Cloud Storage).
  - It has its own schema inference mechanism tailored for cloud file formats, eliminating the need for inferSchema option.
- **Benefits of Auto Loader's Schema Inference:**
  - Simplifies data loading from cloud storage into Delta tables without manual schema specification.
  - Ensures accurate schema inference for cloud file formats, improving data processing efficiency.

Autoloader concept :

- Datatype Hint
- Rescued data
- When you detect column first time, it will fail first time
  To fix this : we have to rerun
  This is done purposefully -> it is a kind of heads up to tell us that new column is created

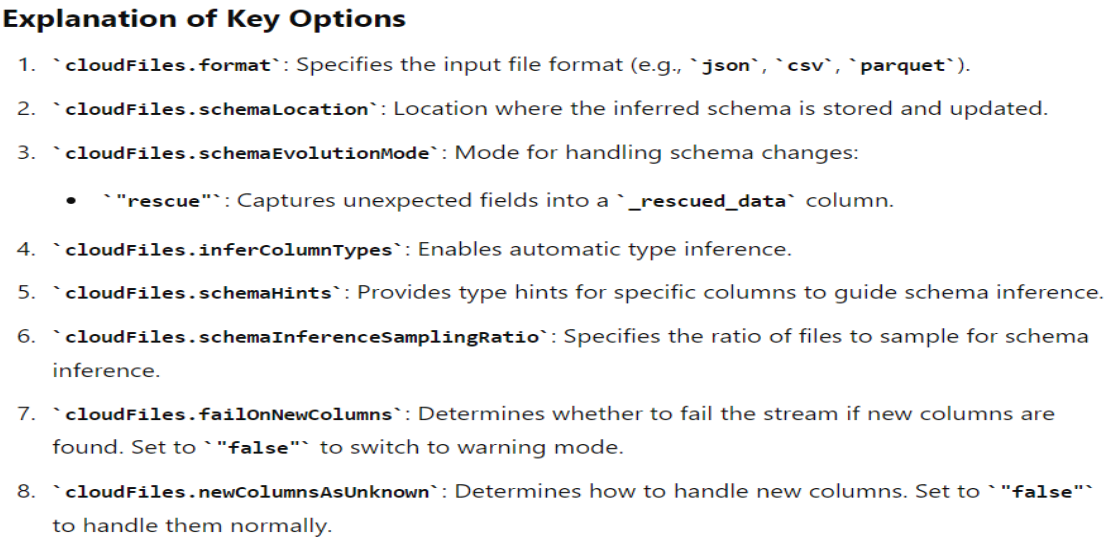
Note :

The data is has to be in cloud then we can use autoloader
We have to move the data into landing zone if we have it in onsite
Checkpointing is used for write and delete
Autoloder is framwork and cloudfile is connecter
First time autoloader will fails

Select * from table_name Where _rescued_data is not null

.option(“cloudFiles.schemaHints”,”id string”)

Images :



Explanation of Key Options

1. `cloudFiles.format`: Specifies the input file format (e.g., `json`, `csv`, `parquet`).
2. `cloudFiles.schemaLocation`: Location where the inferred schema is stored and updated.
3. `cloudFiles.schemaEvolutionMode`: Mode for handling schema changes:
   - `"rescue"`: Captures unexpected fields into a `_rescued_data` column.
4. `cloudFiles.inferColumnTypes`: Enables automatic type inference.
5. `cloudFiles.schemaHints`: Provides type hints for specific columns to guide schema inference.
6. `cloudFiles.schemaInferenceSamplingRatio`: Specifies the ratio of files to sample for schema inference.
7. `cloudFiles.failOnNewColumns`: Determines whether to fail the stream if new columns are found. Set to `"false"` to switch to warning mode.
8. `cloudFiles.newColumnsAsUnknown`: Determines how to handle new columns. Set to `"false"` to handle them normally.

