SPARK ARCHITECTURE AND INTERNAL WORKING MECHANISM:

Spark is an open-source distributed framework. This is mainly used for bigdata processing . **If we want to process millions of records with lightning speed , then we can go for spark. Spark works similar to Hadoop in terms of distributed nature, but it differs in in-memory processing .**

Hadoop adopts in in disc processing model. Spark is atleast 100 times faster in memory processing and atleast 10 times faster in disc processing when compared to Hadoop/traditional systems.

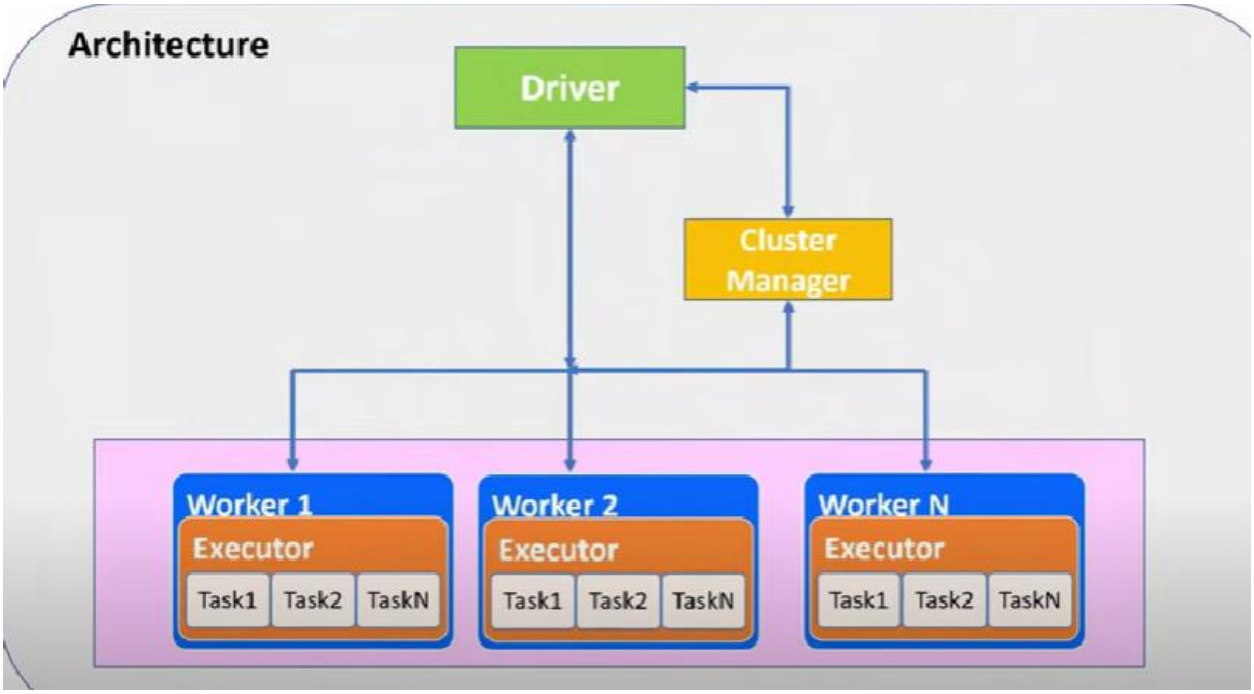
Reason for speed in spark: 2 reasons are there to process millions of data at lightning speed.

1) **Spark adopts in-memory computational model .** in-memory means For Example if someone ask to write our mobile number in piece of paper, it might take couple of seconds because we don't want to refer anything any directory or our mobile phone to retrieve the mobile number. Just we will remember our mobile number.so basically we are retrieving value from our memory, so it won't take much time. If someone asked to write down phone number of distant relative ,then the process we do is first we need to unlock the mobile ,open contacts application in our phone and type keyword to find the person and once we find the person ,we have to retrieve the mobile number and then we have to write it into piece of paper.so this is difference between in memory and accessing from external systems. Spark handles in-memory.
2) **Spark handles parallel processing.** Parallel processing means distributed framework. In any traditional system, data would be stored in 1 disk i.e., only 1 worker node. So, the processor actually has to retrieve the data from the disk to the networks through input output operations and it will bring data to memory and then it will apply certain calculations and also store intermittent result back to disk through network. It will repeat this process to compute the whole operation. So basically it will talk to disk n number of times in order to complete certain task. But coming to distributed nature , there are multiple worker nodes and data is also distributed across multiple worker nodes and each and every worker node can work independently and within worker node ,we might have n number of executors .with in executors we might have multiple cores . So, we can initiate n number of parallel processing in spark.

E.g.: In order to complete construction of building ,if we deploy 1 employee , it can take 100 days .If we deploy 100's of employees ,work can be completed in 1 day**. In traditional system there will be 1 worker node it will complete whole process but coming to spark ,no of workers can be dynamically increased or decreased through this distributed framework.**

SPARK ARCHITECTURE: Spark has well designed layered architecture .It follows master/slave concept. Master is nothing but driver and slave is nothing but worker node . In this layered architecture we have 3 layers (driver ,cluster manager and worker). These 3 layers are very well designed within themselves, and they are integrated /loosely coupled to each other . When we start spark session , all these 3 layers would be coupled, and it will start executing the application.

SPARK ARCHITECTURE:



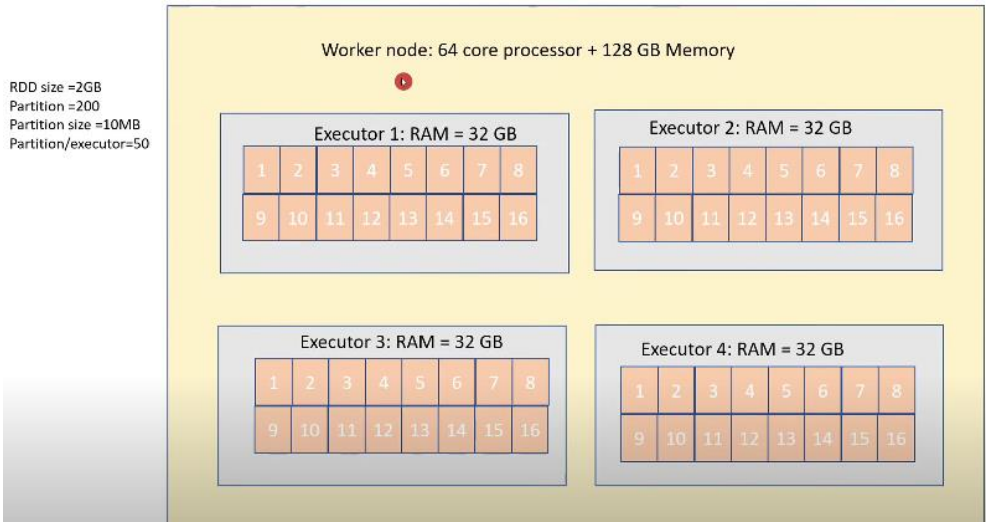
User will submit the code to driver and driver will do n number of processes internally .

When code has to be executed in worker node ,driver will talk to cluster manager like I need few resources to execute few tasks then cluster manager will check the resources and will launch the resources and it will reply back to driver (like created workers for you and you can go ahead with next steps). Once driver gets confirmation from cluster manager , it will start establishing direct connection between worker and driver. And **cluster manager would silently monitor the worker nodes**. **In case if there is any failure with worker nodes then it would be replaced by cluster manager. Cluster manager is responsible to maintain the health of the worker nodes in the cluster.** Driver node will assign the task directly to worker nodes and also worker nodes will maintain the tasks . worker nodes will execute the piece of logic given by driver in the form of jar file and it will execute with the help of executors and then it will return the result back to driver.

In spark architecture , there will be n number of worker nodes . while creating cluster in data bricks we can define minimum number of worker nodes and maximum no of worker nodes. When we start the application , it will start with minimum no of worker nodes . For Example, we defined minimum number of worker nodes as 2 and maximum number of worker nodes as 8 . When application starts it will launch only 2 worker nodes. So, depending on complexity of application submitted by user ,it will keep on increasing other worker nodes. If the program is very complex ,it will keep on increasing until worker node 8 because max worker node we choosed is 8. Beyond 8^{th} worker node it wont increase because we already fixed maximum no of worker node as 8.Similarly when worker load goes down, cluster manager will shut down the worker nodes . it will remove worker node from the execution so that we can save money. Instead of running worker nodes idle , we can remove it(worker nodes). It would be done by spark automatically .

Within 1 worker node, we have n number of executors .

The below screenshot shows how worker node will be internally.



Worker node is nothing but server i.e., 1 computer that is having 64 core processor + 128GB Memory. There are 4 executors created here in this worker node as shown in above screenshot.

So, among(64 core processor + 128GB memory) , if we divide core processor and memory among all executors , then for each executor(in this example 4 executors are there) then we will get 16 core .

These **16 cores can work independently 16 parallel processing can be initiated within this executor**

Within this worker node itself there could be 64 processes can be initiated at a time. 64 tasks can be initiated at a time . we have divided core among executors. Similarly, we have to divide memory among executors. Overall 128GB if we have to divide among each executor , we will get 32 GB memory i.e.: RAM Size.

We will divide 64 core processor and 128GB memory among executors. 4 executors means 64/4 core processor and 128/4 memory among each executor. 8 executors means 64/8 core processor and 128/8 memory among each executor. And these are configurable . By default, data bricks cluster will decide and create. These are configured by developer via spark settings .

For example, there is csv file that is being read by data bricks . size of csv file is 2 GB.

2 GB file we are reading in data bricks , what happens is first step is databricks will create rdd for this 2 GB data. By default, databricks will split file into 200 partitions. So, when it reads external file (csv file) of 2GB when it comes inside databricks it will divide into 200 pieces/partitions . each partition size would be approximately 10MB. And it will create 50 partitions or 50 executors.

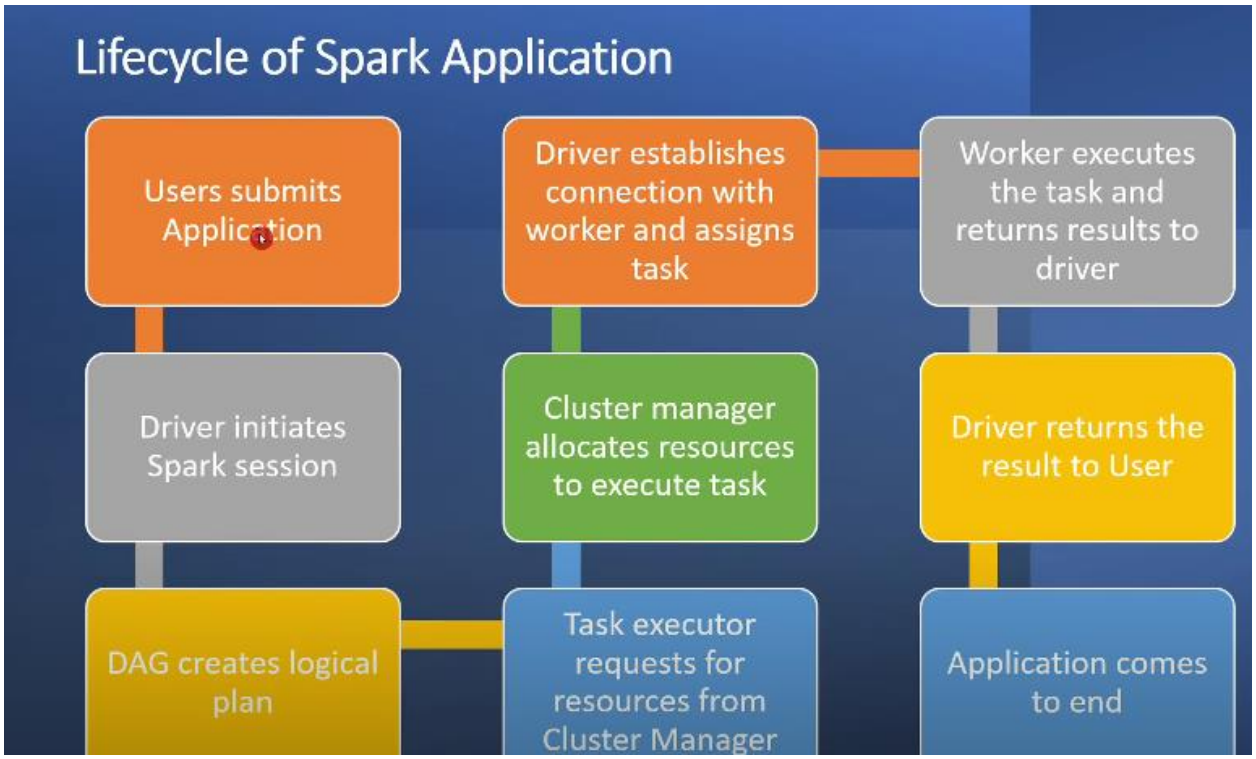
Partition size=file size/no of partitions.

In above screenshot example , there are 200 total partitions and 4 executors. So, when we distribute across executors it would be 50 per executor(200/4) . When there should be some transformation or action to be executed to this executor , what happens is this 16 cores are responsible to handle 50 partitions .what happens is initially this 16 core will pick 1 partition each. So, there should be 16 tasks would be created immediately. Once 16 tasks got completed then again 16 core will pick another 16

partitions 1 partition for each core .then in the 3rd round again it will pick 16 more partitions. 48 partitions got processed. There are 2 more partitions still not processed. What happens is out of these 16 cores , 2 cores will pick up 2 partitions and remaining 14 partitions will become idle . once 2 cores processed 2 partitions, whole process got completed for the executor 1. And 14 partitions for 14 cores in executor 1 is idle for some time. So, this is not good for performance. We can't keep any employee idle . For that reason, we have to configure the parameters , but we have to improve performance by configuring lot of spark parameters .

Number of partitions should be multiple of number of cores in this executor.

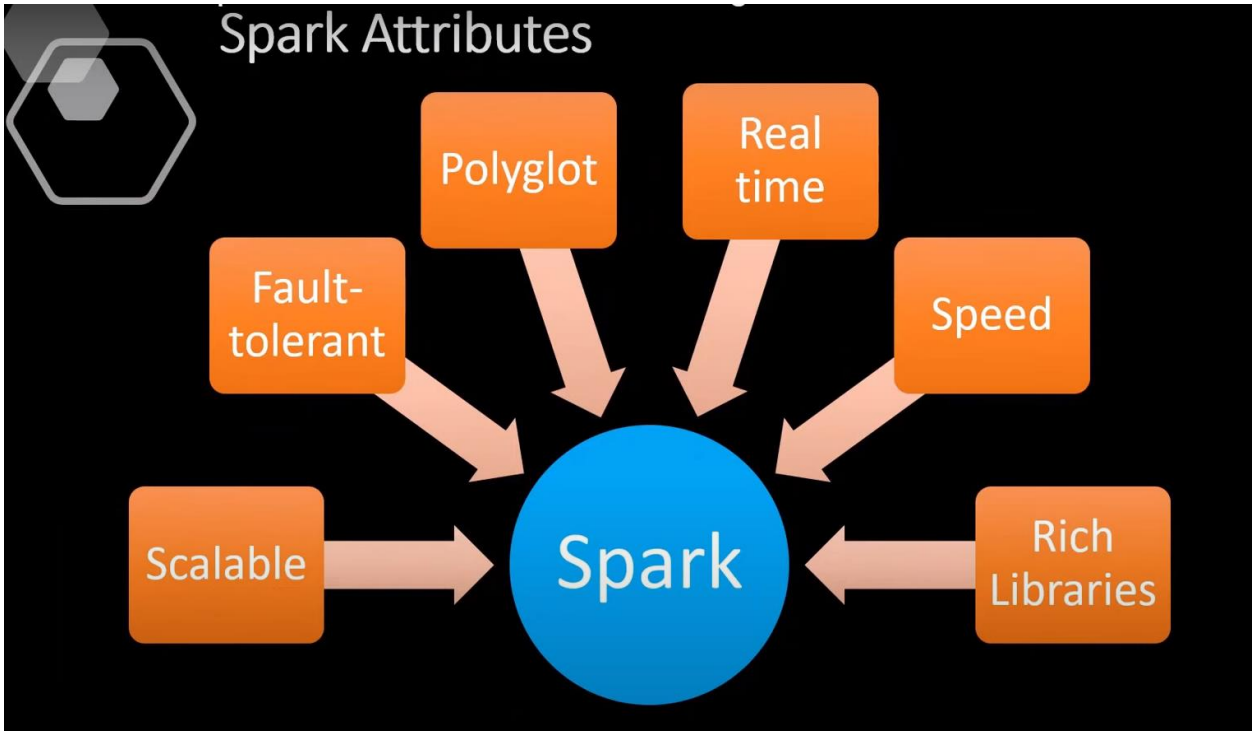
LIFE CYCLE OF SPARK APPLICATION:



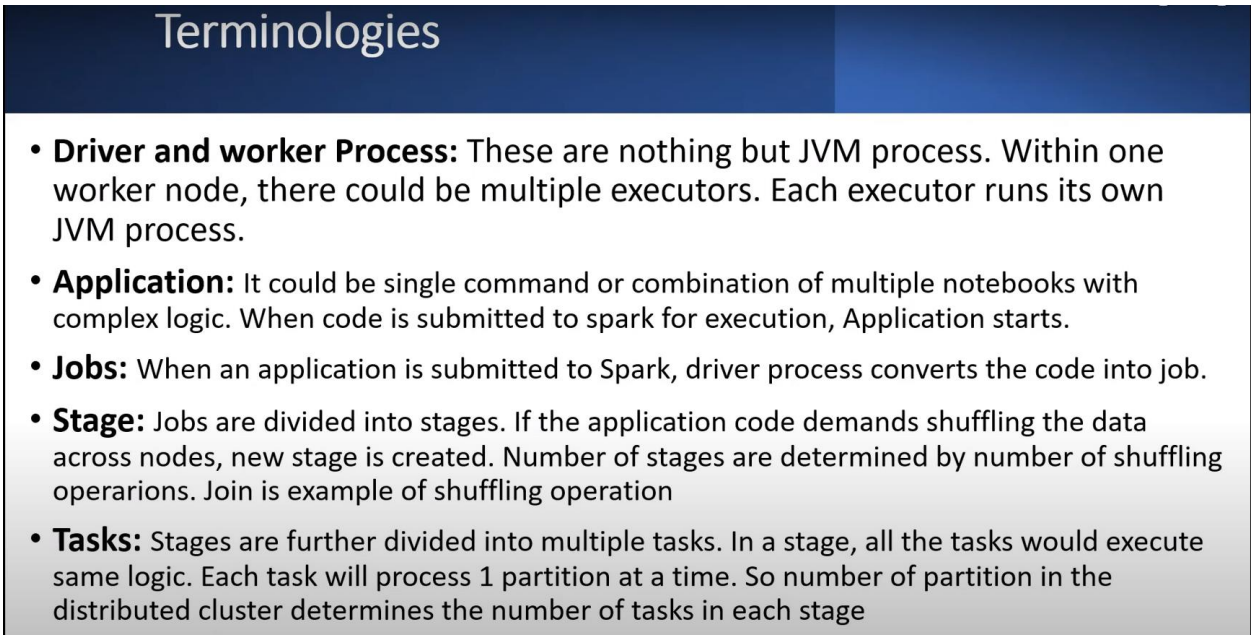
When user submits application(could be pyspark/spark command or multiple notebooks or could be something), When user submit something for execution then driver initiates spark session . code submitted by user that is called application . spark comes to live when user submits application then driver would initiate spark session and internally there is something called DAG (Directed Acyclic Graph). DAG will create logical plans for all transformations that are part of user applications and that will keep the lineage graph . DAG will assign piece of code it will break code into multiple smaller tasks and that would be given to task executor and task executor would request resources from cluster manager . and then Cluster Manager will allocates resources to execute this task and driver establishes connection between worker and driver and driver can start assigning the work to worker nodes in form of JAR files and then worker executes task given by driver and would return the result back to

driver and driver returns the result to user or could store the data to some storage location and application comes to an end.

SPARK ATTRIBUTES:



- **Scalable:** Here the worker nodes can be increased depending on the complexity of workload ,that is called scalable. Initially application should start with minimum number of worker nodes and depending on complexity of work , it will keep on increasing worker nodes until the maximum limit and when the workload goes down , it will automatically remove the worker nodes for execution. That is why it is scalable.
- **Fault-tolerant:** Spark follows lazy evolution model which means when we submit some transformation it will not process data immediately instead of that it will create logical plans to execute the transformation and will keep on building lineage graph in DAG**. If 1 worker node goes down what happens is, we are not losing any data, but we can recover the data using lineage graph in DAG.Thats why it is fault tolerant.**
- **Polyglot:** Spark supports multiple languages for programming . If we have team of developers from different programming background, we can reply them to develop a code . Not only in spark, within notebook itself , we can mix languages . that's why it is called polyglot .
- **Real Time:** Traditional systems are good for batch processing .not only for batch processing ,it can handle streaming data also .let it can be event hubs in azure or from Kafka streaming service. We can receive streaming data in real time and spark can handle .
- **Speed:** Spark is known for its speed due to in memory computational model and distributed framework . so, it enables parallel processing to its higher degree .
- **Rich Libraries:** Spark contains rich libraries for variety of processes, could be SQL/Data science related projects in ML .



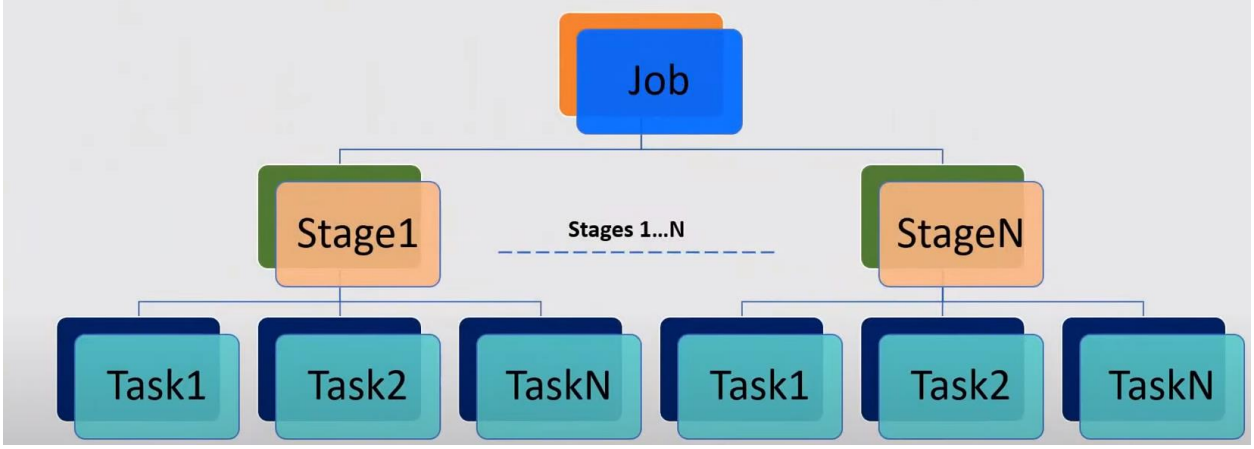
Terminologies:

- **Driver and Worker Process:** When driver and worker got launched , there is JVM process running within that . Driver node has its own JVM process and in worker node each executor will have its own JVM process and within 1 worker node , we will have multiple executors . each executor has its own JVM process .
- **Application:** Application is nothing but piece of code for execution . It could be single command or combination of multiple notebooks with complex logic . When code is submitted to spark for execution, Application starts.
- **Jobs:** When piece of code is submitted by developer to spark , driver splits the logic into multiple stages , first of all it will create one job for application and job would contain multiple stages .
- **Stages:** Incase we have to shuffle data across the nodes in order to perform certain transformation , then new stage would be created . Number of stages would be depending on number of times we need to shuffle the data to complete the transformation . If there is no need to shuffle the data for a job, then it would contain only 1 job . for E.g.: filter is an example for narrow transformation .narrow transformation would not require data shuffling across the nodes.

Summary of above steps:

When user submits the code to driver that is called application and application would contain job and each job is divided into multiple stages based on the need to shuffle the data across the cluster and each stage would consists of multiple tasks . even though there are multiple tasks , each task will do the same operation and 1 task would be created per partition .incase our rdd or data frame contains 200 partitions in the cluster ,then 200 tasks would be created .

Hierarchy:



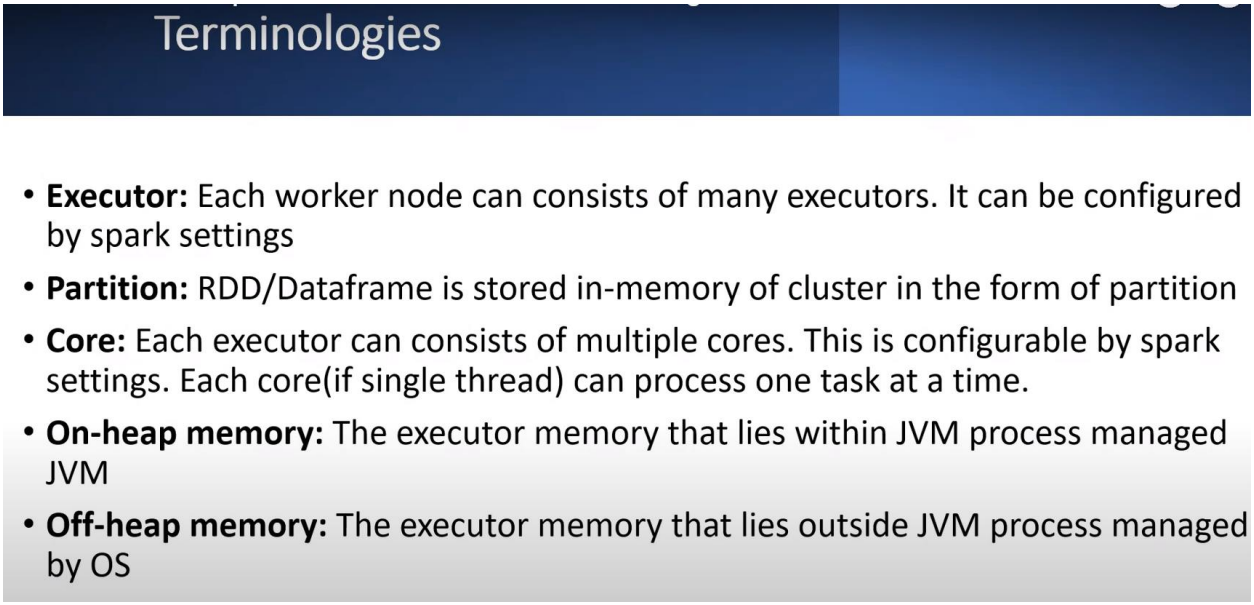
When user submits code , job would be created and job would consists of stages ,it could be stage 1 to stage N ,it should contain N number of stages . Based on the need we shuffle data across executors and each stage would contain multiple tasks tasks 1 till task N. This is purely based on number of partitions stored with in each executor .

Terminologies

- **Transformation:** transforms the input RDD and creates new RDD. Until action is called, transformations are evaluated lazily
- **DAG:** Directed Acyclic Graph keeps track of all transformation. For each transformation, logical plan is created and lineage graph is maintained by DAG
- **Action:** When data output is needed for developer or for storage purpose, action is called. Action would be executed based on DAG and processes the actual data
- **RDD:** Resilient Distributed Dataset is basic data structure of Spark. When spark reads or creates data, it creates RDD which is distributed across nodes in the form of partition.

- **Transformation:** Transforming 1 input RDD or data frame to another by applying certain functions. Transformations are following lazy evaluation model i.e; when we apply some transformations , data processing will not happen immediately but instead of that spark will create logical plans and that would be associated with DAG .

- **DAG:** DAG would keep on adding logical plans for transformation. Let's say there are 10 transformations one by one . for each transformation logical steps will be created and would be associated with DAG .
- **Action:** When we call an action,action would refer DAG and it will execute all the previous 10 transformations and then finally action would actually process the actual data based on DAG and it will return the result to user, or it will store the data to storage location.
- **RDD:** RDD is basic/native structure of spark. When spark reads/works with data , that handles data in the form of RDD.**RDD is distributed data across the cluster in the form of partition**. In later versions we got other API's such as data frame and data set, but RDD is native structure for spark. Even though we are working with data frame on dataset, when we submit the code spark handles that code internally as RDD .

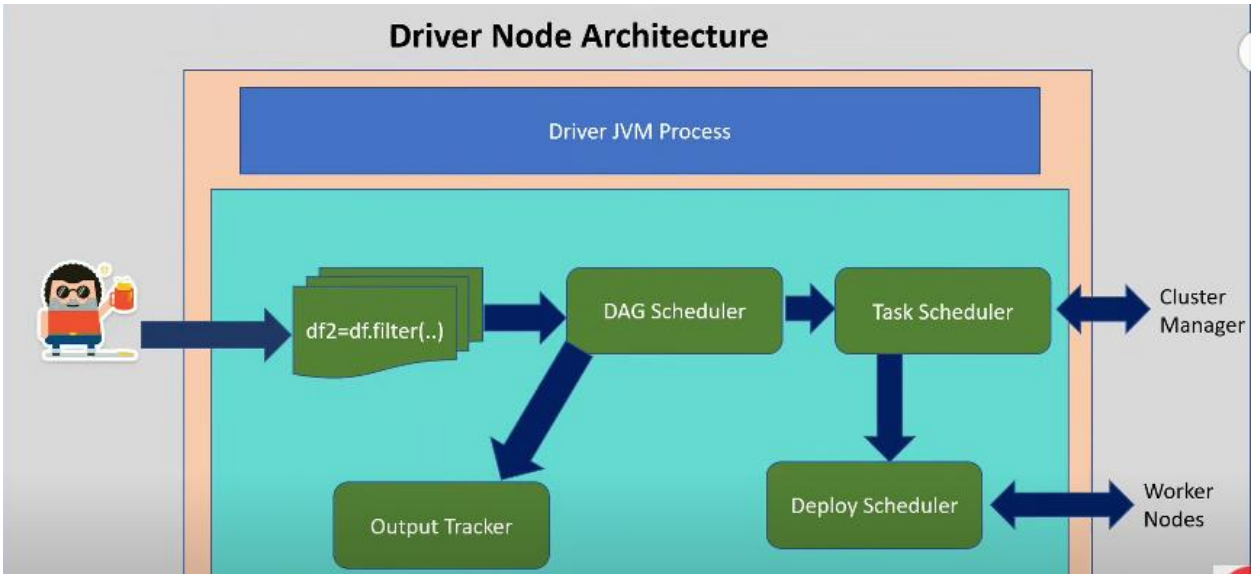


- **Executor:** Each worker node will contain multiple executors and within executor we can have 1 or more cores and each core can work independently and can execute one task at a time. Each task will pick 1 partition and it can work independently.
- **Partition:** RDD/Dataframe that is stored in-memory of cluster in the form of partition .
- **Core:** Core is processor that can work independently, and one core can pick 1 partition at a time, and it can complete 1 task at a time.
- **On-heap memory:** executor runs JVM process. **The memory within JVM Process in executor is called On-heap memory .** If we have to read/write some data within this memory on heap memory i.e; memory within executor , then we need to deserialize or serialize the data.
- **Off-heap memory:** Memory that resides outside JVM process and it is handled by OS directly.

SPARK LIBRARIES: Spark supports multiple rich libraries that are for SQL Process, Streaming process, Machine Learning as part of data science and GraphX (it is like graph like process).Examples of GraphX processes are social networking apps such as fb, linked in, google maps .

SPARK LANGUAGES: Spark is polyglot. It can support multiple languages. If we have team of developers from multiple programming languages, still we can mix them and we can achieve the result . Spark supports languages Scala, Java, Python, SQL, R. In any particular application , we can mix these languages . When I say application let's say we are creating a notebook .the first 40% of logic is written in Scala , 2nd 30% of logic is written in python, again 30% of logic is written in SQL is possible. In 1 notebook , we can mix the languages. In order to mix languages, we have to use magic command.

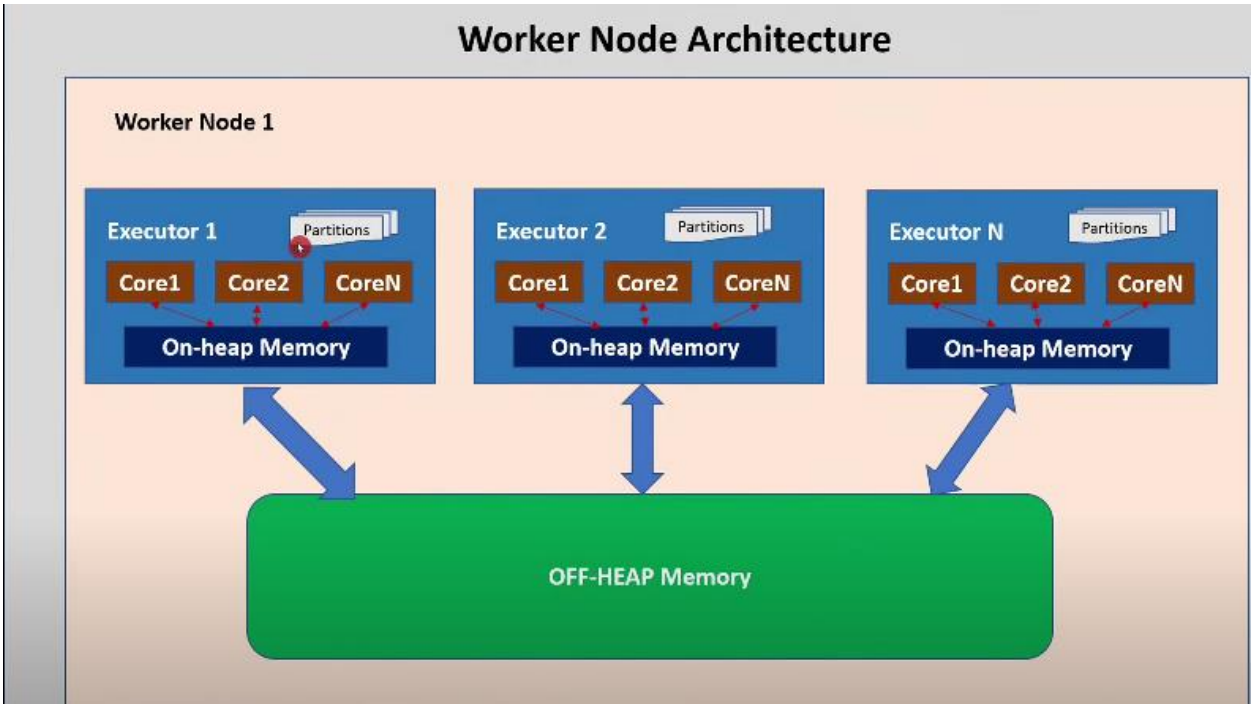
DRIVER NODE ARCHITECTURE:



Driver runs JVM process. When user submits piece of code to driver, then driver initiates spark session . Let's say for example piece of code is df2=df.filter() what happens is DAG Scheduler will create a logical plan for this command/this filter transformation and it will start building lineage graph and when there is an action called in any of the command then what happens is it has to process actual data ,then DAG scheduler will give task to Task Scheduler and Task Scheduler would negotiate the resources with cluster manager in other layer and once resources got allocated by the cluster manager then task scheduler deploying the task to back end i.e; worker nodes . Driver would establish connection to worker nodes through this back end deploy scheduler and worker node will keep on executing and it would return the result back to driver and that output would be tracked by some process called Output Tracker .

These are the major process within driver node architecture.

WORKER NODE ARCHITECTURE:



Worker nodes also runs JVM Process. Within 1 worker node, we will have multiple executors and each executor will consists of 1 or more cores and partitions would be stored with in on-heap memory and each executor would contain data as well in the form of partition. Outside JVM Process ,OFF-HEAP memory would be handled by OS directly . we can store the data within On-heap memory in the form of partition or we can keep data outside on-heap memory i.e; Off-heap memory .

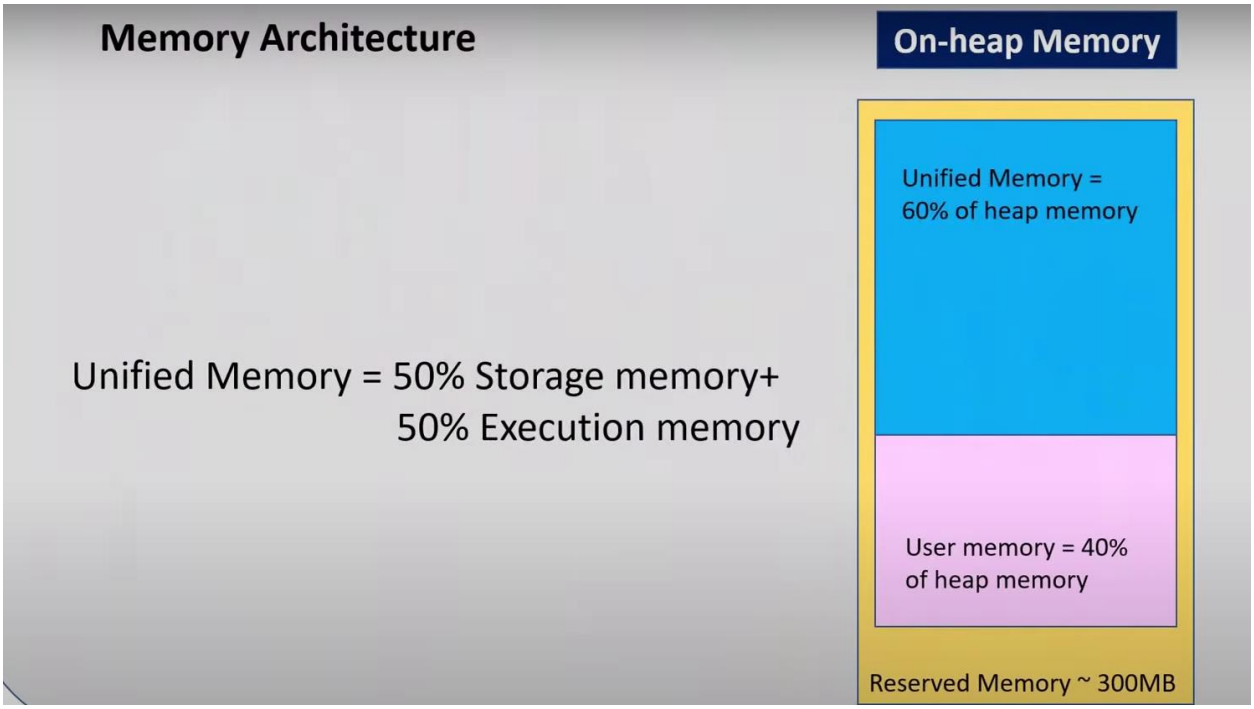
The Advantage of OFF-HEAP memory is 1) we don't need to serialize or deserialize while reading or writing the data to OFF-HEAP memory 2) As part of JVM Process ,there is a process called GC -Garbage Collector . When the memory is full, garbage collector will starts scanning entire memory and it will start removing absolute or very old objects .so that is a very unnecessary process. Garbage collection process is not applicable for OFF-HEAP memory. OFF-HEAP memory is not available in RDD API but in later versions in data set and data frames are available . All the executors they can work with its own on heap memory and also the common OFF-HEAP memory .

STRUCTURE OF ON-HEAP MEMORY:

Structure of on-heap memory is shown as below.

Let's say in worker node, 1 executor is containing 32GB memory(in previous example), Out of 32 GB memory ,300MB is reserved for spark memory i.e; we cannot do anything with reserved memory .**Reserved memory mainly used for some failure recovery incase if the system goes down or crashes then system memory would be used for recovery** . So, we cannot use this reserved memory that would be approximately 300 MB and rest of the memory out of 32GB is divided in to 2 parts → Unified Memory

and User Memory . User Memory will occupy 40% of heap memory which means 40/100 * 32GB = approximately ~13GB . 13GB would be given to user memory and rest of the memory 19GB would be given to unified memory manager and this **user memory is mainly used for storing user objects, metadata.**



Coming to unified memory manager, unified memory contains 60% of heap memory and again unified memory split in to 2 parts Storage Memory(50%) and Execution Memory(50%). So unified memory would be equally divided among storage memory and execution memory. These values are highly configurable in spark setting. Reserved memory we can't do anything but user memory, unified memory(Storage memory+ Execution memory) these are configurable, depending on our need we can configure.

For E.g.: among 19 GB of unified memory , 9.5GB of memory will be given to Storage purpose and another 9.5GB of memory will be given for execution purpose .

For E.g.: in our case Storage is not needed, storage in our use case we won't occupy more storage, but execution would occupy more memory . Let's say we are executing ML algorithm, so it basically executes some complex logic, we need more execution memory , maybe we have given some sample data , so we don't need 50% of storage memory . we can give 20% of unified memory for storage and remaining 80% of unified memory to execution .

In some other cases, if we are going to execute ETL Pipeline , only for staging purposes simply we are collecting our raw data and moving to DWH, in that case what we will do is we will handle millions of records, but we are not executing any complex operation, in that process storage is playing pivotal role. What we can do is we can assign 80% of memory to storage and 20% of memory to execution.

We can configure in such a way that if storage needs more memory and its available in executor then it can use the freely available memory and its vice versa. These kinds of configurations we can do using spark settings

Summary: We have 3 layers in spark architecture driver node layer, cluster manager layer and worker node layer. When user submits application to driver then driver initiates spark session and Between driver node and cluster manager layer, we have DAG scheduler to create logical plan for the transformations/code and will be building lineage graph and DAG will assign piece of code which we received from user in to multiple tasks using task scheduler and task scheduler will negotiate resources from cluster manager in other layer and once resources got allocated by the cluster manager then task scheduler deploying the task to back end i.e.; worker nodes . Driver would establish connection to worker nodes through this back end deploy scheduler and worker node will keep on executing and cluster manager silently monitor worker nodes. If any worker nodes fails , then cluster manager will replace them. driver split the task into multiple worker nodes in the form of JAR file and it would return the result back to driver and that output would be tracked by some process called Output Tracker .

Driver node and each worker node has its own JVM Process. Memory within JVM process in executor is called ON-HEAP memory. Each worker node will have 64GB core processor and 128GB memory.

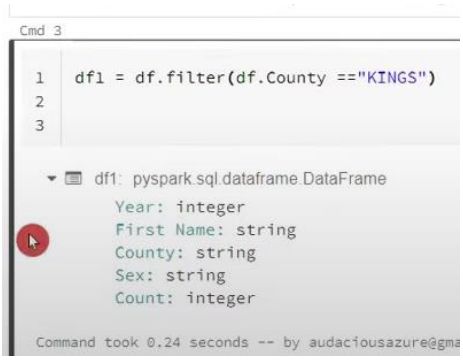
Within worker node we have multiple executors. Within each executor , we have multiple cores , partitions, and heap memory . each core will process independently and will have parallel processing. Each core will pick 1 partition at a time i.e., 1 partition will perform 1 task at a time independently .

ON-HEAP memory is split in to 2 parts . user memory (40% of on heap memory) used for storing user objects and metadata and unified memory(60%) . unified memory again split it into execution memory(50% of unified memory) and Storage Memory(50% of unified memory) . Execution memory and Storage memory can be configurable based up on our needs. There is another memory called Reserved Memory. Reserved Memory is used for failure recovery incase if system crashes or goes down . Reserved Memory will be used for system recovery purposes.

When we execute this piece of code, piece of code is given to driver , driver would start the spark session and also it is read operation, it is not transformation/action so it would read the external data and it would split in the form of partitions and that would be stored within the cluster .

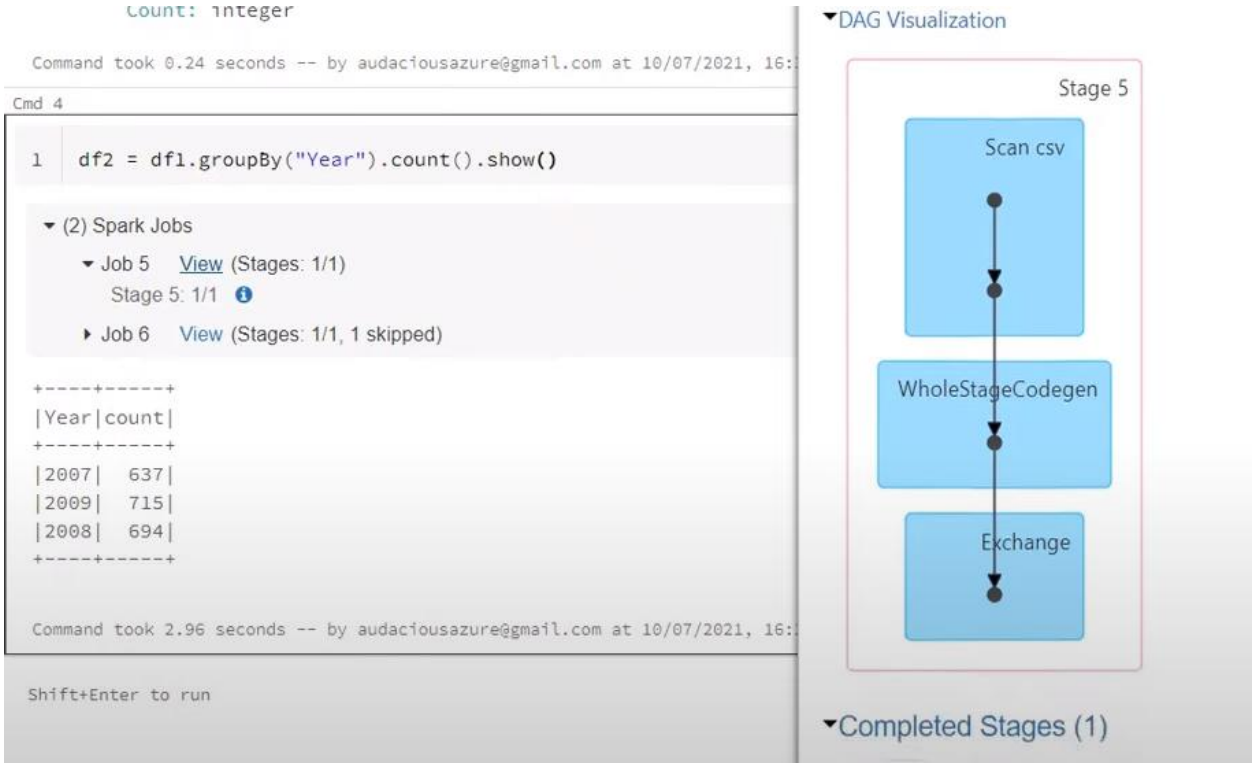
Once code is executed, what happens is data is read by spark from the file system and it has created multiple partitions and data got distributed among worker nodes.

In this below example using filter operation, filter is nothing but transformation which means it will not be executed immediately instead of that it will create logical plan and start constructing lineage graph in DAG



Spark has validated the syntax and it has created logical plan in order to execute this transformation.

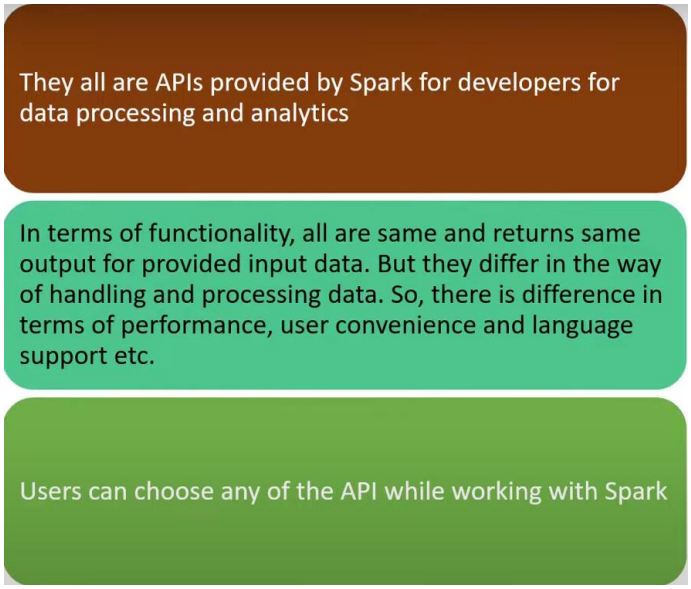
Creating another data frame here by applying groupBy and also showing the output by user. What happens is this is an action and also, we are applying groupBy. groupBy is nothing but wide transformation . it would be shuffling data across the nodes . so basically, here we are calling wide transformation and then it will create one more stage . filter would not create any stage but groupBy will initiate another stage. So totally we will have 2 different stages as shown below.



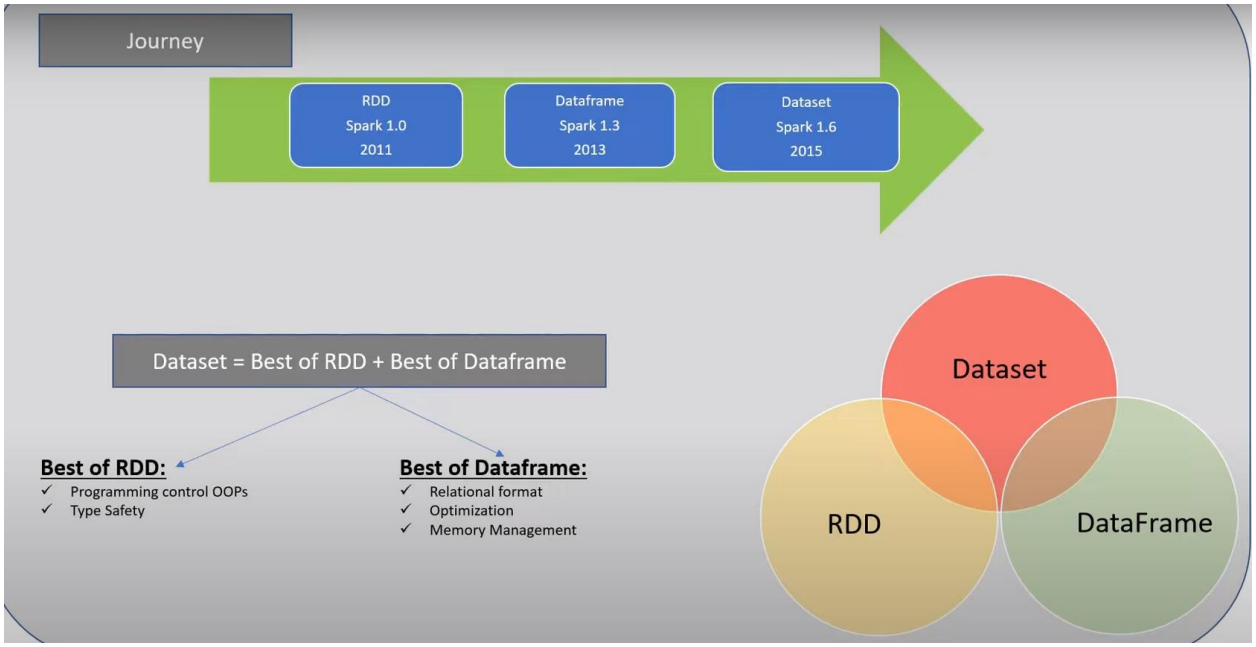
Here if we expand spark jobs → Job View → DAG Visualization got created .

In DAG Visualization ,first it is scanning CSV file , then top of that it is applying the filter then here exchange , we are doing groupBy that is wide transformation then data would be shuffled across executors so that is exchange. So that's how we can verify DAG. Also from DAG execution ,we can get more information about our spark execution how much data operation read or what is shuffled read write .

RDD, DATAFRAME AND DATASET



HISTORY OF API's



Initially spark provided only 1 API i.e., RDD . so users don't have any other choice other than RDD. While working with RDD, it has certain limitations or shortcomings. In order to overcome that shortcomings , DataFrame got introduced. DataFrame was able to successfully overcome those shortcomings that got introduced in RDD. In order to overcome issues in DataFrame, Dataset got introduced. Dataset has got all the best features from RDD and DataFrame.

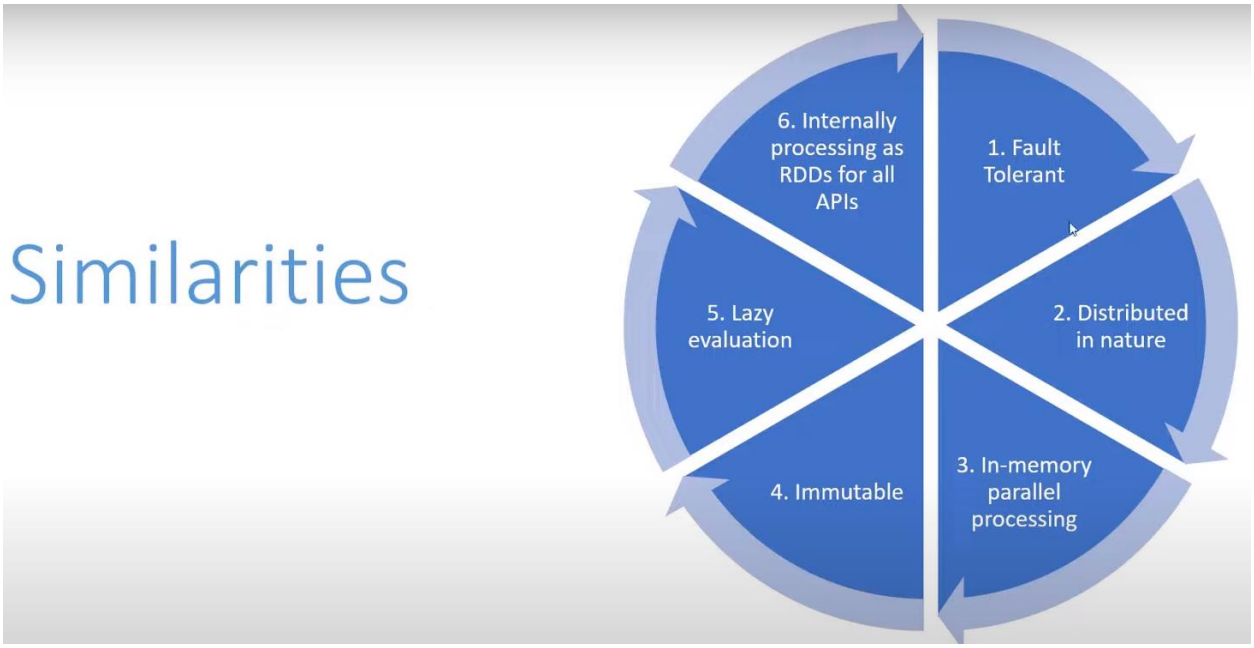
Best of RDD:

RDD provides programming control its OOP style programming which means we have lot of control in programming side, and it has strongly Type Safety which means all the columns would have certain data types. In case if there is any violation then it will throw error at compile time . so, it will save lot of effort to avoid any run time error .

Best of Dataframe:

It is similar to table in relational database. So, it is structured data , that is one of the advantage in data frame and also it has optimization feature . Data frame has certain schema. **Based on the schema ,optimizer can collect the statistics in dataframe. Based on the statistics , it can create n number of logical plans and it can choose best physical plan out of logical plan , that is why data frame can be optimized which is missing in RDD. In Data frame we can do better management because in RDD all the objects would be stored in on-heap memory in the form of JVM objects but in data frame we can save the objects outside on-heap memory i.e., off-heap memory.**

SIMILARITIES BETWEEN THESE API's



Fault-Tolerant: When we are working with spark i.e., if we are executing certain transformation i.e., it will not execute data processing during that time. What happens is it will create a logical plan and based on that it will construct lineage graph that is called DAG. Basically, it will keep all the flow in DAG. If there is some failure in middle of the process, then these API's can construct the data once again using the graph. That is why fault tolerant.

Distributed in nature: Data is distributed in nature. Among these API's , incoming data is distributed across multiple executors in the form of partitions and that is the reason spark can initiate parallel processing and data is distributed across multiple nodes .

In-memory parallel processing: These APIs are working in the concept of In-memory parallel processing which means in traditional system for each, and every time memory will read the data from hard disk, and it will perform certain operations and spark introduces in-memory computation which means all the data will come from in-memory -that is common across all API's .

Immutable: In all this API's, source data could not be modified . For example, if we apply certain transformation on RDD, then it will create another RDD based on the transformation . Basically, it will not change source RDD, it is same for data frame also. That is the reason it is called immutable.

Lazy evaluation: Transformation always work in lazy evaluation model. When we execute certain transformation, it won't process the data immediately. Instead of that it will just create logical plan and it will construct the DAG.

Internally processing as RDDs for all API's: Even though we are working with data frame or datasets, still when we submit the code what happens is internally spark engine will consider it as RDD. It will convert data frame/dataset to RDD, and it will work. RDD is basic structure for spark programming . For user convenience dataframe and dataset got introduced but still internally spark will work as RDD.

DIFFERENCE BETWEEN RDD, DATAFRAME AND DATASET:

RDD	Dataframe	Dataset
Program how to do	Program what to do	Program what to do
OOPs Style API	SQL Style API	OOPs Style API
On-heap JVM objects	Off-heap also used	Off heap also used
Serialization unavoidable	Serialization can be avoided(off heap)	Serialization can be avoided(encoder)
GC impacts performance	GC impact mitigated	GC impact mitigated
Strong Type Safety	Less Type Safety	Strong Type Safety
No optimation	Catalyst Optimizer	Optimization
Compile time error	Run time error	Compile time error
Java, Scala, Python, and R	Java, Scala, Python, and R	Scala and Java
No Schema	Schema Structured	Schema Structured

1) Programmatically we have to tell what to do and how to do .For example we have to perform certain operations then we have to tell what to do and how to do through programmatically . But coming to dataframe and dataset it has optimizer , we can just tell what to do .Based on that it will create multiple logical plans and from that it will choose the best plan to construct the physical plan. So, we don't need to worry about how to do. So, in RDD we might do some mistake , we have to manually tell what to do and how to do .maybe we might not choose best approach but coming to data frame and dataset it will create multiple plans and it will choose the best plan. We don't need to worry about performance.
2) Coming to programming style , RDD and dataset both are providing object-oriented programming style .that is good because we can control lot of things programmatically . Coming to dataframe it is providing traditional SQL style API .
3) Coming to memory management, in RDD All the objects are created in On-heap memory as JVM objects . coming to data frame ,objects can be stored outside on-heap memory i.e., off heap memory. When we store data outside on-heap memory , we are achieving better performance because in on-heap memory , garbage collector will play certain role because when memory is full in on-heap memory then garbage collector will be scanning entire memory and it will remove unwanted very old or absolute objects out of the memory . so, garbage collector unnecessarily will hit the performance.
4) When we store objects outside heap, we don't need serialization because when we are storing as JVM objects then always we have to deserialize and serialize while reading or writing the data out of on heap but that can be avoided in dataframe and dataset

5) GC is nothing but garbage collector, when memory is full in RDD then what happens is garbage collector will start scanning the entire memory and it will start removing very old or absolute objects.so it will take some time, it will hit the performance, but it can be mitigated in data frame and dataset.
6) **RDD and dataset both are Strongly Type Safety which means column type would be defined at compile time .incase if there is any mismatch in the data then it will throw error during compile time but in the data frame we don't have that option incase if there is string value in the integer column or vice versa it will not throw any error during compile time but during run time it will throw error so unnecessarily it will create lot of rework . This is not good feature in data frame but RDD and Dataset both are Strong Type Safety.**
7) In RDD there is no schema enforcement that is the reason we cannot do better optimization but in data frame and dataset there is optimizer .coming to data frame optimizer is catalyst optimizer so it can create multiple logical plans while working with data frame so it can do optimization automatically , that is good feature in data frame which is missing in RDD.
8) RDD and Dataset will give compile time error if there is any mismatch because both are having data type for each column but dataframe will throw error during runtime so in case there is some datatypes mismatch it won't throw any error during compile time, but it will throw error during runtime.
9) Coming to programing languages ,RDD can support JAVA, Scala, python, R languages , same with dataframe but coming to data set it can support only scala and java.
10) In RDD there is no schema enforcement and schema can be enforced in data frame and dataset.

TRANSFORMATION AND ACTION:

These are operations used in spark programming.

Transformation:

Transformation is kind of operation which will transform Dataframe from one form to another. We will get new Dataframe after execution of each transformation operation. Transformation is lazy evaluation based on DAG (Directed Acyclic Graph). Filter, Union etc.,

Until unless we call an action, this transformation will not get executed . so only Lazy evaluation will be performed to create DAG.

Action:

Action:

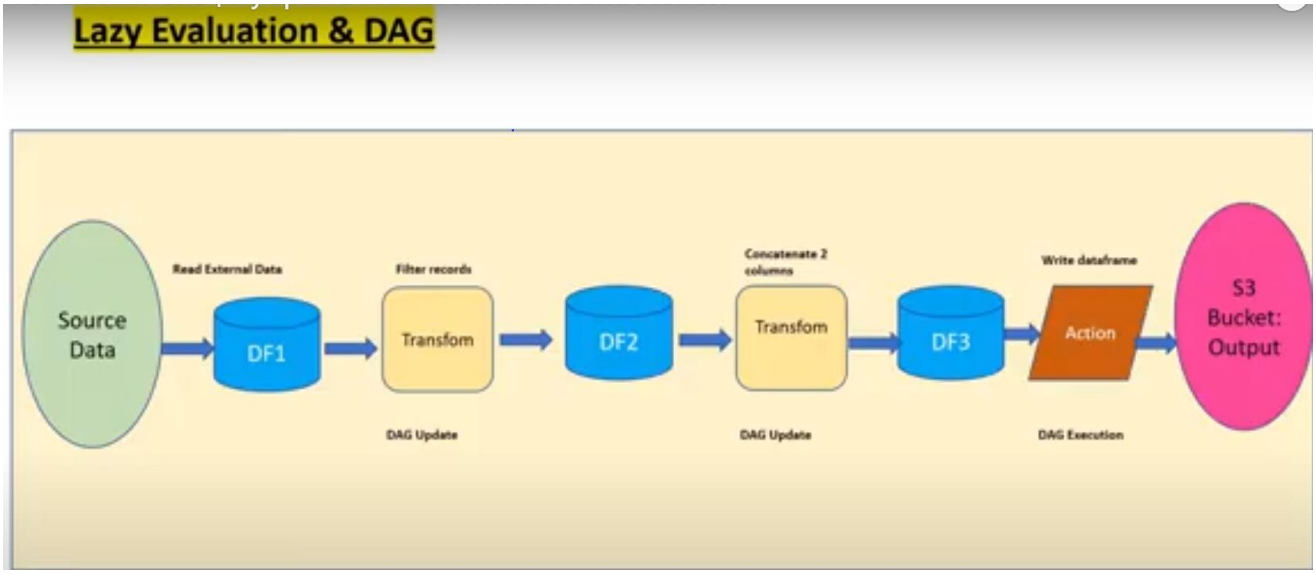
When we want to work with actual data, call the action. Action returns the data to driver for display or storage into storage layer. Count, collect, Save etc.,

When we want to work with actual data, we can call action . so, action would execute all the transformations that are associated with DAG and returns the data to driver for display or storage purpose.

Let's say in spark programming, there are 10 different commands ,let's say first 9 commands are transformation and 10th command is action. Now what happens is when we execute first 9 steps it is

lazy evaluation ,spark engine produces the logical plan and creating DAG and when we execute the 10th step that is action what happens is spark engine refers DAG and it traverses back to step1 and started executing all the steps starting from step1 till step9 , then it has created new dataframe at the end of 9th step that is at the end of the transformation then finally it performs action that is displaying the data by passing the data to driver or storing the data to storage layer .