Session 8

Optimization is done to Save cost, time

Agenda – to learn concept of Optimize, z-order, Vacuum

OPTIMIZE:

Small file issue (5-20 MB) : (Time Optimzation)

If you have small files which are coming every min and your pipeline is scheduled to run every 4 Hour, in this case you will have many small files to process at once

Lets say we have, 500 small file (5 MB) and our query needs to process 200 MB data. In this case, we will have 500 partitions and these will be processed in series (5 at once because 5 core available). Basically files will be in queue (about 100 batch of file file each). This processing will take too much time. If we optimize and merge all file to make 5 files having 100 small files in one big file of 500 MB.In this case there will be only one batch process which will significantly reduce the processing time

Delta table numerous log file issue: (Storage optimization / Cost Opti)

Needs to be confirmed

In this case, whenever you to trnsformation of delta table, it create a log file (Snapshort). Initailly you will be good but eventually you will have lot of log files in delta log folder.

To apply the optimization technique, you need to identify key column that the downstream applications will be using frequently

(Only for understanding)

Let’s say we have 500 small files and each file is around 4 MB. Total size of all file will be 500*4 = 2000MB = 2GB. When we optimize, spark will convert for example 50 file in one file. So in total it will create 10 file (50 * 10 = 500) but at the same time it will convert csv to parquet which will reduce the size of single file from 4 mb(CSV file) to 2 mb(parquet file) for example. Therefore the total size of 2GB(CSV file) will reduce to 1 GB in parquet format for example

Actually it won’t convert but by default when you read and write your csv file in delta table, by default that file is converted into parquet format

Therefore in optimize concept, it will reduce the number of files and also size of each file
Logically it will delete to 500 files and only keep 10 parquet files
Read operation is faster on parquet file (column based fileB)

Below text needs to confirmed

RAW layer ADLS file -> 500 Files -> 5 GB csv file (total of small MB file)
Read and write in delta table -> 500 file -> 2.41 GB parquet files (total of small parquet format file)

Optimize -> 50 file in 1 file
therefore total 10 large file and total size 2.41 GB

Here the drawback is we are deleting those 500 parquet file of size 5GB logically only. They we still sitting in DB delta lake. So previous 5 GB 500 parquet file and now newly created 10 file with total size of 5 GB. Now delta lake will around 10GB of file Viz in effecient. We need to delete(vacuum) there 500 parquet file from storage.

We will use Optimize command in SQL

It will combine small size into large file Max size is 1 gb

Based on each file size , clusture perfomance and number of file it will adjust

To work around the size we can use

SET spark.databricks.delta.optimize.write.maxFileSize = 1g

Spark.conf.set('spark.databricks.delta.optimize.write.maxFileSize' , '1g')

Z-order

Observe which columns like order_id , customer_id, order date is used by downstream application and then prefer z-order technique using that column

Z-order is a technique used in delta lake to optimize query performance by organizing data with delta tables based on value of one or more column. It arranges the data within delta tables based on specific column values, improve data locality and reducing amount of data that needs to be read during query execution

It is recommend to use optimize along with z order

You should not run optimize daily. It depends on use case. You can recommend to run optimize and z order queries once a week.

It will reduce some suffling

OPTIMIZE salesdata ZORDER by (order_date)

Default path of the managed table :

User/hive/warehouse/tablename

Number of files would be same :

Three table timing

Original :25 sec

Optimized :6 sec

Optimized plus Zorrder one :2 sec

OPTIMIZE salesdata ZORDER by (order_date,order_id)

Vacuum:

Clean file that not needed to save storage space to save some cost

Use Case: Vacuum Command

The `VACUUM` command in Delta Lake is used to delete files that are no longer needed. By default, Delta Lake enforces a 7-day retention period to ensure data isn't accidentally deleted too soon.

Example of Vacuum Command Without Retention Check:

With Retention Check (Default Behavior):

```
sql
Copy code
VACUUM my_table;
```

This will remove files older than 7 days, respecting the default retention period.

Disabling Retention Check:

20. **Set Configuration:**

```
python
Copy code
```

```
spark.conf.set('spark.databricks.delta.retentionDurationCheck.enabled'
, 'false')
```

21. **Run Vacuum Command:**

```
sql
Copy code
VACUUM my_table RETAIN 0 HOURS;
```

Dry run

In Databricks, the `VACUUM` command is used to clean up old data files from Delta Lake tables. When you specify zero hours with the `VACUUM` command, it removes all data files that are no longer needed for the current state of the table immediately. This operation can have significant implications for the Time Travel feature.

SESSION 9:

Broadcast Join :

Why we need join :

Within one data set we wont get all value then we will use joins

A broadcast join is a technique used in distributed data processing to optimize joins between a large dataset and a significantly smaller dataset.

For instance, imagine you have a 2GB dataset and a 30MB dataset. Normally, during a join operation, data shuffling occurs, which can cause network congestion and slow down query processing due to the need to transfer data between nodes.

A broadcast join addresses this by replicating the smaller dataset (30MB) to each node in the cluster. This way, each node can perform the join locally without needing to transfer the smaller dataset over the network during computation.

In Spark, if the smaller dataset is less than 10MB, it automatically performs a broadcast join, taking care of partitioning and replication to enhance query performance.

Limitations: Not suitable for large tables because broadcasting large tables can lead to excessive memory usage and potential OOM (Out Of Memory) errors.

spark.conf.set("spark.sql.autoBroadcastJoinThreshold", 10 * 1024 * 1024) # 10 MB

We can check the type of join in the UI

Cluster ->spark ui->sql dataframe->check plan

ByDefault join which is used by spark is ->SortMergJoin

How to apply broadcast join is :

```
Spark.Conf.set(Spark.sql.autoBroadcastjointhresold,10*1024*1024)

Larg_df.join(broadcast(small_df),’product_id’)
```

Partioning and bucketing :

Partitioning and Bucketing

Why We Need It:

- **Partitioning** and **Bucketing** are used to optimize the performance of Spark queries by organizing data in specific ways.

Partitioning:

- **Definition**: Divides the data into smaller, manageable parts based on the values of one or more columns (e.g., date).
- **Use Case**: Useful for large datasets where queries frequently filter on partition columns.

- **Example**:
  ```
  df.write.partitionBy("date").format("parquet").save("/path/to/save")
  ```

Bucketing:

- **Definition**: Organizes data into a fixed number of buckets, based on the hash of a specified column.
- **Use Case**: Useful when the dataset is large and partitioning alone is not sufficient. Bucketing improves join performance by reducing the amount of data shuffled.
- **Example**:

```
df.write.bucketBy(10,
"product_id").format("parquet").saveAsTable("bucketed_table")
```

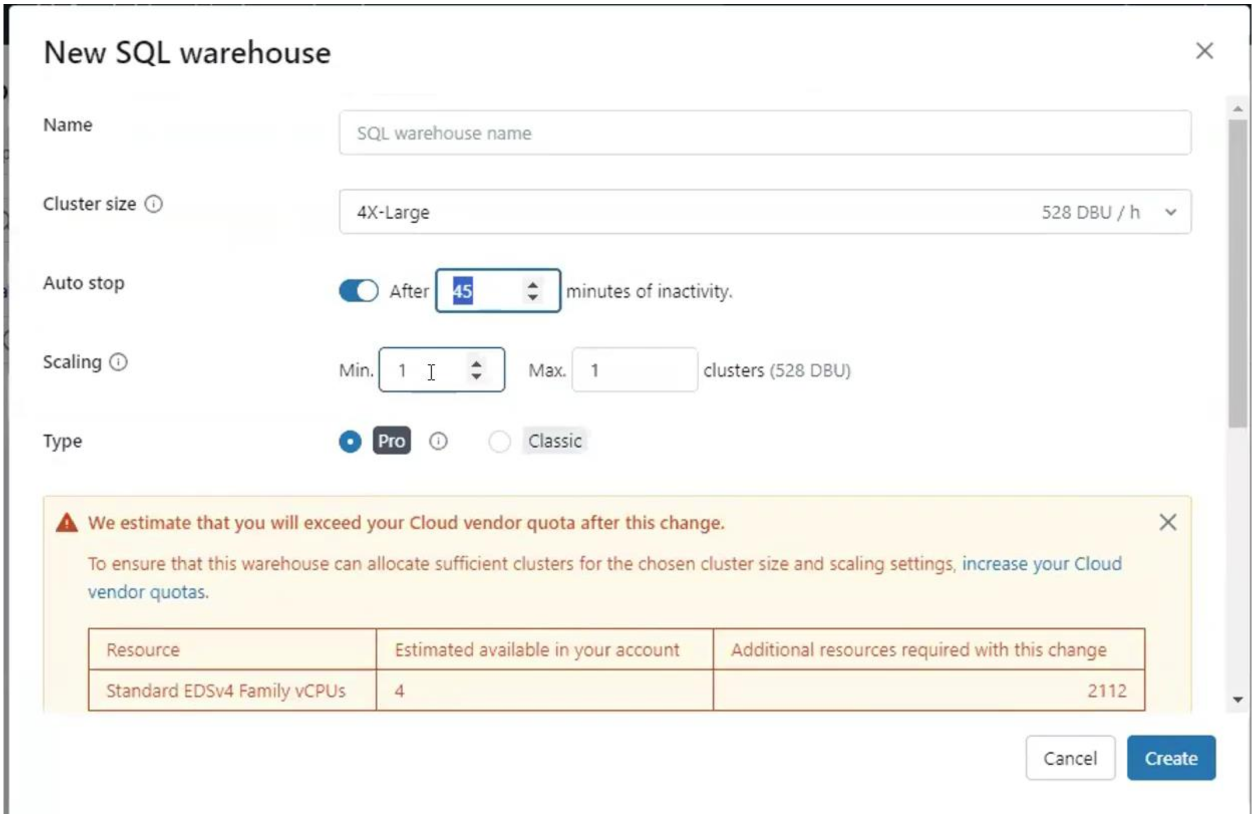
Cluster – Group of VM in backend / All the VM have spark configuration into them