LAZY EVALUATION AND DAG:

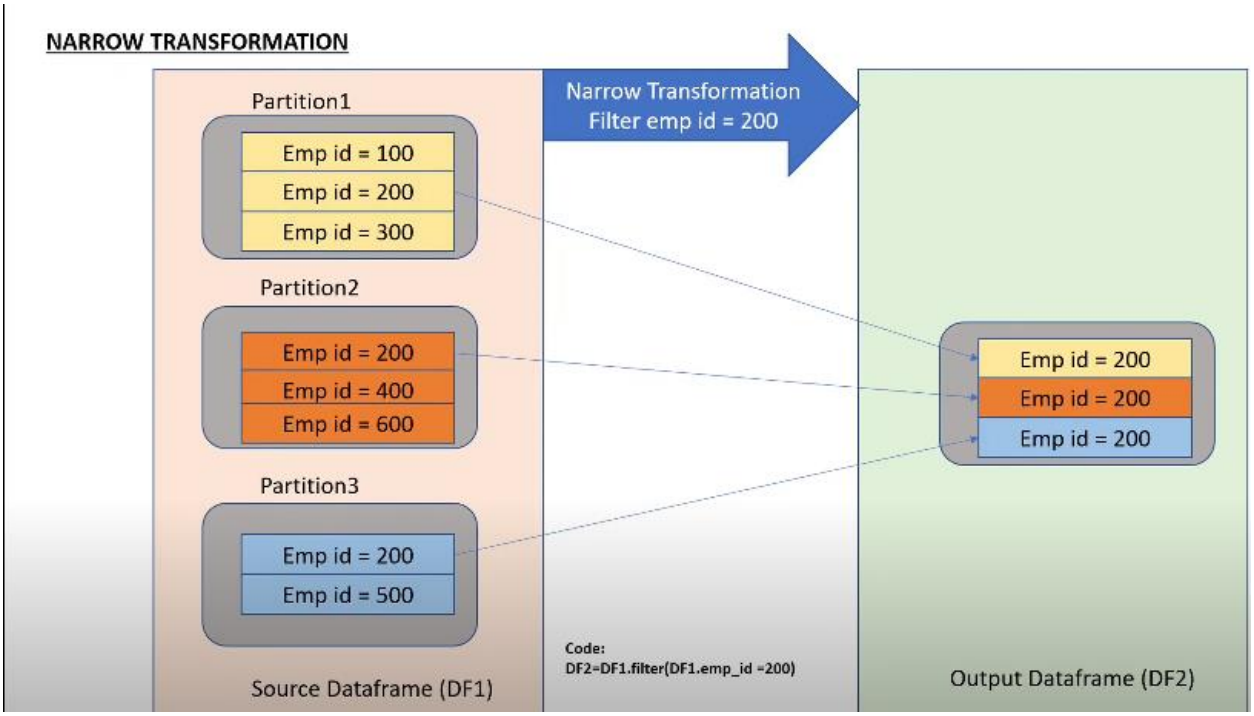


For example, we have an external source data that could be csv/Amazon S3 bucket /could be Azure Data Lake Storage. In the first step, spark engine reads data from external data, and it creates source data frame DF1. On top of source data frame, we are performing some transformation let's say we are filtering certain records based on some condition. Filter is a transformation . So, what happens is process will not immediately produce the data. Instead of that it will create the logical plan and it will update into DAG and at the end of this transformation, it will create another data frame DF2. DF1 is immutable .

Once the data is filtered, then it will not eliminate the data from DF1. But instead of that it will create new data frame DF2 that will contain only the filtered data . Again, we are performing certain transformation , we are concatenating 2 columns first name and last name ,what happens is concatenation is a transformation. It will not work with data, but it will actually create a logical plan and update DAG and at the end of the operation it will create another data frame DF3. Now let's say we are calling action. We want to write the data into S3 bucket or storage layer , what happens is action will refer DAG and it will traverse back to 1st step and it will start executing all the transformations based on logical plan provided by DAG and it will complete steps 1 by 1 and finally it will produce DF3, and the output will be returned into Storage layer.

TRANSFORMATIONS: There are 2 kinds of transformation . one is narrow transformation and second is wide transformation. In narrow transformations, we don't need to shuffle the data across nodes. It is a simple transformation and inexpensive .Wide transformation is very expensive; it will hit the performance because it has to shuffle the data across nodes .

NARROW TRANSFORMATION:



Let's say this is a cluster, within the cluster we have 3 executors' executor 1, executor 2 and executor 3. Within each executor we have 1 partition . partition1 belong to executor 1 . partition2 belong to executor 2 . partition3 belong to executor 3 . now we want to apply filter transformations .I want to filter out all records where emp id =200. For this operation , each and every partition will work on this logic .logic would be provided by driver to the executor and each and every executor can work independently .in order to perform this logic ,they are not depending on other partitions . so that is the reason each and every partition can work independently, and it can return the result back to the driver . so that's the reason it is called narrow transformation. Narrow transformation does not require shuffling data across nodes because each and every partition can work independently.

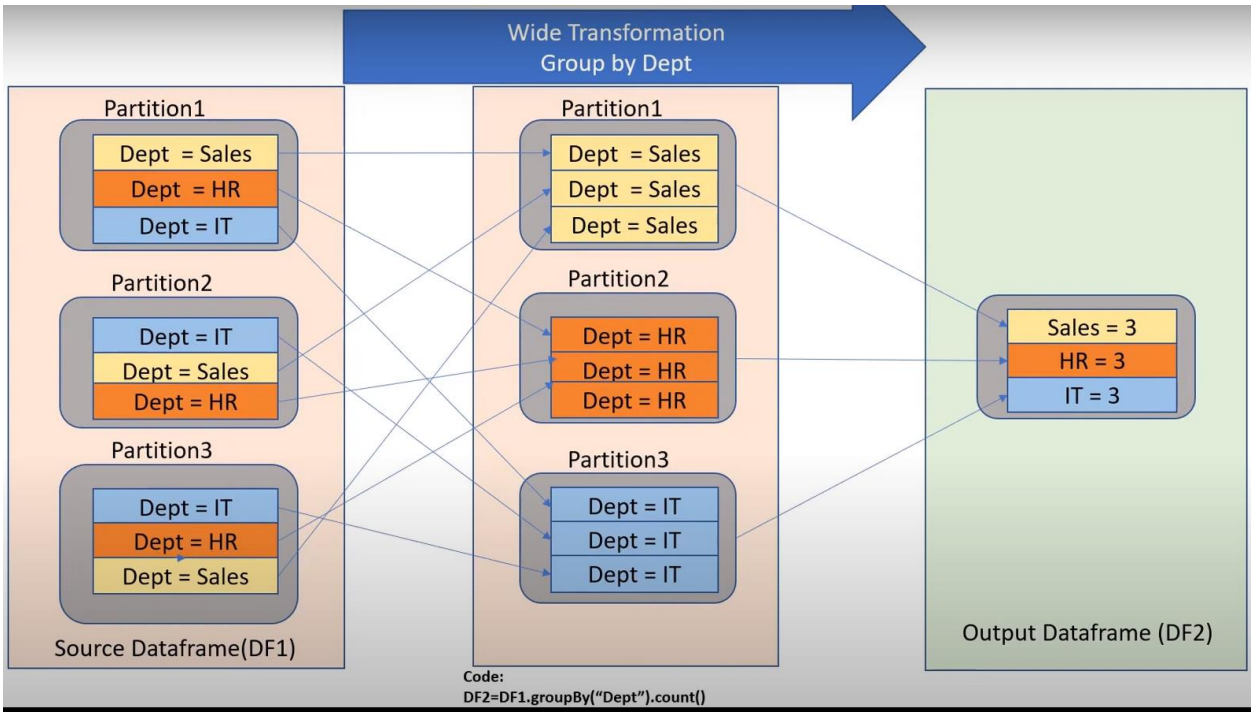
Code: **DF2=DF1.filter(DF1.emp_id=200)**

From DF1, we are filtering based on emp_id column =200 , then output will be DF2. This is the narrow transformation we don't need to shuffle the data across nodes .

WIDE TRANSFORMATION

In Wide transformation, data is shuffled across nodes ,that is why it is expensive, and it will hit the performance . why do we need to shuffle the data ? because each and every partition is depending on other partition to produce the output by applying the transformation .

Let's say here 3 executors. 1 partition in each executor . In 1st partition one record for sales, one record for HR, one record for IT.similarly we have data for partition2 and partition3 .now we have written logic to find number of records for each department. Basically, we have to groupBy department, and we need to find no of records for each department. So groupBy is wide transformation. In order to find number of records for each department , first partition can understand there is 1 record in this partition, but it does not know number of records in other partition so what it needs is we have to shuffle the data , first we have to group all the relevant records into 1 partition , then only it can perform number of records for each department. So, in order to perform that , first it needs to shuffle the data .for example this department sales record will remain in the 1st partition but in the 2nd partition sales record will move to partiton1 . similarly, sales record in partition 3 will move to partition 1. This is how shuffling happens.



Similarly shuffling happens for partition2 and partition 3 . finally, we keep only relevant records in each partition . so now it is easier for spark to find out number of records for each department. Finally, output will be returned to driver . This is an example for wide transformation.

Mostly used transformations are filter, groupBy, sortBy,union. Coming to action, mostly used actions are collect ,first,take,forEach and Save .

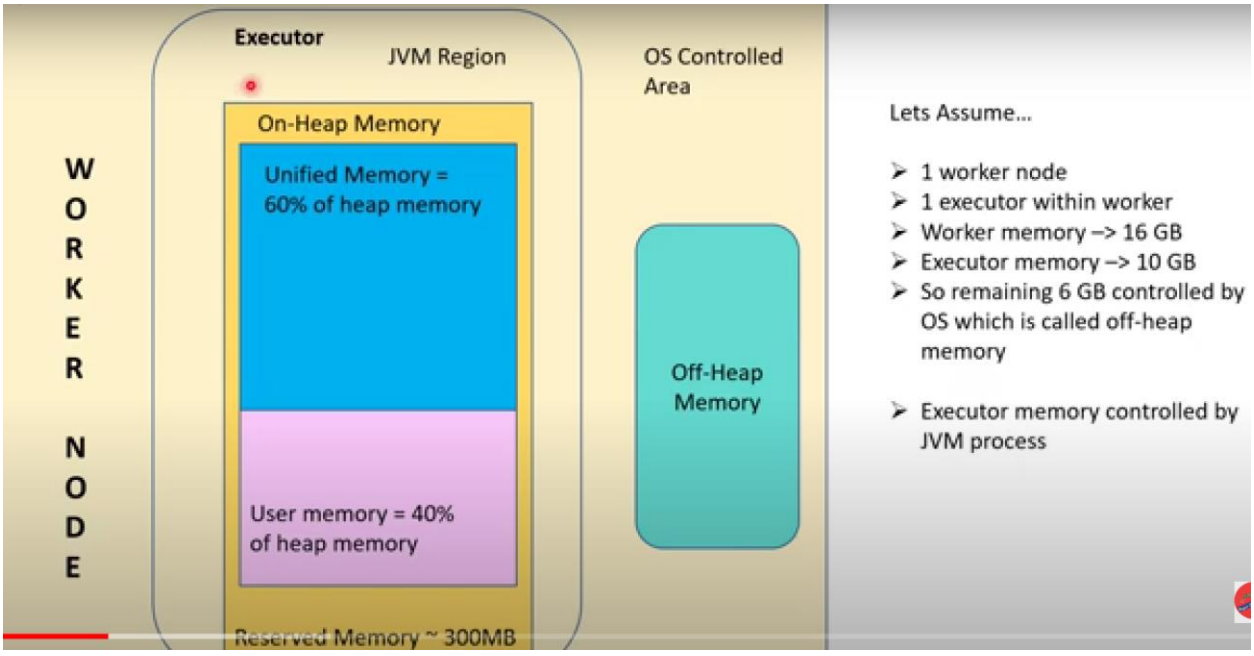
Transformation List:

- map
- filter
- flatMap
- mapPartitions
- mapPartitionsWithIndex
- groupBy
- sortBy
- union
- intersection
- subtract
- distinct
- cartesian
- zip
- sample
- randomSplit
- keyBy
- zipWithIndex
- zipWithUniqueID
- zipPartitions
- coalesce
- repartition
- pipe

Action List:

- reduce
- collect
- aggregate
- fold
- first
- take
- forEach
- top
- treeAggregate
- treeReduce
- forEachPartition
- collectAsMap
- count
- takeSample
- max
- min
- sum
- histogram
- mean
- variance
- stdev
- sampleVariance
- countApprox
- countApproxDistinct
- Save

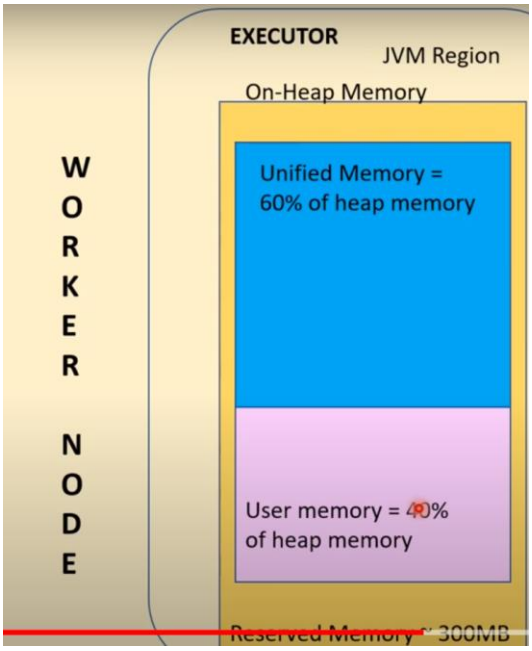
On-Heap vs Off-Heap



Spark architecture contains driver and worker nodes . In worker node, we can have n number of executors . For this example, we can have 1 executor . Let's example worker node is having 16 GB of memory , allocated 10GB for executor . So, there should be 6GB remaining , those 6 GB will be directly controlled

by OS that is called off-heap memory. The memory that is allocated for executor that is mainly controlled by JVM (java virtual machine) is called on-heap memory. The memory that is not controlled by java and controlled by OS is called Off-heap memory. For spark application , we can use either on-heap memory or off-heap memory. In most of the cases in order to design and develop efficient program , we have to use both. There are adv and disadv for both on-heap and off-heap memory.

On-Heap Memory Architecture:



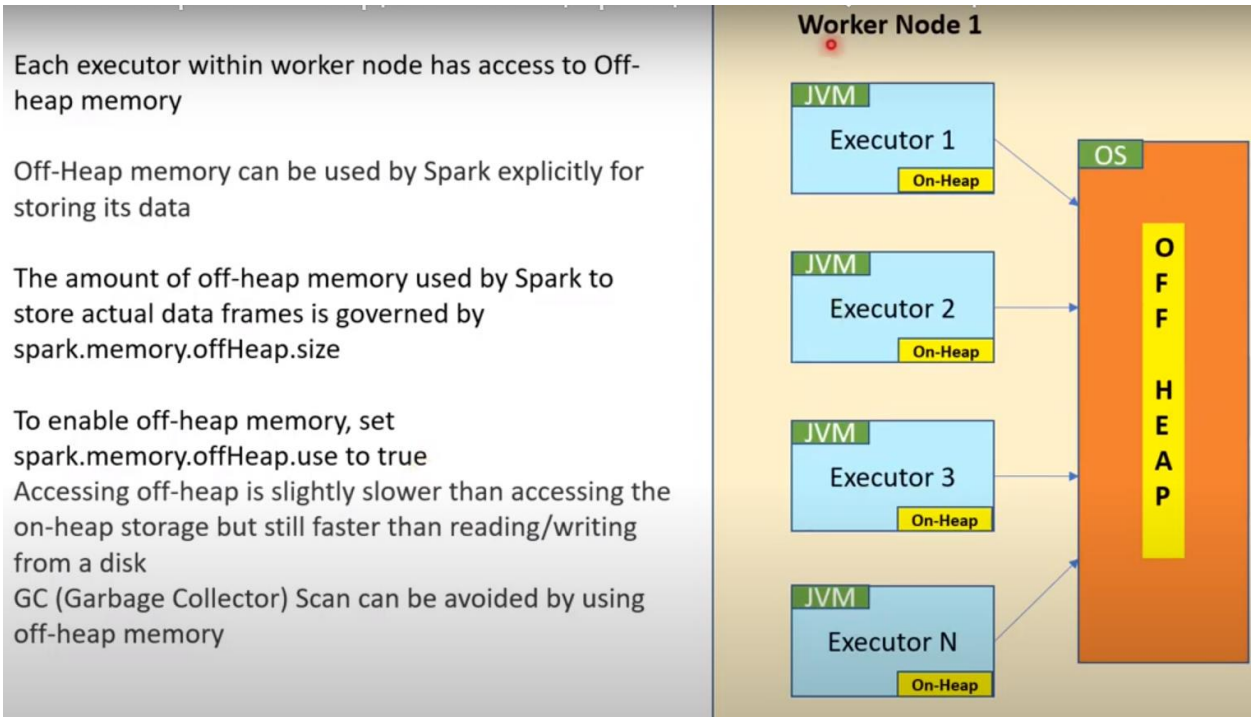
On-heap memory contains 4 regions Reserved Memory , User Memory, Execution Memory, and Storage Memory.

Execution Memory(50%) + Storage Memory(50%) combined as unified memory management. Reserved memory is used for spark internal purposes . if spark crashes ,this reserved memory comes into picture to recover the spark . **When user execute certain code to create lot of metadata kind of information ,that information would be maintained in the user memory.**

Memory management is highly configurable . we can change percent according to our use case by changing spark settings

OFF-Heap Memory : In worker node containing n no of executors . each executor will be controlled by its own JVM process, and each executor will contain its own on-heap memory. Common off heap memory i.e., common across all executors i.e; each executor can access its own on-heap memory and also can access commonly available off-heap memory . On-heap memory is controlled by JVM . Off-heap memory controlled by OS. Let's say executor 1 is having 1 GB of memory . for certain process it's going to execute 1 GB i.e; it cannot accommodate more than 1 GB within its on-heap memory. So, in that case executor can take help from off-heap memory.it can store some objects into off heap memory and data or objects created within off heap memory that is commonly accessible for all executors. Each executor within worker node has access to off-heap memory . off heap memory can be used by spark explicitly for storing its data .we have to change its spark settings. If we have to enable its spark off heap memory ,we must do

using spark settings we have to **set spark.memory.offHeap.use =true**. By default, it will be false .If we have decided to use off-heap memory for our programming, then first we must enable spark.memory.offHeap.use =true . secondly we have to allocate in how much memory we have to allocate into spark heap that can be controlled by another spark setting spark.memory.offHeap.size. Here out of os controlled memory, we must give certain percent or 5 GB to off-heap then this spark setting can help.



Coming to performance, always on heap memory is giving better performance but in case we cannot accommodate on heap memory for our program then we have to store that data into disk . instead of using disk, if we are using off heap memory that is always better . off heap memory is little slower than on heap memory but it is faster than disk . Garbage Collector is part of JVM process . each on heap memory would contain Garbage Collector . When on-heap memory is full, then Garbage collector will start scanning entire memory of on-heap then it will start removing absolute or very old objects from on heap memory. When objects are created and not used frequently ,then it will move the object to old generation and similarly when the on-heap memory is full ,if it does not have space to execute user program then GC will start scanning on heap memory and it will start removing absolute or unwanted objects so that it can give space for new program. But this is not applicable for off-heap memory .

Advantages of OFF-HEAP memory: we don't need to manage objects manually because it is completely automated by JVM process .

GC is not available with OFF-HEAP that is advantage.

Disadvantage of ON-HEAP memory: It will hit the performance . Let's say we are executing 1user program and in the middle of the process ,on heap memory is full then what happens is program execution is paused for some time then GC would start scanning entire memory and then it will start removing some objects . once space is available, then user program will continue which means user program will take more time to complete the execution because of GC scan .

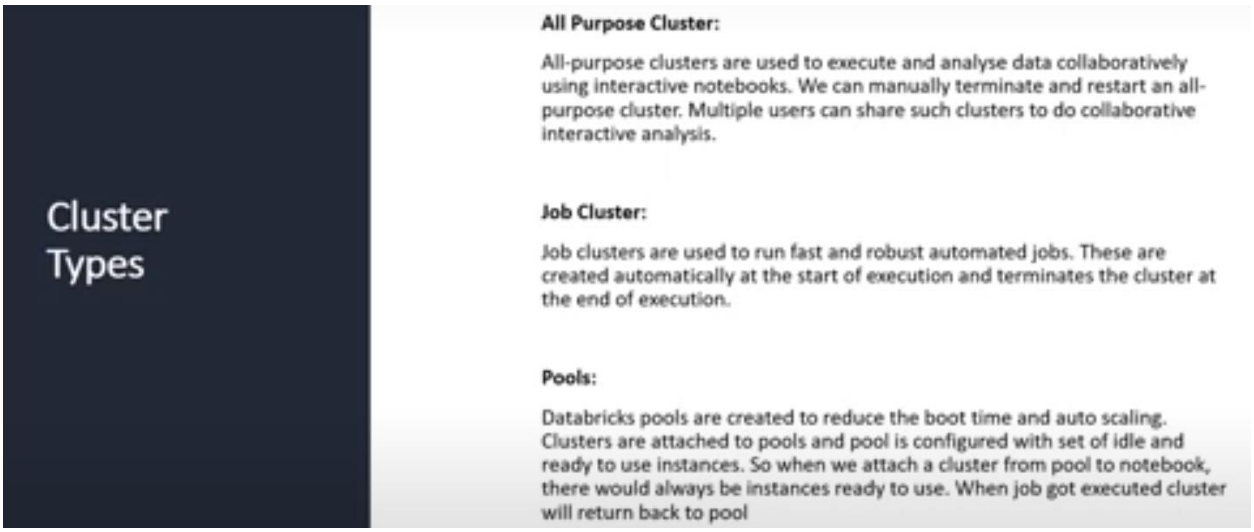
On-Heap	Off-Heap
Better performance than Off-heap because object allocation and deallocation happens automatically	Slower than On-heap but still better than disc performance. Manual memory management
Managed and controlled by Garbage collector within JVM process so adding overhead of GC scans	Directly managed by Operating system so avoiding the overhead of GC
Data stored in the format of Java bytes (deserialized) which Java can process efficiently	Data stored in the format of array of bytes(serialized). So adding overhead of serializing/ deserializing when java program needs to process the data
While processing smaller sets of data that can fit into heap memory, this option is suitable	When need to store bigger dataset that can not fit into heap memory, can make advantage of off-heap memory to store the data outside JVM process

9:57 / 11:55 Scroll for details

In on-heap memory, Data would be stored in deserialized or java bytes . In order to process the data within java environment ,it should be in java bytes i.e; deserialized form in on-heap memory. But coming to off-heap memory OS managed in any OS it should be in serialized form which means data is stored in the form of array of bytes which means in case in java process ,any java process has to access off-heap memory what happens is first it has to deserialize the data which means it has to convert the data from array of bytes to java bytes so that is adding overheads sometime then it will hit the performance.

DATABRICKS-CLUSTER DEPLOYMENT

Cluster is computing infrastructure in data bricks environment . In order to execute any code in data bricks , first we need to create cluster and we need to attach cluster with notebook . **Cluster is set of computational resources with proper configurations** . With the help of cluster, we can execute data engineering, data analytics and data science related workloads.



Cluster types: All Purpose clusters, Job Clusters

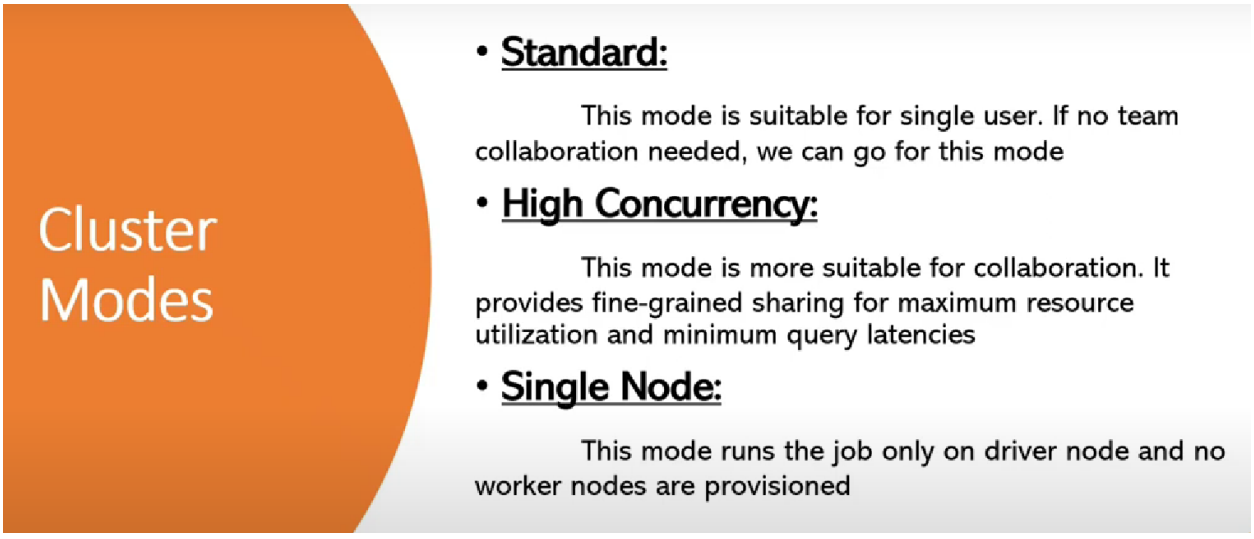
ALL PURPOSE CLUSTERS: All-purpose cluster created by us manually. All-purpose clusters we also call as interactive clusters because all purpose clusters are used mainly for development purpose. So, while

developing we must see the intermittent results .Let's say we are developing 10 commands and we have to see the result one by one for all the commands so that is the reason we can see the output interactively. All-purpose cluster can be used for development purpose and also for job cluster . All-purpose cluster can be terminated , restarted, or edited manually . **Multiple users can share all-purpose cluster**. While **multiple users are using the cluster then computing resource in that cluster would be shared among the developers.**

JOB CLUSTER: Job clusters are mainly used for scheduling jobs . While scheduling a job, we need to configure the cluster parameters. Based on that, cluster would be created during run time and once the job got completed then it would be terminated automatically.so manually we don't have control edit, terminate, or restart. Job clusters are visible only during job runtime.

POOLS: When we have multiple clusters and if we want to combine, then we can create pools and advantage of pool is while creating a pool is we can set parameters such as these many number of instances should be active always or ready to use which means in normal cluster when we are attaching to notebook and executing generally it will take few minutes to warm up, it will take atleast 3 -5 mins, so in order to avoid that we can go for pool. In pool, we can attach multiple clusters and also, we can tell these many nodes should be active all the time. So, which means when we are attaching cluster from the pool then we don't need to wait up for boot up time ,immediately it will start executing and this is mainly used for cost cutting, cost saving, and this is suitable for larger teams. **When there are multiple teams working in a project and one or another team would always need a cluster then this option is suitable .There would always be few instances running up. It could cost but at the same time it is useful for one or another team. So, this option is mainly suitable for larger teams.**

CLUSTER MODES: There are 3 modes. Standard, High Concurrency and Single Node.

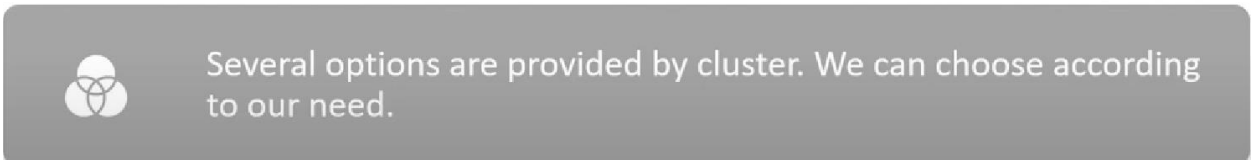


STANDARD MODE: This mode is suitable for single user. If only 1 developer is developing some notebook in data bricks environment , then standard mode can be given. This is not shared among team members and there is no collaboration on this cluster . This is dedicated for 1 particular developer .

HIGH CONCURRENCY MODE: This mode is more suitable for teamwork , team collaboration. This is giving option of sharing the resources among multiple developers . when we work as a team we can go for this option.

SINGLE NODE: Single node cluster will create only driver node and it will eliminate worker node . Single node doesn't have any worker node which means we can save money but at the same time we cannot execute complex jobs because we don't have worker to execute the code.so only driver node will handle all the work in single node.

CLUSTER RUNTIME:



Run time is set of core components that are needed to run cluster . There are multiple choices given by data bricks . Based on our choice , it would differ in usability, performance, and security. Based on our work need there are multiple workloads, it could be data engineering creating data pipelines/ data analytics / can be data science. Depending on our use case, we can choose run time .

CREATION OF CLUSTER IN DATABRICKS WORKSPACE:

Cluster name : can give any name

Cluster Mode: Standard(choosed for sample hands on)

Databricks Runtime Version: There are multiple options we can choose based on our use case. We can choose latest one version.

Autopilot Options : Enable autoscaling , Terminate after ____ minutes of inactivity .

If we click on enable autoscaling, we can select Min Workers and Max Workers. This is useful option for cost saving .Because if we are using executing less complex workloads then it will be used only minimum number of worker nodes so it will save the money . It will not put up all instances immediately .

When we are executing notebook initially if the workload is less then it will put up with 2 instances . Later slowly complexity increases, then spark will automatically increase more number of worker nodes until the maximum limit. So that's why we will enable autoscaling option . If we disable autoscaling, then it will ask for number of worker nodes that is fixed and irrespective of complex or less complex workloads , it will always put up all the worker nodes. Incase if we are going to execute less complex workload and keeping 8 worker nodes unnecessarily ,it will cost more money. So, in order to avoid that always better to use enable autoscaling . But if we know our workload accurately , based on that we can configure workers also without enabling autoscaling.

If we forgot to turn off the cluster after our usage , unnecessarily it will cost . In order to avoid that , we can give terminate this cluster after X minutes of inactivity. By default, it is 120 mins, we can change to 30 mins .

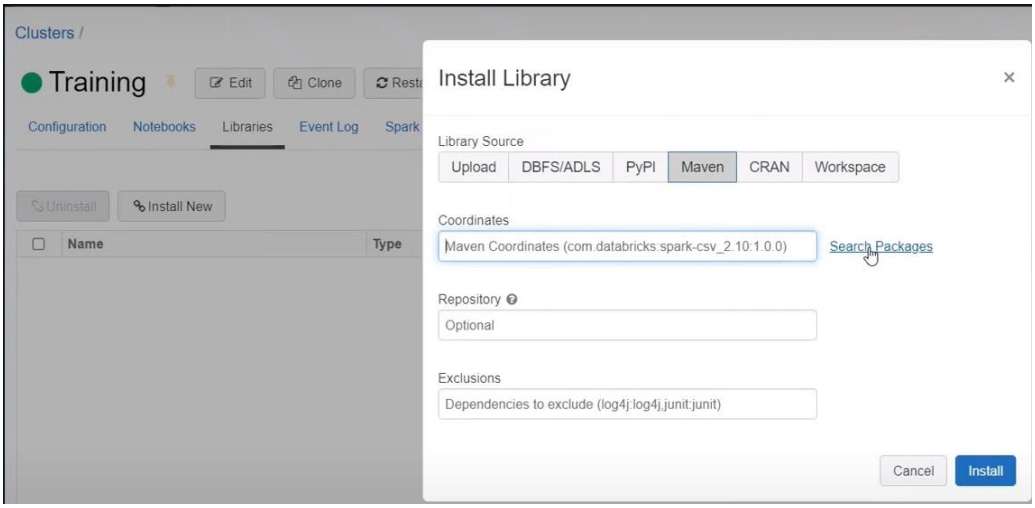
Spot instances: If our workload is not critical ,it is not urgent priority one, then we can go for spot instances which means azure will check for free instances that are available, then it will use for our worker nodes. So, it will be less cost when compared to regular instances . Incase our workload is very critical, business critical then we cannot allow any wait time then we can go for regular worker nodes i.e., azure will not wait for any free instances , immediately it will create new instances then it will start using in our cluster.

Worker Type: Workloads may differ into categories Standard, Memory Optimized, General, Storage Optimized ,Compute Optimized . Few notebooks will do compute optimized which means notebook will do lot of processes and the computing side then we can choose compute optimized worker type. In certain cases, workload would be more on storage side then we can go for Storage Optimized. If workload is on memory side, then we can go for Memory Optimized. Depending up on our use case we can select worker type.

Driver Type: Mostly we can select same as worker, or we can choose any option in the given list. In Databricks, driver type and worker type would be measured in Data Bricks Unit(DBU).

Advanced Options: we can configure few initialization parameters. In spark there are 100 of configuration parameters. For example, default spark shuffling parameters is 200 but we can change it to 500 or 50. Those Spark parameters can be configured in Spark Config. Tags tab mainly used for analysis purpose . For example, we can create tags for each and every cluster . Let's say in our workspace we have 100's of clusters and we want to perform certain analysis on the costing side then we can give tags what we like .Based on multiple categories we can add tags. Coming to Logging tab, while working in cluster, worker and driver node will create lot of logs. If we want to store the logs in to DBFS then we can choose DBFS as destination path. Coming to Init(Initialization) Scripts , if we want to configure certain parameters based on the script, then we can upload initialization script in data bricks file path and we can choose .

Once we created cluster, in Libraries tab we can install libraries for few integrations. For Example, we can integrate data bricks with Kafka then we need to install few relevant libraries in below tabs as shown below. We can also upload few library files in upload tab.



SPARK READER: READING CSV FILE IN DATABRICKS

Syntax:

Syntax:

```
df = spark.read.format(file_type) \
  .option("inferSchema", ) \
  .option("header", ) \
  .option("sep",) \
  .schema(schema_df)
  .load(file_location)

Options:
header='True'
inferSchema='True'
sep=','

Path:
Load(path1)
Load(path1, path2)
Load(folder)

Schema
Schema(schema)
```

Keyword spark in the syntax creates spark session . read key word is used for reading the files. Format key word used to specify type of the file. formats could be csv,parquet,orc,json,avro. Option key word used to specify parameters which are optional not mandatory if needed we can supply or we can ignore. inferSchema is if we want to determine the schema of the csv file then we will specify inferSchema. inferSchema is not recommended because it will hit the performance ,because it will read the data twice. Once to infer the schema ,2nd to create the data frame. So, this is not recommended but during unavoidable situation we can give.

If header is True, first record of csv file will be treated as header.header means column. If we give header as false , it will contain entire csv file as data. Separator option means in our csv file we can use | , ; : or tab . it could be anything. If we want to define that option, we can go for sep keyword.

When we are giving .schema() , inferSchema option is invalid we don't need to provide inferSchema because in schema we want to explicitly give the schema name. When we are giving inferSchema in option, then in .schema() we don't need to give explicitly schema name. for schema first we have to define the schema and then we can use that schema. We can define schema using StructType and StructField, if we want to use StructType and StructField, we want to import few libraries from pyspark or we can use 2nd syntax as shown below.

Schema Definition:

```
schema_define= StructType([StructField('Year', IntegerType(), True),
                           StructField('Name', StringType(), True),
                           StructField('County', StringType(), True),
                           StructField('Sex', StringType(), True),
                           StructField('Count', IntegerType(), False)
                           ])


schema = 'Year INTEGER, Name STRING, County STRING, Sex STRING, Count INTEGER'
```

Once we define schema, we can supply schema to .schema() .

.load() → for loading we have to give file location. File location could be single file, multiple files or we can read entire folder.

Code:

dbutils.fs.ls("/FileStore/tables/baby_names") → If we want to see files uploaded in the dbfs file system then we can use this command. baby_names is folder name.

```
Out[2]: [FileInfo(path='dbfs:/FileStore/tables/baby_names/Baby_Names_2007_2009.csv', name='Baby_Names_2007_2009.csv', size=484978),
 FileInfo(path='dbfs:/FileStore/tables/baby_names/Baby_Names_2010_2012.csv', name='Baby_Names_2010_2012.csv', size=468883),
 FileInfo(path='dbfs:/FileStore/tables/baby_names/Baby_Names_2013_2014.csv', name='Baby_Names_2013_2014.csv', size=362725)]
```

How to read csv file into data frame?

```
df= spark.read.format("csv")\
    .option("inferSchema",True)\
    .option("header",True)\
    .option("sep",",")\
     .load("/FileStore/tables/baby_names/Baby_Names_2007_2009.csv")
```

→ inferSchema true means it will read entire file and it will infer the schema. If we give header True means first record of the file will be treated as a columns. If header is false , first record of the csv file is treated as a data not columns. By default, spark will create column names as c0,c1,c2,c3,c4. In csv file , created comma separator data that's why giving comma as a separator

```
display(df)
print(df.count())
```

Another way of writing options in command is ?

```
df= spark.read.format("csv")\
    .options(inferSchema="True",header="True",sep=",")\
    .load("/FileStore/tables/baby_names/Baby_Names_2007_2009.csv")
display(df)
```

How to read multiple csv files?

```
df= spark.read.format("csv").options(inferSchema="True",sep=","
,header="True").load(["/FileStore/tables/baby_names/Baby_Names_2007_2009.csv","/FileS
tore/tables/Baby_Names/baby_names_2010_2012.csv"])
display(df)
print(df.count())
```

How to read all csv files under a folder?

```
df= spark.read.format("csv")\
.option("inferSchema",True)\
```

```
.option("header",True)\
.option("sep",",")\
.load("/FileStore/tables/baby_names/")
display(df)
print(df.count())
```

if we want to see schema for our csv file,

```
df.printSchema()
```

```
root
 |-- Year: integer (nullable = true)
 |-- First Name: string (nullable = true)
 |-- County: string (nullable = true)
 |-- Sex: string (nullable = true)
 |-- Count: integer (nullable = true)
```

How to create our own schema ?

```
from pyspark.sql.types import StructType,StructField,IntegerType,StringType
schema_defined =StructType([StructField()])

from pyspark.sql.types import
StructType,StructField,IntegerType,StringType,DoubleType
schema_defined = StructType([StructField('Year',IntegerType())
                             ,StructField('First Name',StringType())
                             ,StructField('Country',StringType())
                             ,StructField('Sex' , StringType())
                             ,StructField('Count',IntegerType())
]
)
```

Here we are explicitly supplying the schema using StructType and StructField so this is recommended approach , will perform far better because it will not read the data twice like inferSchema , it will read the data once and will apply schema defined on top of that.

```
df= spark.read.format("csv")\
.option("header",True)\
.option("sep",",")\
.schema(schema_defined)\
.load("/FileStore/tables/baby_names/")

df.printSchema()
display(df)
```

Another approach to create schema

```
schema_alternate = 'Year INTEGER,Name STRING,Country STRING,Sex STRING,Count INTEGER'

df= spark.read.format("csv")\
.option("header",True)\
.option("sep",",")\
.schema(schema_alternate)\
.load("/FileStore/tables/baby_names/")
```

```
df.printSchema()
```

Filter transformation

Filter function in pyspark basically used to limit the data output. If we want to extract subset of master data based on certain condition, then we can go for filter function. Filter can be applied through various techniques as shown below.

- filter(df.column != 50)
- filter((f.col('column1') > 50) & (f.col('column2') > 50))
- filter((f.col('column1') > 50) | (f.col('column2') > 50))
- filter(df.column.isNull())
- filter(df.column.isNotNull())
- filter(df.column.like('%%'))
- filter(df.name.isin())
- filter(df.column.contains(''))
- filter(df.column.startswith(''))
- filter(df.column.endswith(''))

Eg:

```
employee_data = [(10,"Raj Kumar","1999","100","M",2000),
                 (20,"Rahul Rajan","2002","200","F",8000),
                 (30,"Raghav","2010","100",None,6000),
                 (40,"Raja Singh","2004","100","F",7000),
                 (50,"Rama Krish","2008","400","M",1000),
                 (60,"Rasul","2014","500","M",5000),
                 (70,"Kumar Chand","2004","600","M",5000)
]
```

```
employee_schema = ["employee_id","name","doj","employee_dept_id","gender","salary"]
employeeDF =spark.createDataFrame(employee_data,employee_schema)
```

```
employeeDF: pyspark.sql.dataframe.DataFrame
  employee_id: long
  name: string
  doj: string
  employee_dept_id: string
  gender: string
  salary: long
```

```
display(employeeDF)
```

Table +

	employee_id	name	doj	employee_dept_id	gender	salary
1	10	Raj Kumar	1999	100	M	2000
2	20	Rahul Rajan	2002	200	F	8000
3	30	Raghav	2010	100	null	6000
4	40	Raja Singh	2004	100	F	7000
5	50	Rama Krish	2008	400	M	1000
6	60	Rasul	2014	500	M	5000
7	70	Kumar Chand	2004	600	M	5000

7 rows | 11.35 seconds runtime

Display employees whose salary is less than or equal to 5000.

```
display(employeeDF.filter(employeeDF.salary<=5000))    # ==,>,<,<=,>=,!=
```

Table +

	employee_id	name	doj	employee_dept_id	gender	salary
1	10	Raj Kumar	1999	100	M	2000
2	50	Rama Krish	2008	400	M	1000
3	60	Rasul	2014	500	M	5000
4	70	Kumar Chand	2004	600	M	5000

Display employees whose gender is female or doj of employee is 2004.

we can combine multiple filter conditions together

```
display(employeeDF.filter((employeeDF.gender=="F") | (employeeDF.doj==2004)))
```

Table +

	employee_id	name	doj	employee_dept_id	gender	salary
1	40	Raja Singh	2004	100	F	7000

Display employees whose names endswith h

```
display(employeeDF.filter(employeeDF.name.endswith("h")))
```

Table +

	employee_id	name	doj	employee_dept_id	gender	salary
1	40	Raja Singh	2004	100	F	7000
2	50	Rama Krish	2008	400	M	1000

Display employees whose names starts with Raj

```
display(employeeDF.filter(employeeDF.name.startswith("Raj")))
```

Table +

	employee_id	name	doj	employee_dept_id	gender	salary
1	10	Raj Kumar	1999	100	M	2000
2	40	Raja Singh	2004	100	F	7000

Display employees whose names contains kumar

```
display(employeeDF.filter(employeeDF.name.contains("Kumar")))
```

Table +

	employee_id	name	doj	employee_dept_id	gender	salary
1	10	Raj Kumar	1999	100	M	2000
2	70	Kumar Chand	2004	600	M	5000

Display employees whose gender is null

```
display(employeeDF.filter(employeeDF.gender.isNull()))
```

Table +

	employee_id	name	doj	employee_dept_id	gender	salary
1	30	Raghav	2010	100	null	6000

Display employees whose gender is not null

```
display(employeeDF.filter(employeeDF.gender.isNotNull()))
```

Table +

	employee_id	name	doj	employee_dept_id	gender	salary
1	10	Raj Kumar	1999	100	M	2000
2	20	Rahul Rajan	2002	200	F	8000
3	40	Raja Singh	2004	100	F	7000
4	50	Rama Krish	2008	400	M	1000
5	60	Rasul	2014	500	M	5000
6	70	Kumar Chand	2004	600	M	5000

Display employees whose salary equal to 1000 or 5000

We can give list of the values. If the value of the table is matching with any of the value, then it will return that records. I want to filter the records with salary equal to 1000 or 5000 we can get it using isin .

```
display(employeeDF.filter(employeeDF.salary.isin(1000,5000)))
```

Table +

	employee_id	name	doj	employee_dept_id	gender	salary
1	50	Rama Krish	2008	400	M	1000
2	60	Rasul	2014	500	M	5000
3	70	Kumar Chand	2004	600	M	5000

We don't have isnotin function , if we want to get records with isnotin data then we need to keep ~ .

```
display(employeeDF.filter(~employeeDF.salary.isin(1000,5000)))
```

Table +

	employee_id	name	doj	employee_dept_id	gender	salary
1	10	Raj Kumar	1999	100	M	2000
2	20	Rahul Rajan	2002	200	F	8000
3	30	Raghav	2010	100	null	6000
4	40	Raja Singh	2004	100	F	7000

Display employees whose name has kumar in it

like is pattern matching operator. If we want to match particular pattern, we can go for like operator.

```
display(employeeDF.filter(employeeDF.name.like("%Kumar%")))  → %Kumar% means this particular value will be at the start / in the middle or in the end.
```

Table +

	employee_id	name	doj	employee_dept_id	gender	salary
1	10	Raj Kumar	1999	100	M	2000
2	70	Kumar Chand	2004	600	M	5000

ADD , DROP AND RENAME COLUMNS IN DATA FRAME

- **Add Column:**
- DF.withColumn("new_column_name", Value)

- **Rename Column:**
- DF.withColumnRenamed("Old_name", "New_name")

- **Drop Column:**
- DF.drop("Column_name")

```
employee_data = [(10,"Raj","Kumar","1999","100","M",2000),
                 (20,"Rahul","Rajan","2002","200","F",8000),
                 (30,"Raghav","Manish","2010","100",None,6000),
                 (40,"Raja","Singh","2004","100","F",7000),
                 (50,"Rama","Krish","2008","400","M",1000),
                 (60,"Rasul","Kutty","2014","500","M",5000),
                 (70,"Kumar","Chand","2004","600","M",5000)
]

employee_schema = ["employee_id","first_name","last_name",
"doj","employee_dept_id","gender","salary"]
empDF=spark.createDataFrame(employee_data,employee_schema)
display(empDF)
```

(3) Spark Jobs

Table +

	employee_id	first_name	last_name	doj	employee_dept_id	gender	salary
1	10	Raj	Kumar	1999	100	M	2000
2	20	Rahul	Rajan	2002	200	F	8000
3	30	Raghav	Manish	2010	100	null	6000
4	40	Raja	Singh	2004	100	F	7000
5	50	Rama	Krish	2008	400	M	1000
6	60	Rasul	Kutty	2014	500	M	5000
7	70	Kumar	Chand	2004	600	M	5000

7 rows | 1.44 seconds runtime

How to add new column using constant literal for all rows?

```
from pyspark.sql.functions import lit
empDF_AddColumn = empDF.withColumn("Location",lit("Mumbai")).show()
```

(3) Spark Jobs

```
+-----------+----------+---------+----+----------------+------+------+--------+
|employee_id|first_name|last_name| doj|employee_dept_id|gender|salary|Location|
+-----------+----------+---------+----+----------------+------+------+--------+
|         10|       Raj|    Kumar|1999|             100|     M|  2000|  Mumbai|
|         20|     Rahul|    Rajan|2002|             200|     F|  8000|  Mumbai|
|         30|    Raghav|   Manish|2010|             100|  null|  6000|  Mumbai|
|         40|      Raja|    Singh|2004|             100|     F|  7000|  Mumbai|
|         50|      Rama|    Krish|2008|             400|     M|  1000|  Mumbai|
|         60|     Rasul|    Kutty|2014|             500|     M|  5000|  Mumbai|
|         70|     Kumar|    Chand|2004|             600|     M|  5000|  Mumbai|
+-----------+----------+---------+----+----------------+------+------+--------+
```

How to add new column using calculation?

```
empDF_AddColumn = empDF.withColumn("Bonus",empDF.salary*0.1).show()
```

▸ (3) Spark Jobs

```
+-----------+----------+---------+----+----------------+------+------+-----+
|employee_id|first_name|last_name| doj|employee_dept_id|gender|salary|Bonus|
+-----------+----------+---------+----+----------------+------+------+-----+
|         10|       Raj|    Kumar|1999|             100|     M|  2000|200.0|
|         20|     Rahul|    Rajan|2002|             200|     F|  8000|800.0|
|         30|    Raghav|   Manish|2010|             100|  null|  6000|600.0|
|         40|      Raja|    Singh|2004|             100|     F|  7000|700.0|
|         50|      Rama|    Krish|2008|             400|     M|  1000|100.0|
|         60|     Rasul|    Kutty|2014|             500|     M|  5000|500.0|
|         70|     Kumar|    Chand|2004|             600|     M|  5000|500.0|
+-----------+----------+---------+----+----------------+------+------+-----+
```

How to create new multiple columns at a time in pyspark?

```
from pyspark.sql.functions import concat
empDF_AddColumn = empDF.withColumn("Bonus",empDF.salary*0.1)\
                       .withColumn("Name",concat(empDF.first_name,empDF.last_name)).s
how()
```

▸ (3) Spark Jobs

```
+-----------+----------+---------+----+----------------+------+------+-----+------------+
|employee_id|first_name|last_name| doj|employee_dept_id|gender|salary|Bonus|        Name|
+-----------+----------+---------+----+----------------+------+------+-----+------------+
|         10|       Raj|    Kumar|1999|             100|     M|  2000|200.0|    RajKumar|
|         20|     Rahul|    Rajan|2002|             200|     F|  8000|800.0|  RahulRajan|
|         30|    Raghav|   Manish|2010|             100|  null|  6000|600.0|RaghavManish|
|         40|      Raja|    Singh|2004|             100|     F|  7000|700.0|   RajaSingh|
|         50|      Rama|    Krish|2008|             400|     M|  1000|100.0|   RamaKrish|
|         60|     Rasul|    Kutty|2014|             500|     M|  5000|500.0|  RasulKutty|
|         70|     Kumar|    Chand|2004|             600|     M|  5000|500.0|  KumarChand|
+-----------+----------+---------+----+----------------+------+------+-----+------------+
```

If we want to get space between Name column in concatenation, we will do as below.

```
from pyspark.sql.functions import concat,lit
empDF_AddColumn = empDF.withColumn("Bonus",empDF.salary*0.1)\
.withColumn("Name", concat(empDF.first_name,lit(" "),empDF.last_name)).show()
```

▸ (3) Spark Jobs

```
+-----------+----------+---------+----+----------------+------+------+-----+-------------+
|employee_id|first_name|last_name| doj|employee_dept_id|gender|salary|Bonus|         Name|
+-----------+----------+---------+----+----------------+------+------+-----+-------------+
|         10|       Raj|    Kumar|1999|             100|     M|  2000|200.0|    Raj Kumar|
|         20|     Rahul|    Rajan|2002|             200|     F|  8000|800.0|  Rahul Rajan|
|         30|    Raghav|   Manish|2010|             100|  null|  6000|600.0|Raghav Manish|
|         40|      Raja|    Singh|2004|             100|     F|  7000|700.0|   Raja Singh|
|         50|      Rama|    Krish|2008|             400|     M|  1000|100.0|   Rama Krish|
|         60|     Rasul|    Kutty|2014|             500|     M|  5000|500.0|  Rasul Kutty|
|         70|     Kumar|    Chand|2004|             600|     M|  5000|500.0|  Kumar Chand|
+-----------+----------+---------+----+----------------+------+------+-----+-------------+
```

How to rename a column?

```
empDF_RenameColumn = empDF_AddColumn.withColumnRenamed("Name","Full_Name")
display(empDF_RenameColumn)
```

(3) Spark Jobs

empDF_RenameColumn: pyspark.sql.dataframe.DataFrame = [employee_id: long, first_name: string ... 7 more fields]

Table +

	employee_id	first_name	last_name	doj	employee_dept_id	gender	salary	Bonus	Full_Name
1	10	Raj	Kumar	1999	100	M	2000	200	Raj Kumar
2	20	Rahul	Rajan	2002	200	F	8000	800	Rahul Rajan
3	30	Raghav	Manish	2010	100	null	6000	600	Raghav Manish
4	40	Raja	Singh	2004	100	F	7000	700	Raja Singh
5	50	Rama	Krish	2008	400	M	1000	100	Rama Krish
6	60	Rasul	Kutty	2014	500	M	5000	500	Rasul Kutty
7	70	Kumar	Chand	2004	600	M	5000	500	Kumar Chand

7 rows | 1.12 seconds runtime

How to rename multiple columns?

```
empDF_RenameColumn = empDF_AddColumn.withColumnRenamed("Name","Full_Name")\
    .withColumnRenamed("doj","date_of_joining")
display(empDF_RenameColumn)
```

(3) Spark Jobs

empDF_RenameColumn: pyspark.sql.dataframe.DataFrame = [employee_id: long, first_name: string ... 7 more fields]

Table +

	employee_id	first_name	last_name	date_of_joining	employee_dept_id	gender	salary	Bonus	Full_Name
1	10	Raj	Kumar	1999	100	M	2000	200	Raj Kumar
2	20	Rahul	Rajan	2002	200	F	8000	800	Rahul Rajan
3	30	Raghav	Manish	2010	100	null	6000	600	Raghav Manish
4	40	Raja	Singh	2004	100	F	7000	700	Raja Singh
5	50	Rama	Krish	2008	400	M	1000	100	Rama Krish
6	60	Rasul	Kutty	2014	500	M	5000	500	Rasul Kutty
7	70	Kumar	Chand	2004	600	M	5000	500	Kumar Chand

7 rows | 1.02 seconds runtime

How to drop column?

```
empDF_dropColumn=empDF_RenameColumn.drop("Full_Name")
empDF_dropColumn.show()
```

(3) Spark Jobs

```
+-----------+----------+---------+---------------+-----------------+------+------+-----+
|employee_id|first_name|last_name|date_of_joining|employee_dept_id|gender|salary|Bonus|
+-----------+----------+---------+---------------+-----------------+------+------+-----+
|         10|       Raj|    Kumar|           1999|              100|     M|  2000|200.0|
|         20|     Rahul|    Rajan|           2002|              200|     F|  8000|800.0|
|         30|    Raghav|   Manish|           2010|              100|  null|  6000|600.0|
|         40|      Raja|    Singh|           2004|              100|     F|  7000|700.0|
|         50|      Rama|    Krish|           2008|              400|     M|  1000|100.0|
|         60|     Rasul|    Kutty|           2014|              500|     M|  5000|500.0|
|         70|     Kumar|    Chand|           2004|              600|     M|  5000|500.0|
+-----------+----------+---------+---------------+-----------------+------+------+-----+
```

We can drop multiple columns at a time in 1 shot as shown below.

```
empDF_dropColumn=empDF_RenameColumn.drop("Full_Name")\
                                   .drop("Bonus")
display(empDF_dropColumn)
```

(3) Spark Jobs

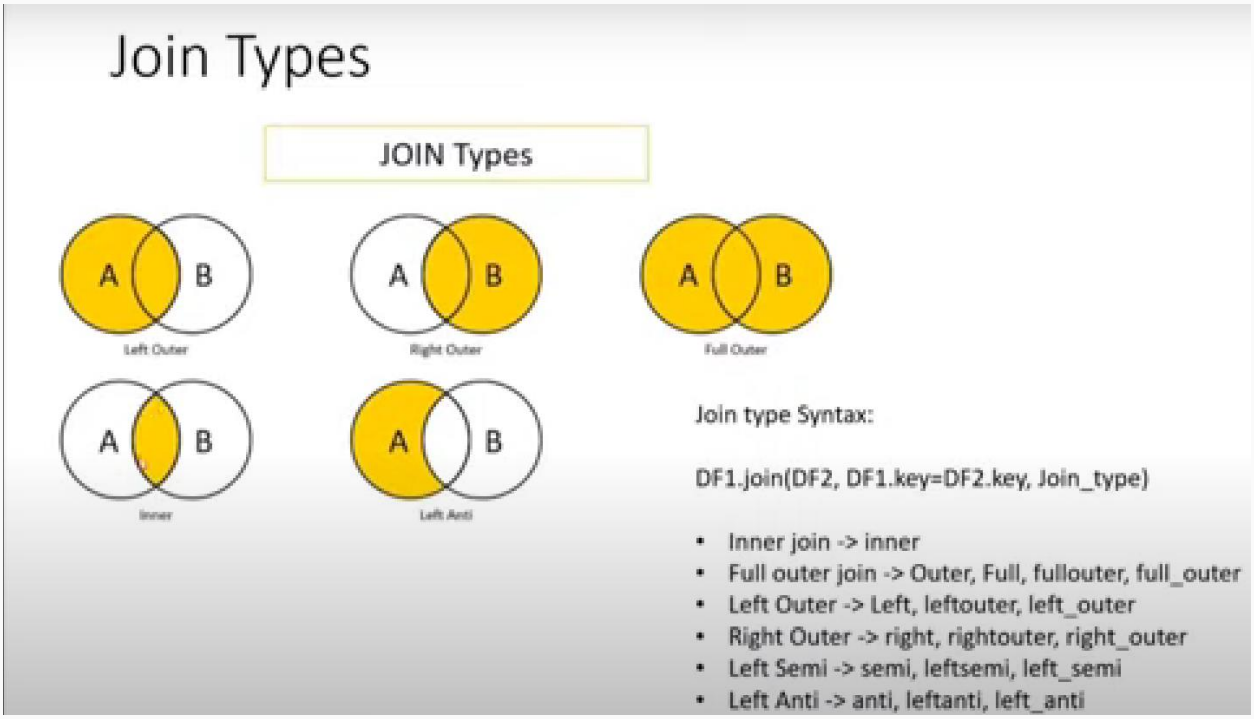
empDF_dropColumn: pyspark.sql.dataframe.DataFrame = [employee_id: long, first_name: string ... 5 more fields]

Table +

	employee_id	first_name	last_name	date_of_joining	employee_dept_id	gender	salary
1	10	Raj	Kumar	1999	100	M	2000
2	20	Rahul	Rajan	2002	200	F	8000
3	30	Raghav	Manish	2010	100	null	6000
4	40	Raja	Singh	2004	100	F	7000
5	50	Rama	Krish	2008	400	M	1000
6	60	Rasul	Kutty	2014	500	M	5000
7	70	Kumar	Chand	2004	600	M	5000

7 rows | 0.88 seconds runtime

PYSPARK JOIN TYPES



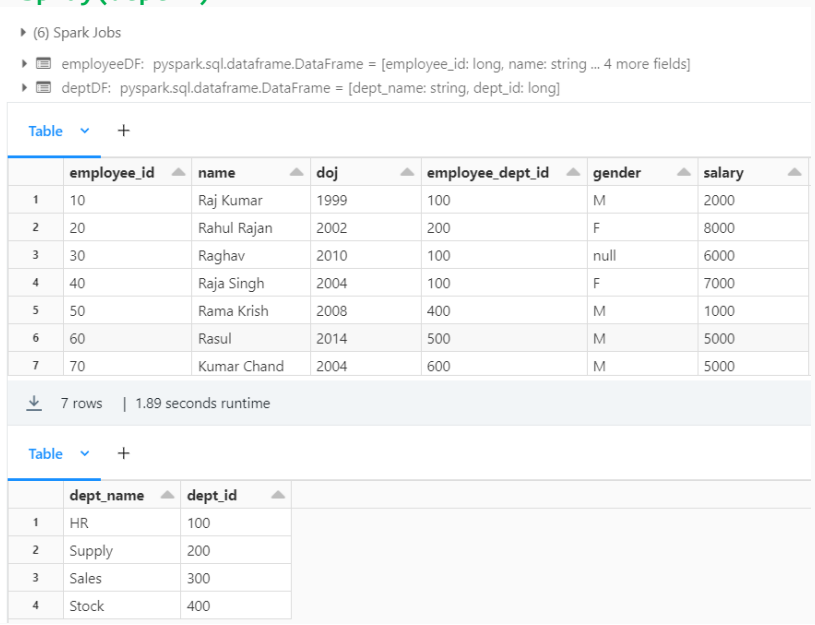
Left outer join will returns common records between 2 data frames and return all records from left data frame regardless of match found or not.in case there is no match found for particular record in right data frame, then all the values for the right table will be null.

Right outer join will returns common records between 2 data frames and return all records from right data frame regardless of match found or not. in case there is no match found for particular record in left data frame, then all the values for the left table will be null.

Full outer join is combination of left outer join + right outer join. Basically, it will return 3 types of datasets . one is common records between 2 data frames. 2nd is all the records from the left data frame regardless of match found or not. All the records from the right table regardless of match found or not. If there is no match found for the record, then the values would be null for the other data frame.

Eg:

```
employee_data = [(10,"Raj Kumar","1999","100","M",2000),
                 (20,"Rahul Rajan","2002","200","F",8000),
                 (30,"Raghav","2010","100",None,6000),
                 (40,"Raja Singh","2004","100","F",7000),
                 (50,"Rama Krish","2008","400","M",1000),
                 (60,"Rasul","2014","500","M",5000),
                 (70,"Kumar Chand","2004","600","M",5000)
                ]

employee_schema = ["employee_id","name","doj","employee_dept_id","gender","salary"]
employeeDF =spark.createDataFrame(employee_data,employee_schema)

department_data =[("HR",100),("Supply",200),("Sales",300),("Stock",400)]
department_schema=["dept_name","dept_id"]
deptDF=spark.createDataFrame(department_data,department_schema)
display(employeeDF)
display(deptDF)
```



▸ (6) Spark Jobs

▸ employeeDF: pyspark.sql.dataframe.DataFrame = [employee_id: long, name: string ... 4 more fields]

▸ deptDF: pyspark.sql.dataframe.DataFrame = [dept_name: string, dept_id: long]

Table +

	employee_id	name	doj	employee_dept_id	gender	salary
1	10	Raj Kumar	1999	100	M	2000
2	20	Rahul Rajan	2002	200	F	8000
3	30	Raghav	2010	100	null	6000
4	40	Raja Singh	2004	100	F	7000
5	50	Rama Krish	2008	400	M	1000
6	60	Rasul	2014	500	M	5000
7	70	Kumar Chand	2004	600	M	5000

7 rows | 1.89 seconds runtime

Table +

	dept_name	dept_id
1	HR	100
2	Supply	200
3	Sales	300
4	Stock	400

How to do inner join between 2 data frames?

```
df_join =employeeDF.join(deptDF,employeeDF.employee_dept_id==deptDF.dept_id,"inner")
```

or `df_join =employeeDF.join(deptDF,employeeDF.employee_dept_id==deptDF.dept_id).`

When we won't specify join condition explicitly , then spark will internally consider it as inner join.

```
display(df_join)
```

(3) Spark Jobs

df_join: pyspark.sql.dataframe.DataFrame = [employee_id: long, name: string ... 6 more fields]

Table +

	employee_id	name	doj	employee_dept_id	gender	salary	dept_name	dept_id
1	10	Raj Kumar	1999	100	M	2000	HR	100
2	30	Raghav	2010	100	null	6000	HR	100
3	40	Raja Singh	2004	100	F	7000	HR	100
4	20	Rahul Rajan	2002	200	F	8000	Supply	200
5	50	Rama Krish	2008	400	M	1000	Stock	400

5 rows | 4.25 seconds runtime

How to do left join between 2 data frames?

```
df_leftjoin =
employeeDF.join(deptDF,employeeDF.employee_dept_id==deptDF.dept_id,"left")
display(df_leftjoin)
```

(6) Spark Jobs

df_leftjoin: pyspark.sql.dataframe.DataFrame = [employee_id: long, name: string ... 6 more fields]

Table +

	employee_id	name	doj	employee_dept_id	gender	salary	dept_name	dept_id
1	10	Raj Kumar	1999	100	M	2000	HR	100
2	20	Rahul Rajan	2002	200	F	8000	Supply	200
3	30	Raghav	2010	100	null	6000	HR	100
4	40	Raja Singh	2004	100	F	7000	HR	100
5	50	Rama Krish	2008	400	M	1000	Stock	400
6	60	Rasul	2014	500	M	5000	null	null
7	70	Kumar Chand	2004	600	M	5000	null	null

7 rows | 2.13 seconds runtime

How to do right join between 2 data frames?

```
df_rightjoin =
employeeDF.join(deptDF,employeeDF.employee_dept_id==deptDF.dept_id,"right")

display(df_rightjoin)
```

(6) Spark Jobs

Table +

	employee_id	name	doj	employee_dept_id	gender	salary	dept_name	dept_id
1	40	Raja Singh	2004	100	F	7000	HR	100
2	30	Raghav	2010	100	null	6000	HR	100
3	10	Raj Kumar	1999	100	M	2000	HR	100
4	20	Rahul Rajan	2002	200	F	8000	Supply	200
5	null	null	null	null	null	null	Sales	300
6	50	Rama Krish	2008	400	M	1000	Stock	400

6 rows | 2.64 seconds runtime

How to do full outer join between 2 data frames?

```
df_fulljoin =
employeeDF.join(deptDF,employeeDF.employee_dept_id==deptDF.dept_id,"full")
display(df_fulljoin)
```

(3) Spark Jobs

Table +

	employee_id	name	doj	employee_dept_id	gender	salary	dept_name	dept_id
1	10	Raj Kumar	1999	100	M	2000	HR	100
2	30	Raghav	2010	100	null	6000	HR	100
3	40	Raja Singh	2004	100	F	7000	HR	100
4	20	Rahul Rajan	2002	200	F	8000	Supply	200
5	null	null	null	null	null	null	Sales	300
6	50	Rama Krish	2008	400	M	1000	Stock	400
7	60	Rasul	2014	500	M	5000	null	null

How to do left semi join between 2 data frames?

Left semi join will return common records between two data frames but in result set it will contain only columns from left table.

```
df_leftsemijoin =
employeeDF.join(deptDF,employeeDF.employee_dept_id==deptDF.dept_id,"leftsemi")
display(df_leftsemijoin)
```

(3) Spark Jobs

Table +

	employee_id	name	doj	employee_dept_id	gender	salary
1	10	Raj Kumar	1999	100	M	2000
2	30	Raghav	2010	100	null	6000
3	40	Raja Singh	2004	100	F	7000
4	20	Rahul Rajan	2002	200	F	8000
5	50	Rama Krish	2008	400	M	1000

How to do left anti join between 2 data frames?

Left anti join is opposite to left semi join. Left semi join returns all common records between 2 data frames but Left anti join will returns all uncommon records between 2 data frames which means in left table the uncommon records it will return.

```
df_leftantijoin =
employeeDF.join(deptDF,employeeDF.employee_dept_id==deptDF.dept_id,"leftanti")

display(df_leftantijoin)
```

(6) Spark Jobs

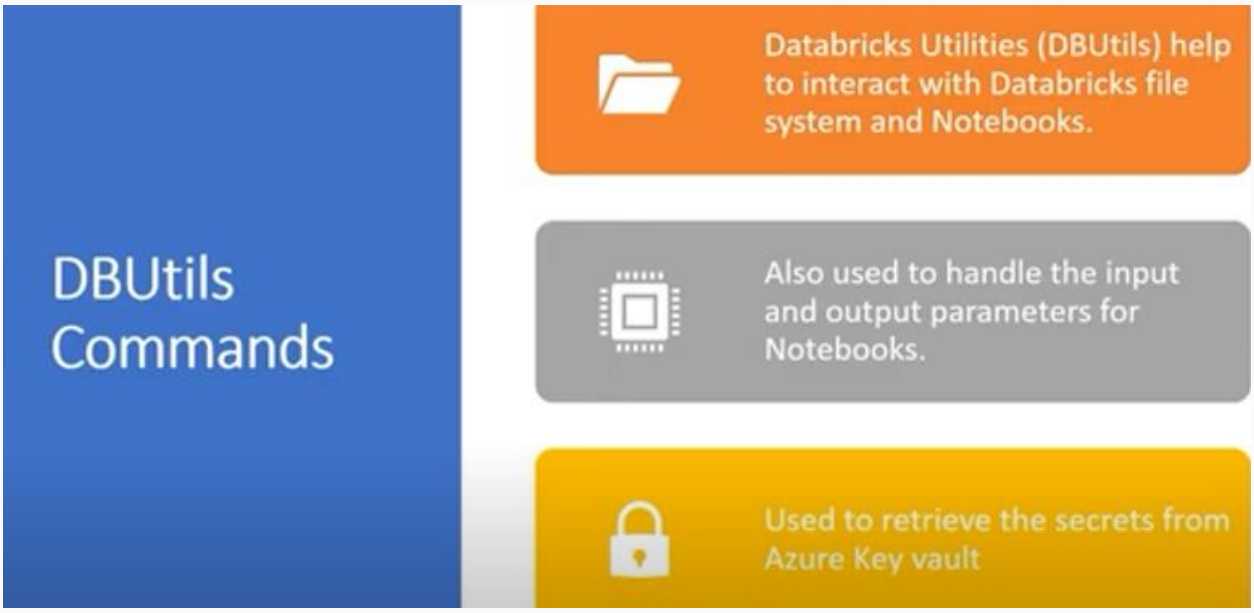
df_leftantijoin: pyspark.sql.dataframe.DataFrame = [employee_id: long, name: string ... 4 more fields]

Table +

	employee_id	name	doj	employee_dept_id	gender	salary
1	60	Rasul	2014	500	M	5000
2	70	Kumar Chand	2004	600	M	5000

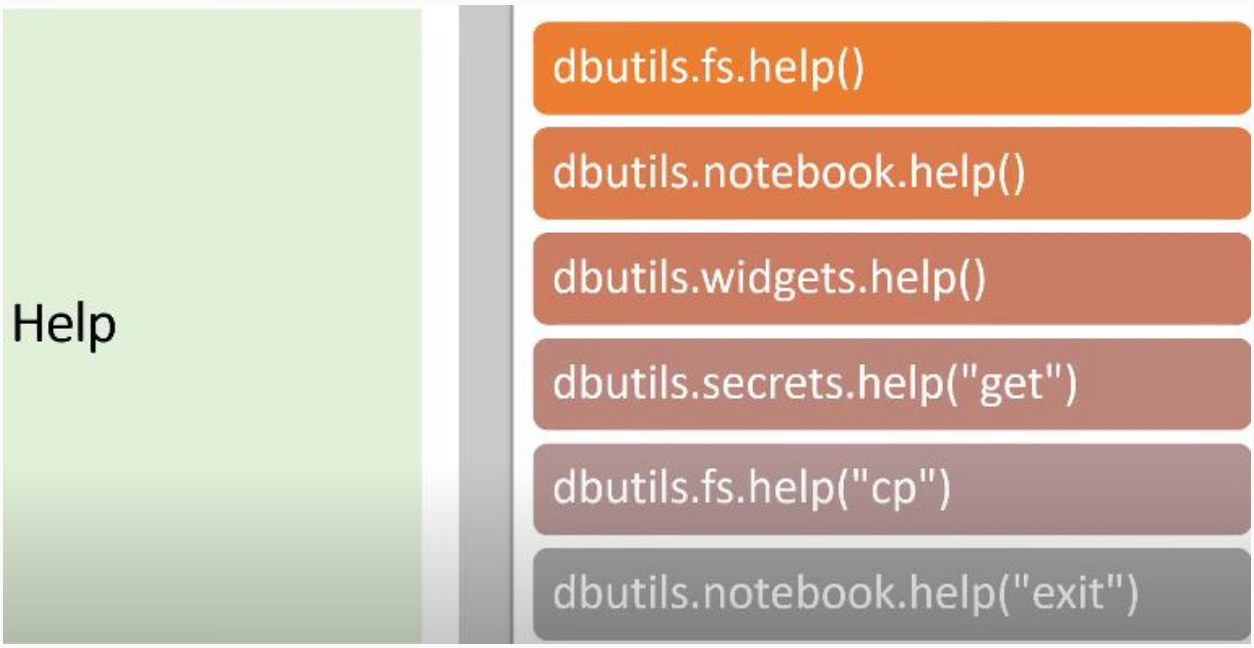
2 rows | 2.28 seconds runtime

UTILITY COMMANDS – DBUtils



These commands are used to interact with Databricks file systems. **These commands are also used to run notebook internally.** For example, one notebook can call another notebook. So, in order to run the notebooks manually from one notebook, we can use DBUtils commands. **This commands also used to handle input and output parameters while calling the other notebooks. Input parameters are handled in the form of widgets in data bricks.** While using data bricks utility commands, we can create input widgets. Another important usage for DBUtils commands is used to retrieve secrets from Azure

Key Vault. While working in data bricks ,mostly we are integrating with external systems such as external data bases. In order to interact with external data bases through data bricks, we have to use connection string. Connection string will consists of username and password for data base but it is always not good idea to save or hardcore the credentials within databricks notebook. In these cases, credentials would kept in key vault. Through DB utility commands , we can retrieve the passwords or secrets.



Data base utility commands are categorized as file systems, notebooks, widgets, and secrets.

dbutils.fs.help() → this command provides utilities for working with file systems.

fsutils methods:

- **cp** : copying file from one directory to another directory.

cp(from path, to path,recurse:boolean=false) if we are copying from 1 folder to another folder and we want to consider all subfolders then we have to copy recursively which means we have to give this recurse value true.

- **head**: used to display output of the file
- **ls**: used to list down all files under a particular directory including subdirectory.
- **mkdirs**: this command used to create new folder or making new directory.
- **mv**: this command used to move file from one directory to other folder.cp used to create another copy file .whereas mv command used to move file from source to destination, i.e., in source, file will not exist , it will move to destination.
- **put**: this command used to create a file
- **rm** : removes file or directory from the file system.

mount : In data bricks , if we have to interact with external file systems, the first step is we have to mount. For example, sometimes we have to work with azure data lake storage/amazon S3 bucket. So, what we have to do is first we have to mount storage location to azure data bricks, then data bricks will treat it as a local folder .

---

dbutils.notebook.help()

notebook methods:

- exit: while secondary notebook got executed, then it can return some value to the master notebook. So that value can be captured using exit.
- run: If we have to execute the notebook, then we can call dbutils.notebook.run(path,timeoutSeconds,parameters/arguments) .

---

dbutils.widgets.help()

widgets methods: we can handle input parameters using various methods.

- **text**: Most used method is text box. We can create text box and we can give some value. We can pass value while calling notebook.
- **combobox/dropdown**: Text box will hold only single value .If we have to handle with multiple values then we can go with combobox or dropdown box. Both are almost same.
- **multiselect**: If we want to go with multiselect option , then we can select multiselect method.
- **getArgument/get**: this commands are used to retrieve the value that are passed through combobox or textbox.
- **remove/removeAll:** If we have to remove already created widgets ,then we will use remove/removeAll. Remove used to remove particular widget. removeAll used to remove all widgets in the notebook.

---

dbutils.secrets.help()

secrets are used to store password/credentials for any system.we cannot hardcore in data bricks, we need to store the secrets in somewhere and these commands are used to retrieve those secrets.

secrets methods:

- **get/getBytes** : These commands will retrieve the data from external system maybe it could be azure key vault or could be secret storing system.
- **list/listScopes** : These commands will list down all the secrets. list command will lists secret metadata for secrets within a scope. listScopes will lists secret scopes.

If I want to get help on particular method cp let's say, then we need to get command as **dbutils.fs.help("cp")**

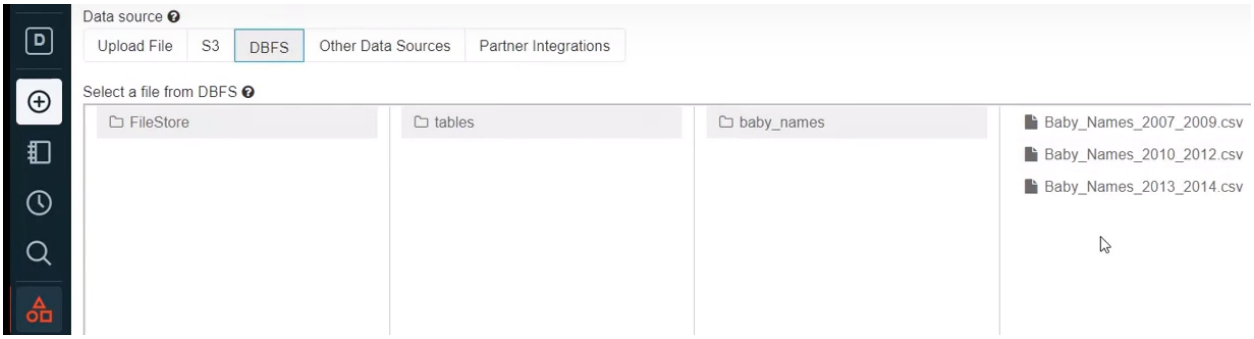
If we want to get help for all commands in fs utility, then dbutils.fs.help().

COMMANDS:

DBUtils File System Commands:

1) **dbutils.fs.ls("/FileStore/tables/baby_names")**

Explanation: This command will list down all files and folders under a folder. Here it accepts a parameter of path. baby_names is a folder as shown below.



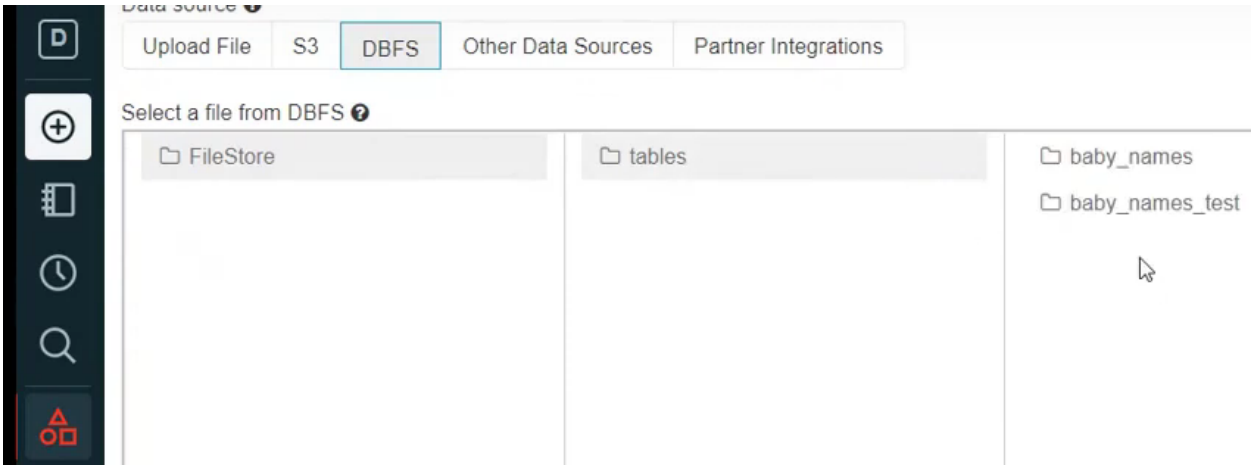
When we execute dbutils.fs.ls("/FileStore/tables/baby_names") , the below output will be displayed. It will display all files names which are there under baby_names folder and its size.

2) **dbutils.fs.head("/FileStore/tables/baby_names/Baby_Names_2007_2009.csv")**

Explanation: head is used to display the contents of a file . For example , if I want to see contents of file Baby_Names_2007_2009.csv,then we will run this command.

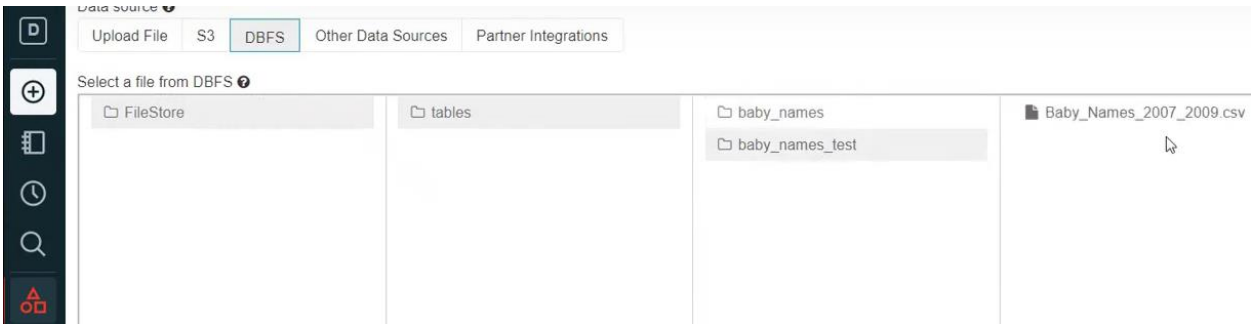
3) **dbutils.fs.mkdirs("/FileStore/tables/baby_names_test/")** output: True

Explanation: This command is used to create a new folder. For example, if I want to create new folder under tables, then using mkdirs command we will execute it.



4) **dbutils.fs.cp("/FileStore/tables/baby_names/Baby_Names_2007_2009.csv","/FileStore/tables/baby_names_test/")**

output: True



Explanation: this command cp used to copy file from one location to another location .

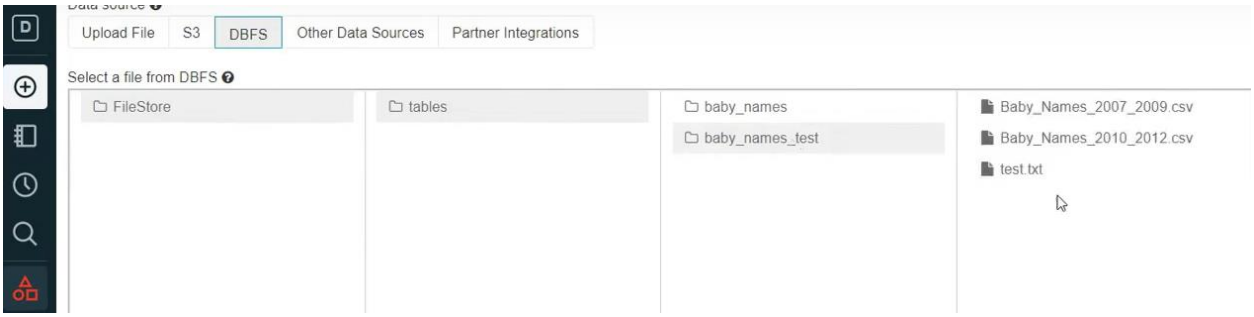
5) **dbutils.fs.mv("/FileStore/tables/baby_names/Baby_Names_2010_2012.csv","/FileStore/tables/baby_names_test/")** output: True

Explanation: mv command used to move file from one location to another location.

6) **dbutils.fs.put("/FileStore/tables/baby_names_test/test.txt","hello world")**

output: wrote 11 bytes.

True



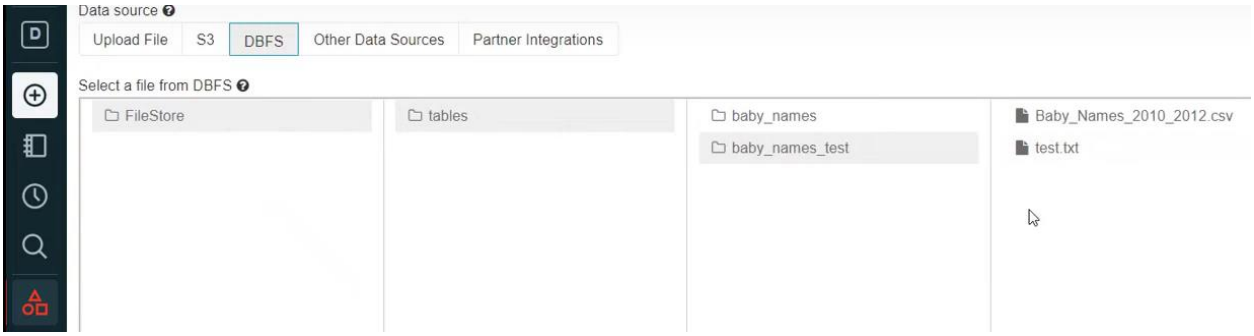
Explanation: put command is used to create new files within the data bricks file system. I want to create a new file under baby_names_test and we want to name the file test.txt and the content of the file would be hello world .

If I want to see contents of test.txt file, we will see it via head command.

dbutils.fs.head("/FileStore/tables/baby_names_test/test.txt") output: 'hello world'

7) **dbutils.fs.rm("/FileStore/tables/baby_names_test/Baby_Names_2007_2009.csv")**

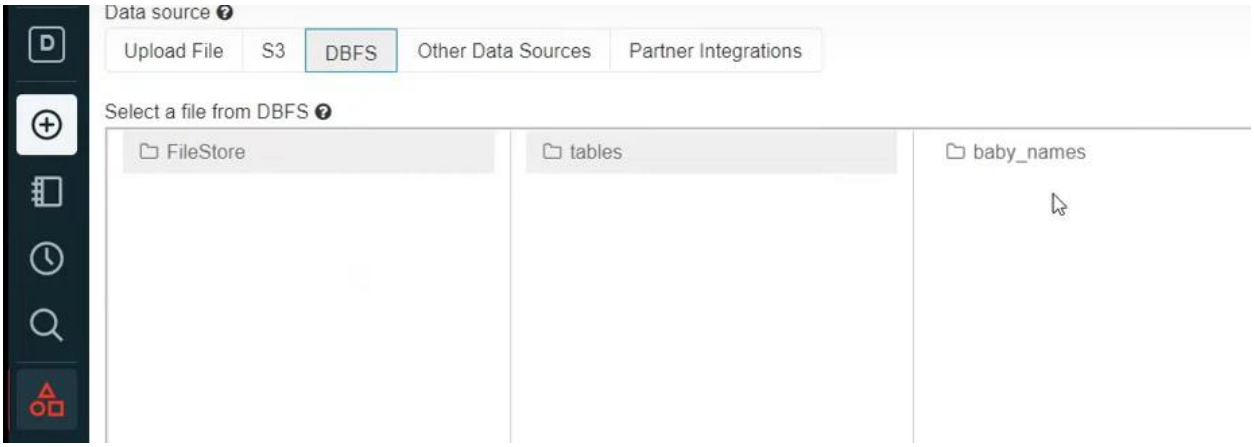
output: True



Explanation: This command is used to delete a file from a location

If we want to remove entire folder files from a location , then we have to give command as below.

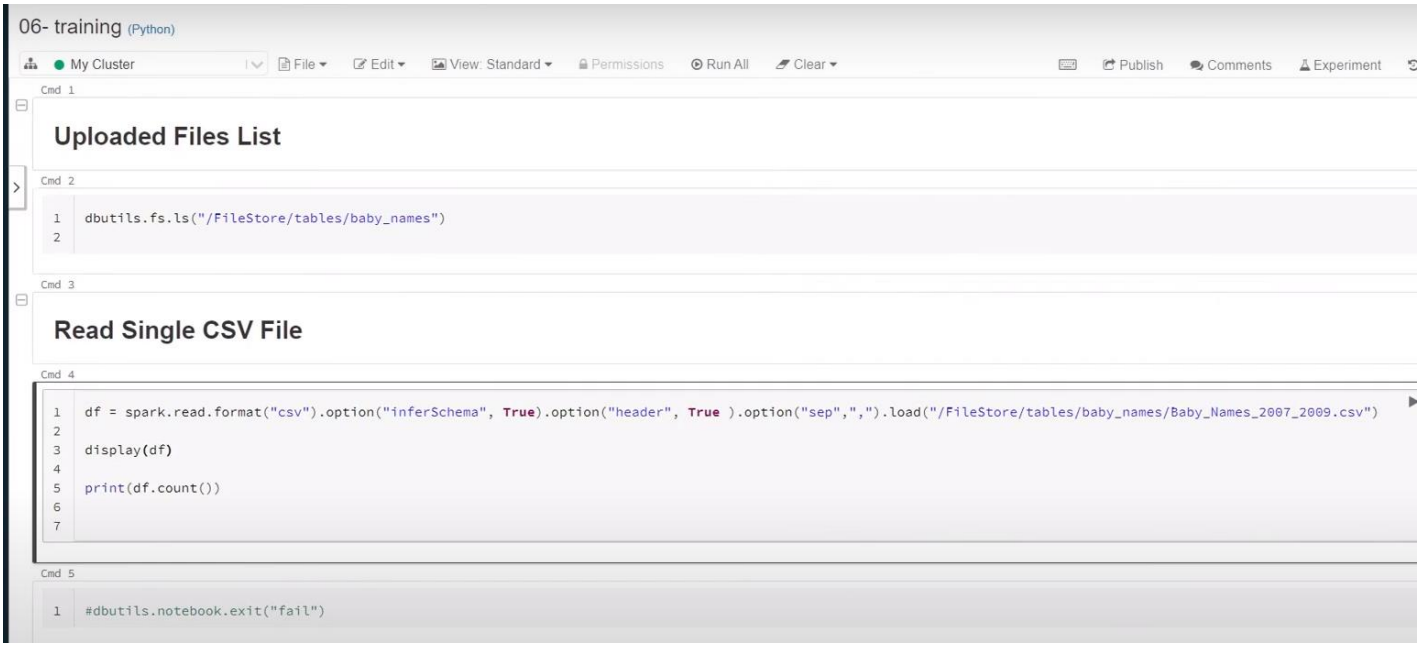
dbutils.fs.rm("/FileStore/tables/baby_names_test/" , True) output: True → we passed another parameter True which means it will recursively remove all the files under this folder . if there is any subfolder ,it will remove that also.



DBUtils Notebook Commands:

In order to execute the notebook, the below is the syntax

dbutils.notebook.run and we have to give notebook name(06-training)



I want to execute 06-training notebook from another notebook 06-DBUtils. So, this notebook(06-training) as shown above internally calling another notebook(06-DBUtils). In order to perform this operation, we will use dbutils.notebook.run command. we need to pass 2 parameters , 1 is notebook name and 2nd is time out in seconds , i.e; till what time spark can wait . if that notebook is not executed with in 60 secs then it will throw error time out error.

Command to run one notebook from another notebook:

dbutils.notebook.run("/06-training",60)

DBUtils Widgets Commands:

Use of widgets: This will accept input value from calling notebook. If there is master notebook that is calling this notebook(06_training_widget) then what will happen is master notebook would pass certain input parameters that would be captured here .

```
1) dbutils.widgets.text("Folder_name","","")
2) dbutils.widgets.text("File_name","","")
```

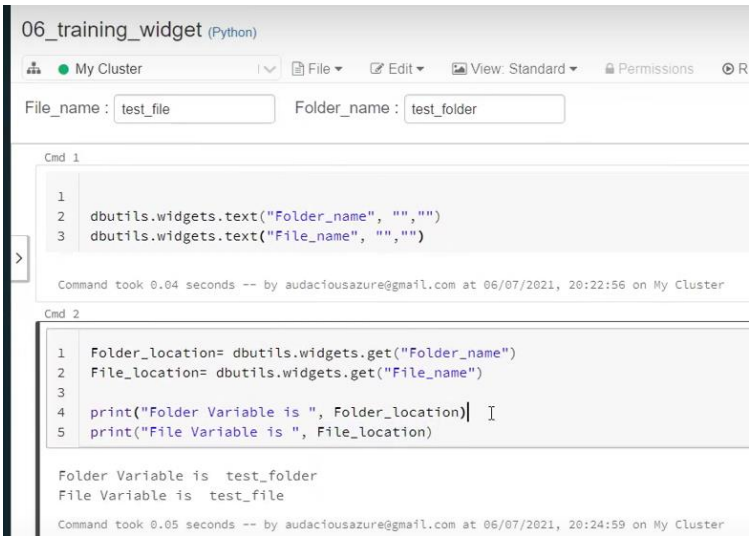
Once we execute above 2 commands, the below widget is created as shown below.

Explanation: we need to provide name for the input box and 2nd parameter we can give value . if we want to pass any default value then we can give it in the 3rd parameter. Basically, it accepts 3 parameters. First, we have to give name and 2nd we have to give value for the input box and 3rd parameter will be default value.

How to get value to some variable?

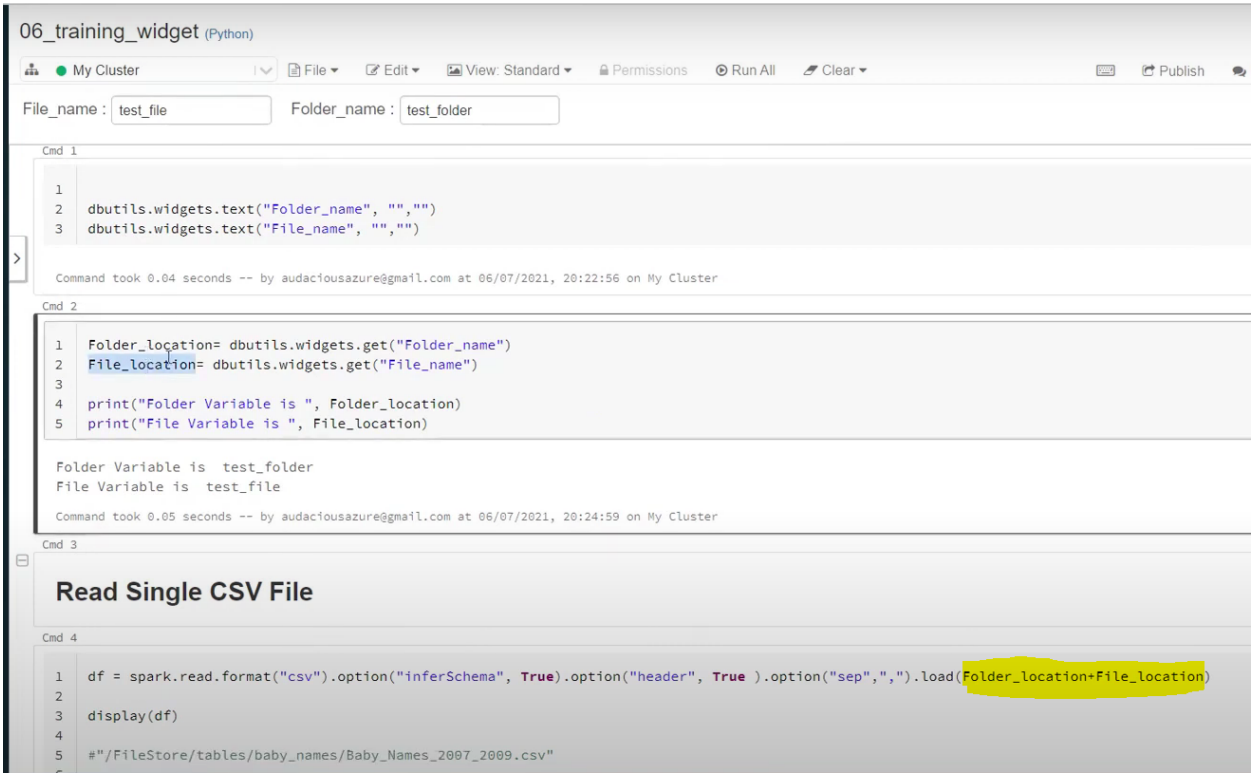
```
3) Folder_location =dbutils.widgets.get("Folder_name")
   File_location = dbutils.widgets.get("File_name")

   print("Folder Variable is" , Folder_location)
   print("File Variable is" , File_location)
```

If we pass File_name as test_file and Folder_name as test_folder as shown below, then if we execute get command , we will get op as below.



06-DBUtils is master notebook , we are going to call other notebook(06_training_widget) in master notebook 06-DBUtils for that syntax is as below. We need to give other notebook name then we have to pass 2 parameters folder name and file name . In order to pass parameters, we have to use $ symbol then folder name

%run ./06_training_widget $Folder_name =”/FileStore/tables/baby_names/” $File_name = “Baby_Names_2013_2014.csv”

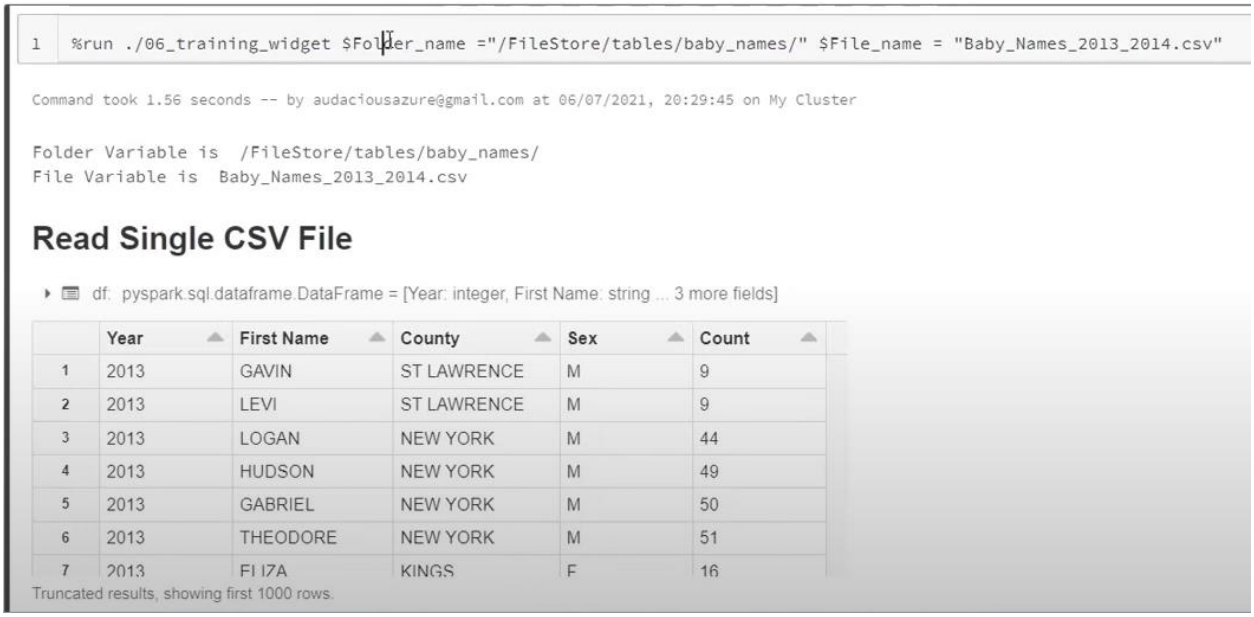


If we execute above command, these input values will be passed to 06_training_widget notebook and values in input widget in 06_training_widget notebook will be folder name

→ **/FileStore/tables/baby_names/** and file name will be → **Baby_Names_2013_2014.csv** and it will create folders as **Folder_location =/FileStore/tables/baby_names/** and File_location = Baby_Names_2013_2014.csv. By combining folder location and file location it will create absolute path and based on that it will create a data frame and that would be displayed as output.

%run ./06_training_widget $Folder_name =”/FileStore/tables/baby_names/” $File_name = “Baby_Names_2013_2014.csv”

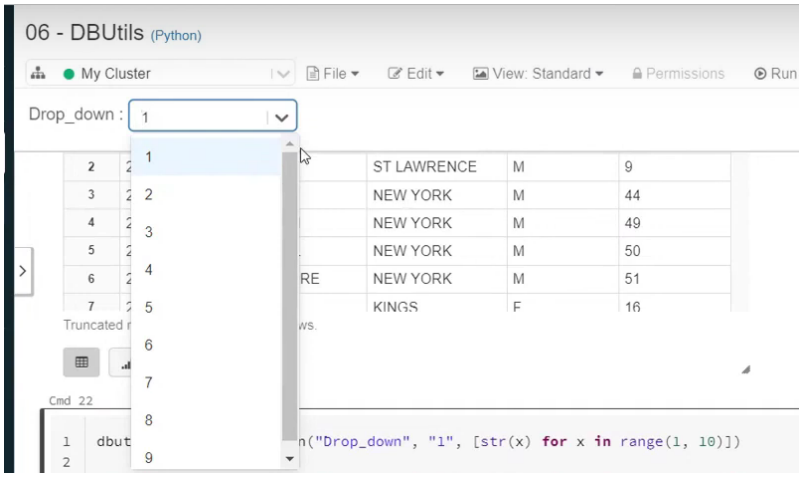
If we execute this command in master notebook 06-DBUtils, below output will be shown.



4) dropdown command:

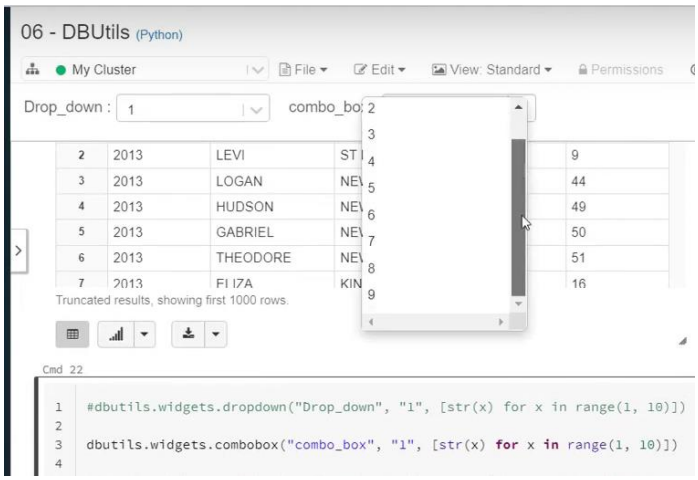
dbutils.widgets.dropdown(“Drop_down”,”1”,[str(x) for x in range(1,10)])

Explanation : we are creating dropdown wizard by giving name Drop_down and giving value 1 that is selected, and it will create 1 to 10 values . using this program, I am creating 10 values for this drop down the list of values . once we execute above command, drop down wizard will be created as shown above.



5) combobox command:

dbutils.widgets.combobox("combo_box","1",[str(x) for x in range(1,10)])

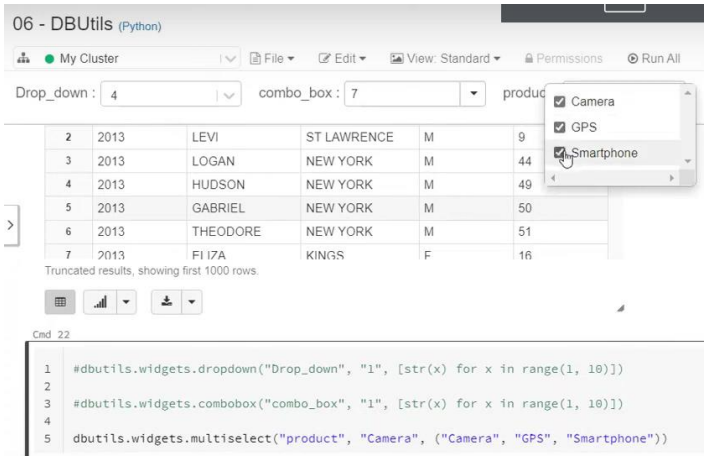


6) multiselect command:

dropdown and combobox we can select only 1 value at a time but if we want to select list of values then we can go with multiselect.

dbutils.widgets.multiselect("product","Camera",("Camera","GPS","Smartphone"))

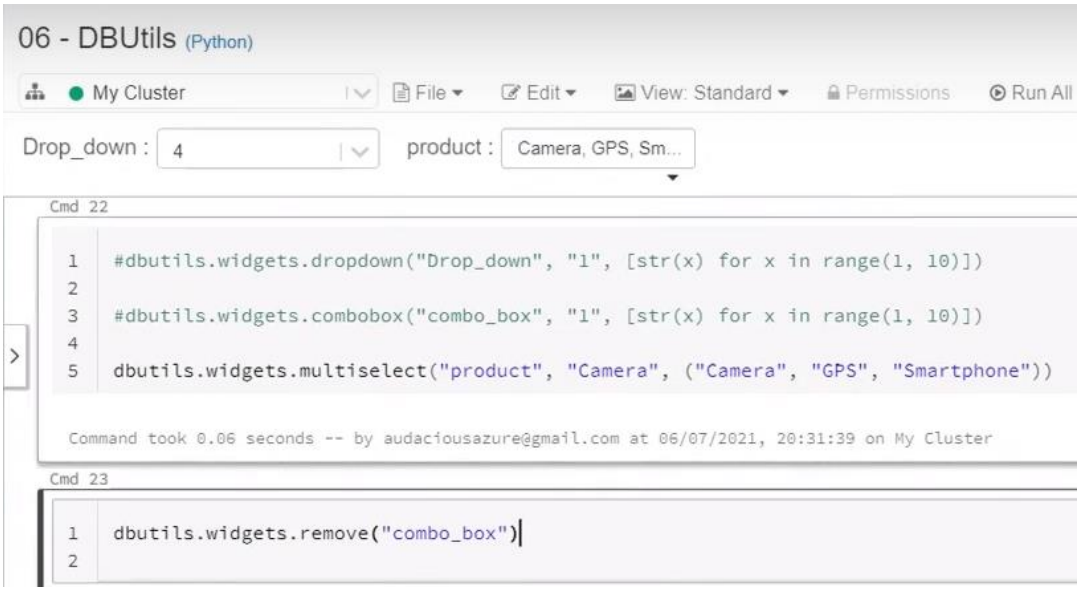
Explanation: name of wizard is product. Product wizard is multiselect box and default value is camera and we have given the list of values camera ,GPS and Smartphone. This is multiselect option.



7) remove command:

If we want to remove wizard 1 particular text box /combobox then we can use command remove.

dbutils.widgets.remove("combo_box") , once we execute this command, combobox will be removed as shown below.

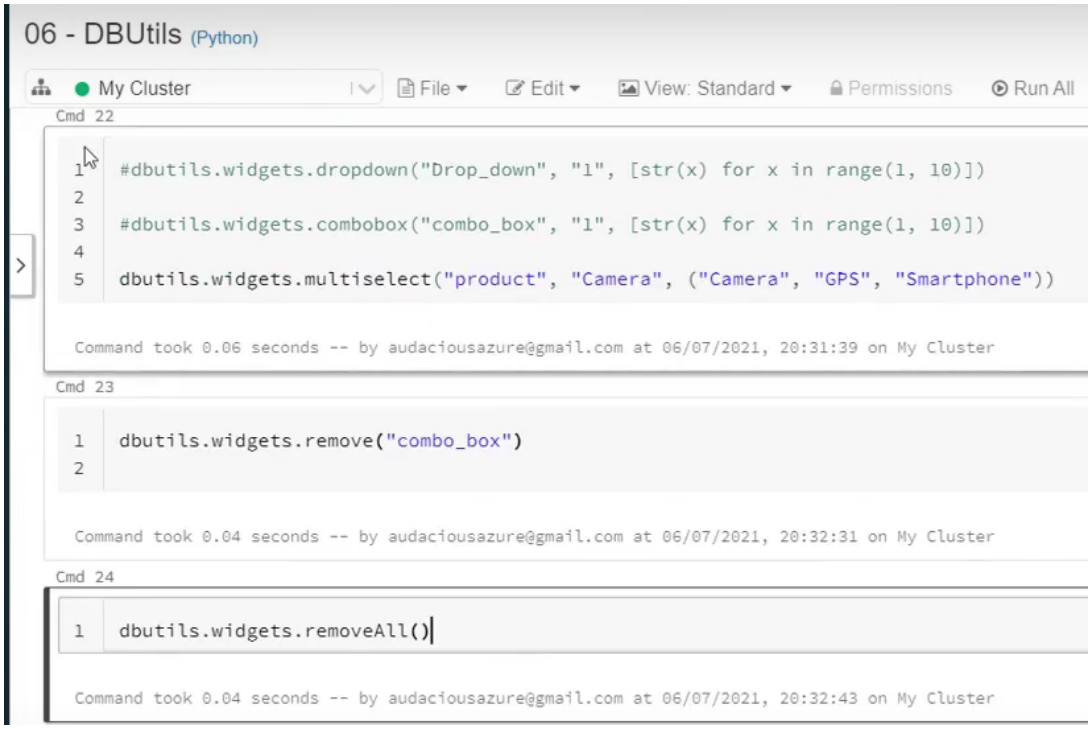


8) removeAll command:

If we want to remove all wizards that we have created, then we can use removeAll function.

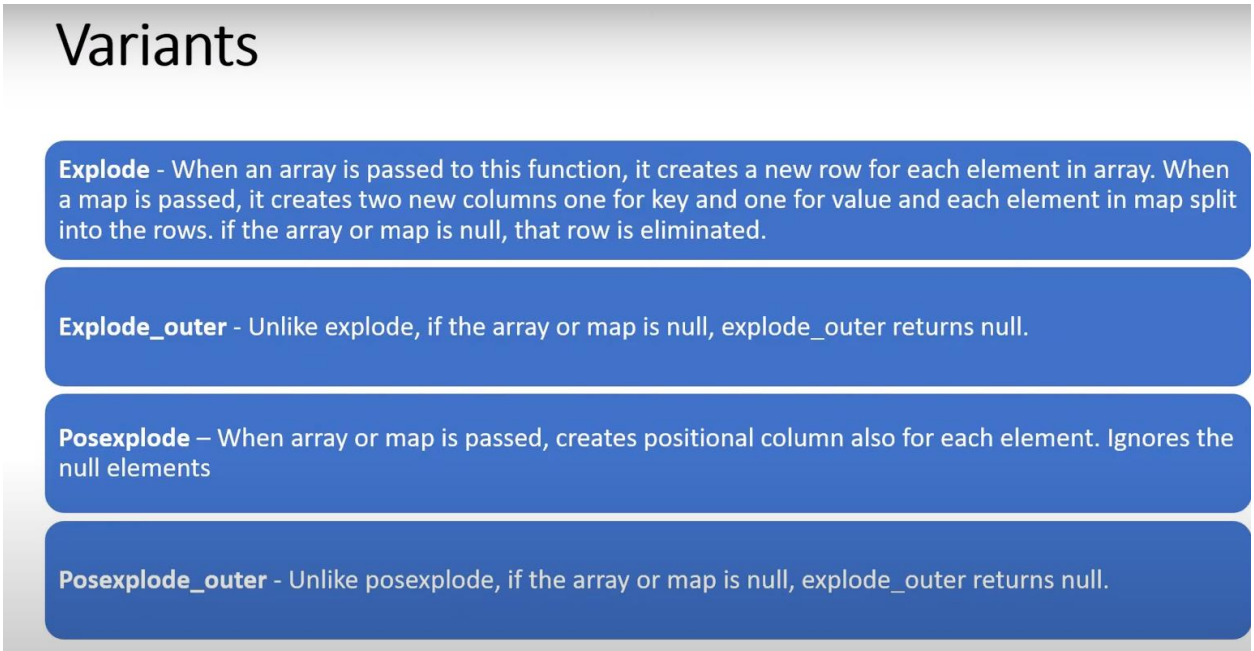
dbutils.widgets.removeAll()

This command will remove all wizards as shown below.



Explode function

Explode function is a pyspark function that returns a new row for each element when we pass array or map. Array would contain list of elements and map would contain list of key value pairs. If we want to split elements in array/map , we can go for explode function. While splitting elements into new row, it will create a new column col by default for array elements. And key and value these would be column names for map.



We can see variants in explode function. We can see 4 variants in explode function. Explode,Explode_outer,Posexplode(positional explode),Posexplode_outer(positional explode outer).

Explode: Explode would create new rows for each elements for array/map type. It will ignore null values .if the array/map is null, then that row is eliminated.

Explode_outer: this will consider null values also. In case array /map value is null then Explode_outer will create a new row for that as well. If we want to include null, then we can go for Explode_outer.

Posexplode: While creating new rows for all the elements for array/map field posexplode also create new column for positions. Position of each element would be captured in position column. Default column name will be pos and posexplode will not consider null values. In case array/map value is null, then it would be dropped.

Posexplode_outer: If we want to consider null value, then we can go for Posexplode_outer.

Eg: **CREATING DATA FRAME WITH ARRAY TYPE**

```
array_appliance = [

    ('Raja',['TV','Refrigerator','Oven','AC']),
    ('Raghav',['AC','Washing Machine',None]),
```

```
    ('Ram',['Grinder','TV']),
    ('Ramesh',['Refrigerator','TV',None]),
    ('Rajesh',None)]

df_app = spark.createDataFrame(array_appliance,['Name','Appliances'])
df_app.printSchema()
display(df_app)
```

(3) Spark Jobs

df_app: pyspark.sql.dataframe.DataFrame = [Name: string, Appliances: array]

```
root
 |-- Name: string (nullable = true)
 |-- Appliances: array (nullable = true)
 |    |-- element: string (containsNull = true)
```

Table +

	Name	Appliances
1	Raja	["TV", "Refrigerator", "Oven", "AC"]
2	Raghav	["AC", "Washing Machine", null]
3	Ram	["Grinder", "TV"]
4	Ramesh	["Refrigerator", "TV", null]
5	Rajesh	null

5 rows | 12.54 seconds runtime

CREATING DATA FRAME WITH MAP TYPE

```
map_brand=[('Raja',{'TV':'LG','Refrigerator':'Samsung','Oven':'Philipps','AC':'Voltas
'}),
           ('Raghav',{'AC':'Samsung','Washing Machine':'LG'}),
           ('Ram',{'Grinder':'Preethi','TV':''}),
           ('Ramesh',{'Refrigerator':'LG','TV':'Croma'}),
           ('Rajesh',None)]
schema = ['Name','Brands']
df_brand =spark.createDataFrame(map_brand,schema)
df_brand.printSchema()
display(df_brand)
```

(3) Spark Jobs

df_brand: pyspark.sql.dataframe.DataFrame = [Name: string, Brands: map]

```
root
 |-- Name: string (nullable = true)
 |-- Brands: map (nullable = true)
 |    |-- key: string
 |    |-- value: string (valueContainsNull = true)
```

Table +

	Name	Brands
1	Raja	{"Refrigerator": "Samsung", "AC": "Voltas", "TV": "LG", "Oven": "Philipps"}
2	Raghav	{"AC": "Samsung", "Washing Machine": "LG"}
3	Ram	{"TV": "", "Grinder": "Preethi"}
4	Ramesh	{"Refrigerator": "LG", "TV": "Croma"}
5	Rajesh	null

5 rows | 1.23 seconds runtime

Explode Array Field

In order to perform explode function, we have to import from the library.

```
from pyspark.sql.functions import explode
df1= df_app.select(df_app.Name,explode(df_app.Appliances))
df1.printSchema()
display(df1)
```

```
root
 |-- Name: string (nullable = true)
 |-- col: string (nullable = true)
```

Table +

	Name	col
1	Raja	TV
2	Raja	Refrigerator
3	Raja	Oven
4	Raja	AC
5	Raghav	AC
6	Raghav	Washing Machine
7	Raghav	null

12 rows | 3.01 seconds runtime

Table +

	Name	col
6	Raghav	Washing Machine
7	Raghav	null
8	Ram	Grinder
9	Ram	TV
10	Ramesh	Refrigerator
11	Ramesh	TV
12	Ramesh	null

12 rows | 3.01 seconds runtime

Explode Map Field

```
from pyspark.sql.functions import explode
df2= df_brand.select(df_brand.Name, explode(df_brand.Brands))
df2.printSchema()
display(df2)
```

```
root
 |-- Name: string (nullable = true)
 |-- key: string (nullable = false)
 |-- value: string (nullable = true)
```

Table +

	Name	key	value
1	Raja	Refrigerator	Samsung
2	Raja	AC	Voltas
3	Raja	TV	LG
4	Raja	Oven	Philipps
5	Raghav	AC	Samsung
6	Raghav	Washing Machine	LG
7	Ram	TV	

10 rows | 1.11 seconds runtime

Table +

	Name	key	value
4	Raja	Oven	Philipps
5	Raghav	AC	Samsung
6	Raghav	Washing Machine	LG
7	Ram	TV	
8	Ram	Grinder	Preethi
9	Ramesh	Refrigerator	LG
10	Ramesh	TV	Croma

10 rows | 1.11 seconds runtime

Rajesh data is not included because it has entire data as null. So null value data is ignored.

Explode outer function to consider NULL Values

```
from pyspark.sql.functions import explode_outer
display(df_app.select(df_app.Name,explode_outer(df_app.Appliances)))
display(df_brand.select(df_brand.Name,explode_outer(df_brand.Brands)))
```

Table +

	Name	col
7	Raghav	null
8	Ram	Grinder
9	Ram	TV
10	Ramesh	Refrigerator
11	Ramesh	TV
12	Ramesh	null
13	Rajesh	null

13 rows | 1.64 seconds runtime

Table +

	Name	key	value
5	Raghav	AC	Samsung
6	Raghav	Washing Machine	LG
7	Ram	TV	
8	Ram	Grinder	Preethi
9	Ramesh	Refrigerator	LG
10	Ramesh	TV	Croma
11	Rajesh	null	null

11 rows | 1.64 seconds runtime

Positional Explode:

```
from pyspark.sql.functions import posexplode
display(df_app.select(df_app.Name,posexplode(df_app.Appliances)))
display(df_brand.select(df_brand.Name,posexplode(df_brand.Brands)))
```

Table +

	Name	pos	col
1	Raja	0	TV
2	Raja	1	Refrigerator
3	Raja	2	Oven
4	Raja	3	AC
5	Raghav	0	AC
6	Raghav	1	Washing Machine
7	Raghav	2	null

12 rows | 1.93 seconds runtime

```
root
 |-- Name: string (nullable = true)
 |-- pos: integer (nullable = false)
 |-- col: string (nullable = true)

root
 |-- Name: string (nullable = true)
 |-- pos: integer (nullable = false)
 |-- key: string (nullable = false)
 |-- value: string (nullable = true)
```

Table +

	Name	pos	key	value
1	Raja	0	Refrigerator	Samsung
2	Raja	1	AC	Voltas
3	Raja	2	TV	LG
4	Raja	3	Oven	Philipps
5	Raghav	0	AC	Samsung
6	Raghav	1	Washing Machine	LG
7	Ram	0	TV	

10 rows | 1.93 seconds runtime

Positional explode will contain one more column which will contain position of each element.

Positional Explode with NULL

```
from pyspark.sql.functions import posexplode_outer
df3= df_app.select(df_app.Name,posexplode_outer(df_app.Appliances))
df4=df_brand.select(df_brand.Name,posexplode_outer(df_brand.Brands))
display(df3)
display(df4)
```

	Name	pos	col
7	Raghav	2	null
8	Ram	0	Grinder
9	Ram	1	TV
10	Ramesh	0	Refrigerator
11	Ramesh	1	TV
12	Ramesh	2	null
13	Rajesh	null	null

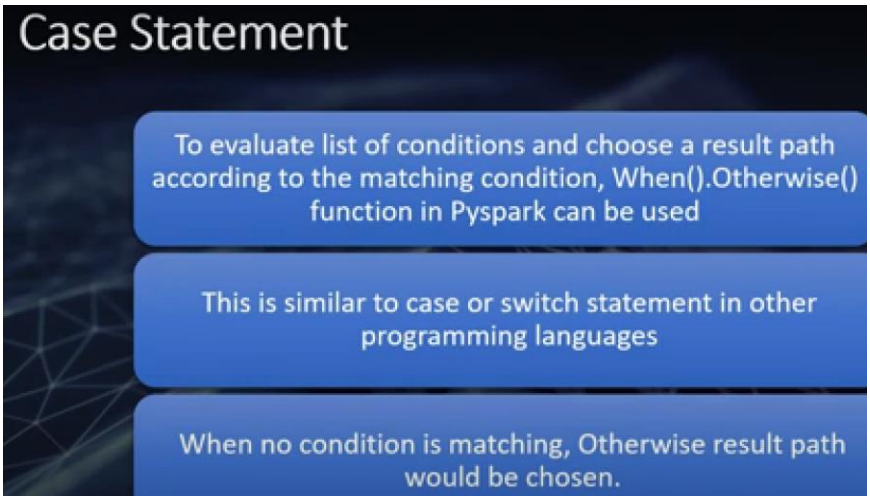
13 rows | 1.72 seconds runtime

Table +

	Name	pos	key	value
5	Raghav	0	AC	Samsung
6	Raghav	1	Washing Machine	LG
7	Ram	0	TV	
8	Ram	1	Grinder	Preethi
9	Ramesh	0	Refrigerator	LG
10	Ramesh	1	TV	Croma
11	Rajesh	null	null	null

11 rows | 1.72 seconds runtime

CASE-ELSE Statement & when otherwise IN PYSPARK:



Case Statement: This function evaluates list of conditions and based on result it will choose one of the result path. If none of the conditions are met, then finally it will go with otherwise path but in case statement it will be called with else statement in SQL languages .

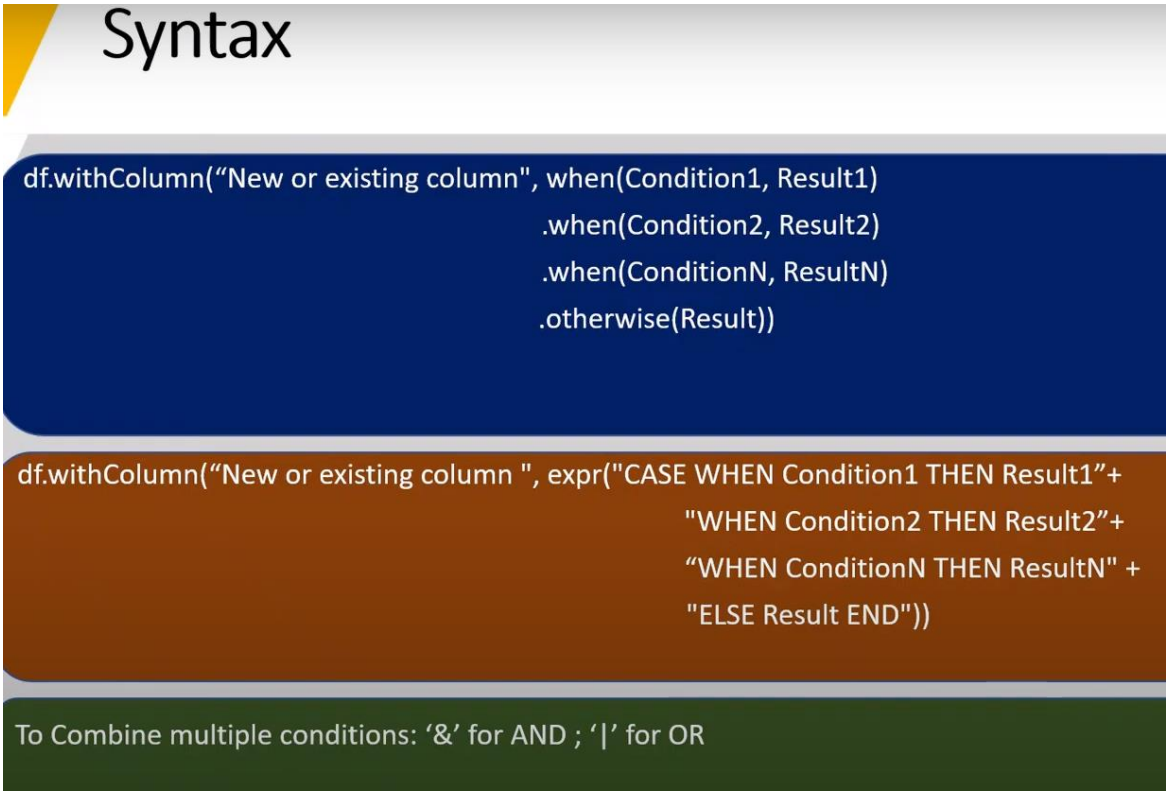
We can write this function in 2 different syntax. 1st is when otherwise and 2nd is case statement(CASE WHEN ELSE).

```
Eg: data_student = [("Raja","Science",80,"P",90),
                    ("Rakesh","Maths",90,"P",70),
                    ("Rama","English",20,"F",80),
                    ("Ramesh","Science",45,"F",75),
                    ("Rajesh","Maths",30,"F",50),
                    ("Raghav","Maths",None,"NA",70)]
schema = ["Name","Subject","Marks","Status","Attendance"]
df= spark.createDataFrame(data_student,schema)
df.printSchema()
display(df)
```

```
root
 |-- Name: string (nullable = true)
 |-- Subject: string (nullable = true)
 |-- Marks: long (nullable = true)
 |-- Status: string (nullable = true)
 |-- Attendance: long (nullable = true)
```

Table +

	Name	Subject	Marks	Status	Attendance
1	Raja	Science	80	P	90
2	Rakesh	Maths	90	P	70
3	Rama	English	20	F	80
4	Ramesh	Science	45	F	75
5	Rajesh	Maths	30	F	50
6	Raghav	Maths	null	NA	70



Updating the existing column using when otherwise column

```
from pyspark.sql.functions import when
df1= df.withColumn("Status",when(df.Marks>=50,"Pass")
                            .when(df.Marks<50,"Fail")
                            .otherwise("Absentee"))
display(df1)
```

(3) Spark Jobs

df1: pyspark.sql.dataframe.DataFrame = [Name: string, Subject: string ... 3 more fields]

Table +

	Name	Subject	Marks	Status	Attendance
1	Raja	Science	80	Pass	90
2	Rakesh	Maths	90	Pass	70
3	Rama	English	20	Fail	80
4	Ramesh	Science	45	Fail	75
5	Rajesh	Maths	30	Fail	50
6	Raghav	Maths	null	Absentee	70

6 rows | 1.71 seconds runtime

Creating new column

```
from pyspark.sql.functions import when
df2= df.withColumn("New_Status",when(df.Marks>=50,"Pass")
                                .when(df.Marks<50,"Fail")
                                .otherwise("Absentee"))
display(df2)
```

(3) Spark Jobs

df2: pyspark.sql.dataframe.DataFrame = [Name: string, Subject: string ... 4 more fields]

Table +

	Name	Subject	Marks	Status	Attendance	New_Status
1	Raja	Science	80	P	90	Pass
2	Rakesh	Maths	90	P	70	Pass
3	Rama	English	20	F	80	Fail
4	Ramesh	Science	45	F	75	Fail
5	Rajesh	Maths	30	F	50	Fail
6	Raghav	Maths	null	NA	70	Absentee

6 rows | 1.21 seconds runtime

Another Syntax Method

```
from pyspark.sql.functions import expr
df3=df.withColumn("New_Status" ,expr("CASE WHEN Marks>=50 THEN 'Pass' "+
                                     "WHEN Marks<50 THEN 'Fail' "+
```

```
                                        "ELSE 'Absentee' END"))
display(df3)
```

▸ (3) Spark Jobs

▸ df3: pyspark.sql.dataframe.DataFrame = [Name: string, Subject: string ... 4 more fields]

Table +

	Name	Subject	Marks	Status	Attendance	New_Status
1	Raja	Science	80	P	90	Pass
2	Rakesh	Maths	90	P	70	Pass
3	Rama	English	20	F	80	Fail
4	Ramesh	Science	45	F	75	Fail
5	Rajesh	Maths	30	F	50	Fail
6	Raghav	Maths	null	NA	70	Absentee

6 rows | 1.51 seconds runtime

MULTI CONDITIONS USING AND and OR Operators

```
from pyspark.sql.functions import when
df4= df.withColumn("Grade", when((df.Marks>=80) & (df.Attendance>=80) ,
"Distinction")
                           .when((df.Marks>=50) & (df.Attendance>=50) , "Good")
                           .otherwise("Average"))
display(df4)
```

▸ (3) Spark Jobs

▸ df4: pyspark.sql.dataframe.DataFrame = [Name: string, Subject: string ... 4 more fields]

Table +

	Name	Subject	Marks	Status	Attendance	Grade
1	Raja	Science	80	P	90	Distinction
2	Rakesh	Maths	90	P	70	Good
3	Rama	English	20	F	80	Average
4	Ramesh	Science	45	F	75	Average
5	Rajesh	Maths	30	F	50	Average
6	Raghav	Maths	null	NA	70	Average

6 rows | 1.23 seconds runtime

```
from pyspark.sql.functions import when
df4= df.withColumn("Grade", when((df.Marks>=80) | (df.Attendance>=80) ,
"Distinction")
                           .when((df.Marks>=50) & (df.Attendance>=50) , "Good")
                           .otherwise("Average"))
display(df4)
```

(3) Spark Jobs

df4: pyspark.sql.dataframe.DataFrame = [Name: string, Subject: string ... 4 more fields]

Table +

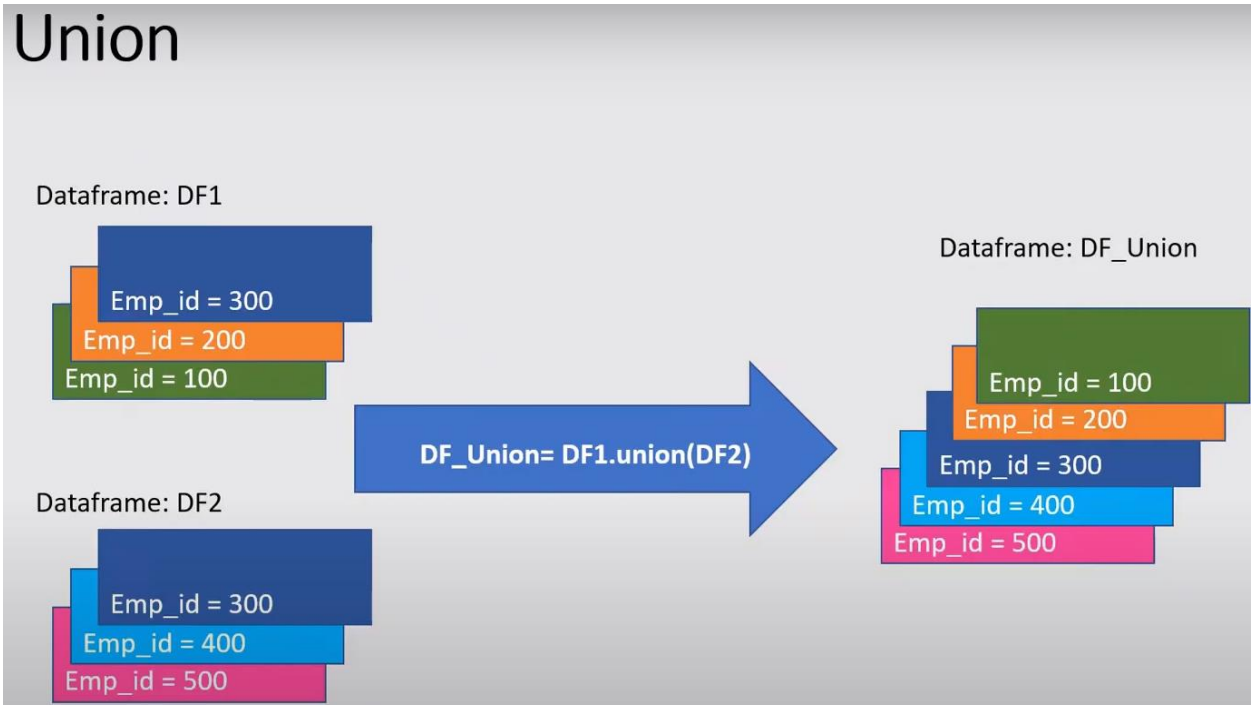
	Name	Subject	Marks	Status	Attendance	Grade
1	Raja	Science	80	P	90	Distinction
2	Rakesh	Maths	90	P	70	Distinction
3	Rama	English	20	F	80	Distinction
4	Ramesh	Science	45	F	75	Average
5	Rajesh	Maths	30	F	50	Average
6	Raghav	Maths	null	NA	70	Average

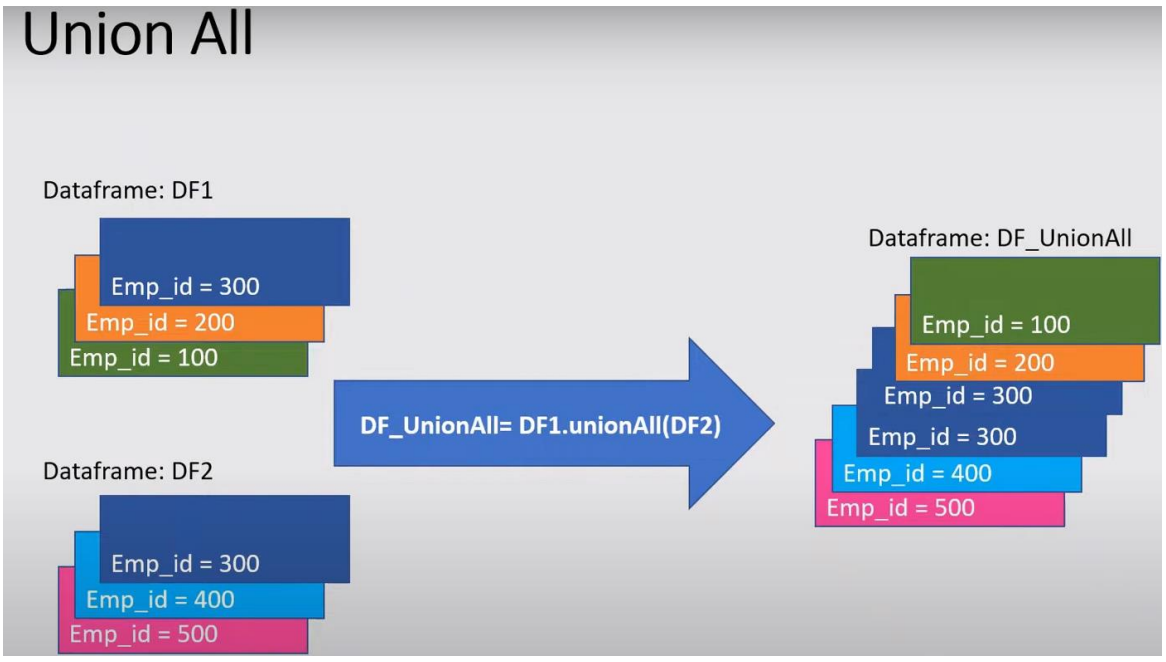
6 rows | 0.96 seconds runtime

Union & UnionAll

Union & UnionAll combine two or more dataframes.Differemce between union and unionAll is union would remove all the duplicate records from resultant data frame until spark version 2.0.0, duplicates can be removed manually by dropDuplicates() but unionAll would retain all the records from the data frame.In order to merge 2 data frames, schema for both data frames should match. If there is any mismatch in schema, then it will through error. Union and unionAll both performs same after spark 2.0.0

This below screenshot is for below spark2.0.0 version. Spark version below 2.0.0 will remove all the duplicates. Now all the records from the dataframe1 will be added to resultant data. Now coming to 2nd data frame emp_id=300 is already added from previous data frame so it would be ignored/eliminated. Only emp_id =400 and emp_id=500 is considered. The resultant data frame is shown below.