Interactive / All purpose cluster

- Manually created using cluster UI
- Genrally used for testing purpose
- Manually start and stop

Policy -

- Unrestircited
- Personal compute – single VM in Back end which will act as driver and worker node (Single node are generally used for small workload (testing))
- Power User Compute – No option of Multinode / Access modes can be changed
- In realtime , access mode is shared as it runs high paraller jobs
- If you want unity catelog, access mode should be share
- Min worker node shoauld be 1 and max 450
- Photo accelaration – use to reduce your cost on spark workload
- Disabling the autoscaling will fix the amount of worker nodes (No min and max)
- Job cluster is tied to job
- Whenever a pipeline is initiated, a job cluster get created in back end. When the execution of pipeline ends, that job cluster gets terminated. Next time when the same pipeline is triggered, a new job cluster gets created and terminated when the exectution ends
- Job compute is scalable i.e it scales automatically
- If you schedule your job using ADF, overall execution time s less but cost would be high but if you use job compute, then overall execution time of a pipeline is high but overall cost would be low
- You can schedule your jobs using ADF and databricks notebooks as well
- Databricks use delta live table and it use job compute only if you schedule your job
- SQL warehouse is a type of compute which we will use to query your data



New SQL warehouse

Name: SQL warehouse name

Cluster size: 4X-Large 528 DBU / h

Auto stop: After 45 minutes of inactivity.

Scaling: Min. 1 Max. 1 clusters (528 DBU)

Type: ● Pro ○ Classic

We estimate that you will exceed your Cloud vendor quota after this change.

To ensure that this warehouse can allocate sufficient clusters for the chosen cluster size and scaling settings, increase your Cloud vendor quotas.

Resource	Estimated available in your account	Additional resources required with this change
Standard EDSv4 Family vCPUs	4	2112

- Pro has new syntax functionalities
- Limited to databricks only
- In real time, when you jobs are scheduled, your cluster would be running. Therefore if you want to query your data in hivemetastore, you don’t want to disturb the running cluster. Therfore just to query the data, use can use SQL warehouse
- Instance pool is used to reduce starting time of cluster
- Minimum ideal – minimum VM in this pool
- Pool is nothing but, it is pool of VM. At any given point of time, n VM shouls be ideal and ready to supply to our cluster
- This pool VM can be used in all purpose cluster as well. This will eliminate the start time of all purpose cluster every time you start the cluster to run the pipeline.
- To reduce the overall execution time we have the option of using pool custer as well
- In non prod , you can use all purpose cluster but in prod all purpose cluster with pool is used. Job compute is recommended to be used in production env as well as it provide automatic scalling.

Example senarios

If you are processing 15TB of data , you can create three cluster. Create 3 linked service with ADF and give 5TB each to each cluster. You can have pool of 3 VM for each cluster. For each cluster you can 1 min and 3-4 max worker node.