UnionAll would consider duplicates also as shown above until spark 2.0.0

How to check spark version

```
from pyspark.sql import SparkSession
spark =SparkSession.builder.master("local").getOrCreate()
print(spark.sparkContext.version)

op: 3.5.0
```

```
employee_data = [(100,"Stephen","1999","100","M",2000),
                 (200,"Philipp","2002","200","M",8000),
                 (300,"John","2010","100","",6000)]
employee_schema=["employee_id","Name","doj","emp_dept_id","gender","salary"]
df1= spark.createDataFrame(employee_data,employee_schema)
display(df1)
```

▸ (3) Spark Jobs

▸ df1: pyspark.sql.dataframe.DataFrame = [employee_id: long, Name: string ... 4 more fields]

Table +

	employee_id	Name	doj	emp_dept_id	gender	salary
1	100	Stephen	1999	100	M	2000
2	200	Philipp	2002	200	M	8000
3	300	John	2010	100		6000

3 rows | 1.14 seconds runtime

```
employee_data = [(300,"John","2010","100","",6000),
                 (400,"Nancy","2008","400","F",1000),
                 (500,"Rosy","2014","500","M",5000)]
employee_schema=["employee_id","Name","doj","emp_dept_id","gender","salary"]
df2= spark.createDataFrame(employee_data,employee_schema)
display(df2)
```

▸ (3) Spark Jobs

▸ df2: pyspark.sql.dataframe.DataFrame = [employee_id: long, Name: string ... 4 more fields]

Table +

	employee_id	Name	doj	emp_dept_id	gender	salary
1	300	John	2010	100		6000
2	400	Nancy	2008	400	F	1000
3	500	Rosy	2014	500	M	5000

3 rows | 1.11 seconds runtime

```
df_union = df1.union(df2)
display(df_union)
```

After spark 2.0.0 version, union and unionAll performing same , both are containing duplicates.

(3) Spark Jobs

Table +

	employee_id	Name	doj	emp_dept_id	gender	salary
1	100	Stephen	1999	100	M	2000
2	200	Philipp	2002	200	M	8000
3	300	John	2010	100		6000
4	300	John	2010	100		6000
5	400	Nancy	2008	400	F	1000
6	500	Rosy	2014	500	M	5000

6 rows | 1.50 seconds runtime

In order to remove duplicates , we will consider dropDuplicates

```
display(df_union.dropDuplicates())
```

(2) Spark Jobs

Table +

	employee_id	Name	doj	emp_dept_id	gender	salary
1	100	Stephen	1999	100	M	2000
2	200	Philipp	2002	200	M	8000
3	300	John	2010	100		6000
4	400	Nancy	2008	400	F	1000
5	500	Rosy	2014	500	M	5000

5 rows | 3.52 seconds runtime

```
df_unionAll =df1.unionAll(df2)
display(df_unionAll)
```

(3) Spark Jobs

df_unionAll: pyspark.sql.dataframe.DataFrame = [employee_id: long, Name: string ... 4 more fields]

Table +

	employee_id	Name	doj	emp_dept_id	gender	salary
1	100	Stephen	1999	100	M	2000
2	200	Philipp	2002	200	M	8000
3	300	John	2010	100		6000
4	300	John	2010	100		6000
5	400	Nancy	2008	400	F	1000
6	500	Rosy	2014	500	M	5000

6 rows | 1.32 seconds runtime

Example for Schema Mismatch case:

```
df2_cut=df2.select(df2.employee_id,df2.Name,df2.doj,df2.emp_dept_id,df2.gender)
df4= df1.union(df2_cut)
```

once we run above command, there is exception as NUM_COLUMNS_MISMATCH because columns mismatch happened in between 2 dataframes(1 df as 6 columns and 2nd df as 5 columns)

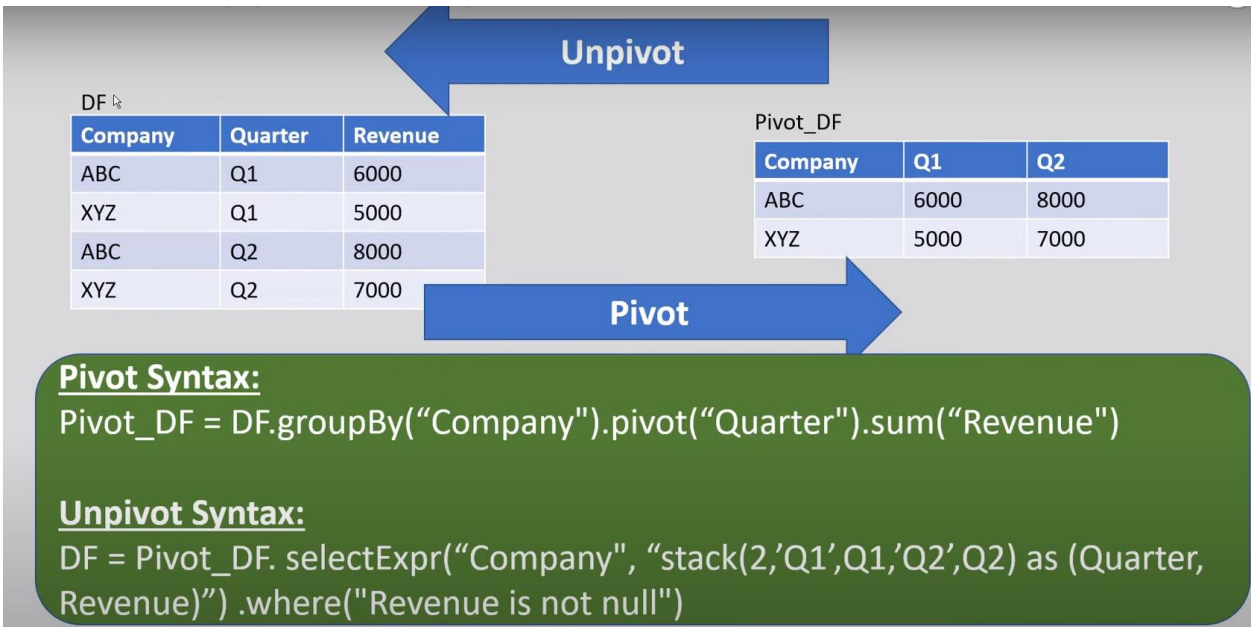
```
AnalysisException: [NUM_COLUMNS_MISMATCH] UNION can only be performed on inputs with the same number of columns, but the first input has 6 columns and the second input has 5 columns. SQLSTATE: 42826;
'Union false, false
:- LogicalRDD [employee_id#205L, Name#206, doj#207, emp_dept_id#208, gender#209, salary#210L], false
+- Project [employee_id#230L, Name#231, doj#232, emp_dept_id#233, gender#234]
   +- LogicalRDD [employee_id#230L, Name#231, doj#232, emp_dept_id#233, gender#234, salary#235L], false
Command took 1.16 seconds -- by paladugusravanthi005@gmail.com at 12/6/2023, 4:43:44 PM on Tra
```

PIVOT AND UNPIVOT IN PYSPARK

These functions are used for data analytics. While doing certain data analysis these functions are unavoidable.

Pivot used to transpose all the list of values from a column into columns .let's say there is a data frame containing millions of records and holding value for employee. Let's say there is a column gender. Even though it contains millions of records still gender column will contain values of male/female and instead of keeping these values in the records , we can convert those values as columns , we can keep 1 column as male column and 2nd column as female column.

Unpivot is quite opposite to pivot. Transposing columns in to list of values to a column.



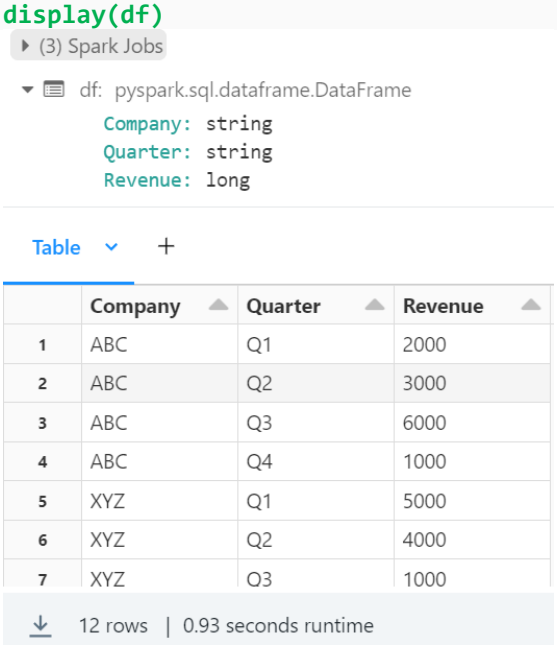
Explanation of Unpivot syntax: there will be no unpivot keyword in unpivot syntax.

In selectExpr we want to retain Company column as it is from original data frame, so mentioned Company. In the next parameter we need to pass it will start with keyword stack and we need to give how many columns you want to convert as a data , in this case we want to convert 2 columns as a data Q1 and Q2, so we need to mention that value as 2. And which column name i.e., Q1 and which value we want to give as a data to the resultant data frame so that is the reason we are passing Q1 and Q1. Similarly, we need to give next column Q2 and its value Q2 in resultant data frame.In order to capture these columns now we have defined which column we want to convert as a row /data, what would be column name for the data that we have to define, we defined as Quarter and Revenue which means it will produce one column as a quarter and it will keep all values from these column names(Q1,Q2) and it will produce one more column revenue and that would populate the values from the matrix value(pivot df) and if we want to avoid producing null records then we can give it in where clause.

Eg: **CREATING SAMPLE DATA FRAME**

```
data = [("ABC","Q1",2000),
        ("ABC","Q2",3000),
        ("ABC","Q3",6000),
        ("ABC","Q4",1000),
        ("XYZ","Q1",5000),
        ("XYZ","Q2",4000),
        ("XYZ","Q3",1000),
        ("XYZ","Q4",2000),
        ("KLM","Q1",2000),
        ("KLM","Q2",3000),
        ("KLM","Q3",1000),
        ("KLM","Q4",5000)]
```

```
schema =["Company","Quarter","Revenue"]
df= spark.createDataFrame(data,schema)
display(df)
```

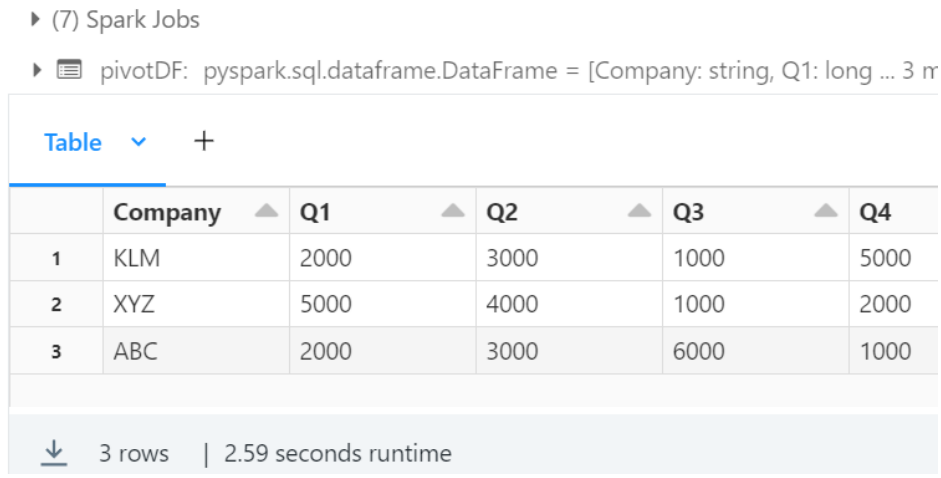


	Company	Quarter	Revenue
1	ABC	Q1	2000
2	ABC	Q2	3000
3	ABC	Q3	6000
4	ABC	Q4	1000
5	XYZ	Q1	5000
6	XYZ	Q2	4000
7	XYZ	Q3	1000

12 rows | 0.93 seconds runtime

PIVOTING A DATA FRAME:

```
pivotDF= df.groupBy("Company").pivot("Quarter").sum("Revenue")
display(pivotDF)
```



	Company	Q1	Q2	Q3	Q4
1	KLM	2000	3000	1000	5000
2	XYZ	5000	4000	1000	2000
3	ABC	2000	3000	6000	1000

3 rows | 2.59 seconds runtime

UNPIVOTING DATA FRAME:

```
unpivotDF = pivotDF.selectExpr("Company","Stack(4,'Q1',Q1,'Q2',Q2,'Q3',Q3,'Q4',Q4) as
(Quarter,Revenue)")
display(unpivotDF)
```

(3) Spark Jobs

unpivotDF: pyspark.sql.dataframe.DataFrame = [Company: string, Quarter: string ... 1 more field]

Table +

	Company	Quarter	Revenue
1	KLM	Q1	2000
2	KLM	Q2	3000
3	KLM	Q3	1000
4	KLM	Q4	5000
5	XYZ	Q1	5000
6	XYZ	Q2	4000
7	XYZ	Q3	1000

12 rows | 1.10 seconds runtime

READ JSON FILE AND FLATTEN JSON

Syntax:

```
df = spark.read

.option("multiline", “true”)

.json(“file_path")
```

Json file :

The below json file is having root : employee . within that first hierarchy starts with emp_id , designation, and attribute. Attribute initiates another hierarchy . within that hierarchy we have Parent_id, status_flag and Department. Department initiates another hierarchy that contains Dept_id, Code,dept_type,dept_flag. This constructs one single record. The same format we can keep n number of records. And also, for each hierarchy we can have multiple values also. For attribute hierarchy we can have different values also parent_id, status_flag and Department has different values. For department hierarchy again we can have different values in it.

Sample Data.json

```
{
"Employee":[
{
 "emp_id": 46066601,
 "Designation": "Manager",
"attribute": [
    {
        "Parent_id": 46555002,
        "status_flag": 1,
        "Department": [
            {
                "Dept_id": 46044403,
                "Code": "ep",
                "dept_type": "MP",
                "dept_flag": 1

            }]
    }]
},
{
    "emp_id": 56555000,
    "Designation": "Supervisor",
    "attribute": [
        {
            "Parent_id": 5605501,
            "status_flag": 1,
            "Department": [
                {
                    "Dept_id": 56044402,
                    "Code": "ep",
                    "dept_type": "P",
                    "dept_flag": 1

                },
                {
                    "Dept_id": 56044000,
                    "Code": "fs",
                    "dept_type": "D",
                    "dept_flag": 0

                }
            ]
        }
    ]
  }
 ]
}
```

Read raw json file:

```
df= spark.read.option("multiline","true").json("/FileStore/tables/json_files/Sample_Data.json")

display(df)
```

▸ (2) Spark Jobs

▸ df: pyspark.sql.dataframe.DataFrame = [Employee: array]

	Employee
1	▸ [{"Designation": "Manager", "attribute": [{"Department": [{"Code": "ep", "Dept_id": 46044403, "dept_flag": 1, "dept_type": "MP"}], "Parent_id": 46555002, "status_flag": 1}], "emp_id": 46066601}, {"Designation": "Supervisor", "attribute": [{"Department": [{"Code": "ep", "Dept_id": 56044402, "dept_flag": 1, "dept_type": "P"}, {"Code": "fs", "Dept_id": 56044000, "dept_flag": 0, "dept_type": "D"}], "Parent_id": 5605501, "status_flag": 1}], "emp_id": 56555000}]

Showing all 1 rows.

Python Function to Flatten Json File: we can use this code in our project also

```
from pyspark.sql.types import *
from pyspark.sql.functions import *
#Flatten array of structs and structs
def flatten(df):
compute Complex Fields (Lists and Structs) in Schema
complex_fields = dict([(field.name, field.dataType)
                        for field in df.schema.fields
                        if type(field.dataType) == ArrayType or type(field.dataType) == StructType])
 while len(complex_fields)!=0:
col_name=list(complex_fields.keys())[0]
print ("Processing :"+col_name+" Type : "+str(type(complex_fields[col_name])))


if StructType then convert all sub element to columns. i.e., flatten structs

if (type(complex_fields[col_name]) == StructType):

expanded = [col(col_name+'.'+k).alias(col_name+'_'+k) for k in [ n.name for n in
complex_fields[col_name]]]

df=df.select("*", *expanded).drop(col_name)

if ArrayType then add the Array Elements as Rows using the explode function i.e. explode Arrays
elif (type(complex_fields[col_name]) == ArrayType):
df=df.withColumn(col_name,explode_outer(col_name))

recompute remaining Complex Fields in Schema

complex_fields = dict([(field.name, field.dataType)

                        for field in df.schema.fields

                        if type(field.dataType) == ArrayType or type(field.dataType) == StructType])

  return df


Apply Flattening Function and Display flattened data

df_flatten = flatten(df)

display(df_flatten)
```

	Employee_Designation	Employee_emp_id	Employee_attribute_Parent_id	Employee_attribute_status_flag	Employee_attribute_Department_Code	Employee_attribute_Department_Dept_id
1	Manager	46066601	46555002	1	ep	46044403
2	Supervisor	56555000	5605501	1	ep	56044402
3	Supervisor	56555000	5605501	1	fs	56044000

BAD RECORDS HANDLING- HOW TO HANDLE CORRUPT RECORDS:

We can handle corrupt records using read option in databricks while reading the databricks. There are 3 methods Permissive , Drop Malformed and Fail Fast.

Permissive method will accept the corrupt record and it will mark corrupt record in a separate column and process will proceed further. There won't be any impact to the entire process.

Drop Malformed : In this method , when there is corrupt record in the input file, it will just ignore the corrupt records, or it will drop the corrupt records and entire process with proceed further. Here also there is no impact to the overall process.

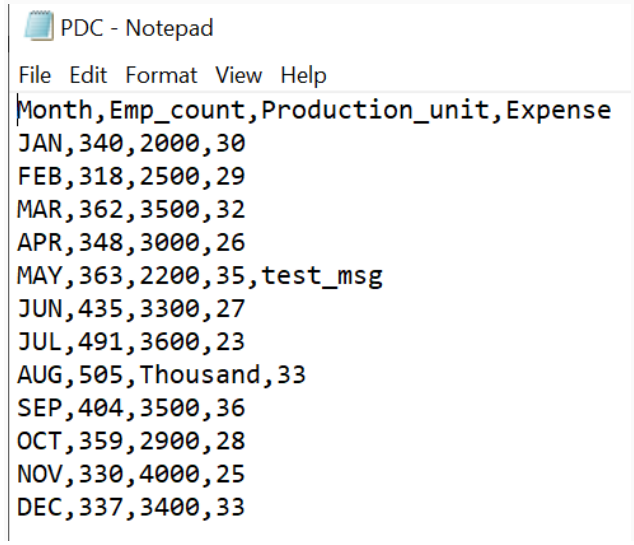
Fail Fast: If there is corrupt record in the input file then entire process will be failed immediately, that is fail fast.

Syntax:

```
df= spark.read.format("csv").option("mode","DROPMALFORMED")
.option("header","true")
.schema(schema)
.load(path)
```

```
df=
spark.read.format("csv").option("header","true").option("inferschema","true").load("/
FileStore/tables/PDC-1.txt")
display(df)
```

The below is the input file



(3) Spark Jobs

df: pyspark.sql.dataframe.DataFrame = [Month: string, Emp_count: integer ... 2

Table +

	Month	Emp_count	Production_unit	Expense
1	JAN	340	2000	30
2	FEB	318	2500	29
3	MAR	362	3500	32
4	APR	348	3000	26
5	MAY	363	2200	35
6	JUN	435	3300	27
7	JUL	491	3600	23

12 rows | 21.99 seconds runtime

	Month	Emp_count	Production_unit	Expense
6	JUN	435	3300	27
7	JUL	491	3600	23
8	AUG	505	Thousand	33
9	SEP	404	3500	36
10	OCT	359	2900	28
11	NOV	330	4000	25
12	DEC	337	3400	33

12 rows | 21.99 seconds runtime

For MAY record , it's not considering last value(test_msg). in input file we have given fifth value also but it is not considered for MAY row record. Coming to row 8 for AUG month it took data as it is because we are inferring schema as it is

```
from pyspark.sql.types import StructField,StructType,IntegerType,StringType
schema = StructType([
    StructField("Month",StringType()),\
    StructField("Emp_count",IntegerType()),\
    StructField("Production_unit",IntegerType()),\
    StructField("Expense",IntegerType()),\
    StructField("corrupt_record",StringType())
])
```

We have added extra column as corrupt_record. This column will indicate us if the column is corrupt or there is a problem with that record.

PERMISSIVE MODE: df1=

```
spark.read.format("csv").option("mode","PERMISSIVE").option("header","true").schema(schema).load("/FileStore/tables/PDC-3.txt")

display(df1)
```

Table +

	Month	Emp_count	Production_unit	Expense	_corrupt_record
1	JAN	340	2000	30	null
2	FEB	318	2500	29	null
3	MAR	362	3500	32	null
4	APR	348	3000	26	null
5	MAY	363	2200	35	MAY,363,2200,35,test_msg
6	JUN	435	3300	27	null
7	JUL	491	3600	23	null
8	AUG	505	null	33	AUG,505,Thousand,33
9	SEP	404	3500	36	null
10	OCT	359	2900	28	null
11	NOV	330	4000	25	null
12	DEC	337	3400	33	null

Here MAY and AUG months are only corrupted so showing only those 2 records as corrupt_records.

DROPMALFORMED MODE:

```
df2=spark.read.format("csv").option("mode","DROPMALFORMED").option("header","true").schema(schema).load("/FileStore/tables/PDC-3.txt")

display(df2)
```

(1) Spark Jobs

df2: pyspark.sql.dataframe.DataFrame = [Month: string, Emp_count: integer ... 3 more fields]

Table +

	Month	Emp_count	Production_unit	Expense	_corrupt_record
1	JAN	340	2000	30	null
2	FEB	318	2500	29	null
3	MAR	362	3500	32	null
4	APR	348	3000	26	null
5	JUN	435	3300	27	null
6	JUL	491	3600	23	null
7	SEP	404	3500	36	null

10 rows | 1.35 seconds runtime

2 corrupted records got removed (among 12 2 records got removed and only 10 records are there in above screenshot).

FAILFAST MODE:

```
df3=
spark.read.format("csv").option("mode","FAILFAST").option("header","true").schema(schema).load("/FileStore/tables/PDC-3.txt")
display(df3)
```

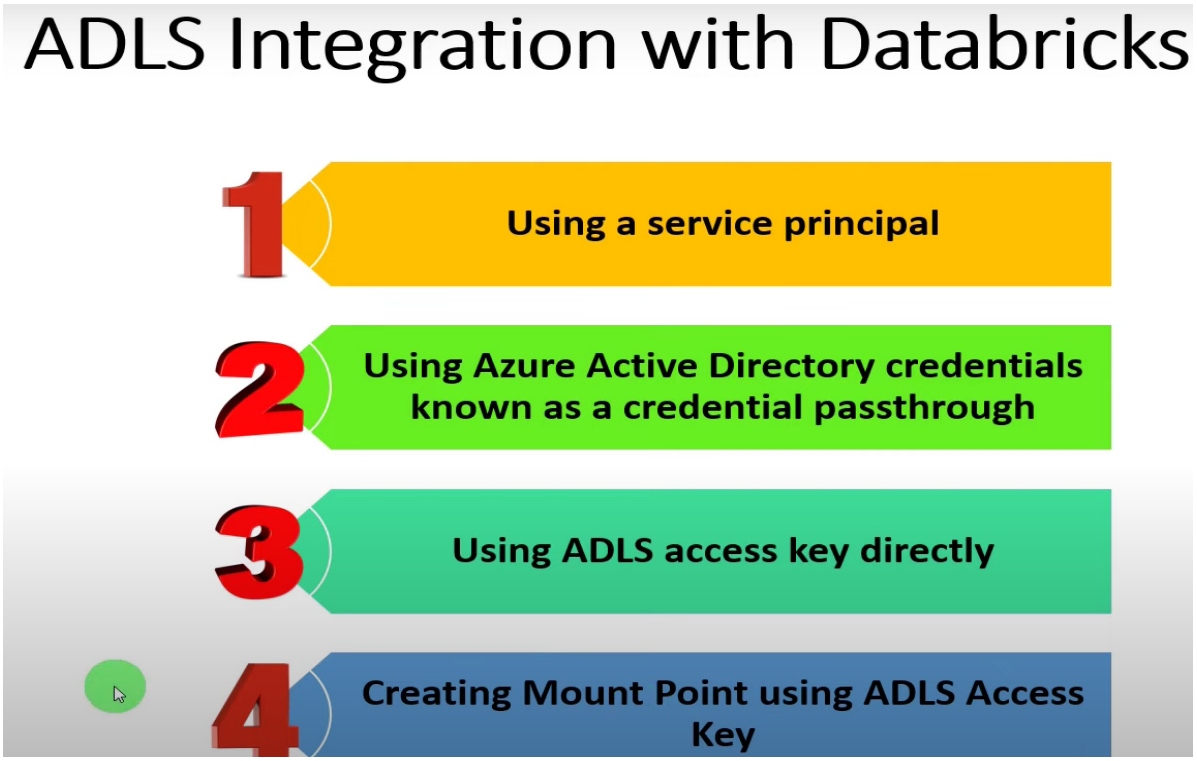
op: This file contains corrupted records, so it got failed.

```
▸ (1) Spark Jobs
⊞FileReadException: Error while reading file dbfs:/FileStore/tables/PDC-3.txt.
Caused by: SparkException: Malformed records are detected in record parsing. Parse Mode: FAILFAST. To process malformed records as null result, try setting the option 'mode' as 'PERMISSIVE'.
Caused by: BadRecordException: org.apache.spark.SparkRuntimeException: [MALFORMED_CSV_RECORD] Malformed CSV record: MAY,363,2200,35,test_msg
Caused by: SparkRuntimeException: [MALFORMED_CSV_RECORD] Malformed CSV record: MAY,363,2200,35,test_msg
Command took 1.67 seconds -- by paladugusravanthi005@gmail.com at 12/7/2023, 4:23:59 PM on Tre
```

HOW TO INTEGRATE AZURE DATA LAKE STORAGE WITH AZURE DATA BRICKS:

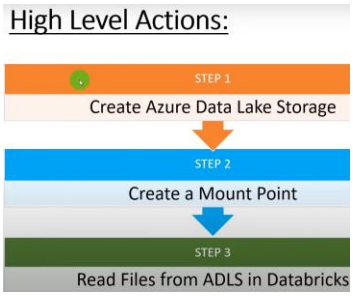
Data bricks can integrate with azure data lake storage for reading and writing files .There are several ways to integrate azure data lake storage with databricks.

- We can integrate azure data lake storage with data bricks using service principal.service principal and oauthentication can be used in this method.
- If we know azure active directory credentials, we can use that also to connect azure data lake storage with data bricks. This method is called credential pass through.
- If we know access key of azure data lake storage , we can connect directly also
- We can create mount point using ADLS access key



Steps to mount ADLS in Data bricks using Mount Point:

Creating ADLS in azure environment. Then create mount point in adb using pyspark code. Once mount point is created, data bricks can integrate ,can read, and write files from ADLS,3rd step is it can read files from ADLS



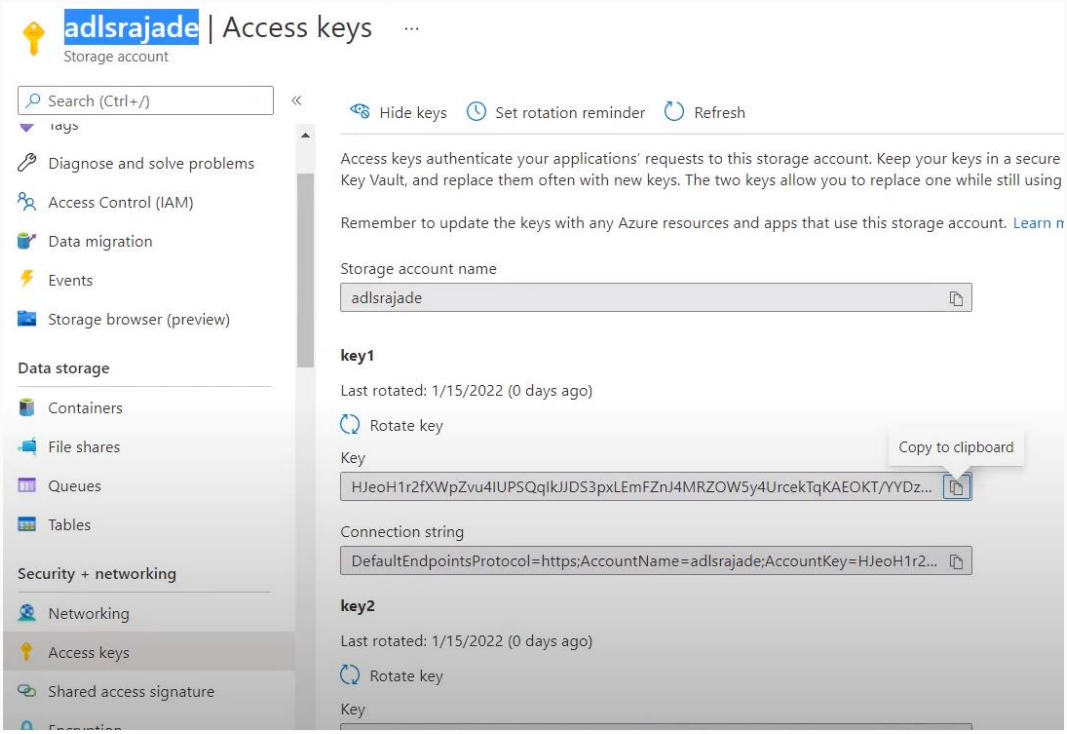
Method-1: ADLS Integration with Databricks using access keys directly

We are going to use access keys directly, for this we have to set spark configuration. To set up spark configuration, we have to pass 2 parameters ,in the first parameter we have to pass storage account

Syntax : `spark.conf.set("fs.azure.account.key.<storage account>.dfs.core.windows.net","<access key>")`

```
spark.conf.set("fs.azure.account.key.adlsrajade.dfs.core.windows.net","<access key>")
```

access key we will get from Storage Account→ Access Keys → Show keys → go to key1 → copy Key as shown below and paste it in access keys section in spark.conf.set command and execute above command.



Now spark configuration is set. Now we can access ADLS with data bricks.

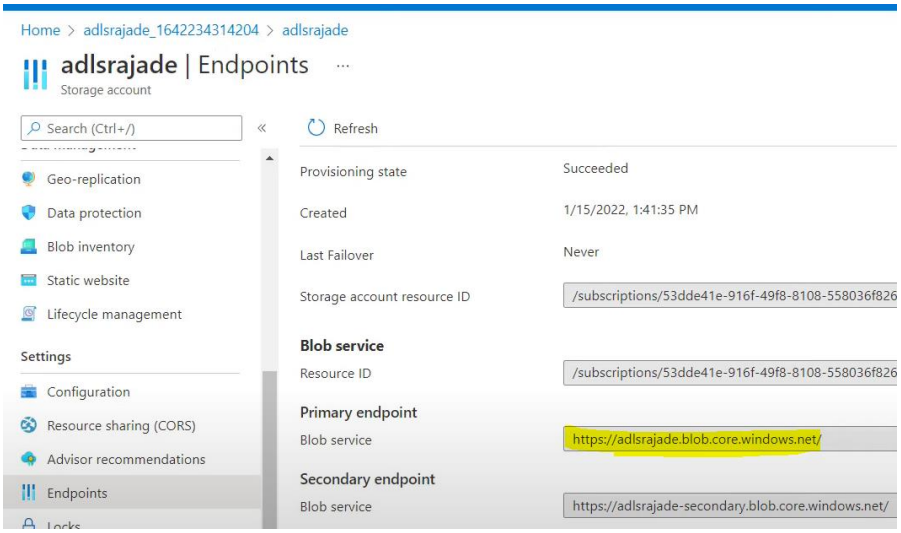
Now if we want to list down all files present in the container, we will use below command,

```
dbutils.fs.ls("abfss://<container_name>@<storage_name>.dfs.core.windows.net/")

dbutils.fs.ls("abfss://container-rajade@adlsrajade.dfs.core.windows.net/")

#set the data lake file location:
file_location= "abfss://container-rajade@adlsrajade.dfs.core.windows.net/"

#Read csv file into data frame df
df =
spark.read.format("csv").option("header","true").option("inferSchema","true").option("delimiter",",").load(file_location)

#display data frame
display(df)
```

Method-2: Creating Mount Point using ADLS Access Key

If we are mounting/integrating ADLS with ADB, adb will treat files as local files only even adls files are not local files if we mount/integrate it.

In below command, in source parameter we can give as adls storage account path/blob storage account path from storage account → endpoints → Blob service link that we need to pass in source as shown below.



Mount_point starts with /mnt then we need to provide particular standard name to this container and in extra configs we need to enter same thing what we did in spark.conf.set command.

Syntax: dbutils.fs.mount(

source= “wasbs://container-rajade@adlsrajade.blob.core.windows.net”,

mount_point = ”/mnt/adls_test”,

extra_configs={“fs.azure.account.key.<storageaccount>.blob.core.windows.net”:”<access key>”})

source= “wasbs://container-rajade@adlsrajade.blob.core.windows.net”,

mount_point = ”/mnt/adls_test”,

extra_configs={“fs.azure.account.key.adlsrajade.blob.core.windows.net”:”<access key>”})

output Out[1]: True i.e.; mount point got created successfully.

---

dbutils.fs.ls(“/mnt/adls_test”)

```
Out[2]: [FileInfo(path='dbfs:/mnt/adls_test/world_population_data.csv', name='world_population_data.csv', size=7138)]
```

---

while working on real time projects, generally we used to create multiple mount points in order to connect with several storage accounts and containers. If we want to see all mount points created for our workspace, we need to use dbutils.fs.mounts() command

dbutils.fs.mounts()

```
Out[2]: [MountInfo(mountPoint='/databricks-datasets', source='databricks-datasets', encryptionType='sse-s3'),
 MountInfo(mountPoint='/databricks/mlflow-tracking', source='databricks/mlflow-tracking', encryptionType='sse-s3'),
 MountInfo(mountPoint='/databricks-results', source='databricks-results', encryptionType='sse-s3'),
 MountInfo(mountPoint='/mnt/adls_test', source='wasbs://container-rajade@adlsrajade.blob.core.windows.net', encryptionType=''),
 MountInfo(mountPoint='/databricks/mlflow-registry', source='databricks/mlflow-registry', encryptionType='sse-s3'),
 MountInfo(mountPoint='/', source='DatabricksRoot', encryptionType='sse-s3')]
Command took 0.20 seconds -- by audaciousazure@gmail.com at 1/15/2022, 2:01:47 PM on test
```

If we want to remove mount points, then we can use

dbutils.fs.unmount(“/mnt/adls_test”)

Out[3]: True

1 spark jobs

/mnt/adls_test has been unmounted.

HOW TO READ TABLES FROM AZURE SQL DATABASE IN DATA BRICKS USING PYSPARK:

First, we will create Azure SQL Database. After creating SQL Database, if we get this ip address is not allowed when we enter username and password in query editor, we need to go to server name → go to firewall settings, we will enable ip address there allow azure resources and services to access this server → we need to enable .and also, we need to create rule name and start ip and end ip with same ip address.

If we want to bring one table from SQL data base to azure data bricks environment and want to create data frame, first we need to build connection string . we need to enter server name , port its default one 1433, and data base name , and its username and

password and jdbc driver value is constant and also, we can create jdbc URL we are supplying .

jdbcHostname = “ss-rajade.database.windows.net”

jdbcPort =1433

jdbcDatabase = “database-rajade”

jdbcUsername = ”rajade”

jdbcPassword =”Test@1234”

jdbcDriver =”com.microsoft.sqlserver.jdbc.SQLServerDriver”

jdbcUrl=f”jdbc:sqlserver://{jdbcHostname}:{jdbcPort};databaseName={jdbcDatabase};user={jdbcUsername};password={jdbcPassword}”

In order to read data from azure SQL database , we need to use below command

Syntax:

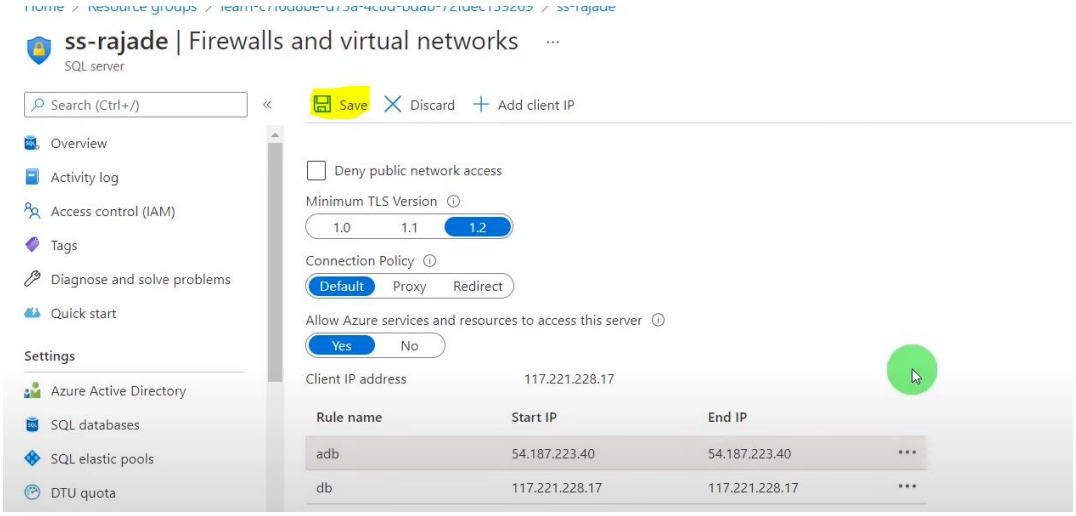
Df= spark.read.format(“jdbc”).option(“url”,jdbcUrl).option(“dbtable”,””).load()

Df= spark.read.format(“jdbc”).option(“url”,jdbcUrl).option(“dbtable”,”SalesLT.Product”).load() →want to read product table from SQL database so mentioned product table here.

Display(Df)

If we execute above command , error can get client with IP address is not allowed to access the server.

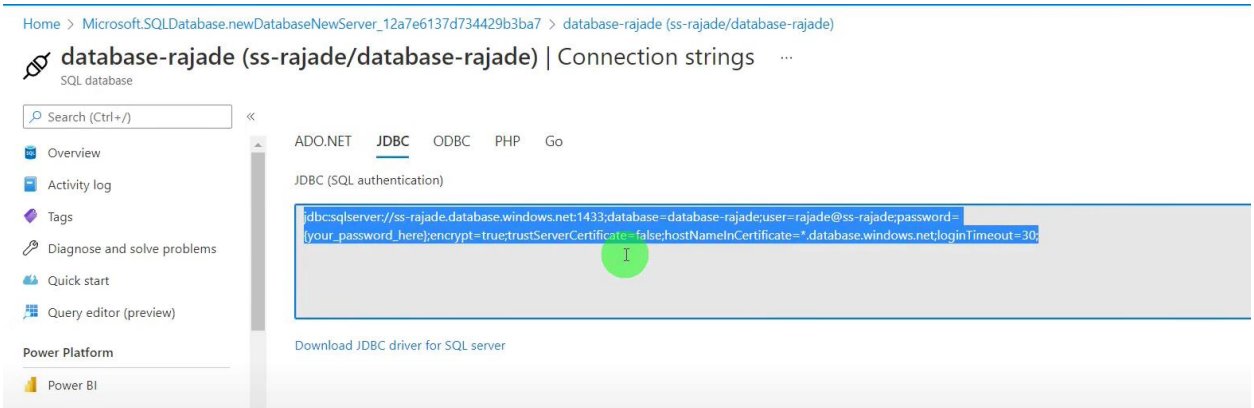
So we need to add data bricks ip address(54.187.223.40) also in SQL server firewall setting.



2nd method of accessing SQL data base to adb:

Instead of building jdbcUrl , if we are going to get connection string from administrator directly, we can use simple method also.

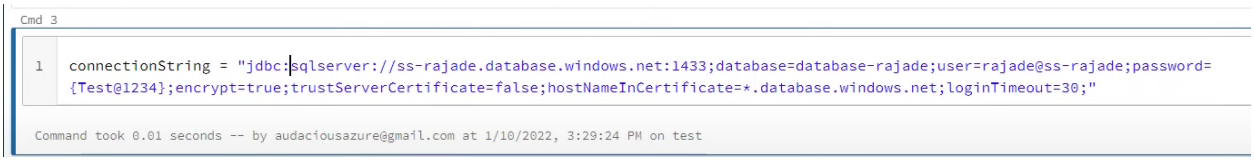
Go to SQL data base → show database connection strings → go to jdbc tab and copy jdbc connection string as shown below.



Now in adb , we will write command as below. Copied jdbc connection string from sql data base we will paste in below in adb.

connectionString = “”

Here from connection string password is not shown so, we need to type our password here as shown below.



Df= spark.read.jdbc(connectionString,””) → need to enter table name here what table we want to read from SQL database.

Df= spark.read.jdbc(connectionString,”SalesLT.Product”)

Display(Df)

Df= spark.read.jdbc(connectionString,”SalesLT.Address”)

Display(Df)

HOW TO BUILD ETL PIPELINE TO LOAD DATA FROM AZURE SQL TO ADLS?

ETL Process: will occur at 3 stages. 1) Extraction would contain fact and dimension tables. We will use JDBC Connection to read data from azure SQL. At the end of read, data frame is created for dimension and fact tables.

2)Transform: we will transform extracted data from azure SQL. In this stage we will apply business rules. Like null values replaced with default values in the dimension tables and going to remove duplicate records from fact table, then we will join fact

and dimension tables based on joining key. Once join is done, will do some aggregation to get some meaningful output at the end of transformation stage.

3) Load the transformed data into Azure SQL/ADLS .so, for that we need to create mount point. Once mount point is done, we can load the transformed data in to ADLS in the form of parquet file.

Code:

Step-1: Extract Tables from Azure SQL

#Define JDBC Connection Parameters:

```
jdbcHostname = “ss-rajade.database.windows.net”

jdbcPort =1433

jdbcDatabase = “asql-rajade”

jdbcUsername = ”rajade”

jdbcPassword =”Test@1234”

jdbcDriver =”com.microsoft.sqlserver.jdbc.SQLServerDriver”

jdbcUrl=f”jdbc:sqlserver://{jdbcHostname}:{jdbcPort};databaseName={jdbcDatabase};user
={jdbcUsername};password={jdbcPassword}”
```

#Read Product Table

```
Df_product
=spark.read.format(“jdbc”).option(“url”,jdbcUrl).option(“dbtable”,”SalesLT.Product”).
load()

display(Df_product)
```

op: we got few null values in column Size and Weight

(1) Spark Jobs

	ProductID	Name	ProductNumber	Color	StandardCost	ListPrice	Size	Weight	ProductCategoryID	ProductModelID	SellSt
1	680	HL Road Frame - Black, 58	FR-R92B-58	Black	1059.3100	1431.5000	58	1016.04	18	6	2002-(
2	706	HL Road Frame - Red, 58	FR-R92R-58	Red	1059.3100	1431.5000	58	1016.04	18	6	2002-(
3	707	Sport-100 Helmet, Red	HL-U509-R	Red	13.0863	34.9900	null	null	35	33	2005-(
4	708	Sport-100 Helmet, Black	HL-U509	Black	13.0863	34.9900	null	null	35	33	2005-(
5	709	Mountain Bike Socks, M	SO-B909-M	White	3.3963	9.5000	M	null	27	18	2005-(
6	710	Mountain Bike Socks, L	SO-B909-L	White	3.3963	9.5000	L	null	27	18	2005-(
	711	Sport-100 Helmet, Blue	HL-U509-B	Blue	13.0863	34.9900	null	null	35	33	2005-(

Showing all 295 rows.

Command took 2.04 seconds -- by audaciousazure@gmail.com at 1/17/2022, 1:48:33 PM on My Cluster

#Read Sales Table

```
Df_sales=spark.read.format(“jdbc”).option(“url”,jdbcUrl).option(“dbtable”,”SalesLT.Sa
lesOrderDetail”).load()
```

```
display(Df_sales)
```

Step-2: Transform the data as per business rules

- **Cleansing Dimension Dataframe – Replace Null Values**

```
df_product_cleansed = Df_product.na.fill({"Size":"M","Weight":100})

display(df_product_cleansed)
```

▸ (1) Spark Jobs

	ProductID	Name	ProductNumber	Color	StandardCost	ListPrice	Size	Weight	ProductCategoryID	ProductModelID	SellSt
1	680	HL Road Frame - Black, 58	FR-R92B-58	Black	1059.3100	1431.5000	58	1016.04	18	6	2002-(
2	706	HL Road Frame - Red, 58	FR-R92R-58	Red	1059.3100	1431.5000	58	1016.04	18	6	2002-(
3	707	Sport-100 Helmet, Red	HL-U509-R	Red	13.0863	34.9900	M	100.00	35	33	2005-(
4	708	Sport-100 Helmet, Black	HL-U509	Black	13.0863	34.9900	M	100.00	35	33	2005-(
5	709	Mountain Bike Socks, M	SO-B909-M	White	3.3963	9.5000	M	100.00	27	18	2005-(
6	710	Mountain Bike Socks, L	SO-B909-L	White	3.3963	9.5000	L	100.00	27	18	2005-(
	711	Sport-100 Helmet, Blue	HL-U509-B	Blue	13.0863	34.9900	M	100.00	35	33	2005-(

Showing all 295 rows.

- **Cleansing Fact Dataframe-Drop Duplicate Records**

```
df_sales_cleansed = df_sales.dropDuplicates()

display(df_sales_cleansed)
```

- **Join fact and dimension dataframes**

```
df_join=df_sales_cleansed.join(df_product_cleansed,df_sales_cleansed.ProductID==df_pr
oduct_cleansed.ProductID,"leftouter").select(df_sales_cleansed.ProductID,
                        df_sales_cleansed.UnitPrice,
                        df_sales_cleansed.LineTotal,
                        df_product_cleansed.Name
                        df_product_cleansed.Color,
                        df_product_cleansed.Size,
                        df_product_cleansed.Weight)

display(df_join)
```

```
df_agg=df_join.groupBy(["ProductID","Name","Color","Size","Weight"]).sum("LineTotal")
.withColumnRenamed("sum(LineTotal)","sum_total_sales")

display(df_agg)
```

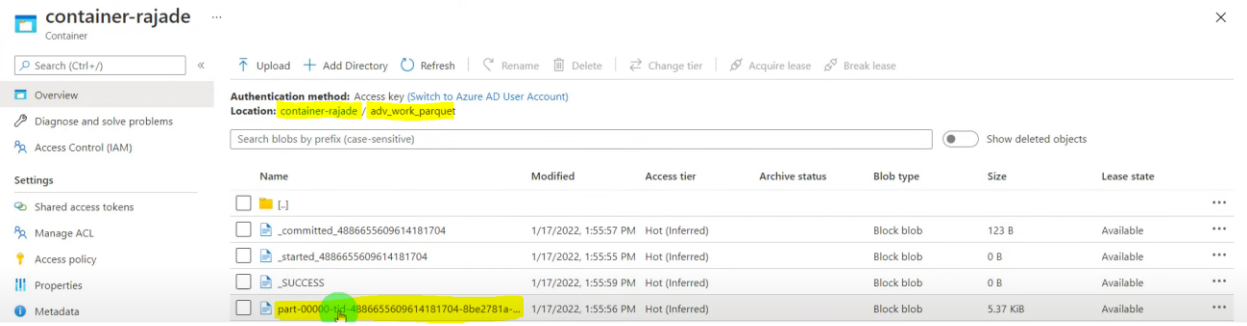
Step3: Load Transformed data into Azure Data Lake Storage

```
#Create Mount Point for ADLS Integration

dbutils.fs.mount(
```

```
source= “wasbs://container-rajade@adlsrajade.blob.core.windows.net”,

mount_point = ”/mnt/adls_test”,

extra_configs={“fs.azure.account.key.adlsrajade.blob.core.windows.net”:”<access key>”})

#List the files under Mount Point

dbutils.fs.ls(“/mnt/adls_test”)

#Write the data in Parquet Format

df_agg.write.format(“parquet”).save(“/mnt/adls_test/adv_work_parquet/”)
```

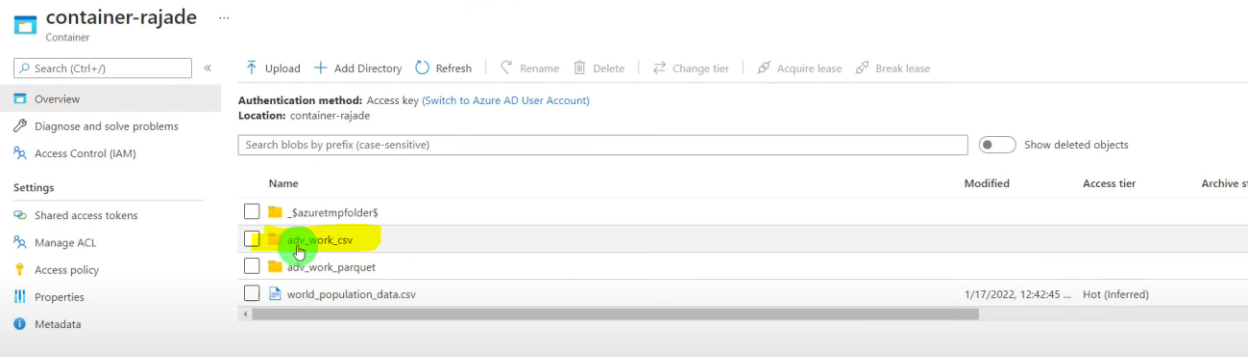
op: 4 spark jobs, file got saved in ADLS container-rajade and adv_work_parquet folder got created.in adv_work_parquet folder, file got created as shown in below screenshot.

Here it has created only 1 part file. Basically, while writing spark default parameter will be 128MB.so if file size is lesser than that, then it will create only 1 partition file, so here that’s why only 1 parquet file got created. We can’t read parquet file data because its not in human readable form. In order to read data in fil, we can write in csv format .




```
#Write the data in csv format

df_agg.write.format(“csv”).option(“header”,”true”).
save(“/mnt/adls_test/adv_work_csv/”)
```

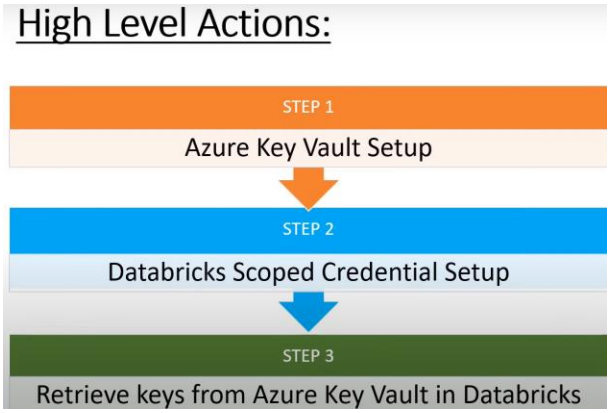
op: 4 spark jobs



#END OF THE PIPELINE

HOW TO INTEGRATE AZURE KEY VAULT WITH AZURE DATA BRICKS:

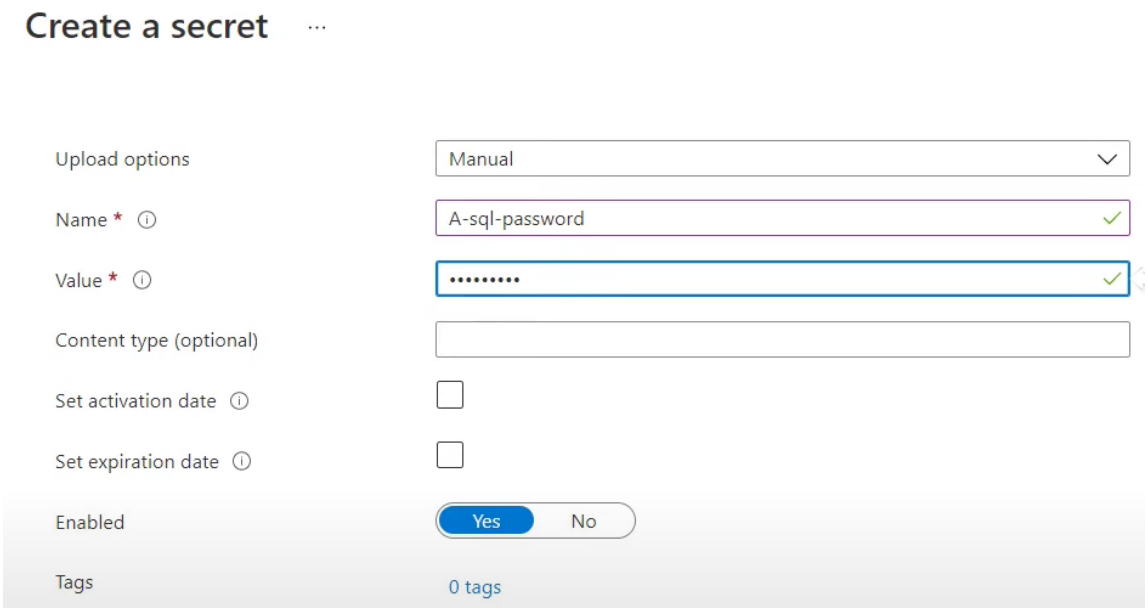
Azure Key Vault is used to secure sensitive information such as passwords, certificates, or any secrets. It's not good practice to hardcore any password in data bricks notebook. In real time projects, it's not good practice to hardcore password in code. so, we can take advantage of azure key vault for our data bricks development. Instead of hardcoring credentials directly, we can save credentials using azure key vault and we can retrieve whenever needed .



- **Azure Key Vault Setup:**
  - **Login to Azure Portal**
  - **Spin the service Azure Key Vault**
  - **Note down DNS name and Resource ID(DNS Name and Resource ID these inputs needed for Databricks Scoped Credential)**
  - **Create a Secret**

We hardcoded the password in above example , we need to create secret for the password value .

First, we will create azure key vault service and then go to azure key vault → settings → secret → Generate/Import secret



Name = A-sql-password is secret name i.e., key

Value: Test@1234 (Here password is Test@1234 that we will mentioned in Value column)

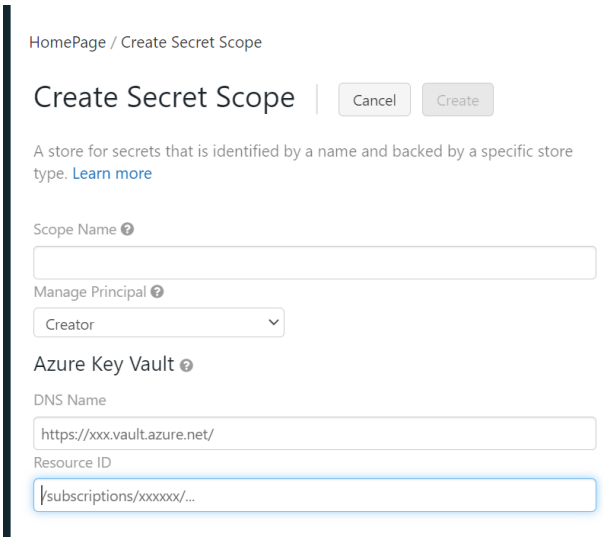
--

Secondly, now we need to create databricks scoped credential

In order to create databricks scoped credential , we need to use URL like below, we need to use our real time project adb URL , on top of ad burl we need to add secrets/createScope

https://community.cloud.databricks.com/?o=2510358043295203#secrets/createScope

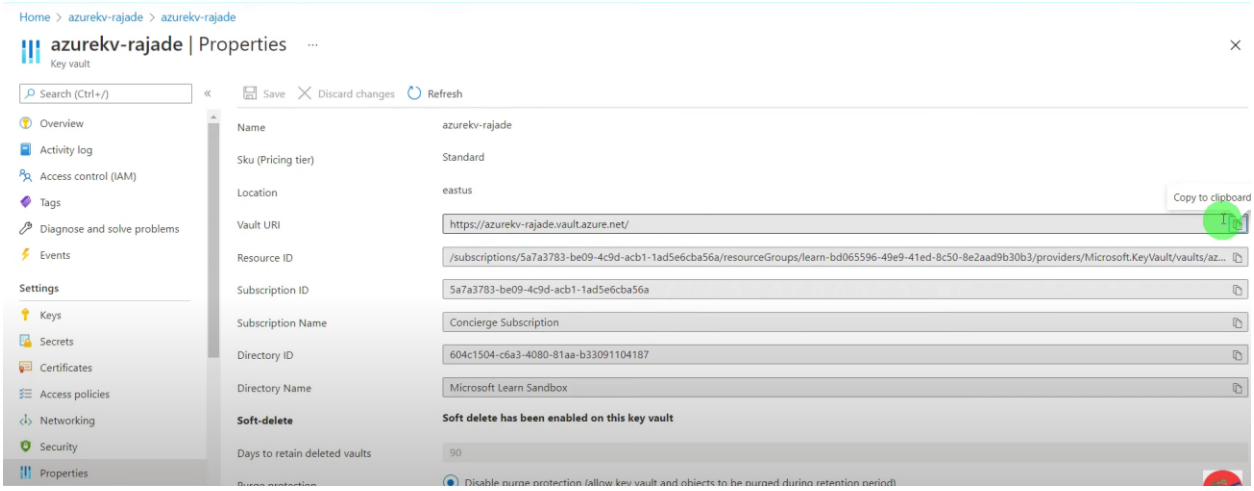
The below is the page where we can create secrets scope for data bricks.



Scope Name: can be anything, in this case we are creating for azure key vault so we will give scope name as azurekv-scope.

DNS Name and Resource ID we will get it from azure key vault portal.

Go to azure key vault → properties → Vault URL is DNS Name. Copy vault URL and Resource ID and paste it in secret scope page and created data base scoped credential



Thirdly, we will retrieve the key using pyspark

```
dbutils.secrets.get(scope = "<scope name>" , key = "<key name>")
```

Below is the code :

```
jdbcHostname = "ss-rajade.database.windows.net"

jdbcPort =1433

jdbcDatabase = "asql-rajade"

jdbcUsername = "rajade"

jdbcPassword = dbutils.secrets.get(scope = "azurekv-scope" , key = "A-sql-password")

jdbcDriver ="com.microsoft.sqlserver.jdbc.SQLServerDriver"

jdbcUrl=f"jdbc:sqlserver://{jdbcHostname}:{jdbcPort};databaseName={jdbcDatabase};user
={jdbcUsername};password={jdbcPassword}"
```

Not only password , we can also retrieve all sensitive information from key vault. For e.g., we don't need to hardcore username, database name, port name, host name, then we can use below code ,

```
jdbcHostname = dbutils.secrets.get(scope = "azurekv-scope" , key = "A-sql-host")

jdbcPort = dbutils.secrets.get(scope = "azurekv-scope" , key = "A-sql-port")

jdbcDatabase = dbutils.secrets.get(scope = "azurekv-scope" , key = "A-sql-dbname")

jdbcUsername = dbutils.secrets.get(scope = "azurekv-scope" , key = "A-sql-User")

jdbcPassword = dbutils.secrets.get(scope = "azurekv-scope" , key = "A-sql-password")

jdbcDriver ="com.microsoft.sqlserver.jdbc.SQLServerDriver"

jdbcUrl=f"jdbc:sqlserver://{jdbcHostname}:{jdbcPort};databaseName={jdbcDatabase};user
={jdbcUsername};password={jdbcPassword}"
```

HOW TO DEVELOP STREAMING APPLICATIONS IN SPARK ENVIRONMENT

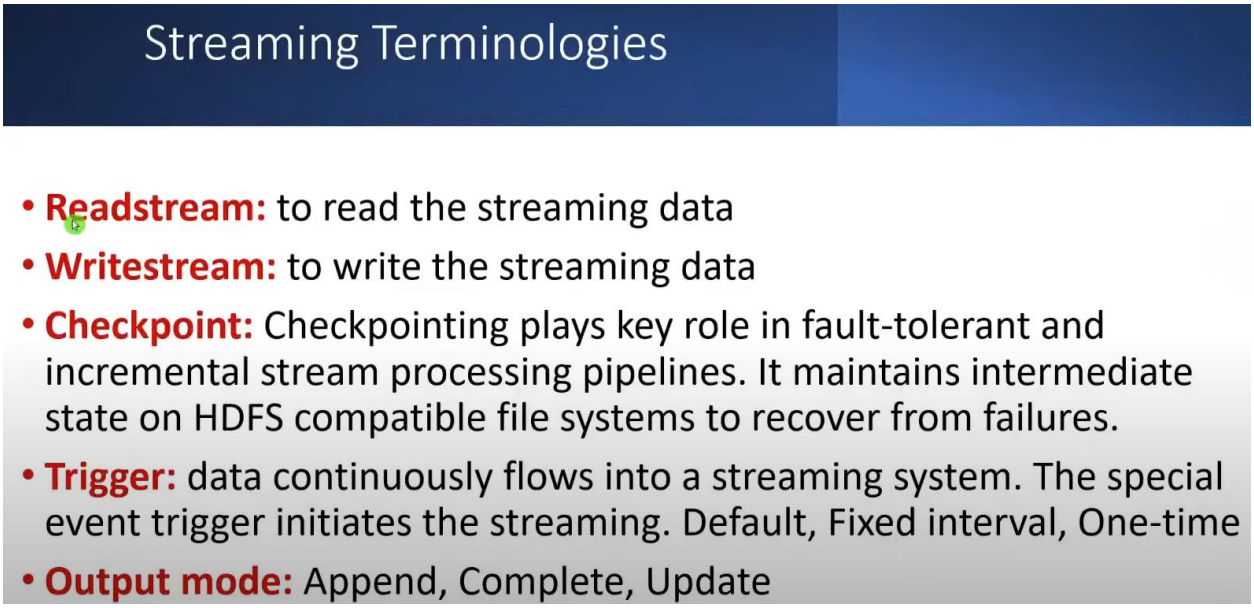
In today's bigdata world, most often we come across situations where we need to deal with real time data and develop streaming applications.

Readstream: Using this function/API , we can consume data from external source systems. Source systems should be data bases or filesystems, or it could be popular streaming servers such as Kafka/Azure event hubs.

Writestream: Once we consumed data from source systems and we have processed, transformed, and applied business aggregations then we can write processed data in to target server using writestream api

Checkpoint: is one of the imp concept in streaming application .checkpoint is used for incremental data loading. If we want to handle incremental data loading, then we need to keep them metadata and track of offset value that got processed in the previous batch.so checkpoint plays key role in that aspect.checkpoint retains offset value that

got generated in the previous run, that would be stored in the checkpoint in the location and when we execute the process next time, spark application first will refer the check point location to retrieve the offset value. Based on offset value it will understand the processed and unprocessed data so it will start picking the unprocessed data and it will process only the incremental data. Another adv of check point is it will help in fault tolerant. In case application gets failed in the middle of the process what happens is when we have to recover this application,application can rerun from the failure point by referring the check point. Checkpoint will give the list of processed and unprocessed data. So, it is very important to create checkpoint location.



Trigger: Trigger is ntg but special event which initiates spark stream application. There are 3 types of trigger, Default,Fixed Interval,One-time . Default trigger is default option.

Default Trigger: Default trigger creates micro patches. Incoming data would be segregated as a micro patch and spark application will start processing one micro patch at a time .once it is completed then it will go to next micro patch. That is default option.

Fixed Interval: We can fix certain time interval, it could be 10sec/1 hr. Based on timing interval , spark streaming application will start processing the incoming data

One time: In case we have to process all the accumulated data in one shot then we can go with one time.

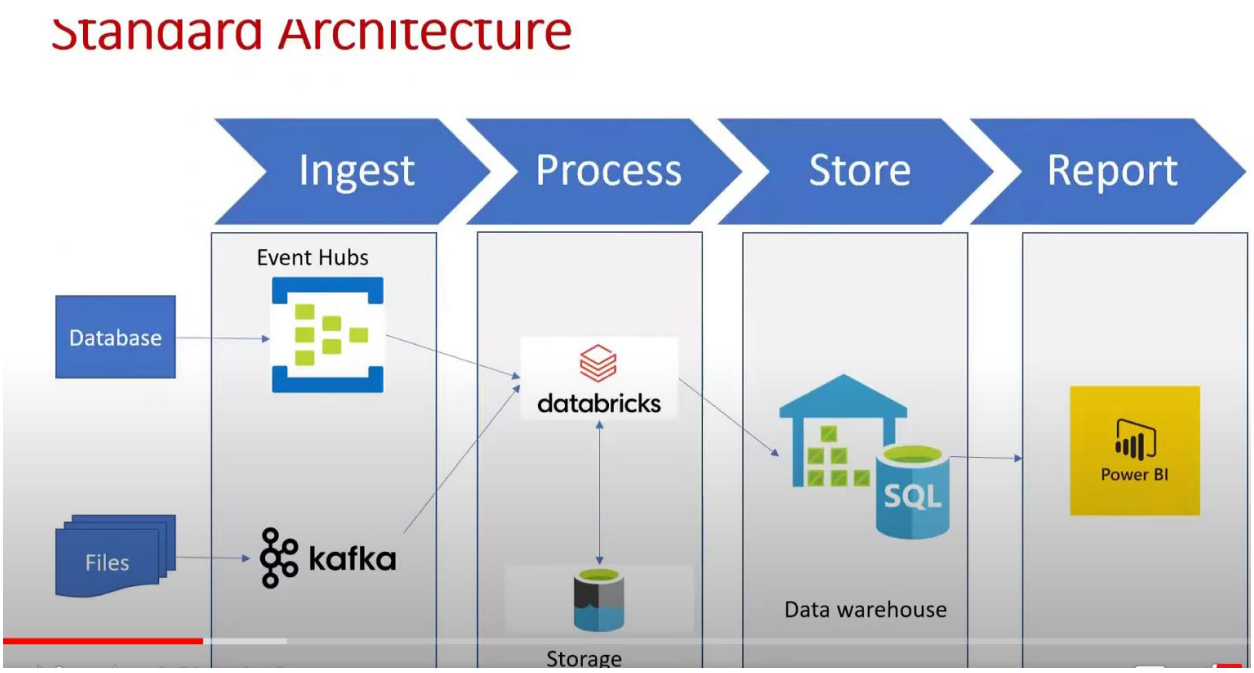
Output modes: There are 3 output modes in spark streaming applications. Append, Complete and Update.

Append mode: If we have to add/accumulate only incremental data to the target then we can go with append mode.

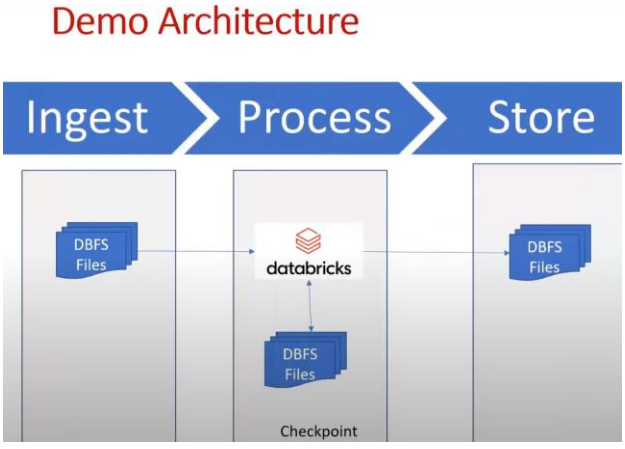
Complete mode: In sudden cases if we have to perform certain aggregation then we have to include complete data

Update Mode: If we need to update existing data based on the latest value then we can go with update mode.

SAMPLE STANDARD ARCHITECTURE FOR STREAMING APPLICATIONS

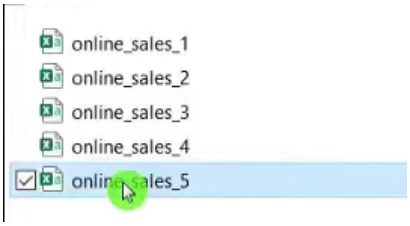


In this architecture , there are 4 processes ingestion, processing, storing, and reporting. Let's say there are external sources databases and file systems and databricks can consume data directly from those sources or it can consume data from the mediator i.e., streaming service like Azure event hubs and kafka. Let's say we are going to use Kafka then what happens is source systems would push data to kafka topic and azure data bricks engine can consume data from kafka topic and incase raw data is needed for some tracking or debugging process then we can persist the data into storage location and storage location can be used for checkpointing also. Checkpoint can keep track of processed data in the form of offset and once data is consumed from kafka topic then databricks engine can apply all the business logic then final aggregated data can move to azure data warehouse and from azure data warehouse, data can flow to reporting either directly can flow to power bi or we can deploy azure services in between for creating semantic layer



Example: There are 5 files , feeding one by one , then databricks will consume in real time and it will start processing and also it will keep track of offset value in checkpoint location. Once data is processed then finally it will move data to target location i.e., stream_write.

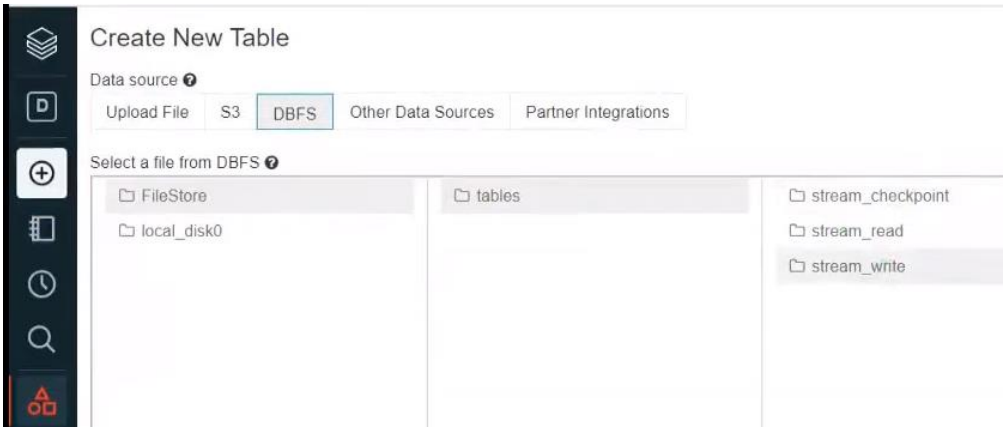
Example Demo: These are the 5 files shown below . using this 5 files we can see how spark streaming application will consume data in real time.



Step-1: we are creating 3 folders using below commands in Data →FileStore→ tables →.

```
# Create folder structure in DBFS file system

dbutils.fs.mkdirs("/FileStore/tables/stream_checkpoint/")

dbutils.fs.mkdirs("/FileStore/tables/stream_read/")

dbutils.fs.mkdirs("/FileStore/tables/stream_write/")
```

Out: True



Step-2: Define schema for our sample streaming data

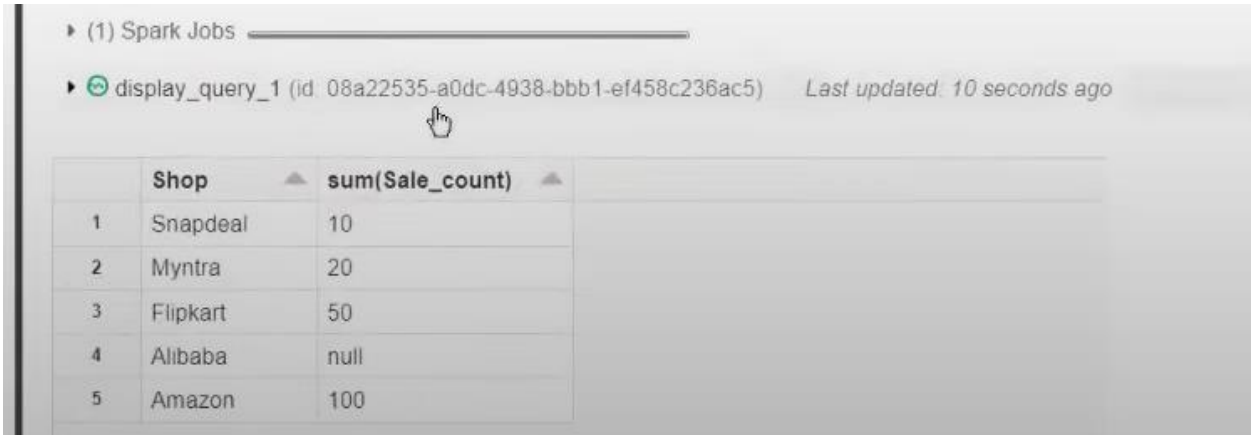
```
from pyspark.sql.types import StructType,StructField,IntegerType,StringType

schema_defined = StructType([StructField('File',StringType()),
                             StructField('Shop',StringType()),
                             StructField('Sale_count',IntegerType())
])
```

Step-3: Read Streaming data → here it will consume data in real time

```
df=spark.readStream.format("csv").schema(schema_defined).option("header",True).option("sep",";").load("/FileStore/tables/stream_read/")

df1=df.groupBy("Shop").sum("Sale_count")

display(df1)
```

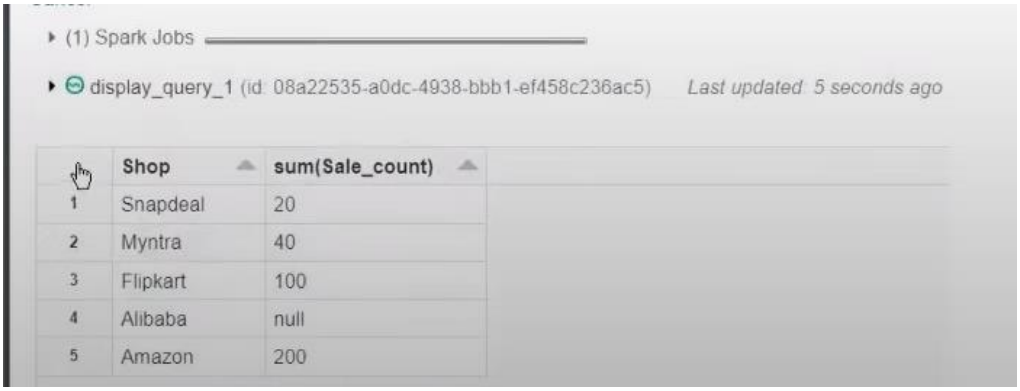
Query returned no results because we haven't attached any file in DBFS path.

Now we uploaded 1st file in another google chrome window, and when we see data again for read streaming data , data is displayed as shown below. That means data is processed for 1st file.



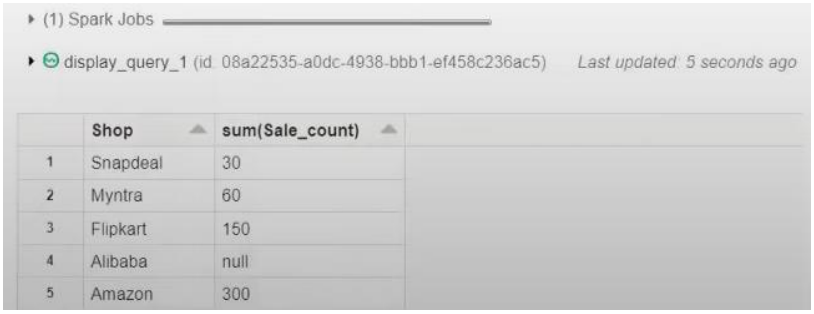
	Shop	sum(Sale_count)
1	Snapdeal	10
2	Myntra	20
3	Flipkart	50
4	Alibaba	null
5	Amazon	100

Now if we add 2nd file in the same location stream_read folder , now aggregation is performed on the 2 files data as shown below.



	Shop	sum(Sale_count)
1	Snapdeal	20
2	Myntra	40
3	Flipkart	100
4	Alibaba	null
5	Amazon	200

Now we uploaded 3rd file , result is as shown below.



	Shop	sum(Sale_count)
1	Snapdeal	30
2	Myntra	60
3	Flipkart	150
4	Alibaba	null
5	Amazon	300

Step-4: Write Streaming data

```
dbutils.fs.mkdirs("/FileStore/tables/stream_checkpoint/")
```

```
dbutils.fs.mkdirs("/FileStore/tables/stream_read/")
```

```
dbutils.fs.mkdirs("/FileStore/tables/stream_write/")
```

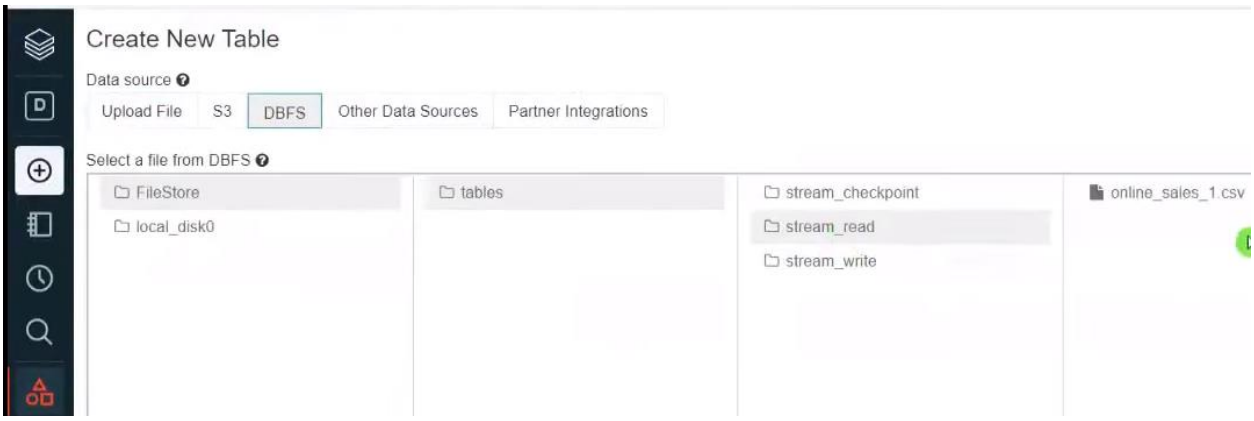
again, creating folders in order to remove files from stream_read folder.

Here we have given path where we want to write data and also, we have mentioned checkpoint location to keep the metadata .If we are writing data in sink, it will create metadata in checkpoint location in order to keep track of processed files.

```
df2=
df.writeStream.format("parquet").outputMode("append").option("path","/FileStore/tables/stream_write/").option("checkpointLocation","/FileStore/tables/stream_checkpoint/")
.start().awaitTermination()
```

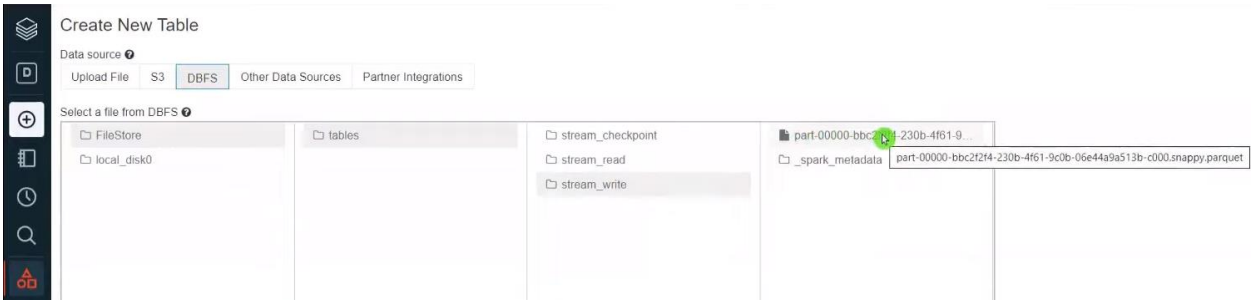
once we execute above command df2 , no data will be shown because no files in stream_read

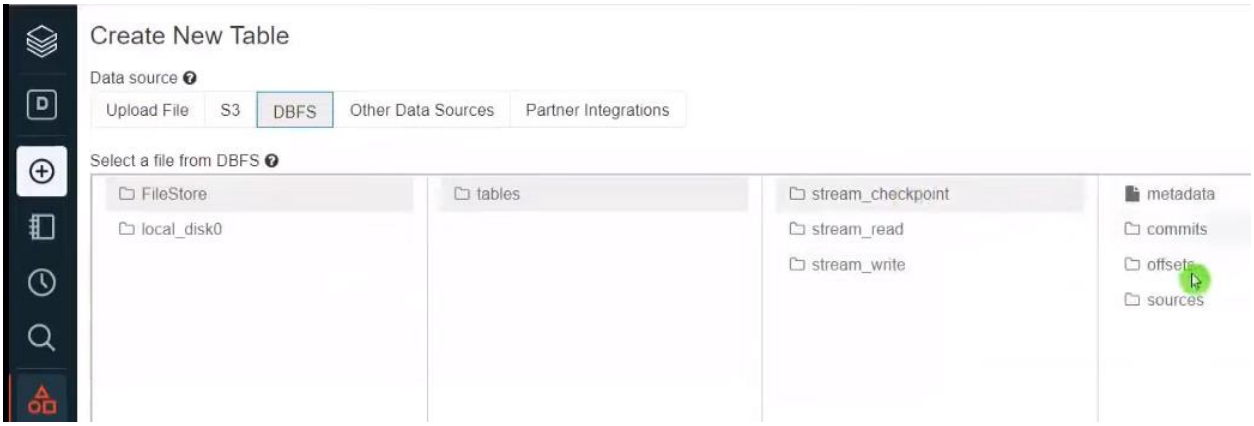
Now once we add 1 file in stream_read as shown below.



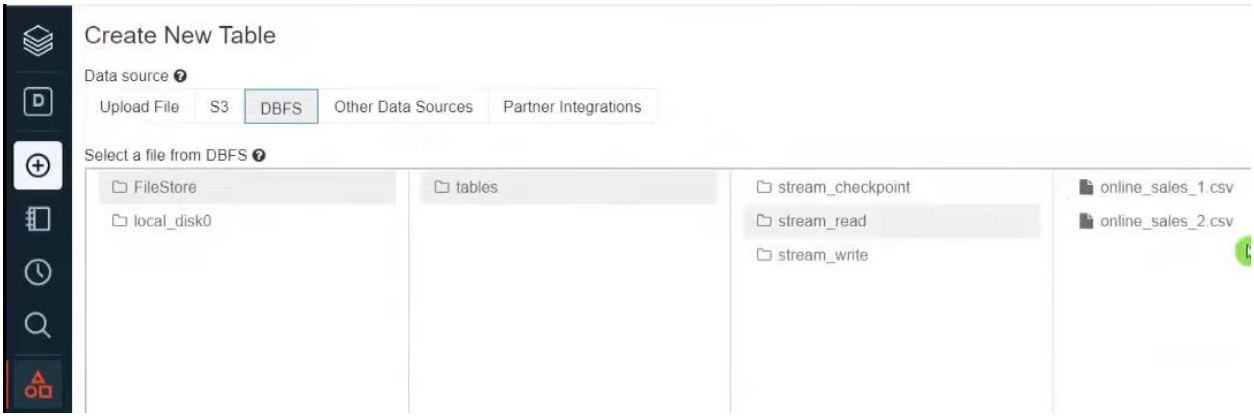
Now code in notebook is still running and if we see stream_write folder, data is uploaded in stream_write folder in the form of parquet file as shown below and also it has created 1 folder for metadata and in checkpoint location(stream_checkpoint) also, we can see it has created metadata. All the folders in stream_checkpoint and spark_metadata folder in stream_write folder both are coupled.in case if we want to make any changes then we cannot remove only 1 folder. Always we want to remove metadata

folder from write location and all the folders from the checkpoint location that means both are coupled

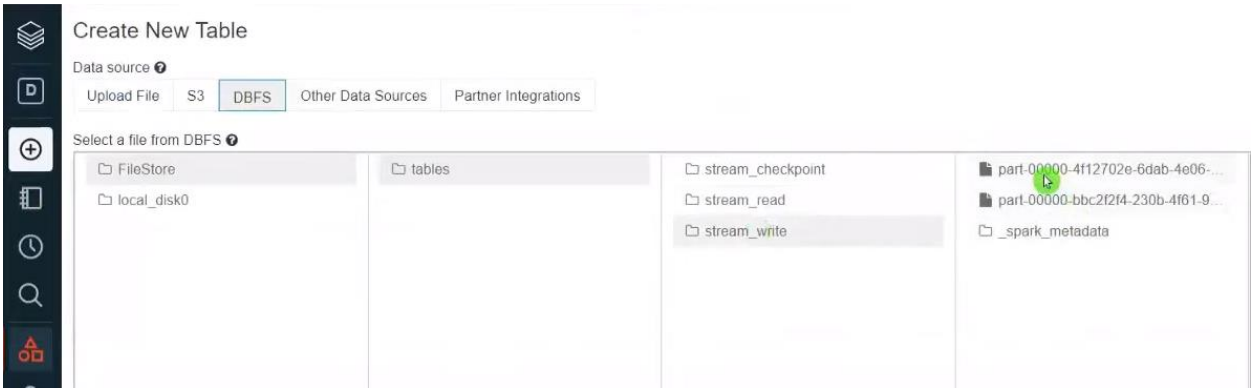




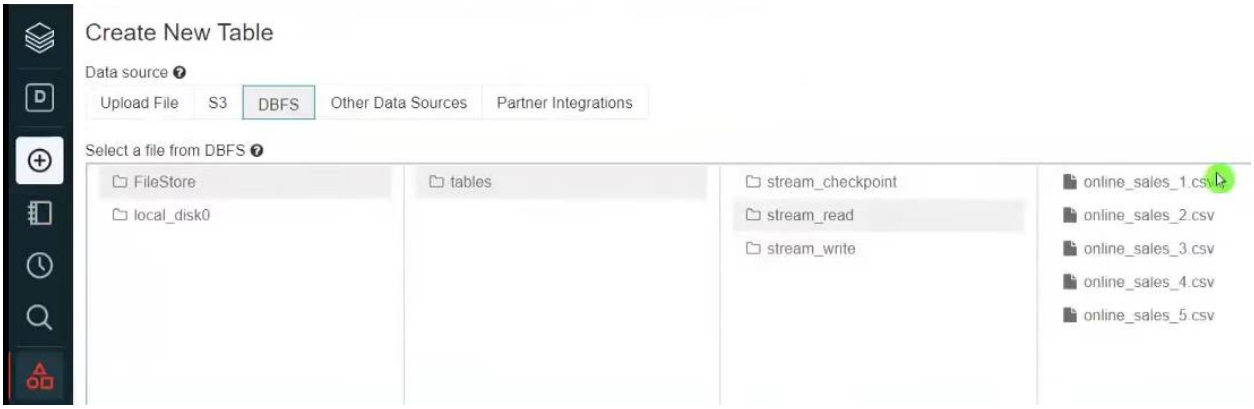
Now we will add 1 more file 2nd file



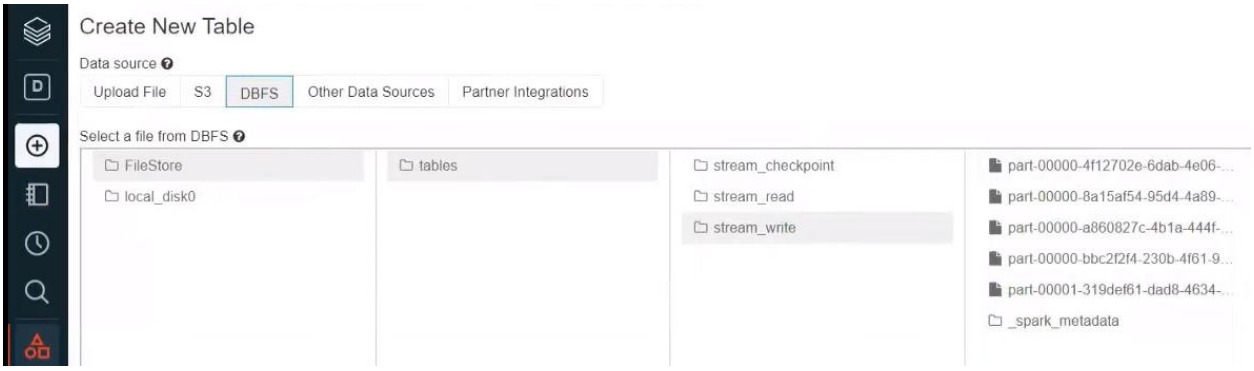
Now our streaming application will pick 2nd file, then it will create parquet file under stream_write folder as shown below.so 2 parquet files got created. 1 parquet file for 1st input file and 2nd parquet file for 2nd input file.



Now 3 more input files got added as shown below.



Code in the notebook is still running, if we see output now in stream_write folder,



Note: for simple example we make usage of DBFS, but in real time projects source will be from kafka or azure event hubs.

Step-5: Verify the written stream output data

```
display(spark.read.format("parquet").load("/FileStore/tables/stream_write/*.parquet")
)
```

```
display(spark.read.format("parquet").load("/FileStore/tables/stream_write/*.parquet"))
```

▸ (3) Spark Jobs

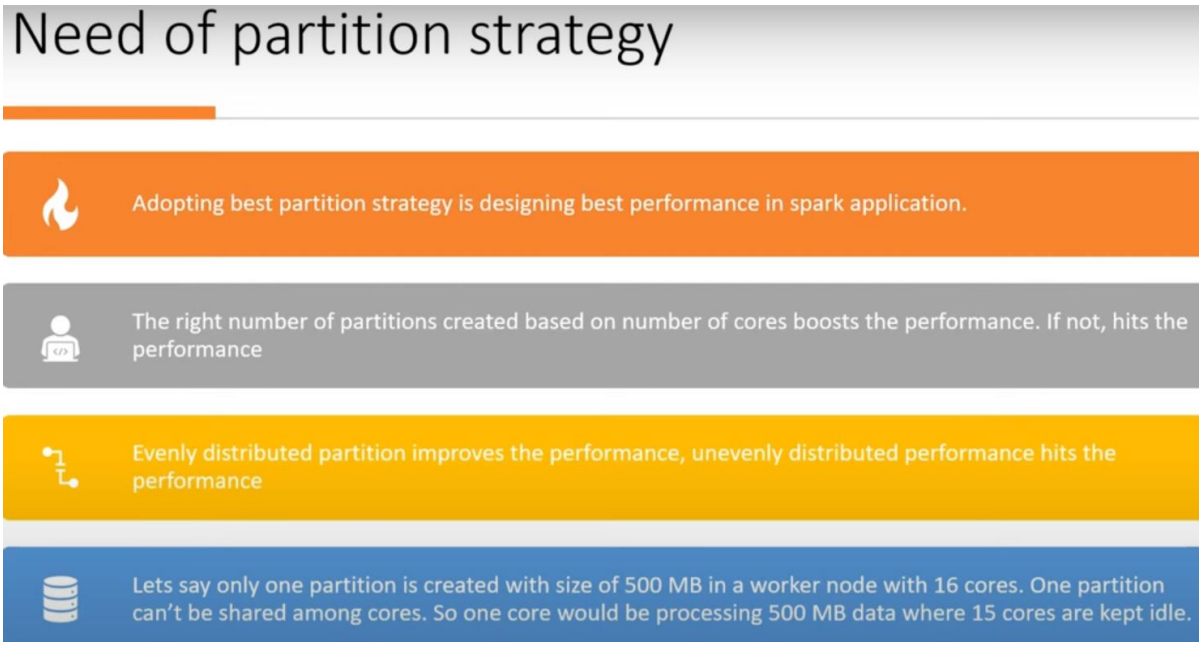
	File	Shop	Sale_count
1	File2	Amazon	100
2	File2	Flipkart	50
3	File2	Myntra	20
4	File2	Snapdeal	10
5	File2	Alibaba	null
6	File5	Amazon	100
7	File5	Flipkart	50

Showing all 25 rows.

SPARK PERFORMANCE OPTIMIZATION

Spark is well known for its speed. Fast computing is mainly because of parallel processing. Partition is the key for parallel processing. If we design partition in proper way that would improve the performance by triggering the parallel processing effectively. If we are not designing the partition properly then we are not effectively using distributed framework. Partition plays important role in performance improvement, error handling and debugging\.

WHY DO WE NEED PARTITION?



- Partition is key for performance , is key for parallel processing and that is key for distributed framework. There are 2 important concepts in partition one is choosing right number of partitions, 2nd is choosing right size of the partitions. Let's say we are having 16 core executors, within that we are created partitions only 10 partitions then what happens is when we are triggering some action out of 16 cores,10 cores will pick 1 partition each.so partition cannot be shared among cores so what happens is 10 cores will pick up 1 partition and start processing but 6 cores will sit idle so we are not effectively using all the cores for processing so that is the reason we need to choose right number

of partitions. Calculation will be like if we have 16 cores in a executor then we can choose 16 partitions at least or it should be multiples of number of cores. So, in this example it should be multiples of 16.so what happens is in 1st iteration all 16 cores will pick up 16 partitions and it will process .once those are completed, in 2nd iteration again those 16 partitions will pick next 16 partitions. so, in this way it will complete all the partitions at same time, so we are not keeping any core idle. So that is the reason we will choose right number of partitions .

- 2nd one is Evenly distributed partitions. Let's imagine we have executor , within that we have 2 cores , and there are 2 partitions ,1 partition is of 1GB size,2nd partition is of 10MB size. Those 2 cores will pick up 1 partition each.so 1st core will process 1GB data and 2nd core will process 10MB data so what happens 2nd core which process 10MB file will complete the process very quickly and it will start sitting idle but other core which is processing 1GB of data it will keep on processing and overall application will take more time for completion so that is the reason we have to choose right size of the partition or we can say partitions to be distributed evenly.

Note: Choosing right number of partitions and evenly distributed partitions these 2 are important areas we need to pay attention .

In order to understand these partitions, first we need to understand 2 parameters one is sc.defaultParallelism and 2nd is spark.sql.files.maxPartitionBytes

sc.defaultParallelism : When we are generating data within spark environment what happens is based on parameter sc.defaultParallelism it would create number of parameters .by default value for the parameter is 8. When we are generating data within spark environment ,it would generate 8 partitions.

When we are reading certain file from external file system let's say we are reading csv file of 1GB from external file system, so number of partitions will be determined by spark.sql.files.maxPartitionBytes, byte default value for the parameter is 128MB. So, if we are reading 1 GB of csv file then what happens is spark.sql.files.maxPartitionBytes this parameter will start packing 128MB of data into each partition which means it will create approximately 7/8 partitions for 1GB of data.

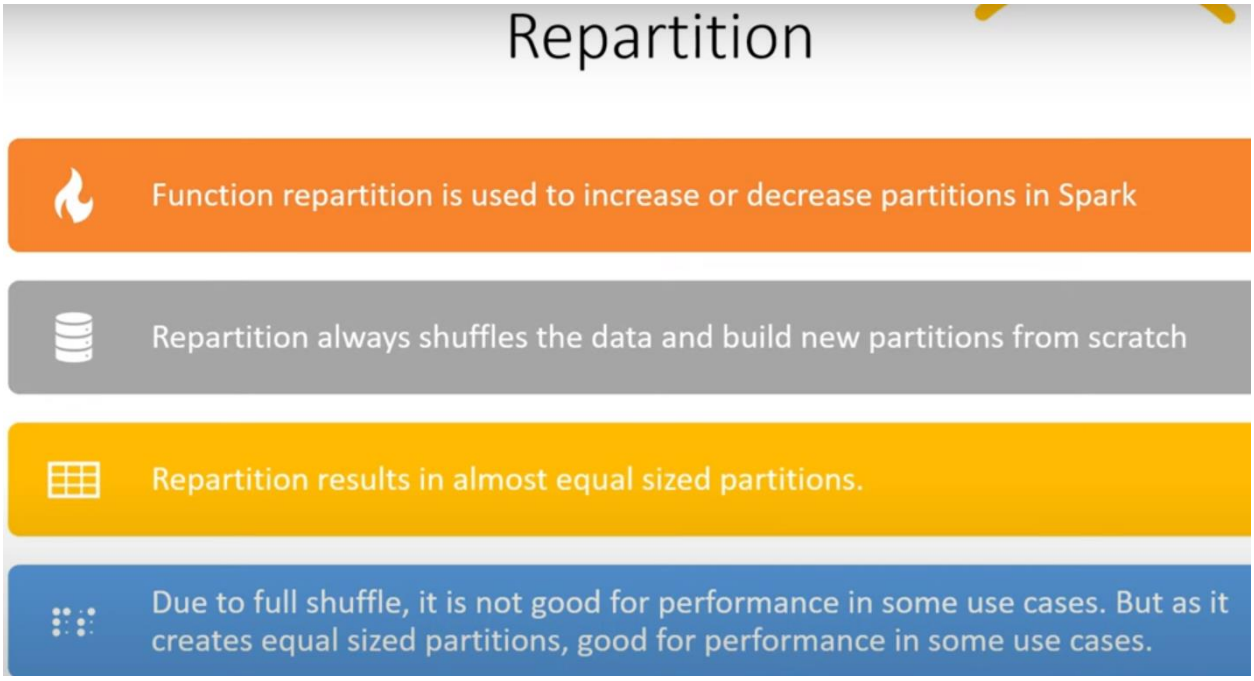
These 2 parameters(sc.defaultParallelism and spark.sql.files.maxPartitionBytes) are configurable. We can change the parameters depending on the situation. Depending on our use case we can change the parameters. Some developers are developing applications which will handle terabytes of data that use case is different. Some developers are developing application which are very simple it will process few GB's of data then that use case is different. Depending on use case , we have to treat this parameters.

maxPartitionBytes will divide the data into multiple partitions of 128MB. So, this is applicable only for the files which are in splitable form. If the file is compressed in snappy/GC format, then it cannot be splitted. Even though file is terabytes of data , still it would be treated as 1 partition.

REPARTITION

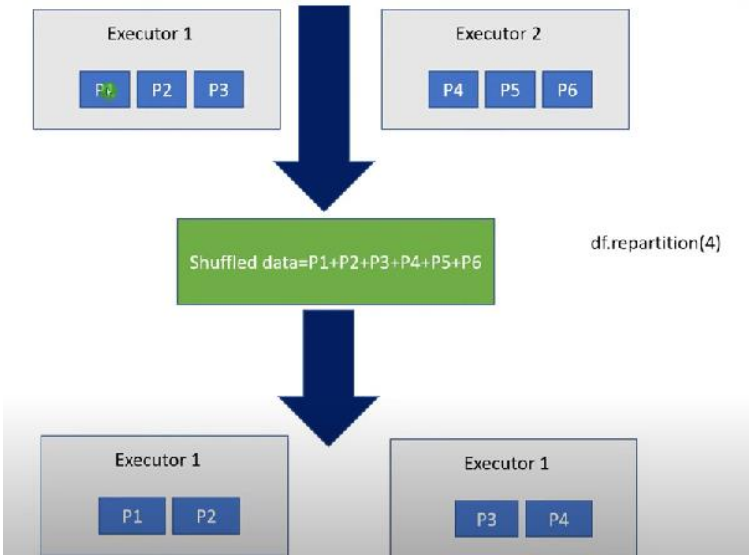
Repartition is spark function which will increase/decrease partitions in spark. Depending on use case that we are working , we are developing solution, so we can decide how many partitions we need. By default, spark will create certain number of partitions but depending on use case if we want to increase/decrease, either we can

configure the parameters or programmatically temporarily we can increase or decrease the partitions for our input data input RDD/df .



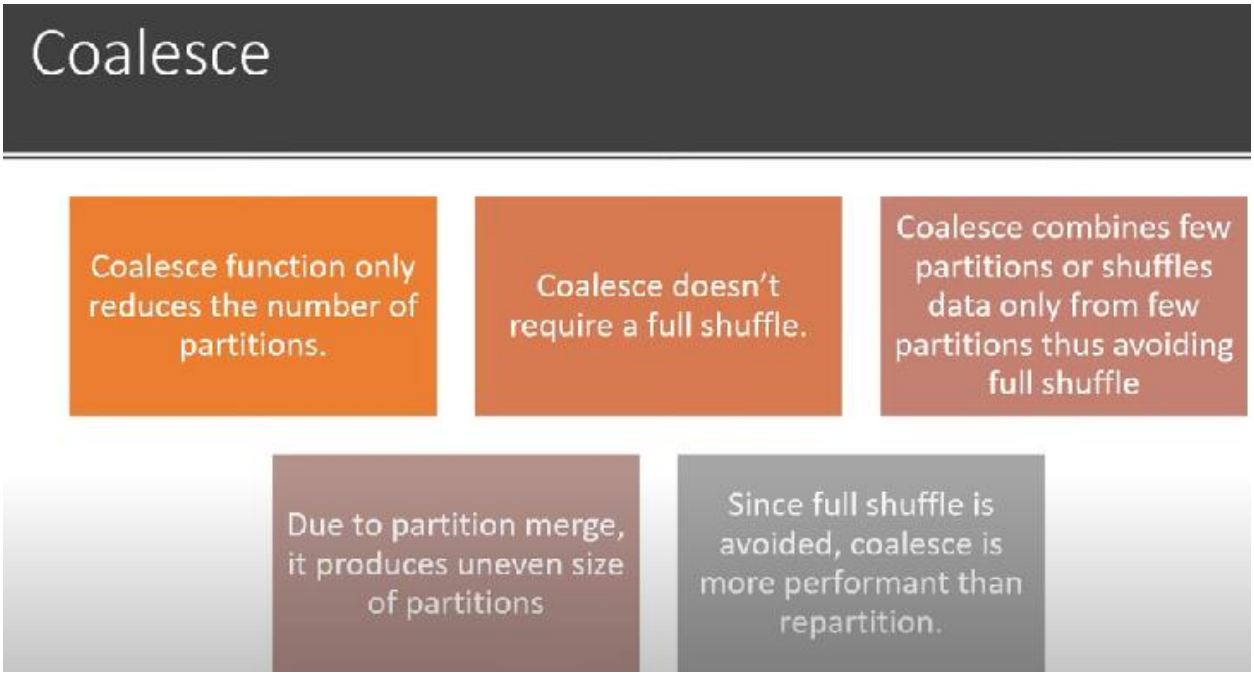
Repartition used to increase/decrease partitions. When we are applying function repartition it would shuffle the data entire data across the node and then it will start building the new partition from scratch which means shuffling is one of the costliest operation in spark but when we are applying repartition it would start with shuffling the data but good part is it will create equal sized partitions which means partition is evenly distributed but at the same time it is triggering the shuffle which will hit the performance so we have to be calculative but at the end of execution it will produce equal sized partitions that will boost the performance.

Example:



Executor1 is having 3 partitions P1,P2,P3 and Executor 2 is having 3 partitions P4,P5,P6. Now if we apply repartition function among these 6 partitions , we want to create 4 partitions then what happens is spark will collect all these data it will start shuffling and it will create the shuffled data that would contain data from all the partitions then it will start splitting data as per our needs. If we use df.repartition(4), it will start splitting shuffled data in to 4 pieces, so 2 partitions for executor 1 and 2 partitions for executor 2.

COALESCE FUNCTION:

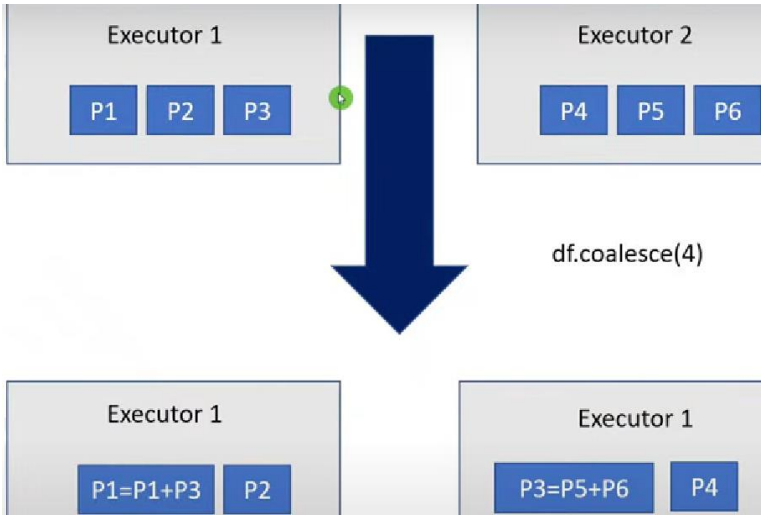


Repartition used for increasing/decreasing partitions but coalesce is used only for decreasing partitions. We cannot increase number of partitions based on coalesce function.

Processing method is completely different for coalesce and repartition. Repartition we will shuffle data across the node then it will produce repartitions but coming to coalesce , coalesce doesn't require a full shuffle . instead of that it will start combining or merging partition within that executor so that it does not require shuffling the partition across the node but at the same time it will produce unevenly distributed partitions. These are difference between repartition and coalesce. Depending on our situation or our use case we have to decide one of the partition strategy.

COALESCE FUNCTION EXAMPLE:

Executor1 has 3 partitions, executor 2 has 3 partitions, when we apply coalesce function df.coalesce(4) , it will convert 6 partitions into 4, what happens is it won't shuffle data across these 2 executors. Instead of that it has to create 4 partitions. 2 partitions per executor ,so it will start combining existing partitions within the executor . so, in executor1 it can combine any of the 2 partitions, it will produce new partition . similarly, in executor2 it will combine any 2 partition and produce P3 and P4 remains as it is.



Example 1:

#Check default parameters

- `sc.defaultParallelism`

Out[1] : 8

- If we are reading certain file from external file system, then file would be splitted based on maxPartitionBytes. By default, it is 128MB

```
spark.conf.get("spark.sql.files.maxPartitionBytes")
```

Out[2]: '134217728b' this is equivalent to 128MB

#Generating data with spark environment

```
from pyspark.sql.types import IntegerType
df= spark.createDataFrame(range(10), IntegerType( ))
df.rdd.getNumPartitions()
```

Out[3] : 8 → 8 partitions this output coming from sc.defaultParallelism

#Verify the data within all partitions

```
df.rdd.glom().collect()
```

1 spark job

If we want to see data for each partition, then we can use function glom. There are 8 partitions in output as shown below.[Row(value=0)] is 1 partition,[Row(value=1)] is 2nd partition,[Row(value=2)] is 3rd partition,[Row(value=3),Row(value=4)] is 4th partition and so on.

```
Out[4]: [[Row(value=0)],
 [Row(value=1)],
 [Row(value=2)],
 [Row(value=3), Row(value=4)],
 [Row(value=5)],
 [Row(value=6)],
 [Row(value=7)],
 [Row(value=8), Row(value=9)]]
```

Example 2:

#Read External file in spark

dbutils.fs.ls("/FileStore/tables/baby_names/")

Out[5] :

```
Out[5]: [FileInfo(path='dbfs:/FileStore/tables/baby_names/Baby_Names_2007_2009.csv', name='Baby_Names_2007_2009.csv', size=484978),
 FileInfo(path='dbfs:/FileStore/tables/baby_names/Baby_Names_2010_2012.csv', name='Baby_Names_2010_2012.csv', size=468883),
 FileInfo(path='dbfs:/FileStore/tables/baby_names/Baby_Names_2013_2014.csv', name='Baby_Names_2013_2014.csv', size=362725)]
```

df=
spark.read.format("csv").option("inferSchema",True).option("header",True).option("sep",",").load("/FileStore/tables/baby_names/")

df.rdd.getNumPartitions()

2 spark jobs will be run.

Out: 3 → 3 partitions (supplied 3 files so 1 partitions for each file)

If file size is greater than 128MB , then it will produce multiple partitions by splitting data of 128MB.

#Change the maxpartitionbytes parameter which changes in no of partitions

Splitting the file based on 200KB, so changed 2nd parameter to 200000. In 2nd command I am getting value for verification

spark.conf.set("spark.sql.files.maxPartitionBytes",200000)

spark.conf.get("spark.sql.files.maxPartitionBytes")

Out[7]: '200000'

df=
spark.read.format("csv").option("inferSchema",True).option("header",True).option("sep",",").load("/FileStore/tables/baby_names/")

df.rdd.getNumPartitions()

2 spark jobs ran

Out[8]: 8 (3 partitions for 1 file, 3 partitions for 2nd file, 2 partitions for 3rd file)

If we change the parameter to 100KB,then

```
spark.conf.set("spark.sql.files.maxPartitionBytes",100000)

spark.conf.get("spark.sql.files.maxPartitionBytes")
```

Now the parameter is set to 100KB, now if we read file and get partitions information again

```
df=
spark.read.format("csv").option("inferSchema",True).option("header",True).option("sep",",").load("/FileStore/tables/baby_names/")

df.rdd.getNumPartitions()
```

2 spark jobs ran

Out[10] : 14(5 partitions for 1st file, 5 partitions for 2nd file, 4 partitions for 3rd file)

This is how we can change default parameters . But in real time there can be tb or gb of data. We need to configure parameters according to our use case

#If we Create single partition with all data .This is not good for performance ,as one core would process entire data while all other cores are kept idle.

REPARTITION EXAMPLE:

```
from pyspark.sql.types import IntegerType

df= spark.createDataFrame(range(10),IntegerType())

df.rdd.glom().collect()
```

(1) Spark jobs

Out[11] 8 partitions

```
Out[11]: [[Row(value=0)],
 [Row(value=1)],
 [Row(value=2)],
 [Row(value=3), Row(value=4)],
 [Row(value=5)],
 [Row(value=6)],
 [Row(value=7)],
 [Row(value=8), Row(value=9)]]
```

We can increase or decrease partitions , first we are increasing partitions to 20

```
df1= df.repartition(20)

df1.rdd.getNumPartitions()
```

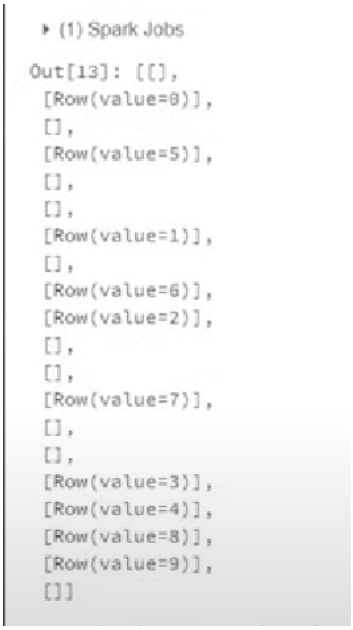
(1) Spark jobs

Out[12] 20 → 20 partitions

We are getting only 10 unique values from 0 to 9 but how do we create 20 partitions out of it ? when we don't have enough data to create partition then it will create empty partition .

We can see 10 empty partitions got created as shown below and 10 numerical values so total 20 partitions got created.

```
df1.rdd.glom().collect()
```



Now we are going to create 2 partitions.

```
df1= df.repartition(2)
```

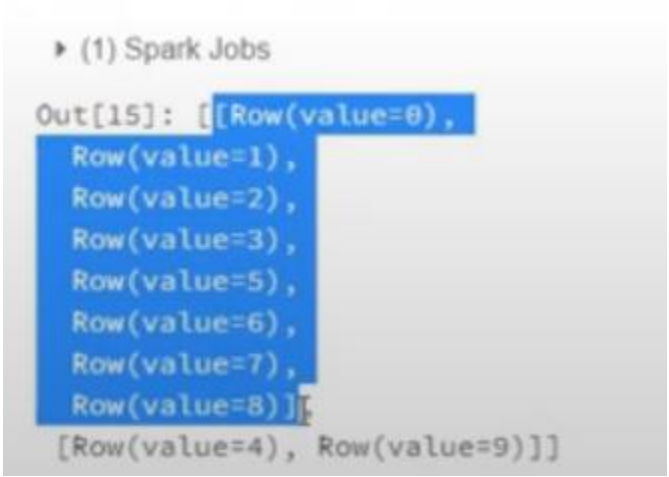
```
df1.rdd.getNumPartitions()
```

(1) Spark jobs

Out[14] : 2 → 2 partitions

```
df1.rdd.glom().collect()
```

here in output 2 partitions got created. Square box starts from and ends with value=8 that is first partition and 2nd partition is value 4 and 9. So total 2 partitions.



COALESCE

df2=df.coalesce(2) → reduced partitions from 8 to 2

df2.rdd.getNumPartitions()

df2.rdd.glom().collect()

(1) Spark jobs

Out[16] :

```
Out[16]: [[Row(value=0), Row(value=1), Row(value=2), Row(value=3), Row(value=4)],
 [Row(value=5), Row(value=6), Row(value=7), Row(value=8), Row(value=9)]]
```

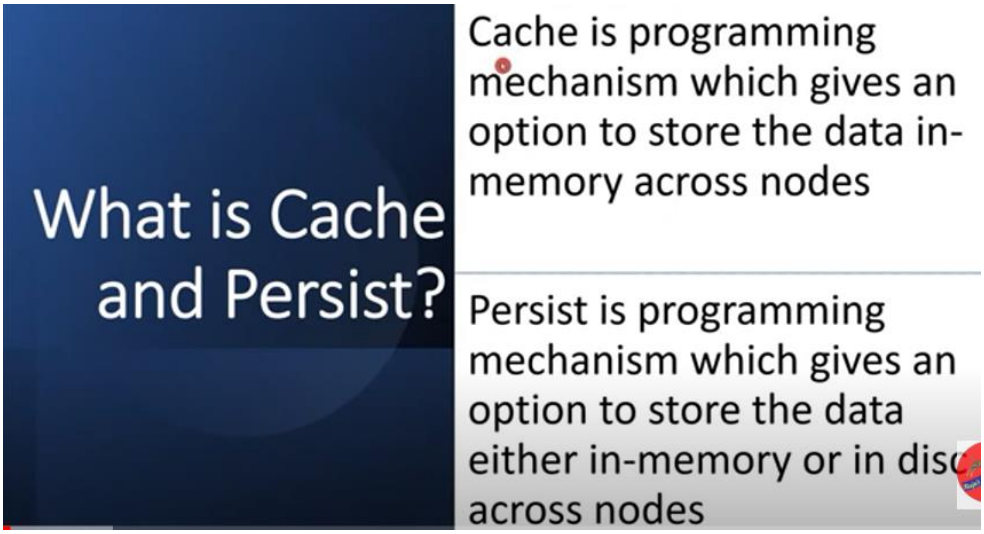
So, 2 partitions got created as shown above.0 to 4 values as 1 partition and 5 to 9 values as 2nd partition.

Note: coalesce is used to reduce number of partitions and it does not require shuffling data across nodes and it would merge partitions within executor. Also coalesce produces unevenly distributed partitions.

Using repartitions , we can increase or decrease number of partitions. It will shuffle data across nodes . It will produce evenly distributed partitions

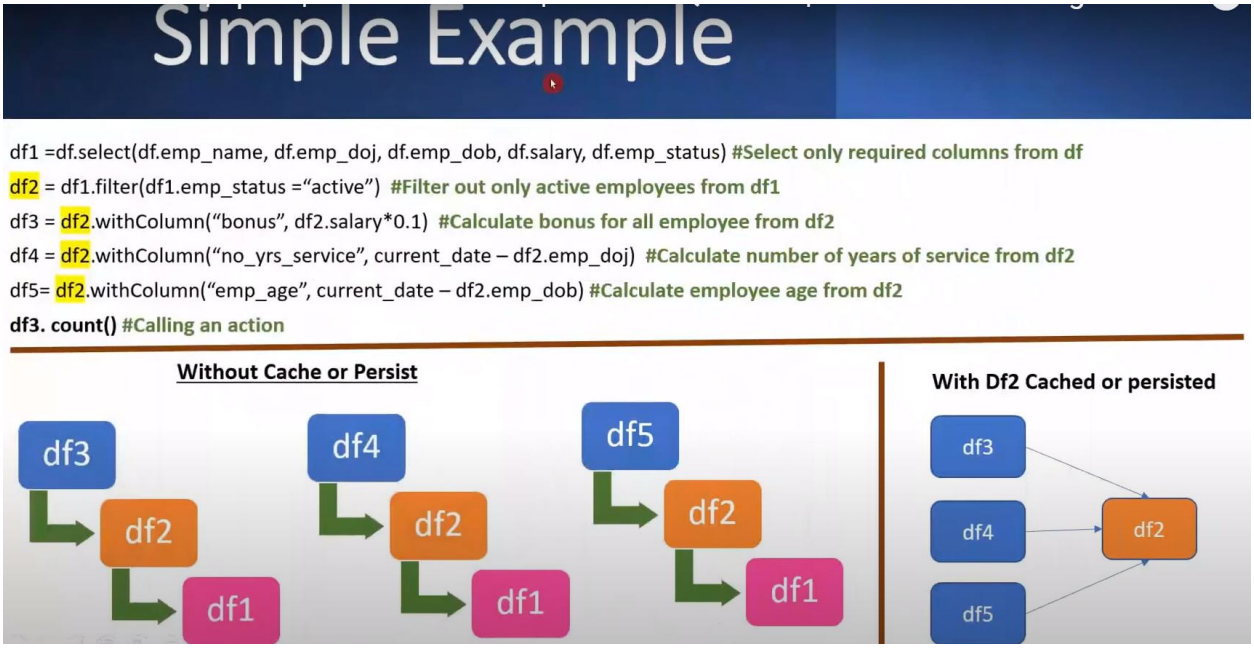
CACHE vs PERSIST

Cache and persist are API's provided by spark. Both are spark function to store data either in memory/disk or combination of both.



When we have to store data in memory then we can use cache. Cache always store the data in memory across nodes. Cache is programming mechanism through which we can store data in-memory across nodes. Coming to persist , persist is another programming mechanism through which we can store the data either in-memory or in disc or combination of both across nodes. As per spark architecture , we already doing in-memory computation then why do we need these cache and persist functions explicitly ?There are transformations and actions in spark. When we execute transformations in spark , there are getting evaluated lazily which means when we are submitting some transformation only logical steps will be created and kept in DAG, but actual execution

will happen only when we call certain action. Let's say there are 1000's of transformation at the end we are calling 1 action then what happens is that action will trigger actual data processing starting from 1st transformation will apply one by one then finally it will apply action so which means when we are triggering certain action then only in memory computation will start but if we want to execute certain transformation intermittently and if we want to store data either in memory /disk then we can use this functions.



In above screenshot, we have given series of transformation and at the end we have given 1 action.

When df3 is called for an action, then flow would go to df3.df3 is depending on df2 and then control goes to df2. df2 is depending on df1 and flow would go to df1 and df1 is depending on df.so df1 will be created first . df2 will be created based on df1 and df3 will be calculated based on df2.

So, all this process df3,df4 and df5 all these are depending on df2 . so df2 is getting recalculated again and again. So, this is time consuming process but in order to improve performance we can calculate this transformation df2 and we can store intermittent result within memory or storage disk of the nodes. We can apply cache/persist on df2 which means all the previous step will be executed and then resultant data will be stored in memory or disk. That is concept of cache. Cache is always storing data in memory. Coming to persist, persist always gives an option, it is flexible we can either store the data into memory or storage disk or can be combination of memory+ storage disk .once we have done cache/persist , process will be simple. If we need df3,then it will directly take the data from df2 because it is already computed once, and result is stored. Same is applicable for df4 and df5.In each and every time, we no need to go for recomputation of df2. In this way we are improving the performance. If we are improving the performance via cache/persist , why we are not doing cache for intermittent transformations. If we are going to persist/cache all the transformations, then it will improve performance and it will be super-fast but on other side any cluster will have limited memory. Let's say our worker

node will contain 128GB of RAM which means it can accept only 128GB of data and out of that we need to store data and also, we need some memory for calculation .so if we are going to dump intermittent result into memory what happens is there is shortage of memory for any computation which means it will throw out of memory error. If we exhaust memory by storing all the data, then we cannot expect better performance because there won't be enough memory for calculation.

Storage Level	Description
MEMORY_ONLY	Same as Cache
MEMORY_ONLY_SER	Persisting the RDD in a serialized (binary) form helps to reduce the size of the RDD, thus making space for more RDD to be persisted in the cache memory. It is space-efficient but not time-efficient
MEMORY_AND_DISK	Stores partitions on disk which do not fit in memory
MEMORY_AND_DISK_SER	Same as "Memory and Disk" but in serialized form
DISK_ONLY	Persists data only in Disk, which would require network input and output operations thus time-consuming. Still better performance than re-computation each time through DAG
DISK_ONLY_2, MEMORY_AND_DISK_2, MEMORY_ONLY_2, MEMORY_AND_DISK_SER_2 MEMORY_ONLY_SER_2	Same as above methods but creates replication in another node.so if any failure of node, still data is accessible through another node

MEMORY_ONLY: In cache data would be stored in the memory. When we gave storage level option MEMORY_ONLY which means intermittent result that would be computed, and data would be stored within memory. In case intermittent result is 150GB and RAM size is 128GB here data size is very huge when compared to memory so what happens is only partial amount of data would go to memory and rest of the data would not get any space in the memory which means rest of the data will always be recalculated using DAG. When we do cache , if data is not fit into the memory what happens is it will store only partial data let's say 80% of data fits into memory,20% of data will always be recalculated .

MEMORY_ONLY_SER: it is same as MEMORY_ONLY.but difference is that when we store any data in memory then it will be in deserialized form (that is space consuming storage type)but here when we are going to apply serialization what happens is that data is serialized, and it is kept in memory which means serialized data will always take lesser space when compared to deserialized data. We can store more data into memory when we do serialization but at the same time if we have to read data from memory what happens is first data has to be deserialized then only programs can use the data, so it is adding 1 more extra process for CPU that is overhurdle.so memory only serialized but is space efficient but not time efficient.

MEMORY_AND_DISK: This option provides storing data between memory and disk which means let's say 80% of data fit in to the memory then it will go to memory and remaining 20% of data will go to disk. This is always better when compared to recomputation of the remaining data . so, if data size does not fit into memory, then remaining data will

be stored in disk. This option is always better when compared to MEMORY_ONLY. As long as data fits into memory as well and good, we can use MEMORY_ONLY if it is not fitting in memory then it will trigger the recomputation for missing partitions. When we are storing any data in disc which means always it will be in serialized format. In this option deserialization would be applied for memory and disk storage will have only serialized data

MEMORY_AND_DISK_SER: same as Memory and Disk but it will be in serialized form. In this option both memory and disk will be serialized.

DISK_ONLY: Here entire data will be stored in disk only in serialized form.

We can apply _2 for all the above options(shown in screenshot) which means replication will happen for each partition which means 1 particular partition will be stored across 2 different nodes. If any one of the node is failed, then data can be retrieved using another node that is advantage. Whichever storage level we need for our use case , we need to go with that.

SUMMARY:

Summary

Level	Space Used	CPU Time	In Memory	On Disk	Serialized
MEMORY_ONLY	High	Low	Y	N	N
MEMORY_ONLY_SER	Low	High	Y	N	Y
MEMORY_AND_DISK	High	Medium	Some	Some	Some
MEMORY_AND_DISK_SER	Low	High	Some	Some	Y
DISK_ONLY	Low	High	N	Y	Y

MEMORY_ONLY level has high space used because in memory data will always be stored in deserialized form which means it will consume more space. And coming to CPU Time it is low because data is already in deserialized form .so when process needs data , it does not need to deserialize once again. Coming to In-Memory, data would be stored in memory.in case it is overflowing then recomputation will be need only for the extra data and On Disk No this will not be stored in disk. Finally Serialized ,by default option is deserialized

MEMORY_ONLY_SER: Here it is in serialization form so it will consume less space and at the same time it will take more time for CPU Processing .it is memory storage and not on disk and it is in serialized

MEMORY_AND_DISK: Here space used is high because even though disk is using serialized form and memory is used in deserialized form .that is the reason memory used is high and coming to CPU Time it will be medium because we have to retrieve certain data from disk which means disk will always require input output operations through network that will take some time but compared to recomputation this input output is always better.

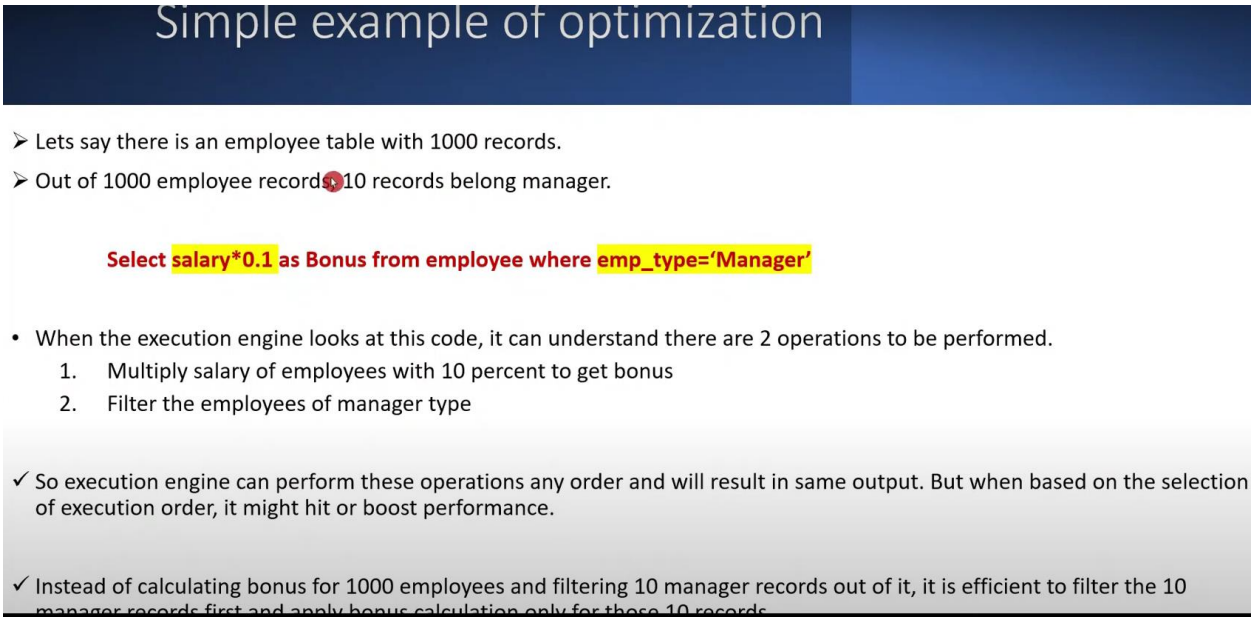
Some of data can go to in-memory and some of data can go to disk. Data stored in disk is serialized but data stored in memory is not serialized

MEMORY_AND_DISK_SER: Space is low due to serialization, and it is not CPU efficient, some of data can be stored in in-memory and some of data can be stored in disk and both memory and disk is serialized.