We basically have to understand how much of data you want to process, after which you need to do load balancing among the cluster. It is not possible in one go. Optimizal setup is achieved overtime.

DLT is declarative ETL framework for the DB data Intelligence Platform

Declarative ETL :

You tell the system what to do and not how to do

Here you describe end results you want to achieve. DLT figures out best results to achieve those results in cost effective manner

Procedural ETL :

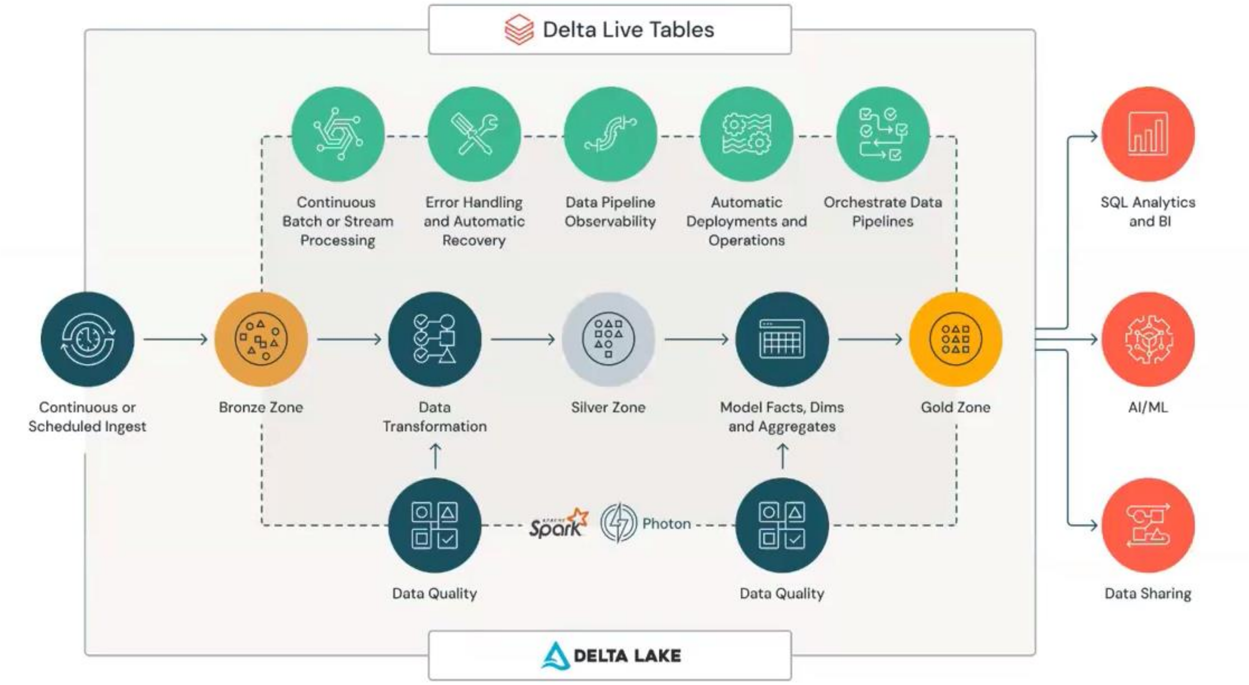
This is what we do in day to day word
We decide what to do and do things on our own

In real time, Implementation of SCD's is not that easy. There is so much of logic that is suppose to be performed. But if we use DLT, you just have to mention what type of SCD's you want and declarative ETL will do it for you

Streaming table is a concept in delta live table
DLT can be used in orchestartion

In case of DLT, you can view linage using hive metasore but in regular case you can only see linage using Unity catelog

To run the DLT, we are suppose to use Job Compute. We cannot use All purpose cluster


Delta Live Tables
Continuous Batch or Stream Processing
Error Handling and Automatic Recovery
Data Pipeline Observability
Automatic Deployments and Operations
Orchestrate Data Pipelines
SQL Analytics and BI
Continuous or Scheduled Ingest
Bronze Zone
Data Transformation
Silver Zone
Model Facts, Dims and Aggregates
Gold Zone
AI/ML
Data Quality
Spark
Photon
Data Quality
Data Sharing
DELTA LAKE