DISK_ONLY: When we use disk only which means all the transformation would be computed and data will be stored into disk only which means when we have to process the data ,it has to retrieve data through networking input and output it might take some time, but it is better approach when compared to recomputation. Space used is low because when we store data in disk which means it is always in serialized form.

CATALYST OPTIMIZER

Catalyst Optimizer is a spark code component written in Scala program. Catalyst Optimizer is a Scala program that is automatically used to find most efficient plan to execute data operations when we submit some user code into spark engine. In order to execute particular statement or particular query, there could be multiple execution path.

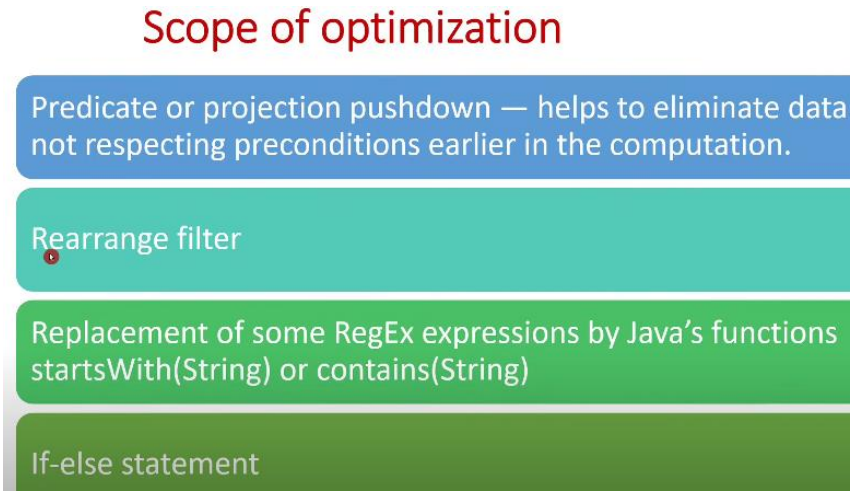


When we submit code(select salary*0.1 as Bonus from employee where emp_type='Manager'), spark execution using catalyst optimizer will create multiple execution plans. 2 approaches are shown above.

SCOPE OF OPTIMIZATION: The below shown few areas where catalyst optimizer applies optimization.

- Predicate/Projection pushdown: When we join multiple tables based on the joining condition it can filter out the records . when we have multiple joining condition first, we have to apply join where it will reduce number of records to the smallest. Whichever way we have written joining condition, internally catalyst optimizer will check, and it will apply predicate push down which means will move the joining condition to top order where it will reduce the data to smallest.

Coming to projection pushdown, when we are working on employee table, employee table is containing 100's of columns but in our final output we need only few columns then there is no point in tracking those 100's of columns in each and every stage so whichever column that is needed for our final output only that is only considered.



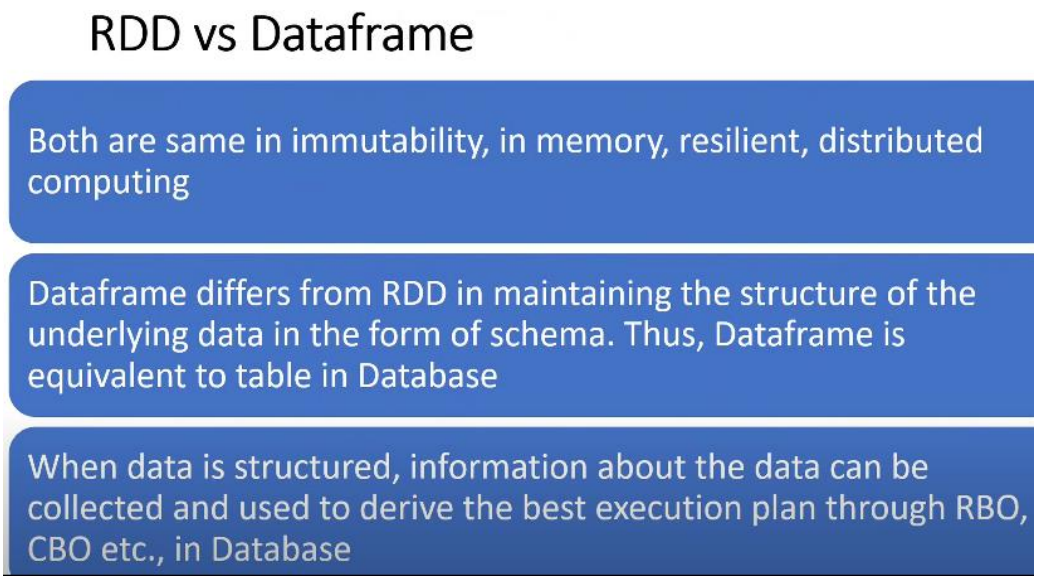
- Rearrange filter: Let's say we are applying multiple filters in 1 action in data frame or query in SQL API, then spark engine catalyst optimizer will check which filter will remove most of the records that we are not interested in then we will get simple dataset and then it will push that filter to the top that is rearranging the filter
- RegEx expression in spark: RegEx expression is costly operation in spark. When we submit code in any language let it be pyspark or Scala what happens is that internally it will go through multiple stages and finally it will be converted into java code because executor will be running on JVM Process. JVM Process can work only with Java code. Finally, code will be converted into java. That is the reason if there are any regular expression in spark then optimizer will look for functions in java. If any functions are available, then it will start considering that function so regular expression will be replaced in spark program.
- If-else statement: If-else containing multiple paths. based on the condition, it will choose one of the path.so optimizer will be smart enough to choose right path by eliminating undesired paths quickly.

This optimizer is only available for dataframe and dataset but not for RDD.

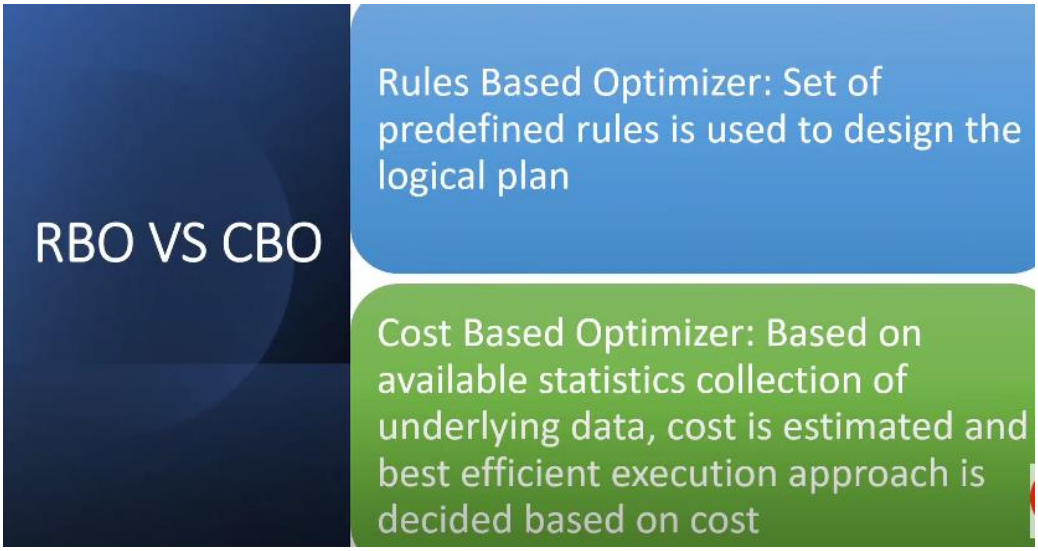
RDD vs Dataframe

- RDD and Dataframe both are same in terms of immutability. Immutability means source dataset cannot be modified. When we apply certain transformation, it will always create new RDD or data frame, but we can never change source data frame. RDD and Dataframe both are working in in-memory which consumes data from the external system then it applies all the transformation it is processing the data in-memory. Resilient: RDD and dataframe both have adopted lazy evaluation model which means when we apply certain transformation, execution will not happen immediately but when we call certain action, it will start applying all the transformations one by one. This is done by DAG. If there is any failure in

building RDD/Dataframe then there is no problem. Using DAG, it can be built once again without any data loss. That is why this is resilient. RDD and dataframe both are consuming data from external system or generating data within its environment then distributing data in the form of partitions across the node. Both RDD and Dataframe are working in distributed computing model.



RBO vs CBO

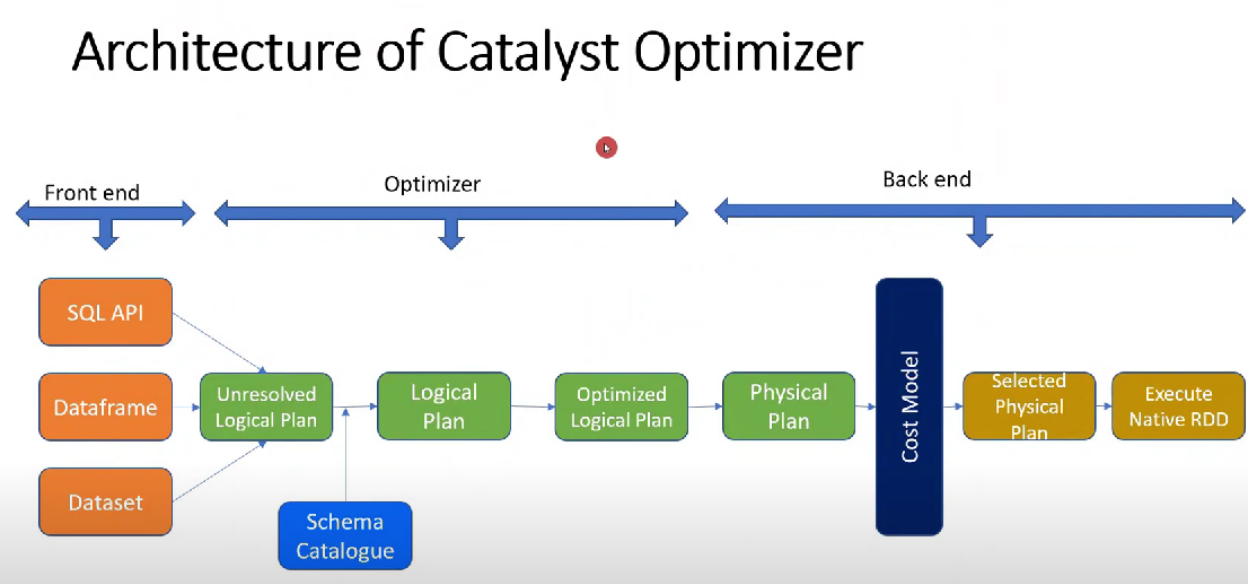


Rules Based Optimizer: When we submit piece of code, let's say select query with where clause .when we submit the query , then execution engine will look at predefined rules, there are already predefined set of rules written in the computing engine, what happens is it will refer predefined rules. based on that it will create logical plan. That we call as Rule Based Optimizer. Rule Based Optimizer will not be suitable for all use cases. Let's say there is 1 application which is processing only few thousands of records with simplest transformation and another application that is handling millions of records with complex transformation so both use cases are different. In this example

have given number of records and complexity of the transformation. But actually, based on N number of parameters , use cases might differ.so this rule based optimizer will blindly follow predefined rules. Based on current situation, optimizer will look for best plan so that is the reason cost-based optimizer got introduced.

Cost Based Optimizer: Cost Based Optimizer it will not blindly follow predefined rules.it will look at the current use case it will refer some data called statistics, when we say statistics, let's consider employee table. In employee table , how many records are there, what is max value, what is min value, how many distinct values we have, how many null values we have. This kind of information will be kept in statistics. Based on statistics, cost-based optimizer will construct the best execution plan. Catalyst Optimizer will take benefit of Rule Based Optimizer and Cost Based Optimizer.

ARCHITECTURE OF CATALYST OPTIMIZER



Architecture of catalyst optimizer segregated in to 3 categories, Front end{this is where we are submitting the user code}, optimizer{major area where catalyst optimizer is contributing that is playing a key role and finally when the best plan is created then it would be submitted to executor for execution to the back end .

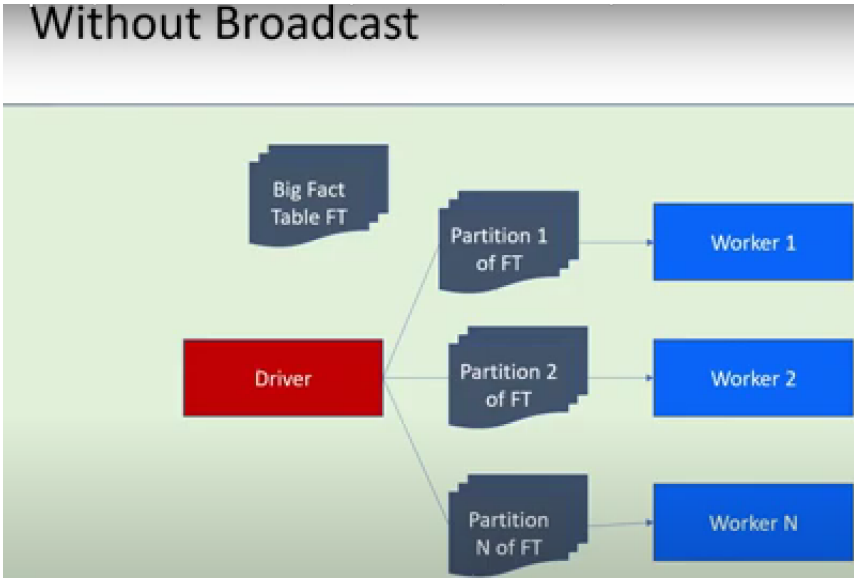
We can develop solution using data frame/SQL API. Let's say we are using SQL API and we are submitting code select * from employee where emp_type='Manager'. When we submit this code for spark engine , first step is it will create unresolved logical plan, i.e., high level plan for the submitted code. Also in this phase, it will just check the semantic and syntax correctness. If we are doing some mistake in the syntax, then computing engine cannot understand what we mean. Semantic verification means we are performing certain operations on employee table, but employee table have been created in this environment before executing this query. In case we are using some object in our query, but that object is not created within the environment then it will throw error it can't execute that we call as semantic. These kinds of parsing activities will be completed in unresolved logical plan stage. Once this plan is created, Logical plan is created from unresolved logical plan. And in the logical plan . In unresolved logical plan, datatype of the column is not derived. For example, in employee table, we are selecting certain columns. If column name is known but we don't know column

data type so column data type will be taken from schema catalogue. In this case we are referring employee table, so it will refer employee schema in the catalog. Based on that it will determine data type and that would be given to next stage Logical Plan. In Logical plan catalyst optimizer is extending Scala program in the form of tree. Tree is one of the Scala data type. Each tree will contain nodes and each node will contain 0 or more child nodes. In each and every node, there will be predefined rules.so here in logical plan stage, rule-based optimizer will play a key role. Rule based optimizer contain list of predefined rules in the form of tree. In each node there will be one rule and this rules can be customized. If we want to add new rule or remove some rule, we can do it. Based on tree model- predefined set of rules ,logical plans will be created. In order to achieve same result , n number of logical plans will be created. Once logical plan is created , it would go to next stage optimized logical plan. Based on the number of logical plans, it will group activities , related activities will be grouped as a micro patch. For example, in order to achieve certain result, there would be mini smaller steps within that and if there are any related/relevant activities then those activities will be grouped as a micro patch. That is one operation done in optimized logical plan stage and 2nd operation is all optimization techniques will be applied i.e., predicate push down/projection push down /rearrange filter those kind of optimization would be applied in optimized logical plan stage . in this stage so far catalyst optimizer understood what to be done. So, coming to physical plan stage , catalyst optimizer will start thinking how to do which means it will start looking at data so Physical plan stage is the place where statistics come to picture . based on statistics all the optimized logical plan will be converted into physical plan. Physical plan will contain steps how to interact with data , how to process actual data . once physical plan is created, cost-based optimizer comes into picture. Cost-based optimizer will start assigning costs. It will start evaluating each physical plan and it will start assigning cost for each model. Based on the cost it will decide which model is containing low cost which means which model will take less time for execution then based on that cost model , it will choose the best efficient plan. Cost based optimizer will pick the best execution query and it will be submitted in the next stage. In the next stage best execution plan that would be converted to java byte code because executor will understand only java code that is run by JVM Process. In this stage ,actual code will be converted to java byte code then java byte code will be submitted to executor for execution.

Summary: When we submit user code what happens is it will first create unresolved logical plan by checking semantic and syntax correctness and in logical plan based on predefined set of rules it will create multiple logical plans and logical plans will be converted as optimized logical plans by applying 2 operations . 1 is grouping relevant operations and creating micro patches and 2nd one is applying all the optimization techniques such as predicate push down. So far it understood what to do (from unresolved logical plan to optimized logical plan). Now it is start thinking how to do . then it will start creating physical plans to process the data. Then cost model will evaluate all the physical plans and it will calculate the cost and based on the cost it will choose the best efficient plan which needs low cost . then once suitable physical plan is selected then that would be converted to java byte code in the next stage and finally it would be submitted to executor for execution.

Broadcast Variable

Broadcast Variable function mostly used for performance tuning. Broadcast Variable is a programming mechanism through which we can keep read-only copy of data into each node of the cluster rather than sending to every node when task needs it. Basically, when we execute some process what happens is task would be send by driver into executors and table would be splitted in the form of partitions and that is distributed across the cluster. Each task will pick up 1 partition and the process would be executed. This is normal process

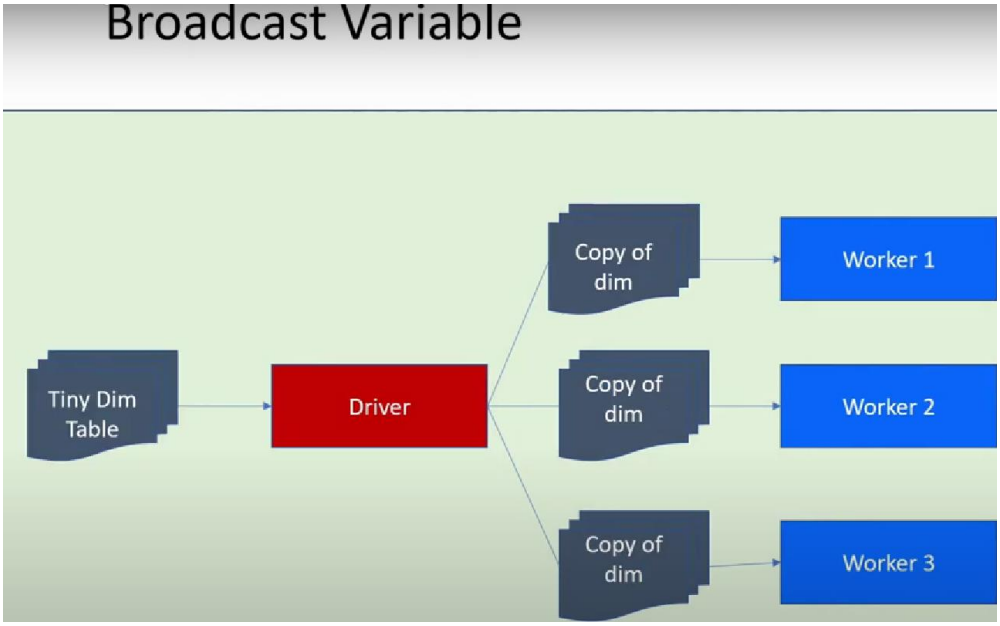


If we are going to process Big Fact Table which contains millions of records. When executor reads data what happens is it would split data into multiple small partitions so let us assume in this case 1 partition is created for each worker node. So, what happens is when drive node sends piece of code in the form of task that would be executed , 1 task will pick 1 partition at a time and it will process. This is regular process.so what happens is 1 particular dataset that could be data frame/table that is splitted into partitions and stored across nodes. This is good for parallel processing but at the same time downside is incase we are doing some joining operation/groupBy operation what happens is that related data will be grouped together so shuffle process will happen.so shuffle is always costly operation so it will hit the performance because data movement will happen across the nodes. This is regular process.

But coming to Broadcast Variable, let us assume we are going to read very small dimension table(containing few 100 or 1000's of records) so what happens is instead of splitting this data into multiple partitions , we can send complete copy of the table into all the worker nodes so what happens process will be very faster because we are joining 1 big fact table with tiny dimension table then dimension table will not be shuffled across this worker nodes. Instead of that we have 1 local copy of this dimensional table into all worker nodes. So, each executor within worker can refer that complete copy of dimension table and it can complete the process without need to shuffle the data across worker nodes. This is advantage of Broadcast Variable.

Summary: so, when we say broad cast variable , it means it will send complete copy of the data into each worker node which means when worker node starts working on the dimensional table first time it will cache the complete dataset into worker node. This

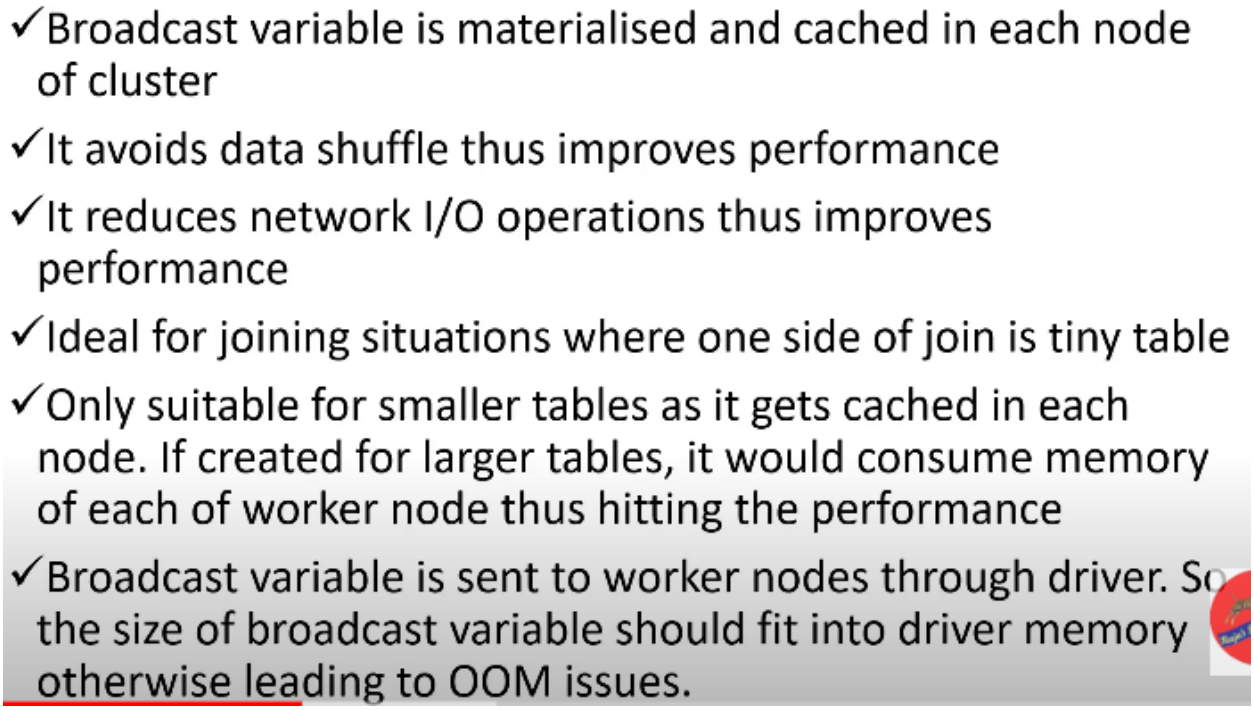
is mainly useful when we are performing joining operations. Let us assume when we are going to write select statement which is joining 2 tables one fact table containing millions of records and another table which is very small tiny dimension table(which contains 100's /few 1000's of records) so what happens is it is not good idea to send fact table or big table into all worker nodes as a broadcast variable because it will completely access the memory within worker nodes.so this smaller table will not make any impact on the memory side.so that is the reason it is always better to send tiny table as a whole copy in the form of cache into worker nodes so that fact table can be splitted and stored across different worker nodes but at the same time dimension table would be saved as a complete copy then each worker node might need to shuffle only fact data but not dimension table thus it will improve the performance



SUMMARY:

- When we have created broadcast Variable which means when executor reads the data for the first time it would cache, and we don't need to send the data again and again with different tasks to the worker nodes. So, when the worker nodes needs data for the first time, it would be cached in the worker node, so it avoids multiple input and output operations to send the data again and again to the worker node.
- Complete copy of the data is stored within 1 worker node which means when it needs to perform certain operation then it does not need to shuffle the data across different nodes because it is already available as a complete copy, so it avoids data shuffle thus improves performance.
- When it does not need to shuffle the data or driver node does not need to send the data along with task each and every time it avoids input output operations through network, so it improves the performance
- This is suitable for tiny tables, but we should be careful in case we are applying broadcast Variable for bigger table what happens is one complete copy of data would be sent to each worker node and it will completely exhaust the memory then it will hit the performance .

Summary



- Complete table will be sent to all worker nodes through driver. Size of tiny dimension table should be smaller than size of the driver. Let us assume driver is containing 12 GB, table containing 15 GB then what happens is it will completely collapse the program and driver will throw out of memory error. So, size of broadcast variable should be less than size of driver memory.

Code: **#Create Sample Dataframe – we can assume this as fact table**

```
Transaction =[
    (100,'Cosmetic',150),
    (200,'Apparel',250),
    (300,'Shirt',400),
    (400,'Trouser',500),
    (500,'Socks',20),
    (100,'Belt',70),
    (200,'Cosmetic',250),
    (300,'Shoe',400),
    (400,'Socks',25),
    (500,'Shorts',100)
]
transactionDF= spark.createDataFrame(Transaction,['Store_ID','Item','Amount'])
transactionDF.show()
```

(3) Spark Jobs

- Job 0 View (Stages: 1/1)
- Job 1 View (Stages: 1/1)
- Job 2 View (Stages: 1/1)

transactionDF: pyspark.sql.dataframe.DataFrame = [Store_ID: long, Item: string ... 1 more field]

Store_ID	Item	Amount
100	Cosmetic	150
200	Apparel	250
300	Shirt	400
400	Trouser	500
500	Socks	20
100	Belt	70
200	Cosmetic	250
300	Shoe	400
400	Socks	25
500	Shorts	100

#Create Sample Dimension Table

```
Store = [
    (100,'Store_London'),
    (200,'Store_Paris'),
    (300,'Store_Frankfurt'),
    (400,'Store_Stockholm'),
    (500,'Store_Oslo')
]
storeDF=spark.createDataFrame(Store,['Store_ID','Store_Name'])
storeDF.show()
```

(3) Spark Jobs

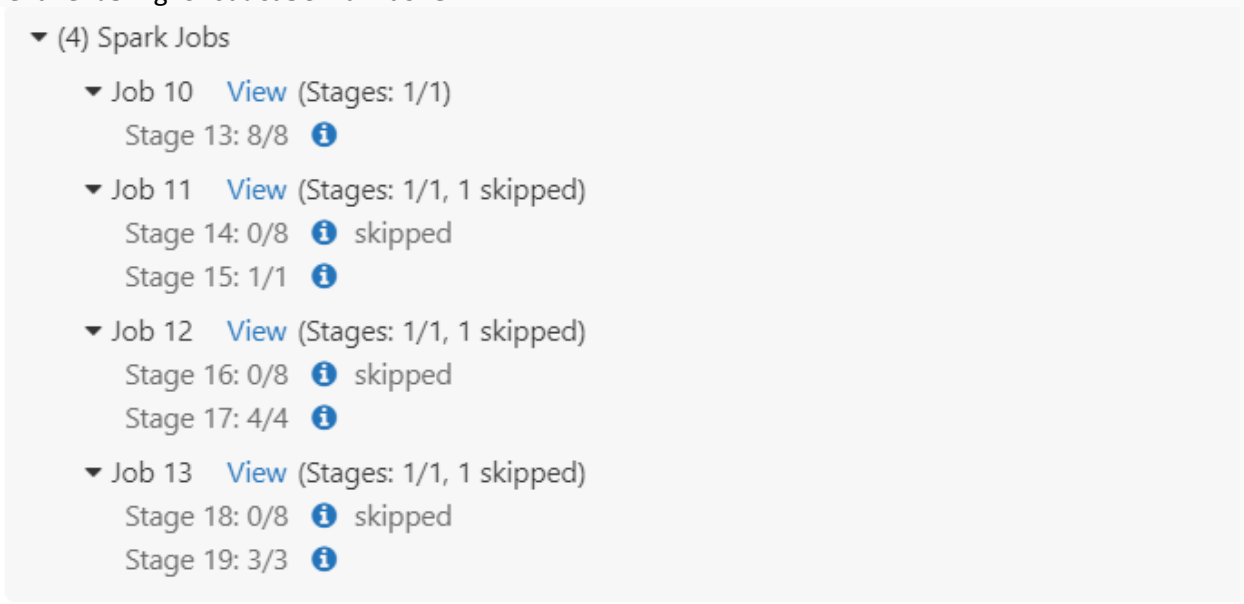
- Job 3 View (Stages: 1/1)
- Job 4 View (Stages: 1/1)
- Job 5 View (Stages: 1/1)

storeDF: pyspark.sql.dataframe.DataFrame = [Store_ID: long, Store_Name: string]

Store_ID	Store_Name
100	Store_London
200	Store_Paris
300	Store_Frankfurt
400	Store_Stockholm
500	Store_Oslo

```
from pyspark.sql.functions import broadcast
joinDF=transactionDF.join(broadcast(storeDF),transactionDF.Store_ID==storeDF.Store_ID
)
joinDF.show()
```

Here storeDF is a tiny table that is going to have few thousands of records, in this example 5 records . so, I want to send this data to all worker nodes so performance can be better. In this example , very less data so there can't be performance improvement. When we are working in real time project, we can see big difference when we are using broadcast variable



joinDF: pyspark.sql.dataframe.DataFrame = [Store_ID: long, Item: string ... 3 more fields]

Store_ID	Item	Amount	Store_ID	Store_Name
100	Cosmetic	150	100	Store_London
200	Apparel	250	200	Store_Paris
300	Shirt	400	300	Store_Frankfurt
400	Trouser	500	400	Store_Stockholm
500	Socks	20	500	Store_Oslo
100	Belt	70	100	Store_London
200	Cosmetic	250	200	Store_Paris
300	Shoe	400	300	Store_Frankfurt
400	Socks	25	400	Store_Stockholm
500	Shorts	100	500	Store_Oslo

If we want to see explain plan for joinDF , then

`joinDF.explain()`

when we run above query, we can see physical plan only

```
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=true
+- == Final Plan ==
   *(2) BroadcastHashJoin [Store_ID#2L], [Store_ID#21L], Inner, BuildRight, false
   :- *(2) Filter isnotnull(Store_ID#2L)
   :  +- *(2) Scan ExistingRDD[Store_ID#2L,Item#3,Amount#4L]
   +- ShuffleQueryStage 0, Statistics(sizeInBytes=200.0 B, rowCount=5, isRuntime=true)
      +- Exchange SinglePartition, EXECUTOR_BROADCAST, [plan_id=325]
         +- *(1) Filter isnotnull(Store_ID#21L)
            +- *(1) Scan ExistingRDD[Store_ID#21L,Store_Name#22]
+- == Initial Plan ==
   BroadcastHashJoin [Store_ID#2L], [Store_ID#21L], Inner, BuildRight, false
   :- Filter isnotnull(Store_ID#2L)
   :  +- Scan ExistingRDD[Store_ID#2L,Item#3,Amount#4L]
   +- Exchange SinglePartition, EXECUTOR_BROADCAST, [plan_id=290]
      +- Filter isnotnull(Store_ID#21L)
         +- Scan ExistingRDD[Store_ID#21L,Store_Name#22]
```

If we want to see complete stages, then we can send parameter to True in explain() command.

`joinDF.explain(True)` → we can see all logical plans, Parsed,Analyzed,Optimized Logical plans and Physical plan

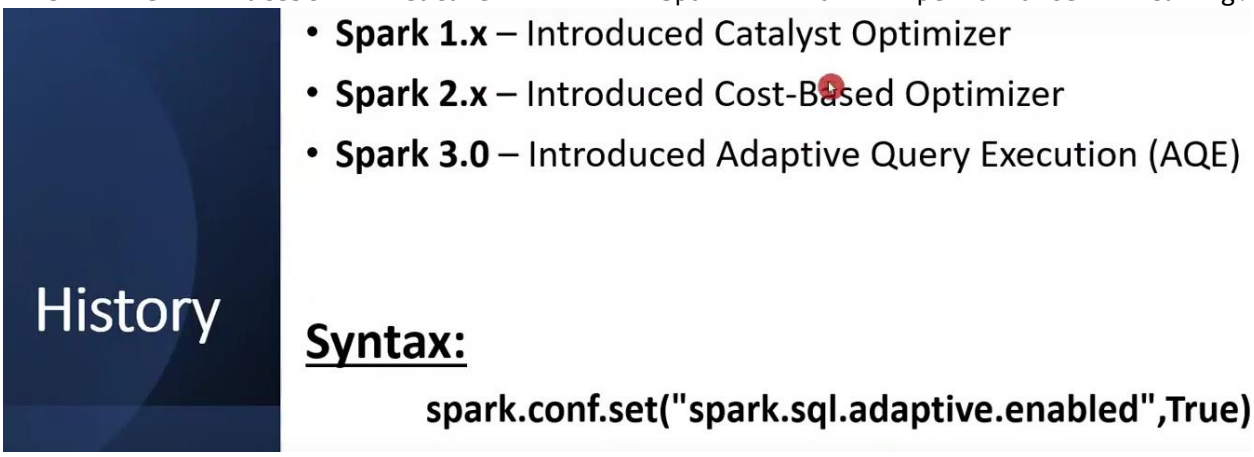
```
== Parsed Logical Plan ==
Join Inner, (Store_ID#2L = Store_ID#21L)
:- LogicalRDD [Store_ID#2L, Item#3, Amount#4L], false
+- ResolvedHint (strategy=broadcast)
   +- LogicalRDD [Store_ID#21L, Store_Name#22], false

== Analyzed Logical Plan ==
Store_ID: bigint, Item: string, Amount: bigint, Store_ID: bigint, Store_Name: string
Join Inner, (Store_ID#2L = Store_ID#21L)
:- LogicalRDD [Store_ID#2L, Item#3, Amount#4L], false
+- ResolvedHint (strategy=broadcast)
   +- LogicalRDD [Store_ID#21L, Store_Name#22], false

== Optimized Logical Plan ==
Join Inner, (Store_ID#2L = Store_ID#21L), rightHint=(strategy=broadcast)
:- Filter isnotnull(Store_ID#2L)
:  +- LogicalRDD [Store_ID#2L, Item#3, Amount#4L], false
+- Filter isnotnull(Store_ID#21L)
   +- LogicalRDD [Store_ID#21L, Store_Name#22], false

== Physical Plan ==
```

```
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=true
+- == Final Plan ==
   *(2) BroadcastHashJoin [Store_ID#2L], [Store_ID#21L], Inner, BuildRight, false
   :- *(2) Filter isnotnull(Store_ID#2L)
   :  +- *(2) Scan ExistingRDD[Store_ID#2L,Item#3,Amount#4L]
   +- ShuffleQueryStage 0, Statistics(sizeInBytes=200.0 B, rowCount=5, isRuntime=true)
      +- Exchange SinglePartition, EXECUTOR_BROADCAST, [plan_id=325]
         +- *(1) Filter isnotnull(Store_ID#21L)
            +- *(1) Scan ExistingRDD[Store_ID#21L,Store_Name#22]
+- == Initial Plan ==
   BroadcastHashJoin [Store_ID#2L], [Store_ID#21L], Inner, BuildRight, false
   :- Filter isnotnull(Store_ID#2L)
   :  +- Scan ExistingRDD[Store_ID#2L,Item#3,Amount#4L]
   +- Exchange SinglePartition, EXECUTOR_BROADCAST, [plan_id=290]
      +- Filter isnotnull(Store_ID#21L)
         +- Scan ExistingRDD[Store_ID#21L,Store_Name#22]
```

In Physical plan, query is getting executed by executor where we can see broadcast join is getting applied.

PERFORMANCE TUNING CONCEPT IN SPARK- ADAPTIVE QUERY EXECUTION

This is latest feature in spark for performance tuning.



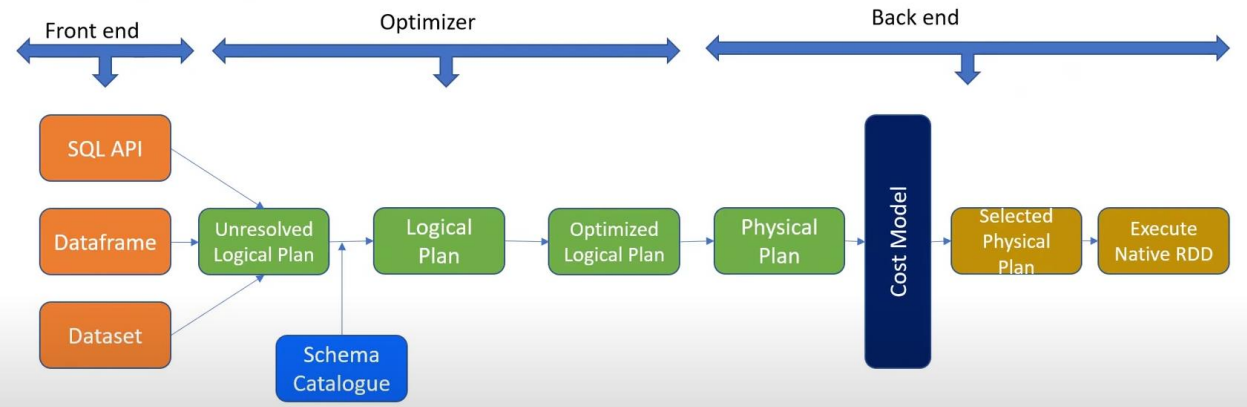
Initially spark introduced catalyst optimizer will rule based optmizer which means based on the predefined set of rules execution plan will be created but in later version spark2.0 onwards it introduced cost based optimizer.Cost based optimizer would actually check statistics collected for the actual data then based on that it will plan the better plan compared to rule based optimzer. In latest version of spark spark 3.0 onwards it introduced one more latest advanced feature – Adaptive Query Execution.

Syntax: **spark.conf.set(“spark.sql.adaptive.enabled”,True)**

If we have to use Adaptive Query execution in our programming then we can enable by using above property. In order to set any spark properties , we can use with this keyword spark.conf.set

From spark 3.0 onwards Adaptive Query execution(AQE) got enabled but if we are using previous version of spark then we need to manually enable using above syntax . If we want to turn off the feature, we can give value as False in above syntax.

ARCHITECTURE OF SPARK OPTIMIZER:



When user submits the code using SQL API /Dataframe into spark engine, the optimizer will start picking the code then it will start creating unresolved Logical Plan, that is the first stage. In this stage optimizer will check syntactic and semantic correctness. If it is passed, then control goes to next stage i.e., logical plan. In unresolved logical plan , it received objects either data frame/tables from the user code but it still doesn't have data type of all objects/columns. So, when it starts pulling logical plan it refers schema catalogue and it will create data type and this logical plan would employee rule-based optimizer for creating many logical plans and rule-based optimizer is kind of tree data type. In tree each node will contain predefined set of rules. Based on predefined set of rules logical plan will be created .Once logical plans are created, then optimization process will start on the created logical plans. In optimized stage it will apply optimization such as predicate push down or changing the order of join conditions, those kinds of optimizations would be applied and also it will start grouping relevant activities into smaller patches. Once optimized logical plan is completed then physical plan will start. In the physical plan, cost-based optimizer CBO will comes to picture. It will start using collected statistics .Based on statistics it will start building physical plan and also on physical plan it will start assigning cost for each task each activity within the plan. Once physical plan is completed and cost is also allocated, then cost based optimizer will choose low-cost plan which means high performance plan (execution plan will perform quickly, will select that plan ,once physical plan is selected then it will give to executor for execution. Once physical plan is selected it will stick to that plan it will not change the plan.so as per physical plan it will convert that as a job and in the job, there could be 10 stages. Plan is already created based on the statistics and all the stages let's say there are 10 stages in the physical plan, it will start executing one by one as per the selected physical plan.

In the existing spark optimizer, cost-based optimizer will look at the collected statistics. Let's say statistics got collected yesterday night, today we are executing query to night so what happens is there is 1 day gap , so there can be many changes in the data base so there can be changes in statistics . so, this cost-based optimizer will not use latest inaccurate statistics , it will always use last collected statistics

so if there is any time gap between the last collected statistics and changes to the database /actual data then it might not be very accurate. So that's the reason AQE got introduced and AQE will start optimizing the query between stages. Without AQE, one physical plan got selected and engine will start executing plan one by one . There are 10 stages let's assume it will start executing one by one but when we enable AQE what happens is selected physical plan it will start executing let's say there are 10 stages and first stage is getting executed at the end of first stage it will write intermittent result of that particular stage into the cluster then AQE will start examining that output of the first stage then based on that it will again start optimizing the plan .that query coming from the intermediate stages that is actual data it will give very accurate statistics. So adaptive query is during execution time , dynamically it will reoptimize the query based on the accurate data(real time data) that is called AQE.

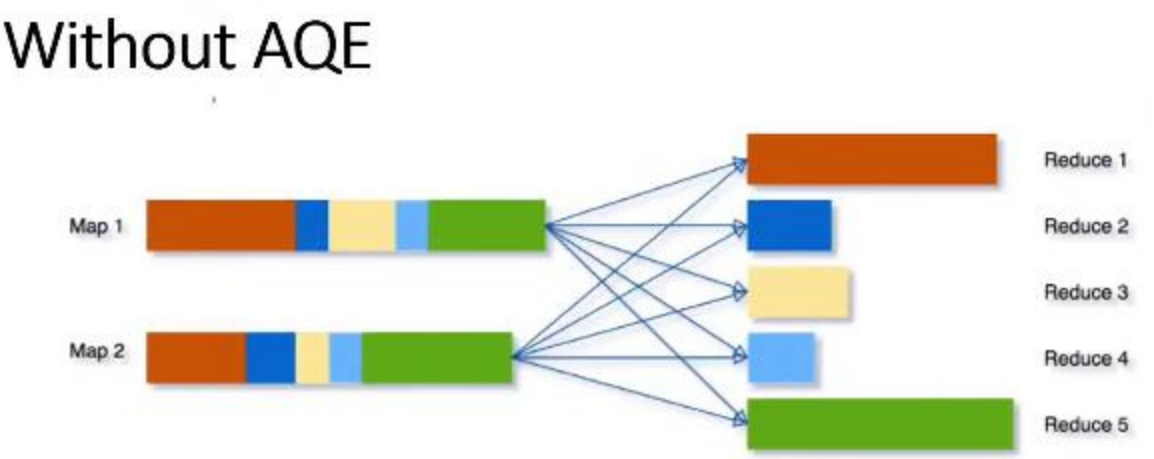
AQE FEATURES:

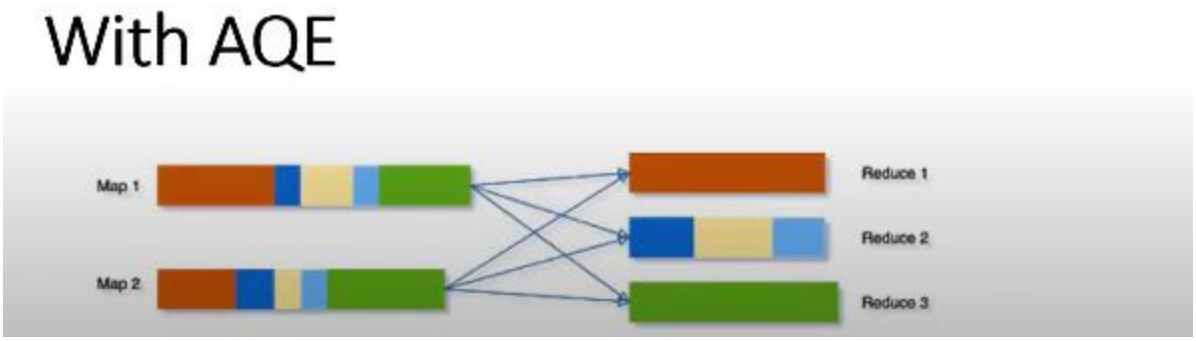
- **DYNAMICALLY COALESCING SHUFFLE PARTITIONS**

We can tune shuffle partitions according to our use case. By default, value will be 200 but that default value is not suitable for all use cases. Depending on our use case we should always change the shuffle partition but when we calculate manually it might not be accurate but when we enable this AQE it will change the shuffle partition during dynamically so it will improve the performance . For example, when we manually configure this shuffle partitions in case if we are going to choose too few partitions which means each partition will be in bigger size that is not good because each core in the executor will take more time for execution and also large partitions leads to data spillage into the disk so that it will hit the performance .

On the other hand, if we choose too many partitions let us say there is 1GB data, but we have created 1000 partitions what will happen is it will create 1MB size of 1000 partitions but that is also not good we should have optimal size of partition. If we are going to choose too many partitions, then size of each partition will be very small that is leading to network I/O operations to shuffle the data .when we are performing certain operations the data would be splitted across multiple partitions then it should collect into it should shuffle data in order to perform certain tasks then it will slow down the performance. It is always difficult to measure and set the number of partitions manually but when we enable AQE it will dynamically choose best number and it will improve the performance.

The below screenshot shows pictorial representation. There are 2 mappings. Let us understand orange colour containing more data , same for green but for other it is smaller in size but in normal cases what happens is each core in the executor will pick up 1 particular partition and it will start processing and it will take more time for orange and green but other partitions other cores will complete task very quickly and will sit idle so that is not good for performance . so, when we enable AQE, it will start merging smaller partitions into one. That is called coalescing . coalescing will decrease number of partitions. In original (without AQE) number of partitions are 5.After applying coalescing through AQE,it will be only 3.so this will improve the performance.





- **DYNAMICALLY SWITCHING JOIN STRATEGIES:**

In Broadcast join , one big fact table containing millions of records and another smaller dimension table containing only few 100's of data then what happens is spark can broadcast that variable into all executor nodes which means instead of distributing that smaller size table into different nodes it will send complete copy of the data into each node. Each node in the cluster would contain one local copy of the complete data . it is very smaller in size so there won't be any impact to the memory. At the same time, it will improve the performance because it will avoid shuffle data because each node will contain complete set of lookup data . this is broadcast join

Broadcast join is not suitable if both tables are not smaller in size. If we are going to broadcast bigger table, it will exhaust memory in the worker node .

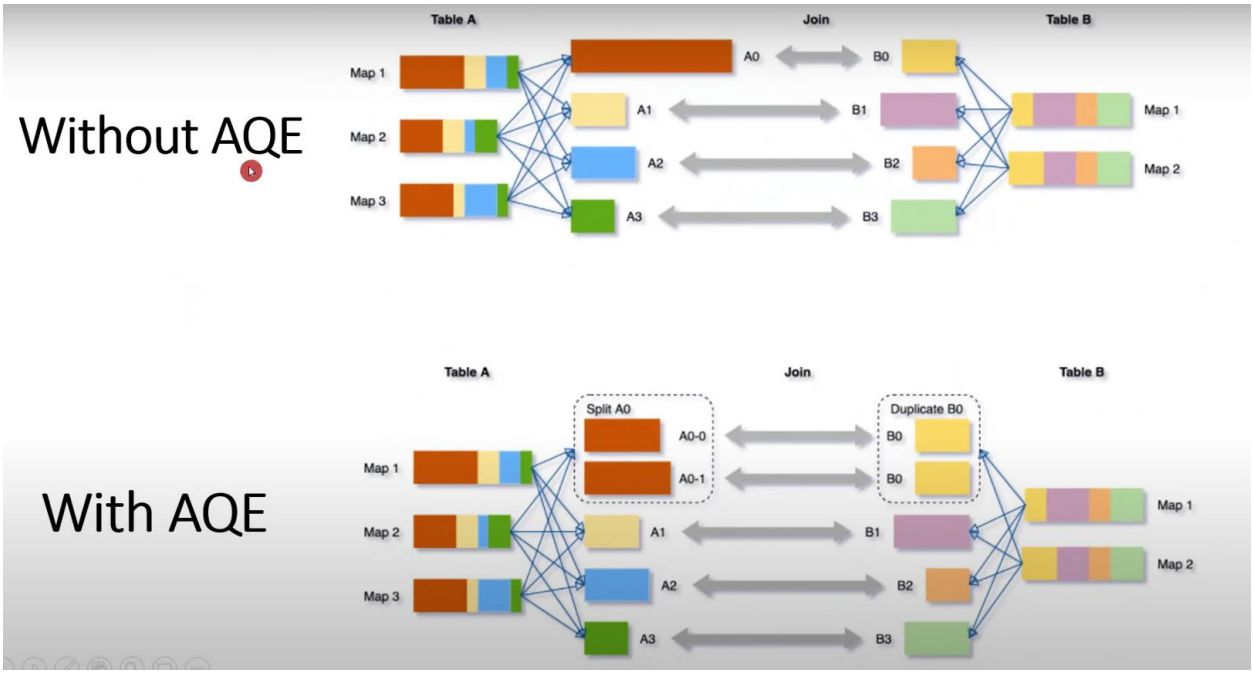
Shuffle join is costly but always not suitable for use-cases same for broadcast. Broadcast is high performance but at the same time not suitable if both tables are big in size.

So, when we enable AQE,automatically it will choose which join method is suitable for particular use case . based on that it will use either shuffle join(sort-merge join) or broadcast join.

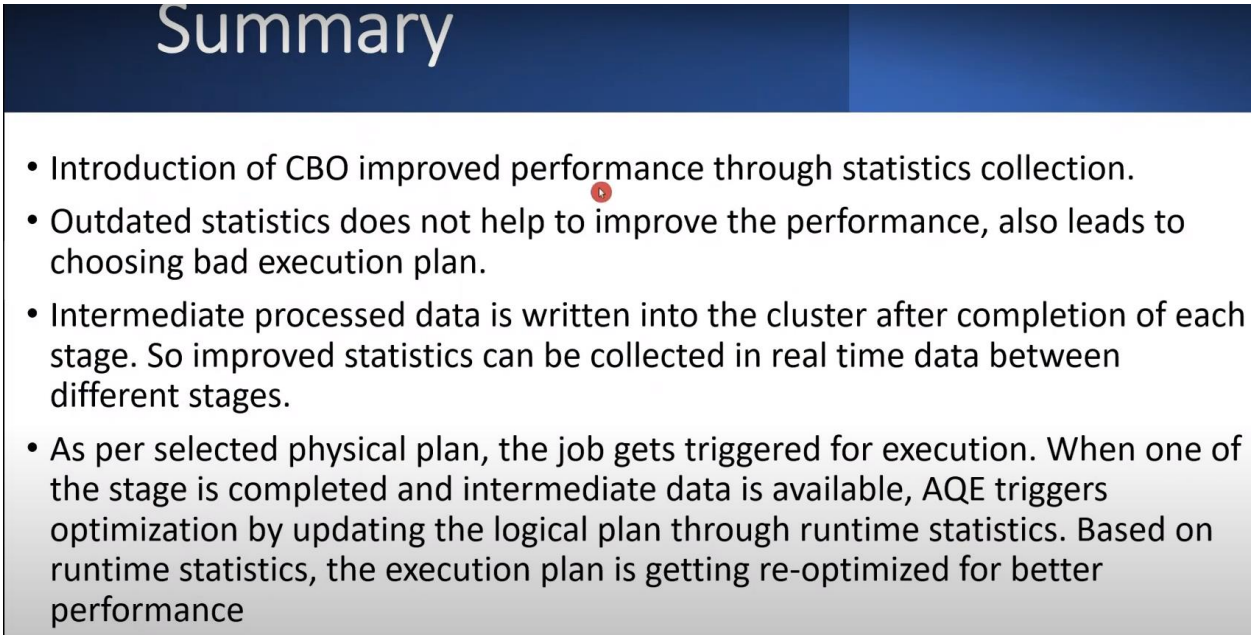
- **DYNAMICALLY OPTIMIZING SKEW JOINS**

Data skew is when we distribute data unevenly then that is called Data Skew. For Example, let us say we are receiving sales data, we have data frame that is containing sales data and 80% of sales is coming from 1 country and 20% of the transaction is

coming from few other 5 countries . In this case if we are going to partition based on country ,what happen is 80% of data will go to one partition and remaining 20% of data will be splitted into 4/5 partition. Here it will create data skew which means 1 partition will be very big in size and other partition will be smaller in size . when we compared all partitions, it will be unevenly distributed . so, in this example one particular partition is containing 80% of the data then what happens when one core executor starts picking this partition ,it will run for longer duration but at the same time other core executors will complete the process very quickly because other partitions are very smaller still other executors will sit idle but still overall process will not be completed until we process that bigger partition so overall it will hit the performance time . so, it's very important not to create data skew. But in case there is data skew there are some concepts we can manually optimize that by some concepts like bucketing or salting . when we enable AQE, automatically it will avoid the data skew .In case in this example if 80% of data goes to 1 particular partition then spark will apply 1 technique internally then it will split data into multiple smaller partitions that is almost equivalent to other partitions . so that is concept of dynamically optimizing skew joins.



The above screenshot shows pictorial representation . Let us say there are 2 tables table A and table B. In without AQE, it will take more time for execution. The executor will take more time to process this one and overall, it will hit the performance because other executors or other cores in the executor should wait for the completion of this particular partition also then only overall process would get completed .so it will increase the overall processing time . so, when we enable this AQE what happens is it will split the bigger partition into multiple smaller pieces and also it will create duplicate of equivalent dimension table . so, in this way data is distributed across multiple cores/executors uniformly then it will improve the performance



Introduction of CBO improved performance through statistics collection but at the same time in CBO,it will always consider latest collected statistics.it will not be very accurate with the real time data so in case if there is huge change between collected statistics and real time data then it will lead to choosing the bad execution plan.so it might not improve the performance and in some cases it might lead to very poor performance plan. So, CBO is good, but it is not suitable for few cases. But when we enable AQE that will get rid of downside of CBO.it will always use statistics from real time data. Based on that it will reoptimize the query between each stages and intermediate process data is written into cluster after completion of each stage. Once intermediate data is materialized ,based on that AQE will start collecting statistics and based on collective statistics it will reoptimize the query between each stages.

In the old plan if physical plan is selected it will stick into one particular plan and based on that it will execute . but when we enable AQE even though it was selected one physical plan, but it will dynamically change the plan as per accurate/real time statistics.

HANDLING NULL VALUES IN PYSPARK

isNull() This function mainly used to filter all the rows where certain column is containing null values

isNotNull() This function mainly used to filter all records where column is not containing null values.

```
data_student =[("Michael","Science",80,"P",90),
               ("Nancy","Mathematics",90,"P",None),
               ("David","English",20,"F",80),
               ("John","Science",None,"F",None),
               ("Blessy",None,30,"F",50),
               ("Martin","Mathematics",None,None,70)]
schema = ["Name","Subject","Mark","Status","Attendance"]
df= spark.createDataFrame(data_student,schema)
```

```
display(df)
```

(3) Spark Jobs

- Job 0 View (Stages: 1/1)
- Job 1 View (Stages: 1/1)
- Job 2 View (Stages: 1/1)

df: pyspark.sql.dataframe.DataFrame = [Name: string, Subject: string ... 3 more fields]

Table +

	Name	Subject	Mark	Status	Attendance
1	Michael	Science	80	P	90
2	Nancy	Mathematics	90	P	null
3	David	English	20	F	80
4	John	Science	null	F	null
5	Blessy	null	30	F	50
6	Martin	Mathematics	null	null	70

6 rows | 14.94 seconds runtime

```
display(df.filter(df.Mark.isNull())) or display(df.filter("Mark IS NULL")) or

from pyspark.sql.functions import col
display(df.filter(col("Mark").isNull()))
```

(3) Spark Jobs

- Job 3 View (Stages: 1/1)
- Job 4 View (Stages: 1/1)
- Job 5 View (Stages: 1/1)

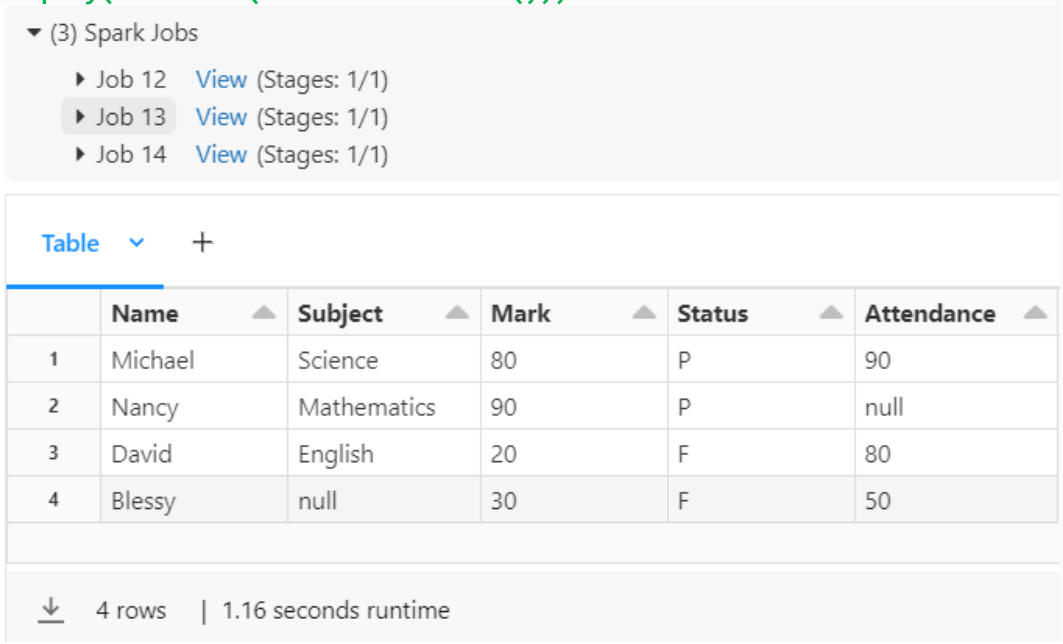
Table +

	Name	Subject	Mark	Status	Attendance
1	John	Science	null	F	null
2	Martin	Mathematics	null	null	70

2 rows | 1.54 seconds runtime

Want to return all records where column is not containing null value

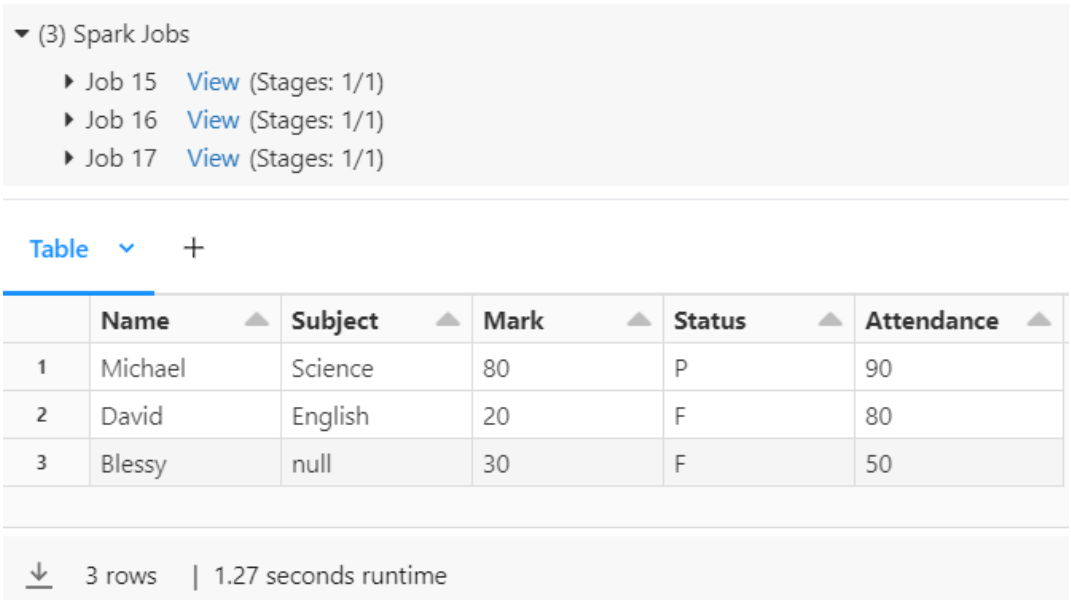
```
display(df.filter(df.Mark.isNotNull()))
```



	Name	Subject	Mark	Status	Attendance
1	Michael	Science	80	P	90
2	Nancy	Mathematics	90	P	null
3	David	English	20	F	80
4	Blessy	null	30	F	50

We can combine multiple conditions also

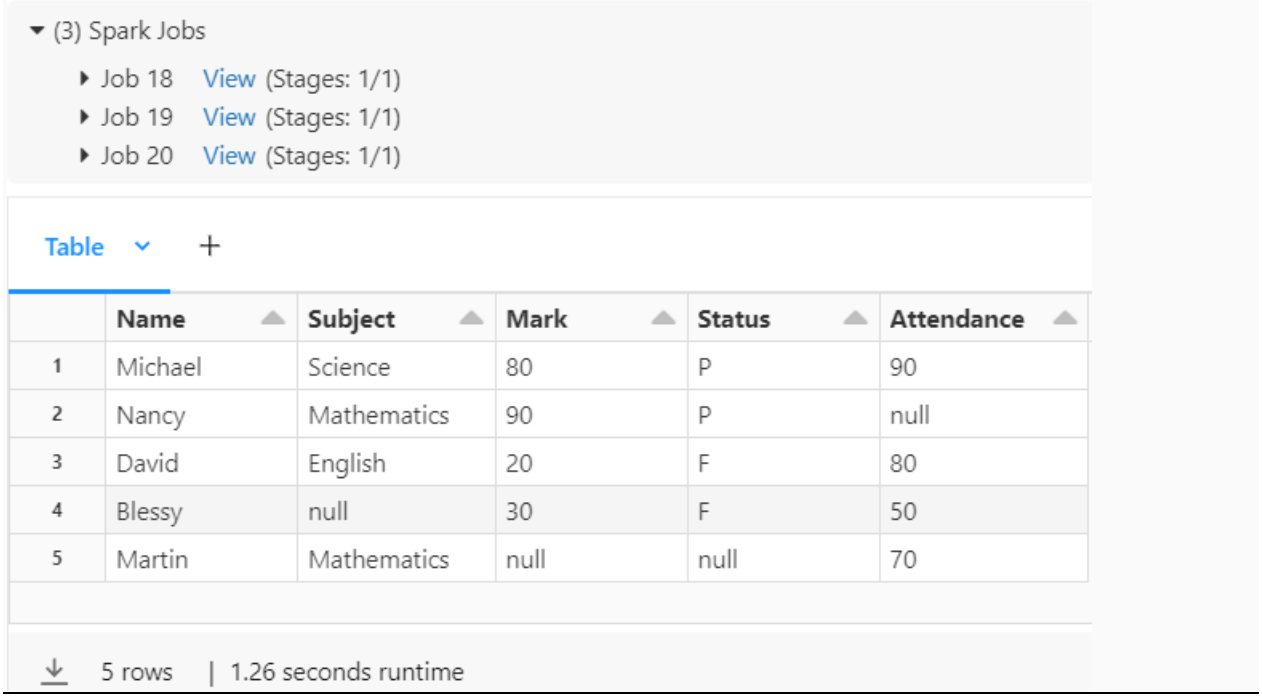
```
display(df.filter(df.Mark.isNotNull() & df.Attendance.isNotNull()))
```



	Name	Subject	Mark	Status	Attendance
1	Michael	Science	80	P	90
2	David	English	20	F	80
3	Blessy	null	30	F	50

If any one of the condition is satisfying, then it will return result.it wont check both conditions to be true . for any one of the condition either 1 or 2 condition , result will be returned.

```
display(df.filter(df.Mark.isNotNull() | df.Attendance.isNotNull()))
```



	Name	Subject	Mark	Status	Attendance
1	Michael	Science	80	P	90
2	Nancy	Mathematics	90	P	null
3	David	English	20	F	80
4	Blessy	null	30	F	50
5	Martin	Mathematics	null	null	70

DROP NULL VALUES IN PYSPARK

na.drop() or dropna() function : This function drops rows if any one or all columns contain null value

FILL MISSING VALUES IN PYSPARK

na.fill() or fillna() function: If we want to populate dummy / default value for null values in the data frame then we can use this function na.fill() or fillna(). Instead of dropping, if we don't want to drop, we can replace null with some other default/dummy value.

df.na.fill(value=0)→ wherever null value is there, it will change to 0 only for numerical columns and string value for the string columns only.

```
data_student =[("Michael","Science",80,"P",90),
               ("Nancy","Mathematics",90,"P",None),
               ("David","English",20,"F",80),
               ("John","Science",None,"F",None),
               ("Martin","Mathematics",None,None,70),
               (None,None,None,None,None)]
schema = ["Name","Subject","Mark","Status","Attendance"]
df= spark.createDataFrame(data_student,schema)
display(df)
```

(3) Spark Jobs

- Job 21 View (Stages: 1/1)
- Job 22 View (Stages: 1/1)
- Job 23 View (Stages: 1/1)

df: pyspark.sql.dataframe.DataFrame = [Name: string, Subject: string ... 3 more fields]

Table +

	Name	Subject	Mark	Status	Attendance
1	Michael	Science	80	P	90
2	Nancy	Mathematics	90	P	null
3	David	English	20	F	80
4	John	Science	null	F	null
5	Martin	Mathematics	null	null	70
6	null	null	null	null	null

6 rows | 1.93 seconds runtime

Command took 1.93 seconds -- by paladugusravanthi005@gmail.com at 12/14/2023, 11:49:37 AM on Tra

#Drop the records with Null value – ALL & ANY

`display(df.dropna()) or display(df.dropna("any"))` → if null is part of any column then that record/rows got removed.

(3) Spark Jobs

- Job 24 View (Stages: 1/1)
- Job 25 View (Stages: 1/1)
- Job 26 View (Stages: 1/1)

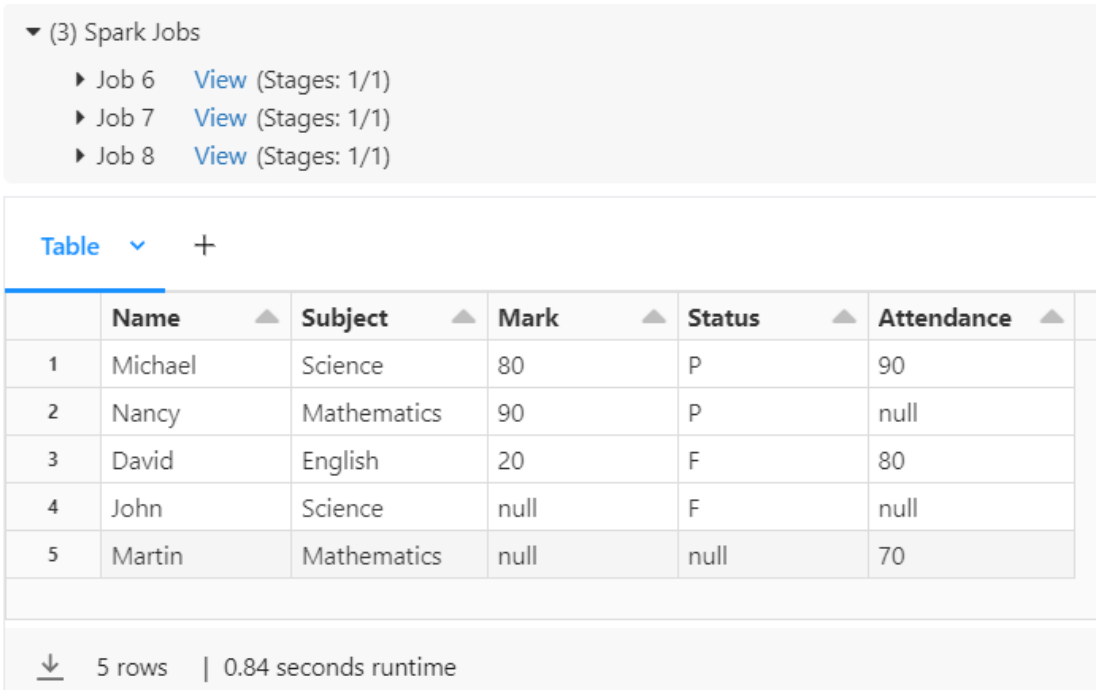
Table +

	Name	Subject	Mark	Status	Attendance
1	Michael	Science	80	P	90
2	David	English	20	F	80

2 rows | 1.12 seconds runtime

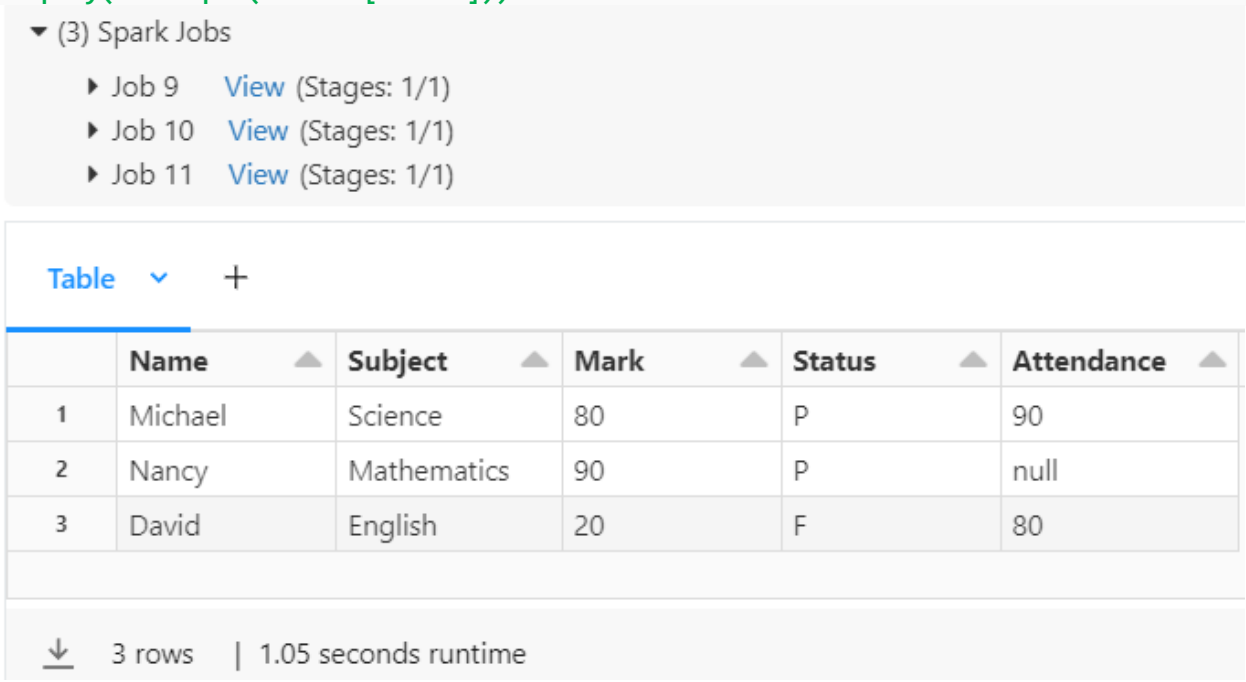
Command took 1.12 seconds -- by paladugusravanthi005@gmail.com at 12/14/2023, 11:53:27 AM on Tra

`display(df.dropna("all"))` → If all the values in record is null then it will remove only that record .

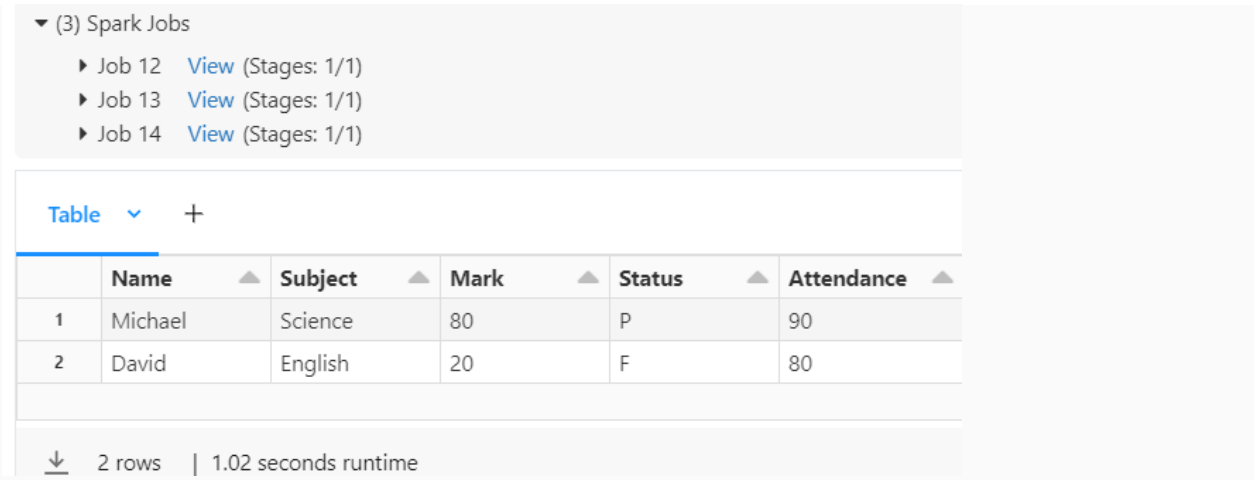


#How to remove records based on particular column

```
display(df.dropna(subset=["Mark"]))
```

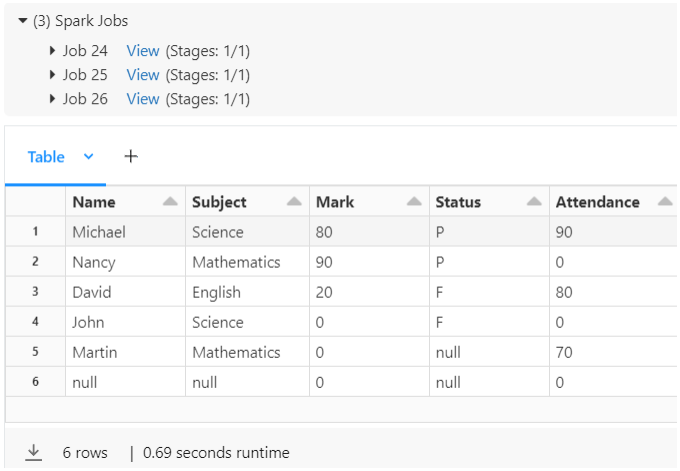


`display(df.dropna(subset=["Mark","Attendance"]))` →It will remove records where column Mark containing null and column attendance containing null

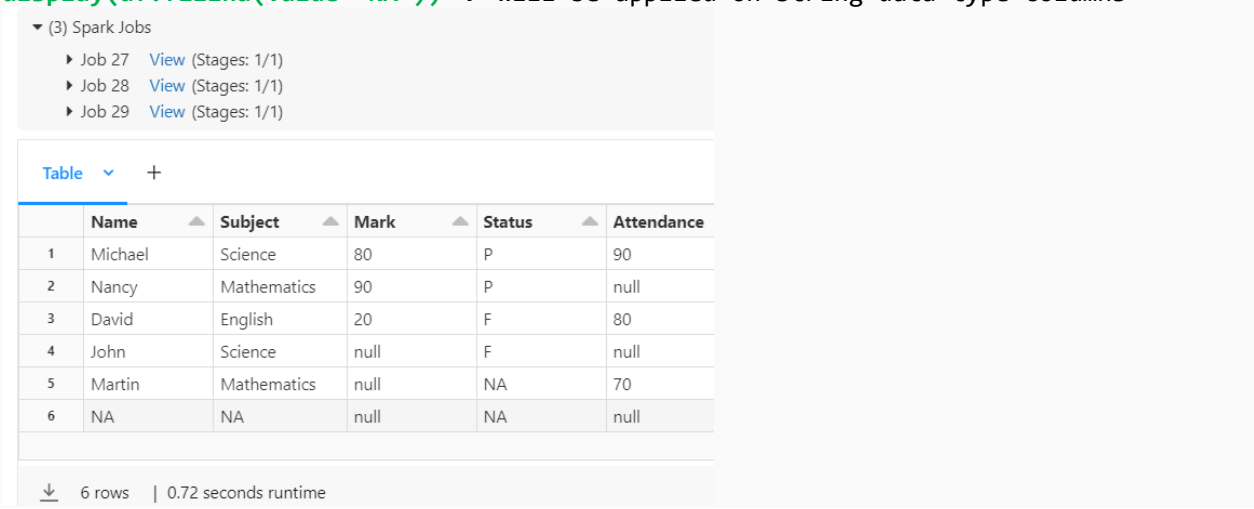


#Fill value for all columns if Null is present

`display(df.fillna(value=0))` or `display(df.na.fill(value=0))` → will be applied on integer data type columns only.



`display(df.fillna(value="NA"))` → will be applied on string data type columns



```
#Fill value for specific columns if contains null
display(df.fillna(value=0),subset=["Mark","Attendance"])
```

(3) Spark Jobs

- Job 33 View (Stages: 1/1)
- Job 34 View (Stages: 1/1)
- Job 35 View (Stages: 1/1)

Table +

	Name	Subject	Mark	Status	Attendance
1	Michael	Science	80	P	90
2	Nancy	Mathematics	90	P	0
3	David	English	20	F	80
4	John	Science	0	F	0
5	Martin	Mathematics	0	null	70
6	null	null	0	null	0

6 rows | 0.81 seconds runtime

```
display(df.fillna({"Name":"No_name","Subject":"English","Mark":0,"Status":"NA","Attendance":50}))
```
→ If we want to keep different default value for each column, then

(3) Spark Jobs

- Job 36 View (Stages: 1/1)
- Job 37 View (Stages: 1/1)
- Job 38 View (Stages: 1/1)

Table +

	Name	Subject	Mark	Status	Attendance
1	Michael	Science	80	P	90
2	Nancy	Mathematics	90	P	50
3	David	English	20	F	80
4	John	Science	0	F	50
5	Martin	Mathematics	0	NA	70
6	No_name	English	0	NA	50

6 rows | 0.96 seconds runtime

UDF: UDF is a piece of code/program which is used to perform certain tasks .

WHERE DO WE USE UDF? Basically, we use UDF to avoid writing same code again and again. Basically, UDF creates reusable code .Let's say we have to call certain function certain action needs to be called again and again in multiple places. Instead of writing code again and again in multiple places, we can write UDF and we can call it in multiple places. UDF uses to maintain modularity of our programming/development.

Syntax starts with def key word for defining UDF and UDF_NAME is user defined function name it could be anything and within brackets we will pass list of parameters to UDF, and we will keep : and in the body we will keep piece of code which is used to perform certain task . once coding is done, programming is developed then finally it ends with keyword return and it will return output

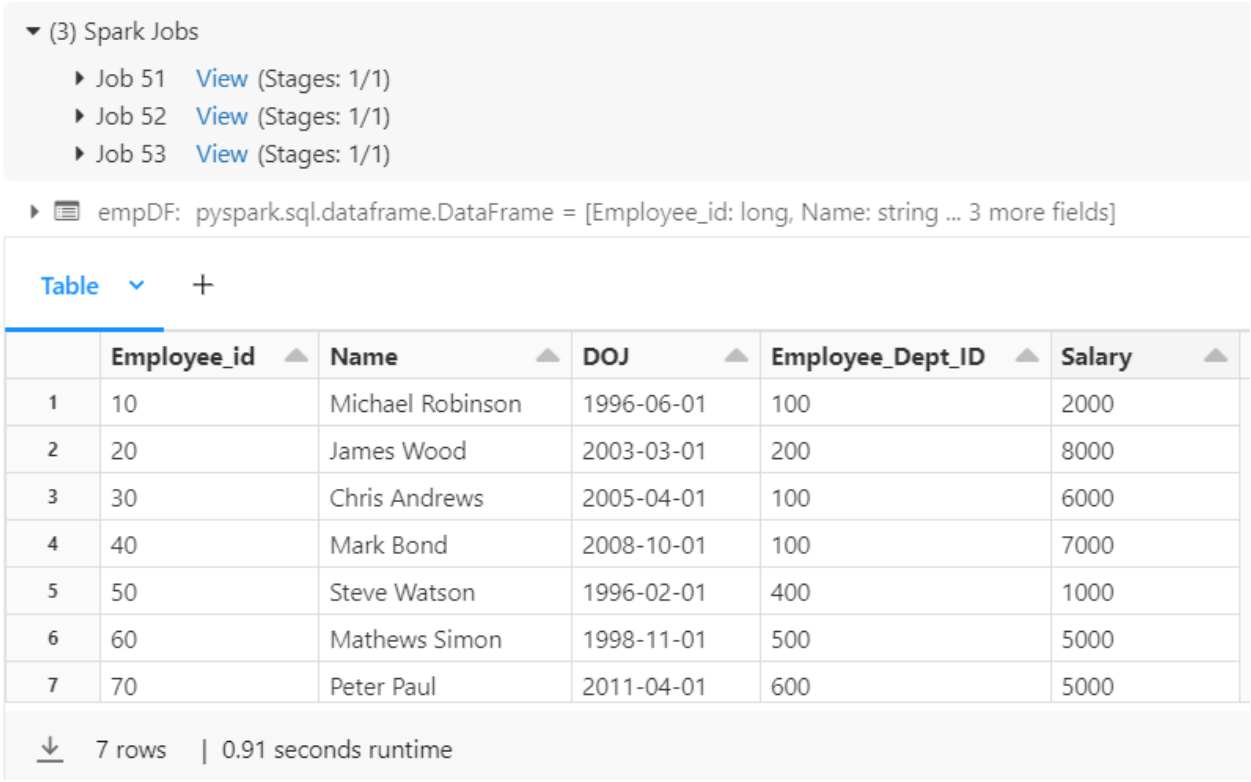
Syntax: **def UDF_NAME(parameters):**

--Code to perform the task

** return RETURN_OUTPUT.**

UDF is a black box from spark engine which means UDF is not recommended for data bricks development or spark development because UDF is black box for spark engine which means spark engine cannot apply any optimization on user defined function so that is the reason UDF is expensive operation in spark development . UDF is expensive because when we are developing user defined function using let's say python which means user defined function that would be python object, but this function would be applied on particular dataset/data frame. Data frame that is actually JVM object which means in Java format.so when UDF has to perform certain action/certain task on the data each and every time the data should be serialized and deserialized because function it's in python format, but data is in java format. So, task can be executed by multiple java API calls .so for each call data to be serialized and deserialized so that program python and java can understand each other so that is the reason UDF is expensive operation so it's always better to avoid UDF and in python there are rich libraries already available so there are so many predefined /prebuilt functions so it's always better for us to use that instead of creating user defined function but still there are certain circumstances during which we need to create user defined function.

```
employee_data =[(10,"Michael Robinson","1996-06-01","100",2000),
                (20,"James Wood","2003-03-01","200",8000),
                (30,"Chris Andrews","2005-04-01","100",6000),
                (40,"Mark Bond","2008-10-01","100",7000),
                (50,"Steve Watson","1996-02-01","400",1000),
                (60,"Mathews Simon","1998-11-01","500",5000),
                (70,"Peter Paul","2011-04-01","600",5000)]

employee_schema =["Employee_id","Name","DOJ","Employee_Dept_ID","Salary"]

empDF = spark.createDataFrame(employee_data,employee_schema)

display(empDF)
```



#Define UDF to rename columns

This particular requirement if needed in multiple places let's say 1000 of data frames, in each and every place , we can't write the code, so we can create UDF in one place so that can be called again and again in multiple other places .that is the requirement

```
import pyspark.sql.functions as f
def rename_columns(rename_df):
    for column in rename_df.columns:
        new_column= "Col_" +column
        rename_df =rename_df.withColumnRenamed(column,new_column)

    return rename_df
```

#Execute UDF

```
renamed_df = rename_columns(empDF)

display(renamed_df)
```

(3) Spark Jobs
- Job 54 View (Stages: 1/1)
- Job 55 View (Stages: 1/1)
- Job 56 View (Stages: 1/1)

renamed_df: pyspark.sql.dataframe.DataFrame = [Col_Employee_id: long, Col_Name: string ... 3 more fields]

Table +

	Col_Employee_id	Col_Name	Col_DOJ	Col_Employee_Dept_ID	Col_Salary
1	10	Michael Robinson	1996-06-01	100	2000
2	20	James Wood	2003-03-01	200	8000
3	30	Chris Andrews	2005-04-01	100	6000
4	40	Mark Bond	2008-10-01	100	7000
5	50	Steve Watson	1996-02-01	400	1000
6	60	Mathews Simon	1998-11-01	500	5000
7	70	Peter Paul	2011-04-01	600	5000

7 rows | 1.02 seconds runtime

If we want to prefix with UDF in all columns, then

```
import pyspark.sql.functions as f
def rename_columns(rename_df):
    for column in rename_df.columns:
        new_column= "UDF_" +column
        rename_df =rename_df.withColumnRenamed(column,new_column)

    return rename_df
```

```
renamed_df = rename_columns(empDF)
display(renamed_df)
```

(3) Spark Jobs
- Job 66 View (Stages: 1/1)
- Job 67 View (Stages: 1/1)
- Job 68 View (Stages: 1/1)

renamed_df: pyspark.sql.dataframe.DataFrame = [UDF_Employee_id: long, UDF_Name: string ... 3 more fields]

Table +

	UDF_Employee_id	UDF_Name	UDF_DOJ	UDF_Employee_Dept_ID	UDF_Salary
1	10	Michael Robinson	1996-06-01	100	2000
2	20	James Wood	2003-03-01	200	8000
3	30	Chris Andrews	2005-04-01	100	6000
4	40	Mark Bond	2008-10-01	100	7000
5	50	Steve Watson	1996-02-01	400	1000
6	60	Mathews Simon	1998-11-01	500	5000
7	70	Peter Paul	2011-04-01	600	5000

7 rows | 0.61 seconds runtime

```
#UDF TO CONVERT NAME INTO UPPER CASE


from pyspark.sql.functions import upper,col

def upperCase_col(df):
    empDF_upper = df.withColumn('name_upper',upper(df.Name))
    return empDF_upper
```

```
Up_Case_DF=upperCase_col(empDF)
display(Up_Case_DF)
```

(3) Spark Jobs

- Job 75 View (Stages: 1/1)
- Job 76 View (Stages: 1/1)
- Job 77 View (Stages: 1/1)

Up_Case_DF: pyspark.sql.dataframe.DataFrame = [Employee_id: long, Name: string ... 4 more fields]

Table +

	Employee_id	Name	DOJ	Employee_Dept_ID	Salary	name_upper
1	10	Michael Robinson	1996-06-01	100	2000	MICHAEL ROBINSON
2	20	James Wood	2003-03-01	200	8000	JAMES WOOD
3	30	Chris Andrews	2005-04-01	100	6000	CHRIS ANDREWS
4	40	Mark Bond	2008-10-01	100	7000	MARK BOND
5	50	Steve Watson	1996-02-01	400	1000	STEVE WATSON
6	60	Mathews Simon	1998-11-01	500	5000	MATHEWS SIMON
7	70	Peter Paul	2011-04-01	600	5000	PETER PAUL

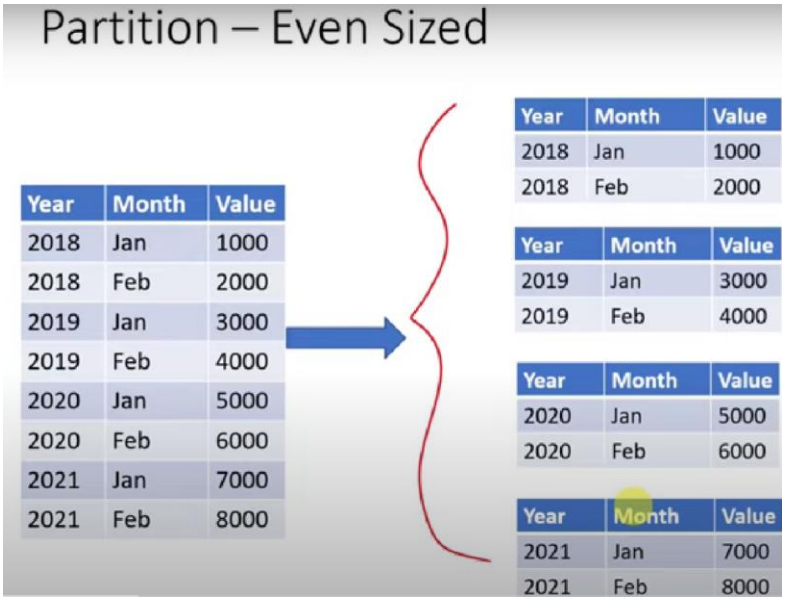
7 rows | 0.67 seconds runtime

DATA SKEWNESS

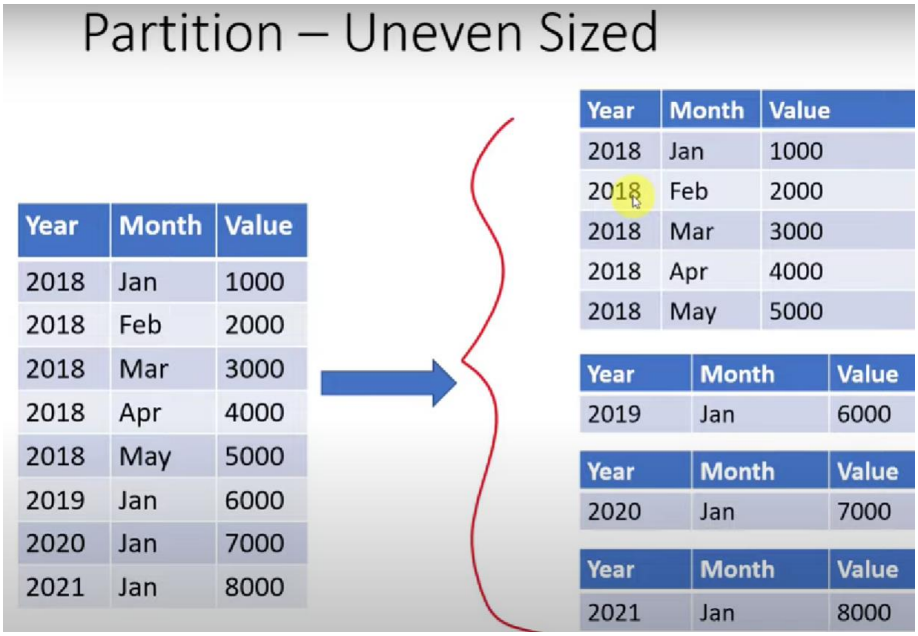
Data Skewness is one of the widely used terminology in big data project. While processing big data file normally we will split the data into multiple smaller chunks that is called partition . while creating partition if we don't distribute data evenly it will create unevenly sized partitions which means 1 partition will be bigger than other partitions. As a result, while processing that particular partition by a core within executor it will take long time when compared to other partition which means smaller partition will process very quickly and corresponding cores will sit idle and only 1 core which is processing bigger file it will keep on running for longer duration. As a result, over all execution time of that particular core/task. As a result, it will impact overall execution time of that particular stage. As a result, it will impact overall execution time of the job. So, this is how it will downgrade the performance.

Let us take below example containing 1 file with 3 columns Year, Month and Value. In real time partition can contain millions of records. The below file has 8 records and data is distributed evenly and there are 2 records for each year .let us assume year is partition key. Based on year we are going to partition this particular file, so it

will produce 2 records per partition .In this case it has produced evenly distributed partition. Now almost all partitions are almost same in size . As a result, while processing this file all the cores would execute in parallel. As a result, we can achieve highest parallelism



On the other hand, let us assume a file which is containing 8 records, out of that there are 5 records for year 2018 and 1 record for 2019,1 record for 2020 and 1 record for 2021 . If we are going to create partition based on year key, then it will create unevenly distributed partitions which means there would be 5 records for 1 partition and 1 record for other years. as a result, this is going to produce uneven sized partition. In real time project there could be millions/billions of records as a result it is going to hit the performance heavily.



Impact of Data Skew

When we are going to supply evenly distributed partitions, let's say there are 4 partitions evenly distributed, as a result even execution time from all the core for all the task would be same so we can achieve highest parallelism. In case we are going to supply unevenly distributed partitions which means 1 partition is bigger when compared to other partitions .As a result core/task for the bigger partitions will take more time. As a result, overall execution time will be impacted.

Smaller use case how data skew is getting created , what would be the root cause?

Let us assume there is an online retail store and their primary location is German and they set up business very well in German and they are producing 10M records 10M transactions per day .slowly they are expanding there business to other countries such as France , Italy, and Sweden. Let us assume France , Italy and Sweden haven't established business in better way still there are lot of improvements they have to perform .

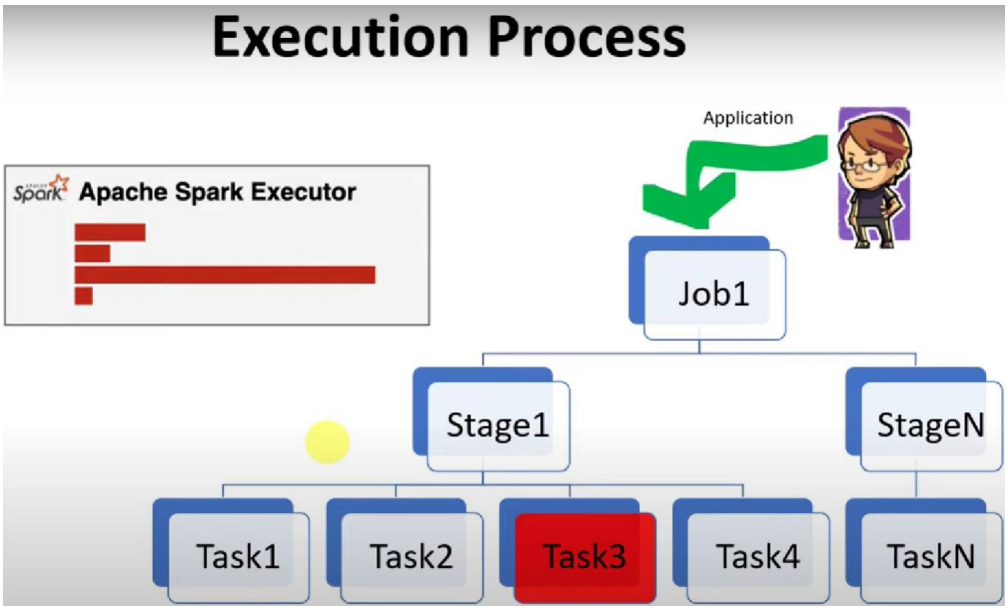
Root Cause of Data Skew

Country	Data Count per Day
Germany	10 M records
France	0.5 M records
Italy	0.1 M records
Sweden	50 K records

Currently they are running just pilot version which means in France they are just producing 0.5M records which is $1/20^{th}$ of prime location Germany and coming to Italy it is just producing 0.1M records 100 times smaller than Germany and coming to Sweden

it is just producing 50K transactions per day which is 200 times smaller than prime location Germany .These transaction data will be stored in 1 particular fact table. When we are going to partition based on country_id, definitely it will produce unevenly sized partition which means Germany it will be very bigger when compared to other country ID's . This is one simple example how data skew is getting produced. If we find there is data skew in our application as per the nature, we need to put some proper mechanism to handle that otherwise it is going to hit our application.

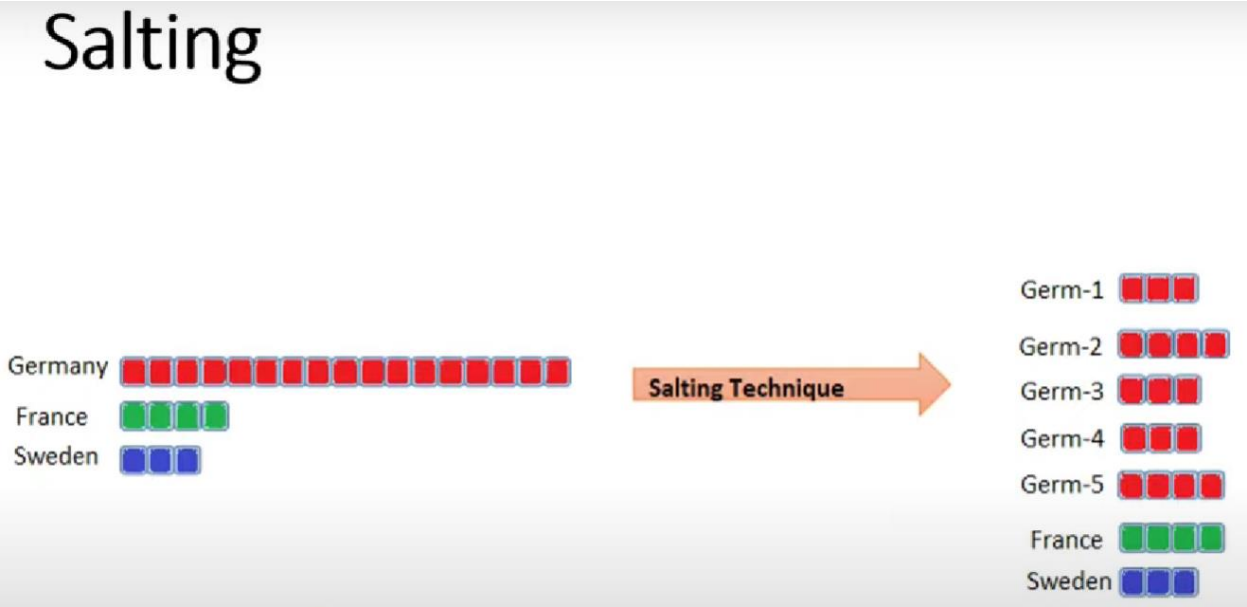
EXECUTION PROCESS: When we are submitting some application/code in to spark environment by the user, first it will create jobs.jobs would be divided into multiple stages and stages would be splitted into multiple tasks .If we have to perform some wide transformation then 1 stage would be created which means in our application there is one wide transformation then it would contain 2 stages and similarly if we are going to have 10 wide transformations in our application which means it will have 10+1 =11 stages . Let us assume in this particular application there is stage1 and we are supplying unevenly distributed partitions, one particular partition is bigger than other partitions so what happens when we are submitting application what happens is within each stage task would be created. As per spark architecture, each core within the executor will pick one particular partition and these partitions are not sharable across the core so each core will pick one particular partition and it will start processing that is called task. In this case what happens is 3 tasks Task1, Task2 and Task 4 it would be completed quickly but 3rd task which is processing bigger partition it will keep on running for longer duration so other 3 core will sit idle even though it has completed its task it cannot move to next stage, or it cannot help Task3 because 1 particular partition will not be sharable so these cores can't help particular tasks. At the same time these cores can't move to next stage because as per spark architecture before completing 1 stage , next stage cannot be started because stage2 would be depending on resultant data of stage1 and stage1 will be depending on resultant data of all the tasks within the stage. As a result, what happens is one particular core will be processing the partition file for longer duration while other cores are sitting idle, so this is how we are loosing advantage of parallelism. Now we have understood what will be impact in the execution process if we are going to create data skew in the partition.



MULTIPLE WAYS TO HANDLE DATA SKEW:

- Salting Technique: It is one of the popular technique to handle data skew.
- Apply Skew-hint with spark SQL
- Use broadcast join if we are going to join 2 tables out of that 1 table is very smaller in size
- We can enable AQE if we are going to use spark3 or higher version

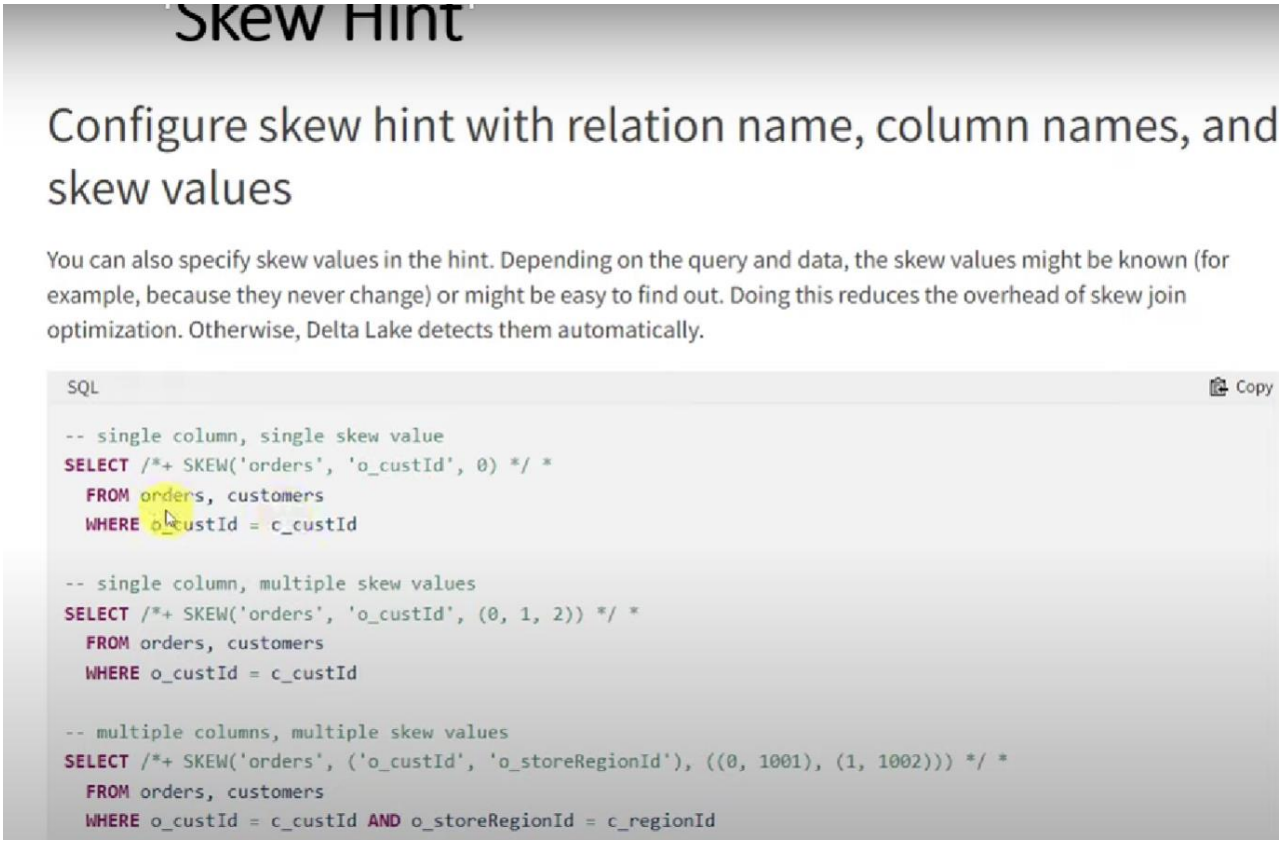
SALTING TECHNIQUE:



Salting technique is in case if we are going to have data skew for a particular partition, then we will create pseudo-ID .Based on that we will split the bigger partition into multiple smaller partitions .Let us assume there is one file partitioned based on country_id. For Germany it is containing so many records when compared to France and Sweden. So now we have to split this Germany into multiple smaller chunks so for that we have to create some pseudo_id. So that is called salting.so these smaller chunks are almost equal to other partitions. So, we can achieve this through certain coding concept that is called salting. Salting is one of the technique to avoid data skew.

Skew Hint: If we are using spark SQL, we can specify SKEW , if we are expecting Skew in our data then we can explicitly specify while processing that data which means when we are specifying through hint what happens is spark engine will prepare query accordingly and it will optimize while processing .it's always better to use skew hint if we expect data skew in our file.

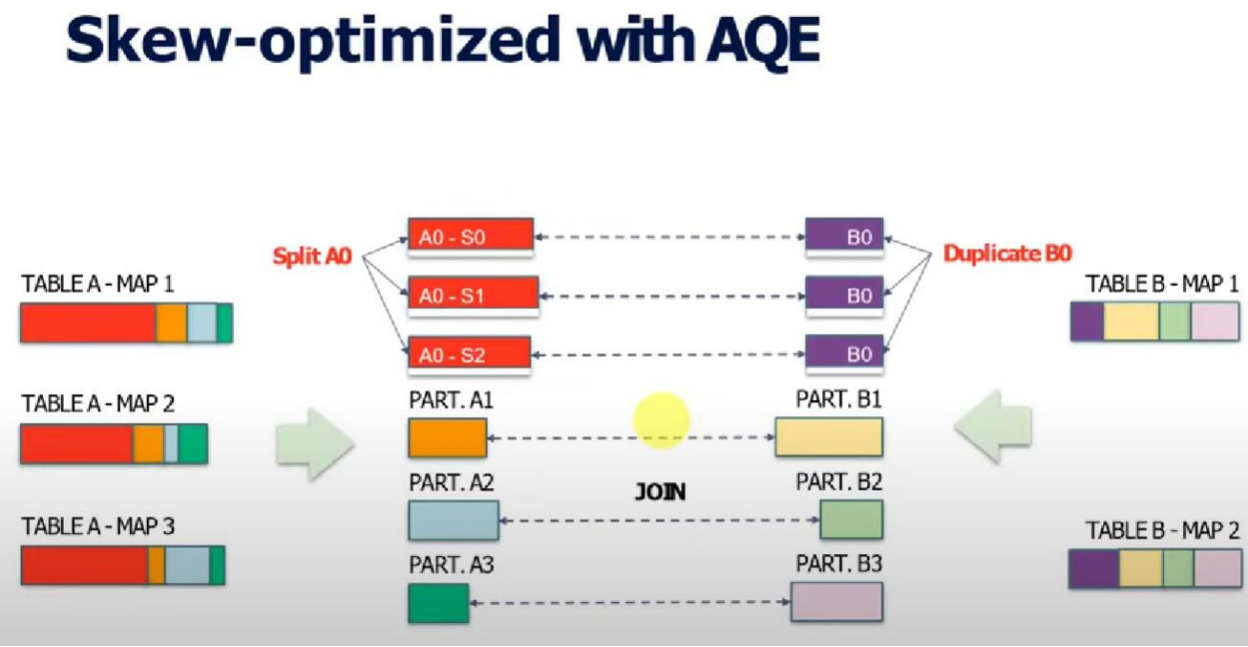
Here 2 tables getting joined orders and customers, for that in order table there is 1 column cust_id that is skewed and cust_id 0 if cust_id 0 containing more records compared to other id then we can explicitly specify that. If we expect data skews for multiple ID's we can specify that also from order table , particular cust_id column and data skew for these id's 0,1,2 . we can also specify skew values for multiple columns and multiple skew values. This is another way to handle data skew.



Broadcast Variable: In case we are joining 2 tables one is facts and another is dimension and fact will always be bigger when compared to dimension and dimension table very small, then what we can do is instead of splitting that smaller table into multiple partitions and storing across cluster what we can do is we can send small copy of the tiny table into all worker nodes/all the executors. As a result, data cannot be shuffled while performing the joining.so this method is suitable only if 1 method is smaller. Incase if table is not smaller, we are going to use broadcast Variable ,it will add memory pressure to the cluster across all the workers. So, it is going to again impact our application.

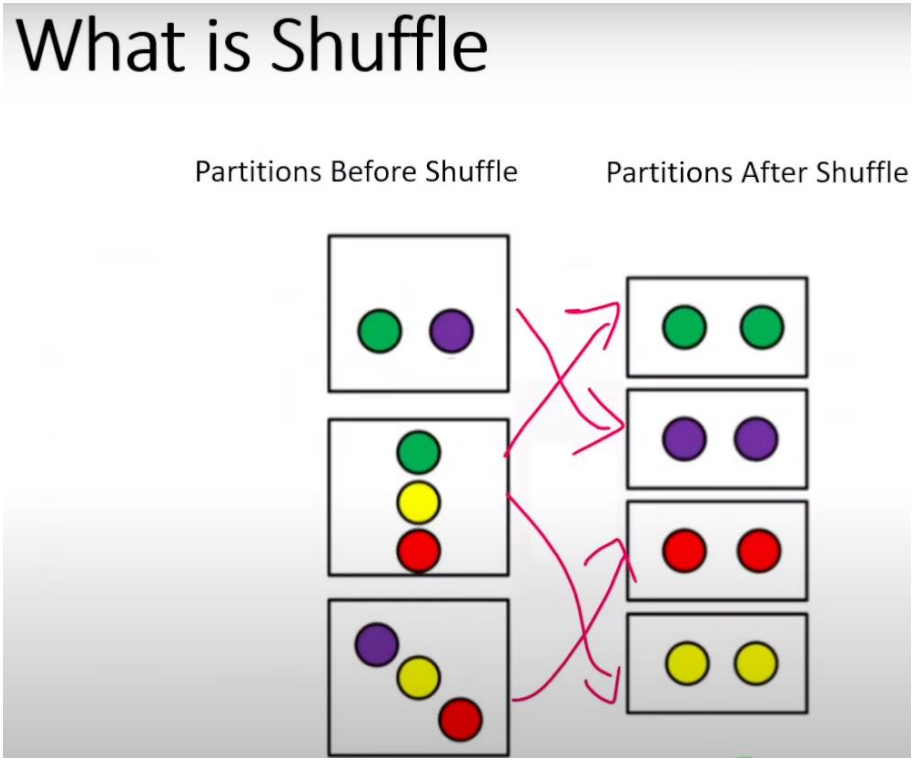
AQE: If we are going to enable AQE,it will take care of our performance optimization. Basically, it will perform these operations 1st is coalescing shuffle partitions. If we are having many number of partitions it will combine partitions and it will make optimized size so that is called coalescing. 2nd is it will automatically switch join strategies like broadcast join or sort merge join. 3rd is optimizing skew joins that is part of AQE.

Let's say when we are going to enable AQE, and we are joining 2 tables. In one table data is skewed, red colour shown below that particular key is skewed its creating data skew. What happens is we don't need to create any coding mechanism manually to create salting technique, it would be handled by AQE automatically and it will create salting key. Based on that it will split data skew into uniform sized partitions. Simply we can enable AQE and rest will be taken care by spark engine, we don't need to do anything.



SHUFFLING PARAMETER

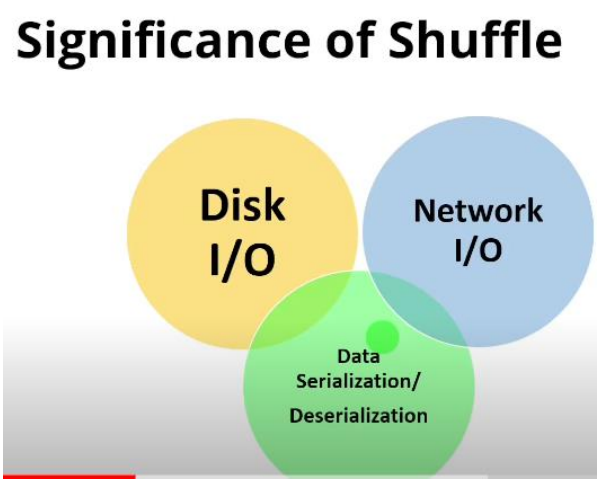
This is one of the performance optimization technique in spark databricks development. In order to understand shuffling parameter first we need to understand shuffle.



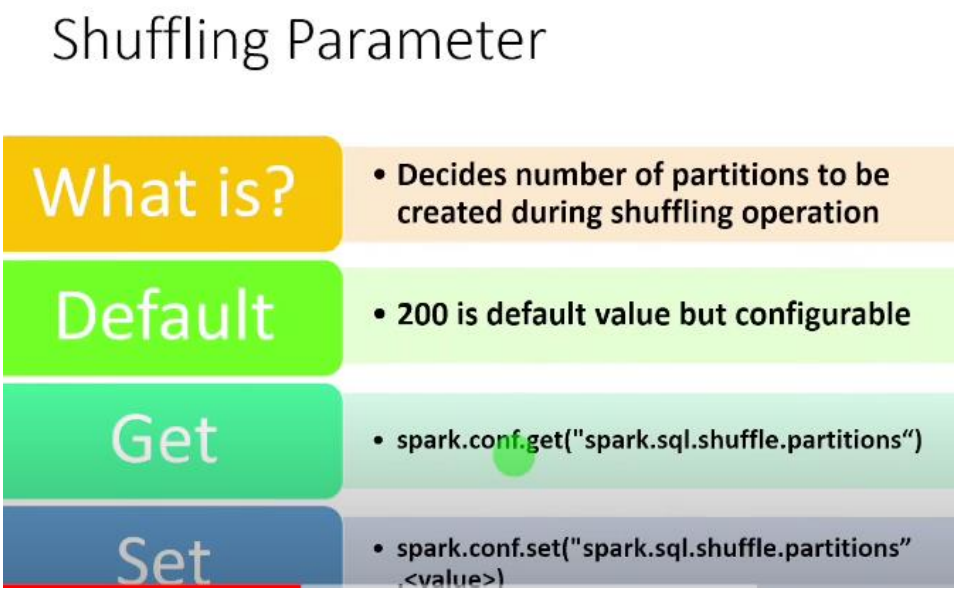
Shuffle is exchanging data across executors. Why should we need to shuffle the data ? In order to achieve certain result in certain calculations ,it's important to shuffle the data across executors. Shuffle is mainly used in spark transformations. Spark

transformations are classified as narrow transformation and wide transformation. Narrow transformation does not require shuffling in order to produce the output. On the other hand, wide transformation, in order to perform certain calculation, let it be groupBy,aggregate or join operation in order to get output for those operations data in one particular executor is depending on the data from another partition from another executor. So, in order to produce desired output, it's very important to sort and shuffle the data. In the above screenshot we can clearly see that if we want to calculate number of balls per colour, we have to sort the data ,bring the same colour of balls into one executor before performing the calculation. So, this is how shuffle happens across executors.

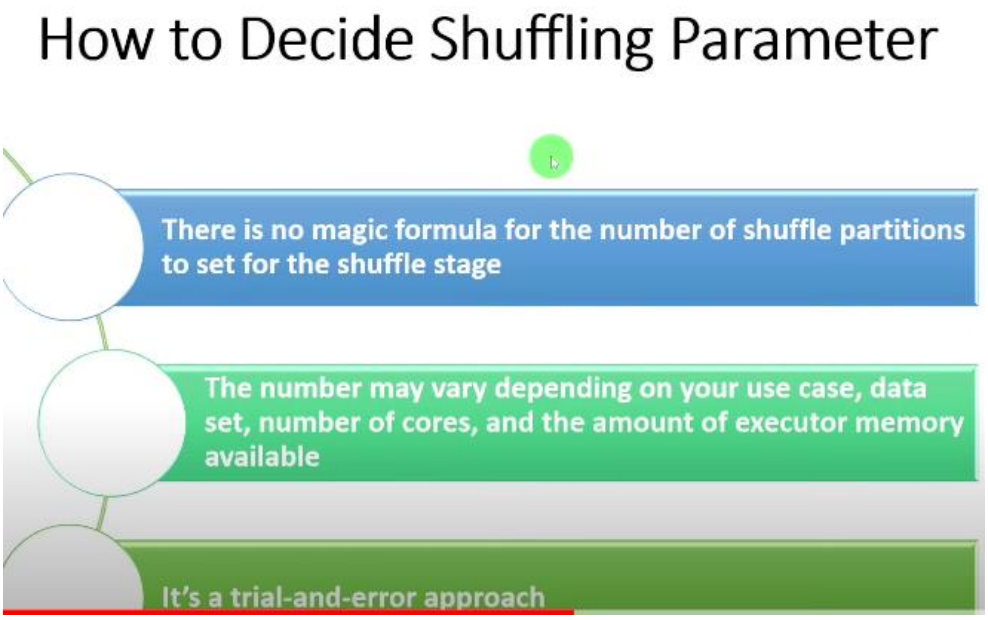
SIGNIFICANCE OF SHUFFLE:



Shuffle is one of the costliest operation in spark development. So, it's very important to give attention to shuffling parameters. Shuffling involves lot of disk I/O operations and network I/O operations, and it leads to data serialization and deserialization. These are the factors hitting the performance in any spark application. That is the reason we have to choose right shuffling parameters.



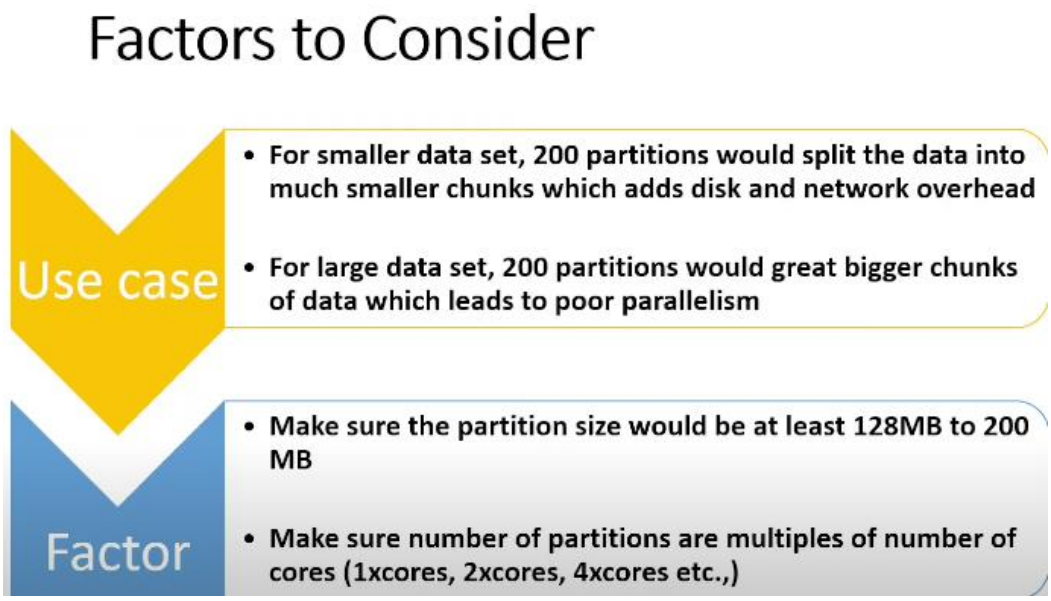
Shuffle parameter is while exchanging the data in between stages how many number of partitions to be created that is called shuffling parameter. By default, number is 200 which means if we are performing wide transformation which requires data shuffling in between the stages which means at the end of each stage 200 partitions will be created. If we want to understand what is default value or set value from environment, then we can use this command spark.conf.get("spark.sql.shuffle.partitions") → this will give number of partitions shuffle parameters set for our environment and in case if we have to set new value for this parameter then we have to use this command spark.conf.set("spark.sql.shuffle.partitions",<value>). Value can be less than 200 or more than 200. Depending on our use case we can set any value



How to decide shuffling parameter? There is no magic number to get shuffling parameter. Shuffling parameter differs from use case to use case and it differs based on many factors such as our use case, data set , number of cores and amount of executor memory available. These are the factors defining shuffling parameter from use case to use case.it is purely trial and error approach, we don't have magic number. It's purely depending on our use case.

FACTORS TO CONSIDER WHILE CHOOSING SHUFFLING PARAMETER:

Its depending on our use case. Let's say we have smaller dataset in our application. Let's say it is 1GB and 200 partitions for 1GB it would be huge number which means each partition will be very small in size, so this is going to add overhead to our disk and network operations. On the other hand, we have large data set which is close to 1TB.If we are creating 200 partitions out of 1TB , again each partition will be very big in size.so we are not going to achieve benefit of parallelism. Even that is also not suitable so how we can choose what are the factors to consider? Always choose partition size should be atleast 120MB to 200MB.This is the standard size for any partition block and make sure no of partitions that are getting created that is equivalent to multiples of no of cores in our use case. If we are going to have 16 number of cores in our cluster, then we can make atleast 16 partitions or 32,64,128 . we can make number of partitions in this style(can be multiples of cores – 1*cores,2*cores,4*cores etc)



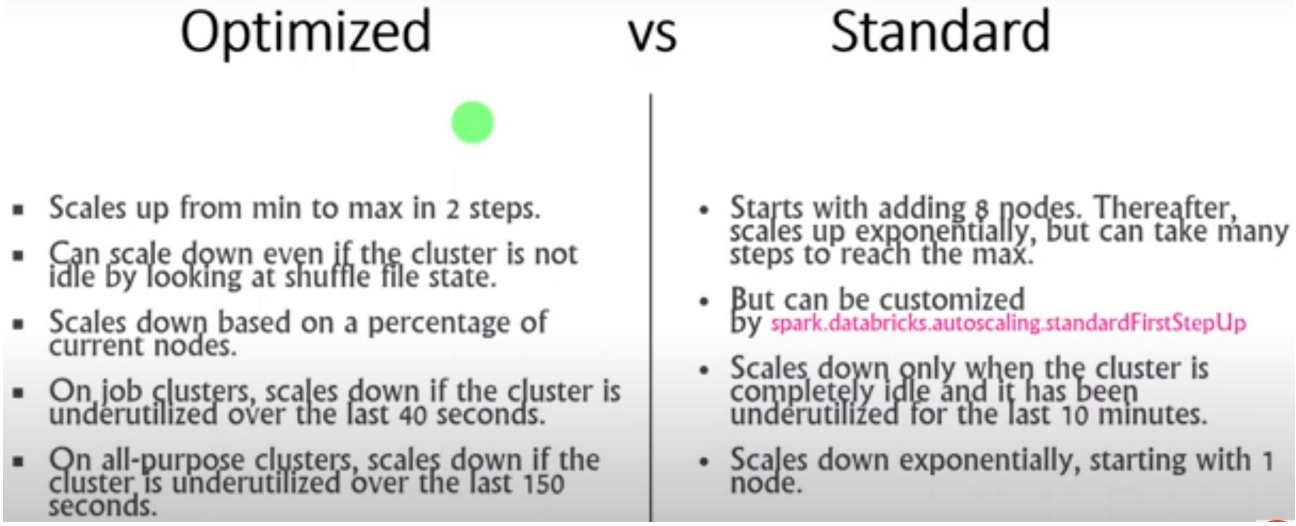
OPTIMIZED SCALING

Autoscaling is one of the important feature in performance optimization and also cost saving in data bricks environment. Autoscaling is one of the feature provided by the data bricks to choose/change number of worker nodes dynamically. While creating cluster we need to enable autoscaling for this feature. Also, we need to fix range of number min and max number. So, depending on workload by executing some application, data bricks will choose number of workers according to workload. Let's say we have given range 2 to 8 , depending on workload data bricks can choose number of worker nodes either 2 or 8 or anything between them. Autoscaling is one of the feature based on which data bricks can choose number of worker nodes dynamically depending on workload.

There are 2 types of auto scaling. Optimized and standard. There are 2 types of cluster types Automated/job cluster and All Purpose/Interactive cluster. Coming to job/automated cluster, always optimized scaling technique will be applied .Coming to interactive/all purpose cluster , autoscaling method will be depending on the environment .While creating cluster , if we choose standard pricing tier then standard autoscaling will be applicable for that environment. While creating databricks environment if we go with premium pricing tier then optimized autoscaling will be applicable for that environment.

DIFFERENCE BETWEEN OPTIMIZED AND STANDARD SCALING:

- In optimized scaling , data bricks can scale up number of worker nodes from minimum to maximum in 2 steps. Let's say, we have chosen range from 8 to 64 in our cluster creation and depending on the workload data bricks can reach the max no of workers i.e., 64 in 2 steps , starting from 8 and maybe it can jump directly to 32 and immediately it can reach max 64. So, it can scale up from min to max in 2 steps.so it can boost the performance in this way. Coming to standard it will exponentially increase worker nodes depending on the workload. Even though we are going to get complex workload still data bricks will start adding nodes exponentially. So we are losing some performance in this approach.



- Coming to optimized autoscaling , If data bricks engine forecasts some less workload in some time then it will start reducing number of worker nodes but that is not applicable in standard worker nodes and in standard worker node it will always starts with 8 number of worker nodes and this parameter is configured using spark.databricks.autoscaling.standardFirstSetUp here we can give even 2 /4 which means when it starts it will start adding those many number of nodes. Coming to job cluster in optimized, in job cluster we will always use optimized even though it is standard/premium pricing tier .
- Coming to job cluster in optimized, if cluster is underutilized over last 40 seconds, then it start scaling down which means it is start saving our cost. But coming to standard it will start scaling down only if cluster is idle for sometime and also it is underutilized for last 10 minutes. Let's say in optimized it is 40 sec and in standard it is 10 mins which is approximately 9 mins we are wasting our money.
- In optimized autoscaling for all-purpose clusters that is interactive clusters it scale down number of worker nodes if it is underutilized for 150 secs. So overall we can say using optimized autoscaling , we can aggressively change number of worker nodes depending on the workload. Coming to standard it's in exponential manner even though we have complex workload or less workload , it will exponentially increase/decrease number of worker nodes in standard. That's the main difference.

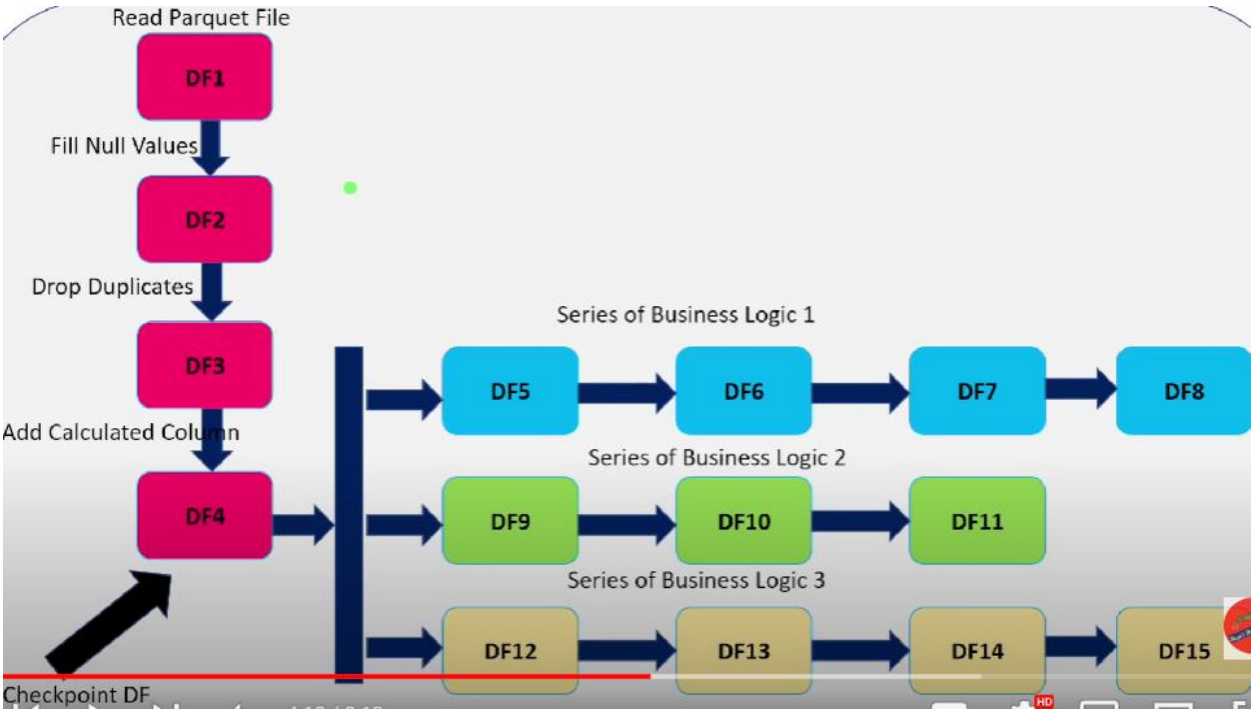
Dataframe CheckPoint

Checkpoint is a process of materializing or persisting data frame resultant data into disk. It is one of the performance optimization technique used in databricks development. Let us assume we are developing databricks notebook with 100's of transformations and few actions. When we are calling an action, it would traverse back through the dependent transformations which means it will go through transformation1 which is needed for this action then starting from transformation1, it will start executing subsequent transformations. Finally, it will get result for that particular action and that would be returned to user. This is the normal process.

When we are having 100's of transformations and all of those transformations are needed for multiple actions , when there is action called , each and every time those 100's of transformations will be recomputed again and again, and result will be returned to

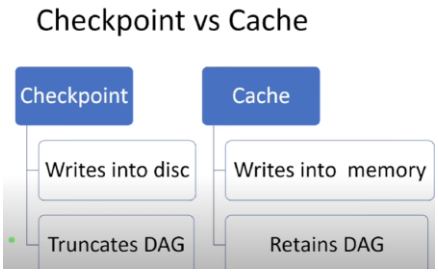
user. So, this will hit the performance because 100's of transformations would be recomputed again and again. So, to handle this situation databricks has given 2 approaches.one is checkpoint and 2nd one is cache. In check point let's say there are 100 transformations , out of 100 transformations we want to create checkpoint for first 50 transformations which means all the transformations starting from transformation1 until transformation 50 would be executed and intermediate result data would be stored in a disk. When certain action is called which is depending on 100 transformations then action would traverse back until transformation51 it will not go back because transformation 51 can be directly executed based on the resultant data that is returned to the disk.so trnsformation51 will take data from the disk and it will complete rest of the transformation . So, when there are multiple actions being called and all of these actions are depending on 100's of transformations then what happens is first 50 transformations can't be recomputed again and again. Instead of that persisted data in the disk can be used for the transformation for further transformation. In this way we can improve the performance. We can avoid recomputing first 50 transformations.

Even though checkpoint looks similar to cache there are some differences. Major difference is cache intermediate data would be returned to memory but in check point intermediate resultant dataset will be returned to disk. When we are creating checkpoint until transformation50 which means lineage graph before until transformation50 will be truncated which means resultant data is returning to disk. At the same time, we cannot retrieve previous lineage graph starting from transformation1 until transformation50 but coming to cache even though we are materializing or persisting intermediate result into memory but still it would retain lineage graph so if there is some network or some failure using lineage graph those 50 transformations can be recomputed .these are differences between checkpoint and cache. Both are looking similar but depending on our requirement we can use any of this approach



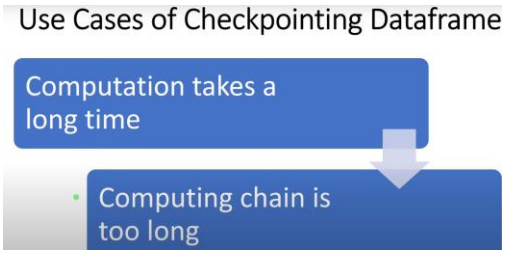
In our notebook we are doing multiple transformations DF1,DF2,DF3,DF4 and there are multiple branches .let's say there is an action which is depending on DF8,DF8 is

depending on DF7 and DF7 is depending on DF6 and DF6 is depending on DF5 and DF5 is depending on DF4.here there is a hierarchy. Similarly, there is an action here DF11 and it would traverse back to DF9 and DF9 depending on DF4 and DF4 is depending on DF3 and so on. Similarly, DF15 action will be done. In this particular example when we are having actions here, each and every time it will traverse back, and first 4 transformations will be recomputed again and again .In this use case what we can do is instead of recomputing all dataframes again and again we can create a check point for DF4 which means starting from DF1 until DF4 all steps will be executed once and resultant data will be returned to disk. When there is action called depending on DF8,DF8 will traverse back to DF7,DF7 to DF6,DF6 to DF5 and DF5 will not be depending on DF4, instead of that it can read data directly from disk so in this way computation will be stopped until DF4, recomputation will not be happened from DF1 to DF4, instead of that DF5 will take data directly from the disk. This is how we can avoid recomputation.



In checkpoint resultant dataset will be returned to the disk whereas in cache intermediate resultant dataset will be returned to the memory. In check point once intermediate result is created and returned to disc then it would truncate lineage graph in DAG. Whereas in cache even though it is writing intermediate result into memory still it would retain lineage graph in DAG.

USE CASES FOR CHECK POINT:

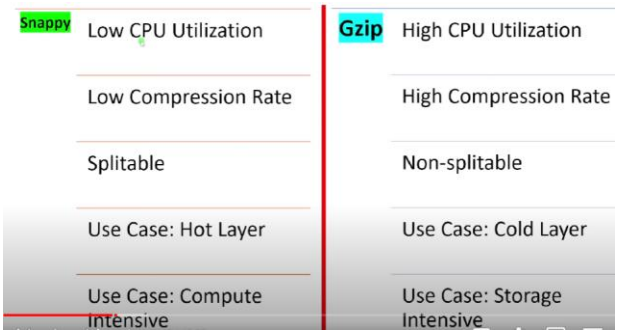


Let's say we are having 10 transformations ,out of 10 transformations first 3 transformations are taking too much time.so when we are calling certain actions which are depending on 3 data frames again and again it will recompute and it will take lot of time for recomputation.in this case what we can do is we can create check point for these 3 transformations which means complex or time consuming computation will be done only once and resultant data will be returned to disk and further subsequent transformations can refer check pointed data in disk.

Next use case is computing chain is too long , in our notebook there are several 100's of transformations finally we are calling an action and action would traverse across all these transformation. In order avoid computing chain length we can go for checkpointing. These are the 2 use cases to use check point in our databricks environment.

COMPRESSION METHODS- SNAPPY vs GZIP

In any big data projects choosing right file format improve the performance. There are many bigdata file formats available in the market such as parquet,avro,orc,json and csv. Each and every file format contains unique use cases.it means one particular file format is not suitable for all use cases. We have to understand our use case. While working with big data projects choosing right compression method will hit or boost the performance.



- Snappy consumes low CPU Utilization which means when we are performing snappy compression it won't occupy more CPU resources which means other rest of other processes can get some resources for their computation. That is advantage of snappy. Coming to Gzip compression it will consume more CPU which means it will occupy more resources of the server which means other processes might be affected to get some resources, so this is not good.
- Snappy has Low compression rate whereas Gzip has High compression rate. Let's say we are having 1 GB file we are going to compress that one, if we are going to apply snappy compression may be let's say it is producing 500MB whereas when we go with Gzip , even it can compress even better like 1GB to 2MB.At the end of compression, snappy compressed file will occupy more space in their disc and Gzip compressed file will occupy less space in the disk. In this point Gzip scores better than snappy.
- Snappy is splitable format whereas Gzip is non-splitable format. Now a days in bigdata we are going with massively parallel processing which means it's a distributed and also parallel processing, so parallel processing is very important to achieve heavy calculation in short duration so that file format should be splitable then only multiple cores can access the same file for quick computation. In that case snappy is splitable so we can improve the calculation. Coming to Gzip, it's not splitable so one core from a server can access the file at a time so it will hit the performance of any calculation. So, in this particular point snappy scores better than Gzip. Snappy is better in terms of Low CPU Utilization and splitable. Gzip is better in terms of high compression rate.
- Hot Layer is suitable for snappy. For example, we are compressing data that is accessed frequently by our application then that is called hot layer, we are going to access data multiple times again and again. In that case we can go with snappy because when we are accessing data multiple times again and again, each time it's going to consider less CPU, which means rest of other process running on same server will get some resources for their calculation. But if we are going with Gzip in hot layer what happens is Gzip compression itself will occupy more CPU ,memory ,resources of the server then other process will be affected .

so hot layer it's better to go with snappy. For example, if we are not going to access data frequently it is mainly for storage purpose then we can go with cold layer in Gzip.

- In Snappy data is splitable which means in parallel processing we can achieve any calculation very quickly so that's the reason if our use case is more compute intensive then it's better to go with splitable format that is snappy. If our use case is more storage intensive for example for auditing purpose, we have to maintain our historical 15 years back old data we have to store in some layer then we have to go with Gzip because this is going to give high compression rate which means compressed file will occupy less amount of disc space. So, we don't need to spend more money for disc storage. But if we are going with snappy for the use case storage intensive what happens is snappy will create bulky or bigger size compressed files, so we have to pay more for storage disc.

```
csvFile = "/FileStore/tables/bigdata_training/read_files/Baby_Names.csv"
csvDF = spark.read.option('sep',",").option("inferSchema","true").csv(csvFile)
display(csvDF)
```

▸ (3) Spark Jobs

	_c0	_c1	_c2	_c3	_c4
1	Year	First Name	County	Gender	Count
2	2007	ZOEY	KINGS	F	11
3	2007	ZOEY	SUFFOLK	F	6
4	2007	ZOEY	MONROE	F	6
5	2007	ZOEY	ERIE	F	9
6	2007	ZOE	ULSTER	F	5
7	2007	ZOE	WESTCHESTER	F	24

```
csvDF.write.format("parquet").option("compression","snappy").save("/FileStore/tables/bigdata_training/write_files/snappy_parquets")
```

(1) Spark job is run.

```
%fs
ls /FileStore/tables/bigdata_training/write_files/snappy_parquets/
```

here in screenshot 3 are supporting files and 4th row is final file

	path	name
1	dbfs:/FileStore/tables/bigdata_training/write_files/snappy_parquets/_SUCCESS	_SUCCESS
2	dbfs:/FileStore/tables/bigdata_training/write_files/snappy_parquets/_committed_1389686368330815512	_committed_13
3	dbfs:/FileStore/tables/bigdata_training/write_files/snappy_parquets/_started_1389686368330815512	_started_13896
4	dbfs:/FileStore/tables/bigdata_training/write_files/snappy_parquets/part-00000-tid-1389686368330815512-f6ca6be9-60bb-4ff1-bc8b-d9ef6028b6e7-11-1-c000.snappy.parquet	part-00000-tid-1

```
2  ls /FileStore/tables/bigdata_training/write_files/snappy_parquets/
```

		name	size
1		_SUCCESS	0
2		_committed_1389686368330815512	123
3		_started_1389686368330815512	0
4	4ff1-	part-00000-tid-1389686368330815512-f6ca6be9-60bb-4ff1-bc8b-d9ef6028b6e7-11-1-c000.snappy.parquet	153843

Size of snappy file is 153KB

```
csvDF.write.format("parquet").option("compression","gzip").save("/FileStore/tables/bigdata_training/write_files/gzip_parquets")
```

(1) Spark job is run.

```
%fs

ls /FileStore/tables/bigdata_training/write_files/gzip_parquets/
```

	path	name
1	dbfs:/FileStore/tables/bigdata_training/write_files/gzip_parquets/_SUCCESS	_SUCCESS
2	dbfs:/FileStore/tables/bigdata_training/write_files/gzip_parquets/_committed_6203988734748797603	_committed_62
3	dbfs:/FileStore/tables/bigdata_training/write_files/gzip_parquets/_started_6203988734748797603	_started_62039
4	dbfs:/FileStore/tables/bigdata_training/write_files/gzip_parquets/part-00000-tid-6203988734748797603-f66605fb-5753-4e31-b5bd-54b49c22d5a0-12-1-c000.gz.parquet	part-00000-tid-6

```
2  ls /FileStore/tables/bigdata_training/write_files/gzip_parquets/
```

		name	size
1		_SUCCESS	0
2		_committed_6203988734748797603	119
3		_started_6203988734748797603	0
4	4e31-	part-00000-tid-6203988734748797603-f66605fb-5753-4e31-b5bd-54b49c22d5a0-12-1-c000.gz.parquet	119736

Size of gzip file is 119KB.

Split FUNCTION

Split function is used to split single column into multiple columns based on logic/ based on certain regular expression

Syntax:

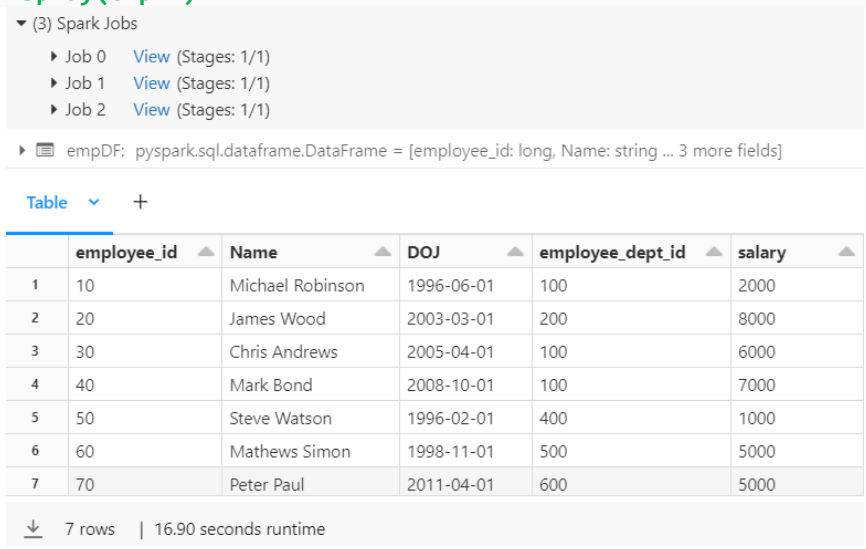
```
pyspark.sql.functions.split(str,pattern,limit=-1)
```

str → input string it can be column from data frame which column we want to split into multiple columns

pattern → Based on which pattern we want to split the string

limit → How many times we want to apply this pattern. By default, it will apply for whole string. If we want to split specific number of times, we can specify limit.

```
employee_data =[(10,"Michael Robinson","1996-06-01","100",2000),
                (20,"James Wood","2003-03-01","200",8000),
                (30,"Chris Andrews","2005-04-01","100",6000),
                (40,"Mark Bond","2008-10-01","100",7000),
                (50,"Steve Watson","1996-02-01","400",1000),
                (60,"Mathews Simon","1998-11-01","500",5000),
                (70,"Peter Paul","2011-04-01","600",5000)]
employee_schema =["employee_id","Name","DOJ","employee_dept_id","salary"]
empDF= spark.createDataFrame(employee_data,employee_schema)
display(empDF)
```

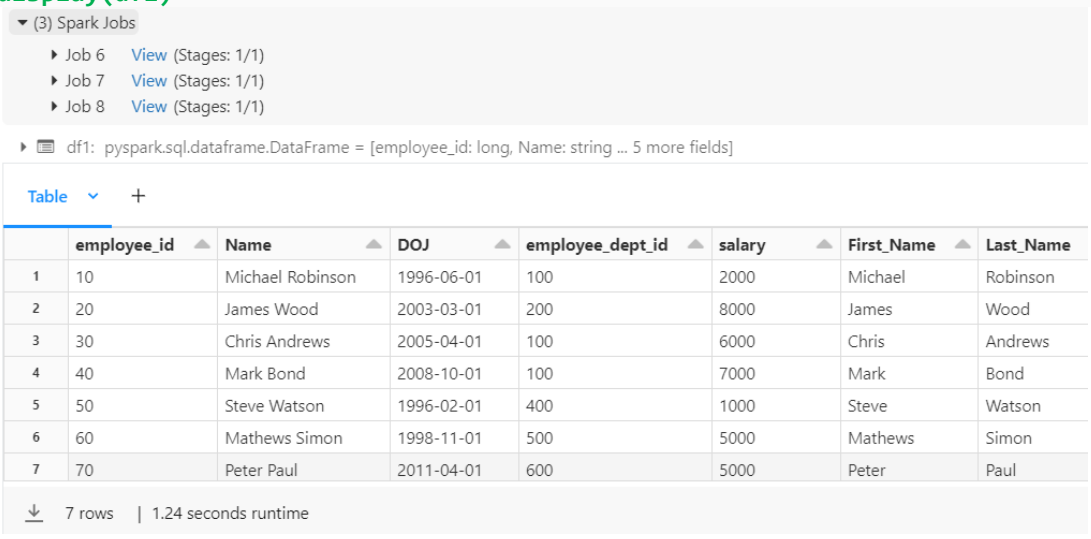


	employee_id	Name	DOJ	employee_dept_id	salary
1	10	Michael Robinson	1996-06-01	100	2000
2	20	James Wood	2003-03-01	200	8000
3	30	Chris Andrews	2005-04-01	100	6000
4	40	Mark Bond	2008-10-01	100	7000
5	50	Steve Watson	1996-02-01	400	1000
6	60	Mathews Simon	1998-11-01	500	5000
7	70	Peter Paul	2011-04-01	600	5000

7 rows | 16.90 seconds runtime

#First method of split

```
from pyspark.sql.functions import split
df1 = empDF.withColumn('First_Name',split(empDF['Name'],' ').getItem(0)) \
           .withColumn('Last_Name',split(empDF['Name'],' ').getItem(1))
display(df1)
```

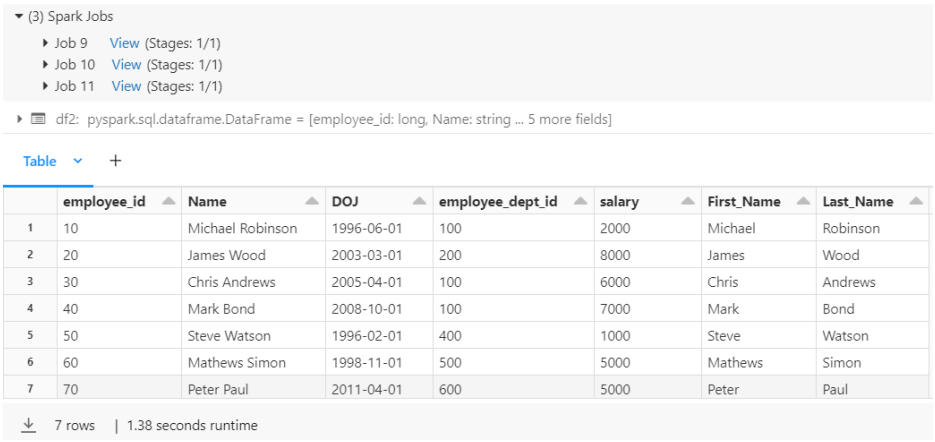


	employee_id	Name	DOJ	employee_dept_id	salary	First_Name	Last_Name
1	10	Michael Robinson	1996-06-01	100	2000	Michael	Robinson
2	20	James Wood	2003-03-01	200	8000	James	Wood
3	30	Chris Andrews	2005-04-01	100	6000	Chris	Andrews
4	40	Mark Bond	2008-10-01	100	7000	Mark	Bond
5	50	Steve Watson	1996-02-01	400	1000	Steve	Watson
6	60	Mathews Simon	1998-11-01	500	5000	Mathews	Simon
7	70	Peter Paul	2011-04-01	600	5000	Peter	Paul

7 rows | 1.24 seconds runtime

#Second Method of Split

```
import pyspark
split_col = pyspark.sql.functions.split(empDF['Name'],' ')
df2= empDF.withColumn('First_Name',split_col.getItem(0)) \
    .withColumn('Last_Name',split_col.getItem(1))
display(df2)
```



(3) Spark Jobs
- Job 9 View (Stages: 1/1)
- Job 10 View (Stages: 1/1)
- Job 11 View (Stages: 1/1)

df2: pyspark.sql.dataframe.DataFrame = [employee_id: long, Name: string ... 5 more fields]

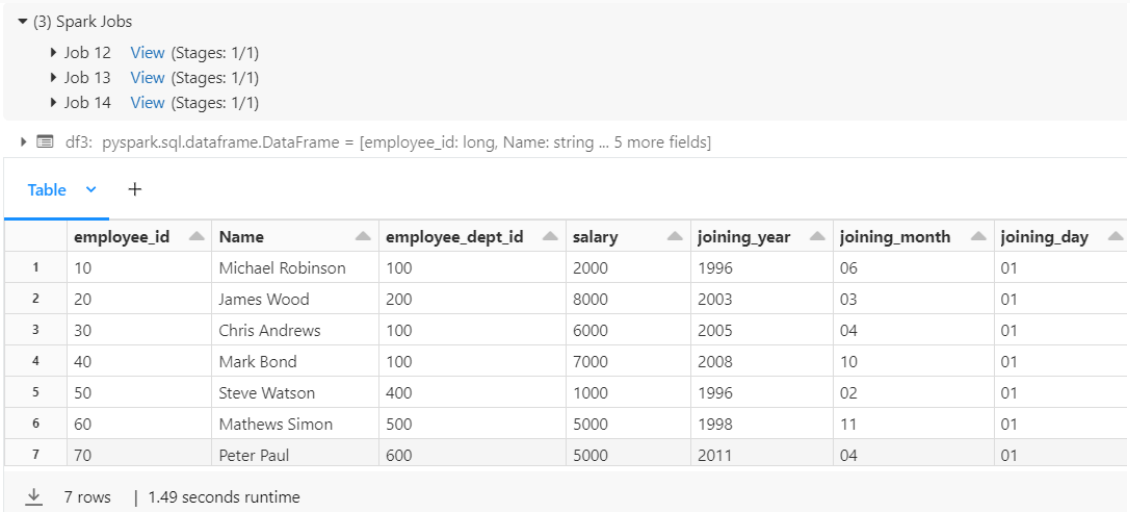
Table +

	employee_id	Name	DOJ	employee_dept_id	salary	First_Name	Last_Name
1	10	Michael Robinson	1996-06-01	100	2000	Michael	Robinson
2	20	James Wood	2003-03-01	200	8000	James	Wood
3	30	Chris Andrews	2005-04-01	100	6000	Chris	Andrews
4	40	Mark Bond	2008-10-01	100	7000	Mark	Bond
5	50	Steve Watson	1996-02-01	400	1000	Steve	Watson
6	60	Mathews Simon	1998-11-01	500	5000	Mathews	Simon
7	70	Peter Paul	2011-04-01	600	5000	Peter	Paul

7 rows | 1.38 seconds runtime

#Third Method of Split

```
split_col= pyspark.sql.functions.split(empDF['doj'],'-')
df3=
empDF.select("employee_id","Name","employee_dept_id","salary",split_col.getItem(0).al
ias('joining_year'),split_col.getItem(1).alias('joining_month'),split_col.getItem(2).
alias('joining_day'))
display(df3)
```



(3) Spark Jobs
- Job 12 View (Stages: 1/1)
- Job 13 View (Stages: 1/1)
- Job 14 View (Stages: 1/1)

df3: pyspark.sql.dataframe.DataFrame = [employee_id: long, Name: string ... 5 more fields]

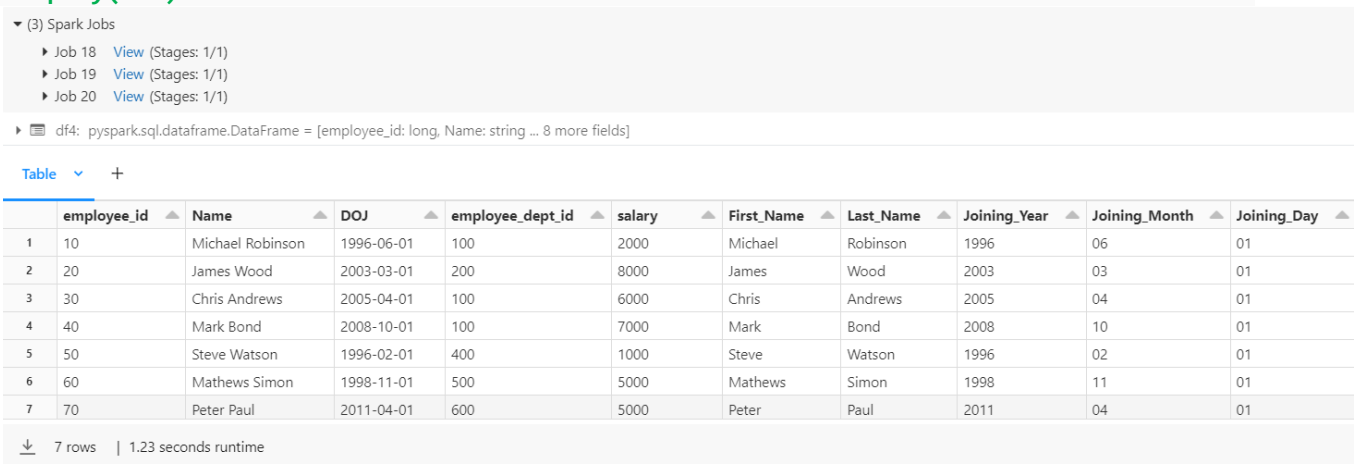
Table +

	employee_id	Name	employee_dept_id	salary	joining_year	joining_month	joining_day
1	10	Michael Robinson	100	2000	1996	06	01
2	20	James Wood	200	8000	2003	03	01
3	30	Chris Andrews	100	6000	2005	04	01
4	40	Mark Bond	100	7000	2008	10	01
5	50	Steve Watson	400	1000	1996	02	01
6	60	Mathews Simon	500	5000	1998	11	01
7	70	Peter Paul	600	5000	2011	04	01

7 rows | 1.49 seconds runtime

#We can combine multiple splits

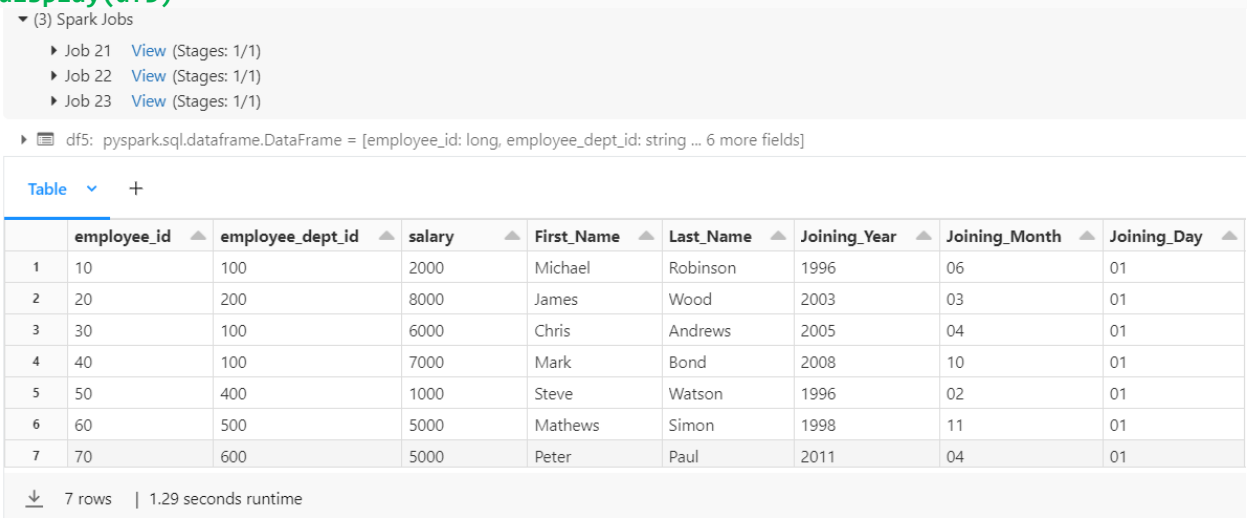
```
df4= empDF.withColumn('First_Name',split(empDF['Name'],' ').getItem(0)) \
    .withColumn('Last_Name',split(empDF['Name'],' ').getItem(1)) \
     .withColumn('Joining_Year',split(empDF['DOJ'],'-').getItem(0)) \
     .withColumn('Joining_Month',split(empDF['DOJ'],'-').getItem(1)) \
     .withColumn('Joining_Day',split(empDF['DOJ'],'-').getItem(2))
display(df4)
```



	employee_id	Name	DOJ	employee_dept_id	salary	First_Name	Last_Name	Joining_Year	Joining_Month	Joining_Day
1	10	Michael Robinson	1996-06-01	100	2000	Michael	Robinson	1996	06	01
2	20	James Wood	2003-03-01	200	8000	James	Wood	2003	03	01
3	30	Chris Andrews	2005-04-01	100	6000	Chris	Andrews	2005	04	01
4	40	Mark Bond	2008-10-01	100	7000	Mark	Bond	2008	10	01
5	50	Steve Watson	1996-02-01	400	1000	Steve	Watson	1996	02	01
6	60	Mathews Simon	1998-11-01	500	5000	Mathews	Simon	1998	11	01
7	70	Peter Paul	2011-04-01	600	5000	Peter	Paul	2011	04	01

#Split and drop splitted columns

```
df5= empDF.withColumn('First_Name',split(empDF['Name'],' ').getItem(0)) \
    .withColumn('Last_Name',split(empDF['Name'],' ').getItem(1)) \
     .withColumn('Joining_Year',split(empDF['DOJ'],'-').getItem(0)) \
     .withColumn('Joining_Month',split(empDF['DOJ'],'-').getItem(1)) \
     .withColumn('Joining_Day',split(empDF['DOJ'],'-').getItem(2)) \
     .drop(empDF['Name']) \
     .drop(empDF['DOJ'])
display(df5)
```



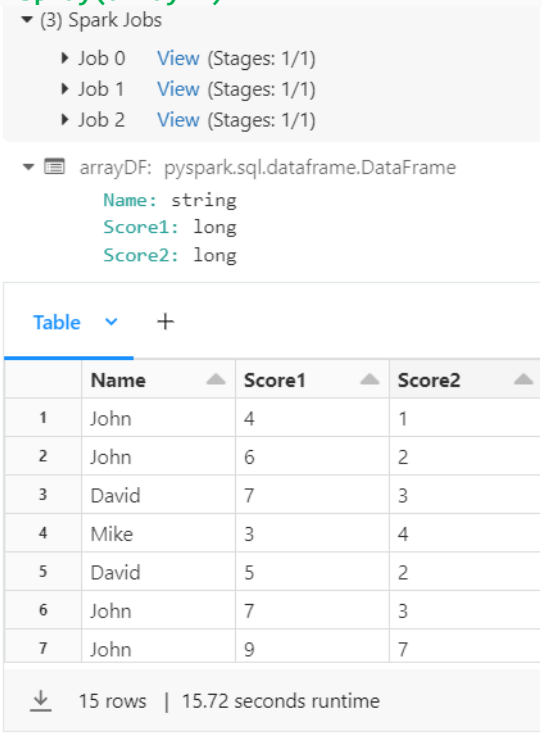
	employee_id	employee_dept_id	salary	First_Name	Last_Name	Joining_Year	Joining_Month	Joining_Day
1	10	100	2000	Michael	Robinson	1996	06	01
2	20	200	8000	James	Wood	2003	03	01
3	30	100	6000	Chris	Andrews	2005	04	01
4	40	100	7000	Mark	Bond	2008	10	01
5	50	400	1000	Steve	Watson	1996	02	01
6	60	500	5000	Mathews	Simon	1998	11	01
7	70	600	5000	Peter	Paul	2011	04	01

Arrays_Zip FUNCTION IN PYSPARK

arrays_zip is a pyspark function that returns a merged array of structs in which Nth Struct contains all N-th values of input arrays. It means it merges all the elements by its position.

#create sample dataframe

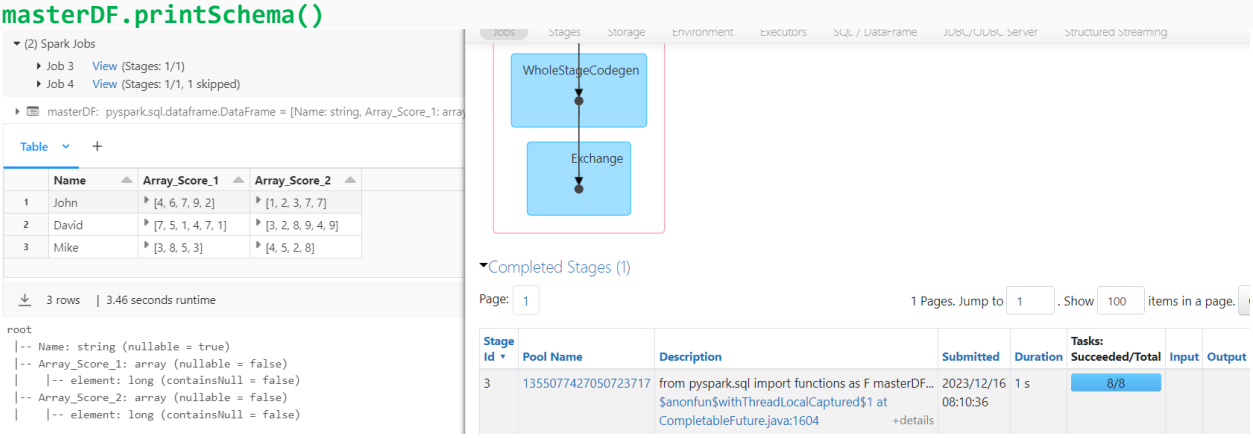
```
array_data =[
    ("John",4,1),
    ("John",6,2),
    ("David",7,3),
    ("Mike",3,4),
    ("David",5,2),
    ("John",7,3),
    ("John",9,7),
    ("David",1,8),
    ("David",4,9),
    ("David",7,4),
    ("Mike",8,5),
    ("Mike",5,2),
    ("Mike",3,8),
    ("John",2,7),
    ("David",1,9)]
array_schema =["Name","Score1","Score2"]
arrayDF= spark.createDataFrame(array_data,array_schema)
display(arrayDF)
```



#Convert sample dataframe into Array Dataframe

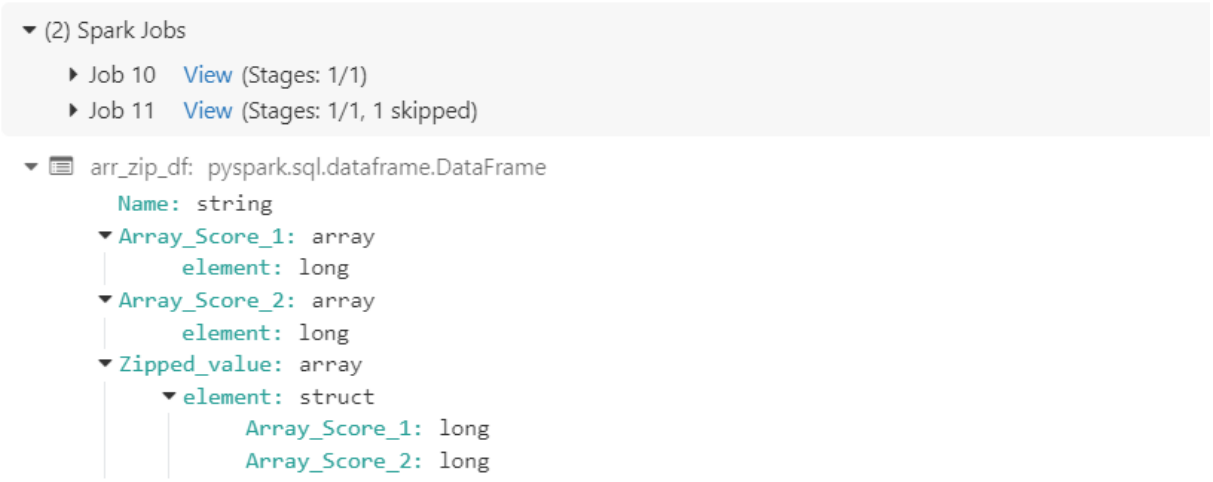
In order to use this function, collect_list, arrays_zip or explode we need to import certain libraries from pyspark.sql.

```
from pyspark.sql import functions as F
masterDF=arrayDF.groupBy("Name").agg(F.collect_list('Score1').alias('Array_Score_1'),
F.collect_list('Score2').alias('Array_Score_2'))
display(masterDF)
masterDF.printSchema()
```



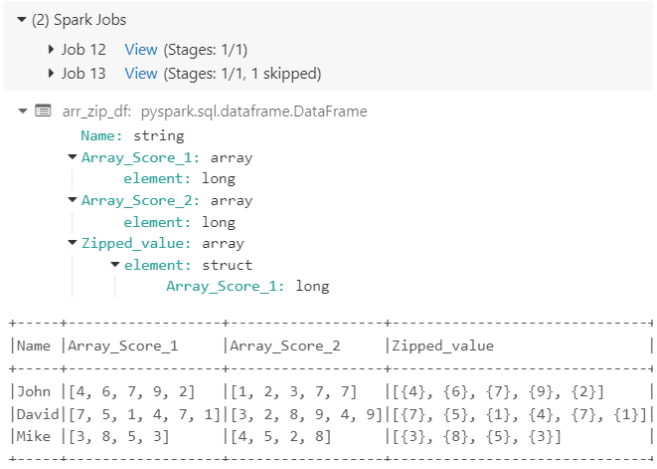
#Apply arrays_zip function on Array DF

```
arr_zip_df=masterDF.withColumn("Zipped_value",F.arrays_zip("Array_Score_1","Array_Sco
re_2"))
arr_zip_df.show(10,False)
```



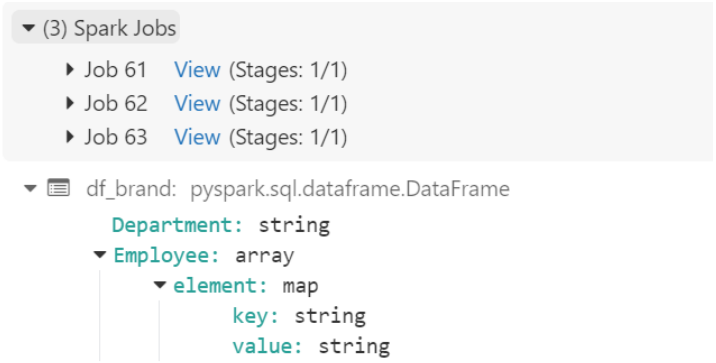

```
+-----+------------------+------------------+-----------------------------------------------+
|Name |Array_Score_1     |Array_Score_2     |Zipped_value                                   |
+-----+------------------+------------------+-----------------------------------------------+
|John |[4, 6, 7, 9, 2]   |[1, 2, 3, 7, 7]   |[{4, 1}, {6, 2}, {7, 3}, {9, 7}, {2, 7}]       |
|David|[7, 5, 1, 4, 7, 1]|[3, 2, 8, 9, 4, 9]|[{7, 3}, {5, 2}, {1, 8}, {4, 9}, {7, 4}, {1, 9}]|
|Mike |[3, 8, 5, 3]      |[4, 5, 2, 8]      |[{3, 4}, {8, 5}, {5, 2}, {3, 8}]               |
+-----+------------------+------------------+-----------------------------------------------+
```

If we pass only 1 array, then it will split array into separate elements as shown below.



#Practical use case to flatten data using arrays_zip and explode

```
empDF=[
    ('Sales_Dept',[{'emp_name':'John','Salary':'1000','yrs_of_service':'10','Age':'33
'},
                   {'emp_name':'David','Salary':'2000','yrs_of_service':'15','Age':'4
0'},
                   {'emp_name':'Nancy','Salary':'8000','yrs_of_service':'20','Age':'4
5'},
                   {'emp_name':'Mike','Salary':'3000','yrs_of_service':'6','Age':'30'
},
                   {'emp_name':'Rosy','Salary':'6000','yrs_of_service':'8','Age':'32'
}]),
    ('HR_Dept',[{'emp_name':'Edwin','Salary':'6000','yrs_of_service':'8','Age':'31'},
                   {'emp_name':'Tomas','Salary':'3000','yrs_of_service':'4','Age':'26
'},
                   {'emp_name':'Sarah','Salary':'12000','yrs_of_service':'22','Age':'
49'},
                   {'emp_name':'Stella','Salary':'15000','yrs_of_service':'25','Age':
'52'},
                   {'emp_name':'Kevin','Salary':'4000','yrs_of_service':'5','Age':'27
'}])
]

df_brand = spark.createDataFrame(empDF,schema =['Department','Employee'])
df_brand.printSchema()
display(df_brand)
```




```
root
 |-- Department: string (nullable = true)
 |-- Employee: array (nullable = true)
 |    |-- element: map (containsNull = true)
 |    |    |-- key: string
 |    |    |-- value: string (valueContainsNull = true)
```

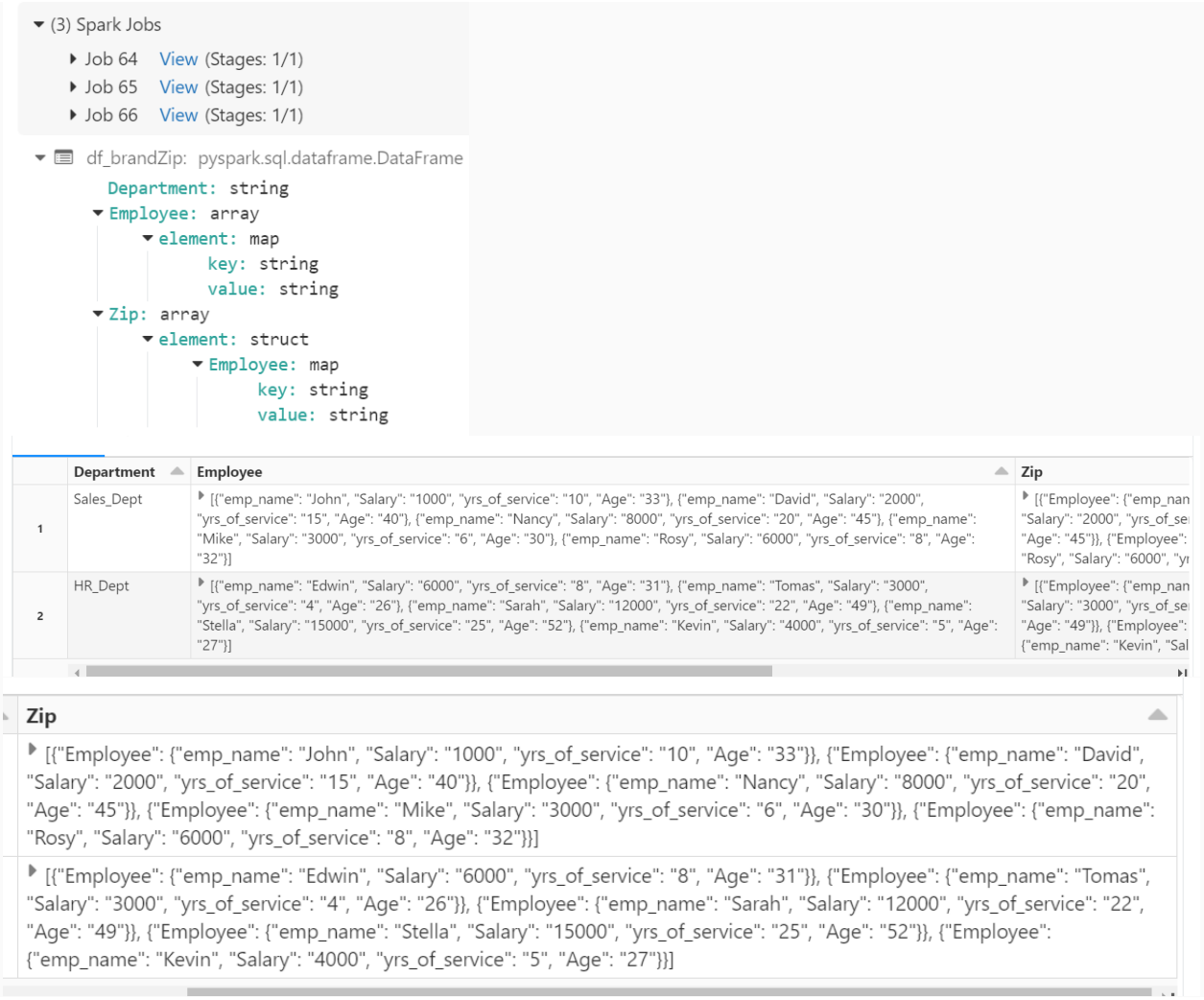
Table +

	Department	Employee
1	Sales_Dept	[{"emp_name": "John", "Salary": "1000", "yrs_of_service": "10", "Age": "33"}, {"emp_name": "David", "Salary": "2000", "yrs_of_service": "15", "Age": "40"}, {"emp_name": "Nancy", "Salary": "8000", "yrs_of_service": "20", "Age": "45"}, {"emp_name": "Mike", "Salary": "3000", "yrs_of_service": "6", "Age": "30"}, {"emp_name": "Rosy", "Salary": "6000", "yrs_of_service": "8", "Age": "32"}]
2	HR_Dept	[{"emp_name": "Edwin", "Salary": "6000", "yrs_of_service": "8", "Age": "31"}, {"emp_name": "Tomas", "Salary": "3000", "yrs_of_service": "4", "Age": "26"}, {"emp_name": "Sarah", "Salary": "12000", "yrs_of_service": "22", "Age": "49"}, {"emp_name": "Stella", "Salary": "15000", "yrs_of_service": "25", "Age": "52"}, {"emp_name": "Kevin", "Salary": "4000", "yrs_of_service": "5", "Age": "27"}]

2 rows | 0.79 seconds runtime

#Apply Arrays_zip

```
df_brandZip=df_brand.withColumn("Zip",F.arrays_zip(df_brand["Employee"]))  → Here for zip column it created separated element. It splitted elements of array into separate elements.
display(df_brandZip)
```



Now we have to split separate elements into separate rows , for that we can use explode function and we can achieve that.

```
df_brand_exp =df_brandZip.withColumn("Explode",F.explode(df_brandZip.Zip))
display(df_brand_exp)
```

(3) Spark Jobs

- Job 67 View (Stages: 1/1)
- Job 68 View (Stages: 1/1)
- Job 69 View (Stages: 1/1)

df_brand_exp: pyspark.sql.dataframe.DataFrame

```
Department: string
Employee: array
    element: map
        key: string
        value: string
Zip: array
    element: struct
        Employee: map
            key: string
            value: string
Explode: struct
    Employee: map
        key: string
        value: string
```

Table +

	Department	Employee	Zip
1	Sales_Dept	[{"emp_name": "John", "Salary": "1000", "yrs_of_service": "10", "Age": "33"}, {"emp_name": "David", "Salary": "2000", "yrs_of_service": "15", "Age": "40"}, {"emp_name": "Nancy", "Salary": "8000", "yrs_of_service": "20", "Age": "45"}, {"emp_name": "Mike", "Salary": "3000", "yrs_of_service": "6", "Age": "30"}, {"emp_name": "Rosy", "Salary": "6000", "yrs_of_service": "8", "Age": "32"}]	[{"Employee": {"emp_r "Salary": "2000", "yrs_of "Age": "45"}}, {"Employe "Rosy", "Salary": "6000",
2	Sales_Dept	[{"emp_name": "John", "Salary": "1000", "yrs_of_service": "10", "Age": "33"}, {"emp_name": "David", "Salary": "2000", "yrs_of_service": "15", "Age": "40"}, {"emp_name": "Nancy", "Salary": "8000", "yrs_of_service": "20", "Age": "45"}, {"emp_name": "Mike", "Salary": "3000", "yrs_of_service": "6", "Age": "30"}, {"emp_name": "Rosy", "Salary": "6000", "yrs_of_service": "8", "Age": "32"}]	[{"Employee": {"emp_r "Salary": "2000", "yrs_of "Age": "45"}}, {"Employe "Rosy", "Salary": "6000",
3	Sales_Dept	[{"emp_name": "John", "Salary": "1000", "yrs_of_service": "10", "Age": "33"}, {"emp_name": "David", "Salary": "2000", "yrs_of_service": "15", "Age": "40"}, {"emp_name": "Nancy", "Salary": "8000", "yrs_of_service": "20", "Age": "45"}, {"emp_name": "Mike", "Salary": "3000", "yrs_of_service": "6", "Age": "30"}, {"emp_name": "Rosy", "Salary": "6000", "yrs_of_service": "8", "Age": "32"}]	[{"Employee": {"emp_r "Salary": "2000", "yrs_of "Age": "45"}}, {"Employe "Rosy", "Salary": "6000",
4	Sales_Dept	[{"emp_name": "John", "Salary": "1000", "yrs_of_service": "10", "Age": "33"}, {"emp_name": "David", "Salary": "2000", "yrs_of_service": "15", "Age": "40"}, {"emp_name": "Nancy", "Salary": "8000", "yrs_of_service": "20", "Age": "45"}, {"emp_name": "Mike", "Salary": "3000", "yrs_of_service": "6", "Age": "30"}, {"emp_name": "Rosy", "Salary": "6000", "yrs_of_service": "8", "Age": "32"}]	[{"Employee": {"emp_r "Salary": "2000", "yrs_of "Age": "45"}}, {"Employe "Rosy", "Salary": "6000",
5	Sales_Dept	[{"emp_name": "John", "Salary": "1000", "yrs_of_service": "10", "Age": "33"}, {"emp_name": "David", "Salary": "2000", "yrs_of_service": "15", "Age": "40"}, {"emp_name": "Nancy", "Salary": "8000", "yrs_of_service": "20", "Age": "45"}, {"emp_name":	[{"Employee": {"emp_r "Salary": "2000", "yrs_of

		Explode
1	: "1000", "yrs_of_service": "10", "Age": "33"}}, {"Employee": {"emp_name": "David", 40"}}, {"Employee": {"emp_name": "Nancy", "Salary": "8000", "yrs_of_service": "20", e", "Salary": "3000", "yrs_of_service": "6", "Age": "30"}}, {"Employee": {"emp_name": Age": "32"}}]	{"Employee": {"emp_name": "John", "Salary": "1000", "yrs_of_service": "10", "Age": "33"}}
2	: "1000", "yrs_of_service": "10", "Age": "33"}}, {"Employee": {"emp_name": "David", 40"}}, {"Employee": {"emp_name": "Nancy", "Salary": "8000", "yrs_of_service": "20", e", "Salary": "3000", "yrs_of_service": "6", "Age": "30"}}, {"Employee": {"emp_name": Age": "32"}}]	{"Employee": {"emp_name": "David", "Salary": "2000", "yrs_of_service": "15", "Age": "40"}}
3	: "1000", "yrs_of_service": "10", "Age": "33"}}, {"Employee": {"emp_name": "David", 40"}}, {"Employee": {"emp_name": "Nancy", "Salary": "8000", "yrs_of_service": "20", e", "Salary": "3000", "yrs_of_service": "6", "Age": "30"}}, {"Employee": {"emp_name": Age": "32"}}]	{"Employee": {"emp_name": "Nancy", "Salary": "8000", "yrs_of_service": "20", "Age": "45"}}
4	: "1000", "yrs_of_service": "10", "Age": "33"}}, {"Employee": {"emp_name": "David", 40"}}, {"Employee": {"emp_name": "Nancy", "Salary": "8000", "yrs_of_service": "20", e", "Salary": "3000", "yrs_of_service": "6", "Age": "30"}}, {"Employee": {"emp_name": Age": "32"}}]	{"Employee": {"emp_name": "Mike", "Salary": "3000", "yrs_of_service": "6", "Age": "30"}}
5	: "1000", "yrs_of_service": "10", "Age": "33"}}, {"Employee": {"emp_name": "David", 40"}}, {"Employee": {"emp_name": "Nancy", "Salary": "8000", "yrs_of_service": "20",	{"Employee": {"emp_name": "Rosy", "Salary": "6000", "yrs_of_service": "8", "Age": "32"}}

We have created separate row for each employee.

#Flatten fields from exploded list

Now if we want to convert these key values into separate columns

```
df_brand_output=df_brand_exp.withColumn("employee_emp_name",df_brand_exp['Explode.Employee.emp_name'])\
                            .withColumn("employee_yrs_of_service",df_brand_exp['Explode.Employee.yrs_of_service'])\
                            .withColumn("employee_salary",df_brand_exp['Explode.Employee.Salary'])\
                            .withColumn("Employee_Age",df_brand_exp['Explode.Employee.Age']).drop("Explode").drop("Zip").drop("Employee")
```

```
df_brand_output: pyspark.sql.dataframe.DataFrame
  Department: string
  employee_emp_name: string
  employee_yrs_of_service: string
  employee_salary: string
  Employee_Age: string
```

```
display(df_brand_output)
```

	Department	employee_emp_name	employee_yrs_of_service	employee_salary	Employee_Age
1	Sales_Dept	John	10	1000	33
2	Sales_Dept	David	15	2000	40
3	Sales_Dept	Nancy	20	8000	45
4	Sales_Dept	Mike	6	3000	30
5	Sales_Dept	Rosy	8	6000	32
6	HR_Dept	Edwin	8	6000	31
7	HR_Dept	Tomas	4	3000	26

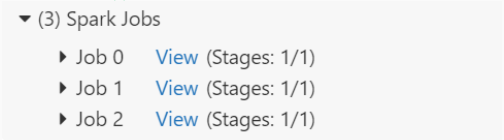
10 rows | 0.53 seconds runtime

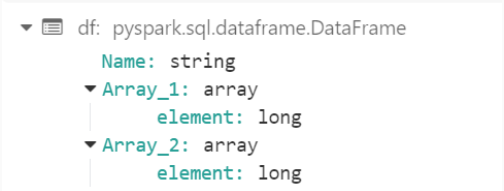
Array_intersect function : This functions are mainly used when working with array data. Array_intersect is one of pyspark function that is used to return common elements across two arrays without any duplication which means when we input two array into this function, it will just filter out only common elements that are present across these two array data.

If we are going to apply array_intersect,then it will return only common elements present across these two array which means it will first start comparing first element in first array is this common across two array then it would be dropped. Next coming to 2nd element in first array, is it common across two array , if this is also not available then it will be dropped. Next coming to 3rd element in first array, if it is common across 2 array, then that element will be considered for final output etc. and if it has duplicates, it will remove duplicate element.

```
empDF=[
    ('John',[4,6,7,9,2],[1,2,3,7,7]),
    ('David',[7,5,1,4,7,1],[3,2,8,9,4,9]),
    ('Mike',[3,9,1,6,2],[1,2,3,5,8])
```

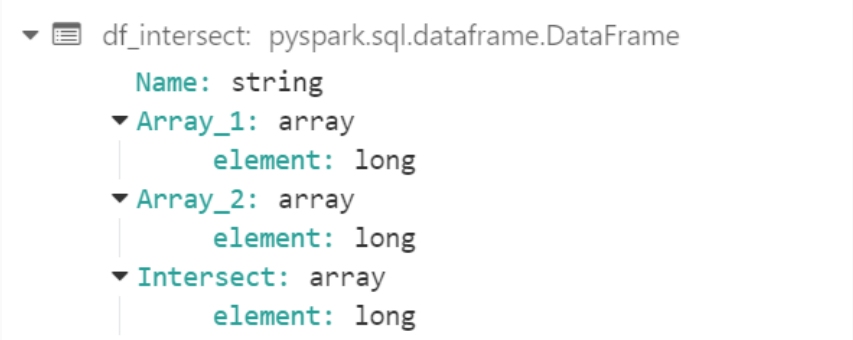
```
    ]
df= spark.createDataFrame(empDF,["Name","Array_1","Array_2"])
df.show()
```






```
+-----+-----------------+-----------------+
| Name|          Array_1|          Array_2|
+-----+-----------------+-----------------+
| John|   [4, 6, 7, 9, 2]|   [1, 2, 3, 7, 7]|
|David|[7, 5, 1, 4, 7, 1]|[3, 2, 8, 9, 4, 9]|
| Mike|   [3, 9, 1, 6, 2]|   [1, 2, 3, 5, 8]|
+-----+-----------------+-----------------+
```

```
#Apply array_intersect
from pyspark.sql import functions as F
df_intersect=df.withColumn("Intersect",F.array_intersect(df["Array_1"],df["Array_2"])
)
df_intersect.show()
```

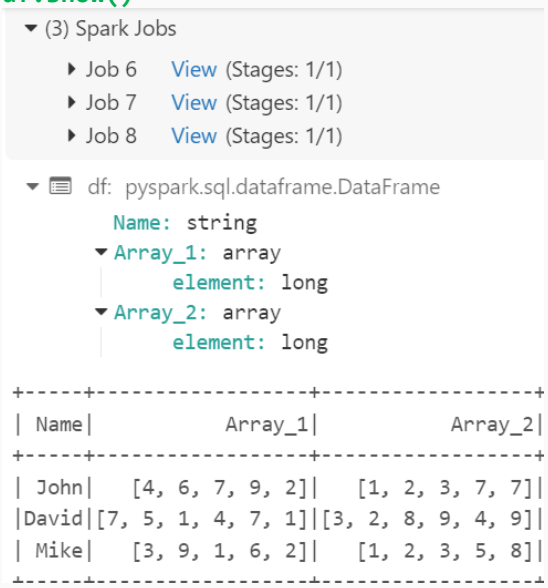



```
+-----+-----------------+-----------------+---------+
| Name|          Array_1|          Array_2|Intersect|
+-----+-----------------+-----------------+---------+
| John|   [4, 6, 7, 9, 2]|   [1, 2, 3, 7, 7]|   [7, 2]|
|David|[7, 5, 1, 4, 7, 1]|[3, 2, 8, 9, 4, 9]|      [4]|
| Mike|   [3, 9, 1, 6, 2]|   [1, 2, 3, 5, 8]|[3, 1, 2]|
+-----+-----------------+-----------------+---------+
```

Array_except : is one of pyspark function that returns list of elements which are present in first array but not in second array which means it is returning all the uncommon elements only from 1st array when compared to 2nd array.

When we are applying array_except what happens is it will start comparing 2 arrays. If any elements present in first array but not in 2nd array, then that would be returned as output.so here in syntax we have to give order of array. For example, if we give Array1 as a first parameter ,which means it will return all elements from 1st array. In case if we want to find out uncommon elements from array2, then array2 will be given as first parameter and array to be compared is given in the second place.

```
empDF=[
    ('John',[4,6,7,9,2],[1,2,3,7,7]),
    ('David',[7,5,1,4,7,1],[3,2,8,9,4,9]),
    ('Mike',[3,9,1,6,2],[1,2,3,5,8])
     ]
df= spark.createDataFrame(empDF,["Name","Array_1","Array_2"])
df.show()
```




```
+-----+------------------+------------------+
| Name|           Array_1|           Array_2|
+-----+------------------+------------------+
| John|   [4, 6, 7, 9, 2]|   [1, 2, 3, 7, 7]|
|David|[7, 5, 1, 4, 7, 1]|[3, 2, 8, 9, 4, 9]|
| Mike|   [3, 9, 1, 6, 2]|   [1, 2, 3, 5, 8]|
+-----+------------------+------------------+
```

```
from pyspark.sql import functions as F
df_except =df.withColumn("Arrayexcept",F.array_except(df["Array_1"],df["Array_2"]))
df_except.show()
```

(3) Spark Jobs

- Job 9 View (Stages: 1/1)
- Job 10 View (Stages: 1/1)
- Job 11 View (Stages: 1/1)

df_except: pyspark.sql.dataframe.DataFrame

```
Name: string
Array_1: array
    element: long
Array_2: array
    element: long
Arrayexcept: array
    element: long
```

```
+-----+------------------+------------------+-----------+
| Name|           Array_1|           Array_2|Arrayexcept|
+-----+------------------+------------------+-----------+
| John|   [4, 6, 7, 9, 2]|   [1, 2, 3, 7, 7]|  [4, 6, 9]|
|David|[7, 5, 1, 4, 7, 1]|[3, 2, 8, 9, 4, 9]|  [7, 5, 1]|
| Mike|   [3, 9, 1, 6, 2]|   [1, 2, 3, 5, 8]|     [9, 6]|
+-----+------------------+------------------+-----------+
```

```
from pyspark.sql import functions as F
df_except =df.withColumn("Arrayexcept",F.array_except(df["Array_2"],df["Array_1"]))
df_except.show()
```

(3) Spark Jobs

- Job 12 View (Stages: 1/1)
- Job 13 View (Stages: 1/1)
- Job 14 View (Stages: 1/1)

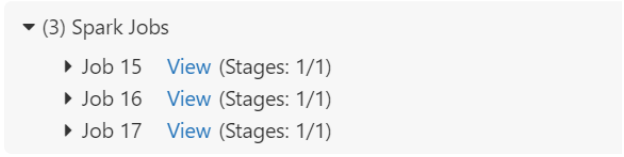
df_except: pyspark.sql.dataframe.DataFrame

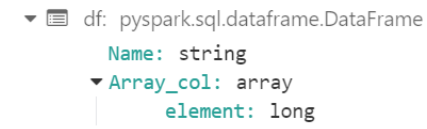
```
Name: string
Array_1: array
    element: long
Array_2: array
    element: long
Arrayexcept: array
    element: long
```

```
+-----+------------------+------------------+------------+
| Name|           Array_1|           Array_2| Arrayexcept|
+-----+------------------+------------------+------------+
| John|   [4, 6, 7, 9, 2]|   [1, 2, 3, 7, 7]|      [1, 3]|
|David|[7, 5, 1, 4, 7, 1]|[3, 2, 8, 9, 4, 9]|[3, 2, 8, 9]|
| Mike|   [3, 9, 1, 6, 2]|   [1, 2, 3, 5, 8]|      [5, 8]|
+-----+------------------+------------------+------------+
```

Array_sort function Array_sort is array function used to sort elements within array in ascending order

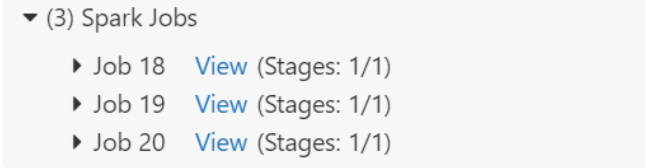
```
empDF=[
    ('John',[4,6,7,9,2]),
    ('David',[7,5,1,4,7,1]),
    ('Mike',[3,9,1,6,2])
]
df= spark.createDataFrame(empDF,["Name","Array_col"])
df.show()
```

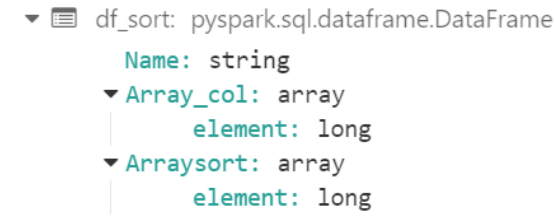





```
+-----+-----------------+
| Name|        Array_col|
+-----+-----------------+
| John|  [4, 6, 7, 9, 2]|
|David|[7, 5, 1, 4, 7, 1]|
| Mike|  [3, 9, 1, 6, 2]|
+-----+-----------------+
```

```
from pyspark.sql import functions as F
df_sort =df.withColumn("Arraysort",F.array_sort(df["Array_col"]))
df_sort.show()
```





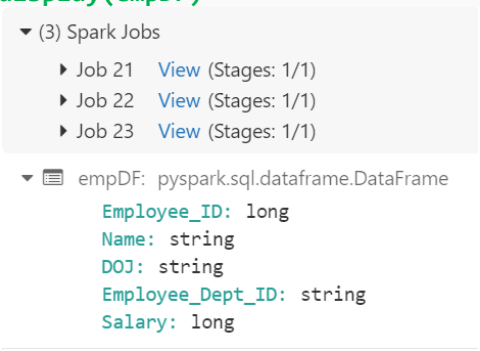

```
+-----+-----------------+-----------------+
| Name|        Array_col|        Arraysort|
+-----+-----------------+-----------------+
| John|  [4, 6, 7, 9, 2]|  [2, 4, 6, 7, 9]|
|David|[7, 5, 1, 4, 7, 1]|[1, 1, 4, 5, 7, 7]|
| Mike|  [3, 9, 1, 6, 2]|  [1, 2, 3, 6, 9]|
+-----+-----------------+-----------------+
```

Join(not a table join) , String Join This is a python function that returns string by joining all the elements of an iterable data such as list, tuple etc. separated by a string separator.

Syntax: **String_Separator.join(iterable data)**

String Separator should be , ; or string

```
employee_data =[(10,"Michael Robinson","1996-06-01","100",2000),
                (20,"James Wood","2003-03-01","200",8000),
                (30,"Chris Andrews","2005-04-01","100",6000),
                (40,"Mark Bond","2008-10-01","100",7000),
                (50,"Steve Watson","1996-02-01","400",1000),
                (60,"Mathews Simon","1998-11-01","500",5000),
                (70,"Peter Paul","2011-04-01","600",5000)]
employee_schema=["Employee_ID","Name","DOJ","Employee_Dept_ID","Salary"]
empDF=spark.createDataFrame(employee_data,employee_schema)
display(empDF)
```



	Employee_ID	Name	DOJ	Employee_Dept_ID	Salary
1	10	Michael Robinson	1996-06-01	100	2000
2	20	James Wood	2003-03-01	200	8000
3	30	Chris Andrews	2005-04-01	100	6000
4	40	Mark Bond	2008-10-01	100	7000
5	50	Steve Watson	1996-02-01	400	1000
6	60	Mathews Simon	1998-11-01	500	5000
7	70	Peter Paul	2011-04-01	600	5000

7 rows | 1.31 seconds runtime

#create iterable data → want to get list of columns as elements

```
column_list = empDF.columns
print(column_list)
```

Output: ['Employee_ID', 'Name', 'DOJ', 'Employee_Dept_ID', 'Salary']

#Join string

```
joined_string = ",".join(column_list)  → here data is separated with , so used comma and column_list is having list of elements
print(joined_string)
```

Output: Employee_ID,Name,DOJ,Employee_Dept_ID,Salary output produced is lengthy string which is containing all columns with comma separated value.

USE CASES

This below use case applicable for table join operation

This is useful while joining tables

```
joinstring = " AND ".join(list(map(lambda x:("Target." + x + "=Source."+x),column_list)))
print(joinstring)
```

Output : Target.Employee_ID=Source.Employee_ID AND Target.Name=Source.Name AND Target.DOJ=Source.DOJ AND Target.Employee_Dept_ID=Source.Employee_Dept_ID AND Target.Salary=Source.Salary

---

This is useful while creating update statement

If we want to update target table based on source table ,then we can use this syntax

```
updatestring = ",".join(list(map(lambda x: ("Target." + x + " = Source."+x),column_list)))
print(updatestring)
```

Output: Target.Employee_ID = Source.Employee_ID,Target.Name = Source.Name,Target.DOJ = Source.DOJ,Target.Employee_Dept_ID = Source.Employee_Dept_ID,Target.Salary = Source.Salary

---

This is mainly useful for insert statement

While inserting data into target table then in the insert clause we have to specify list of columns

```
sourceinsertstring =",".join(list(map(lambda x: ("Source." +x),column_list)))
print(sourceinsertstring)
```

Output:
Source.Employee_ID,Source.Name,Source.DOJ,Source.Employee_Dept_ID,Source.Salary

Partition By FUNCTION: This function is mainly used to write data into disc or memory. In order to write data into memory, there are other concepts as well coalesce and repartition.Partition By function used also in window functions that is aggregate functions.

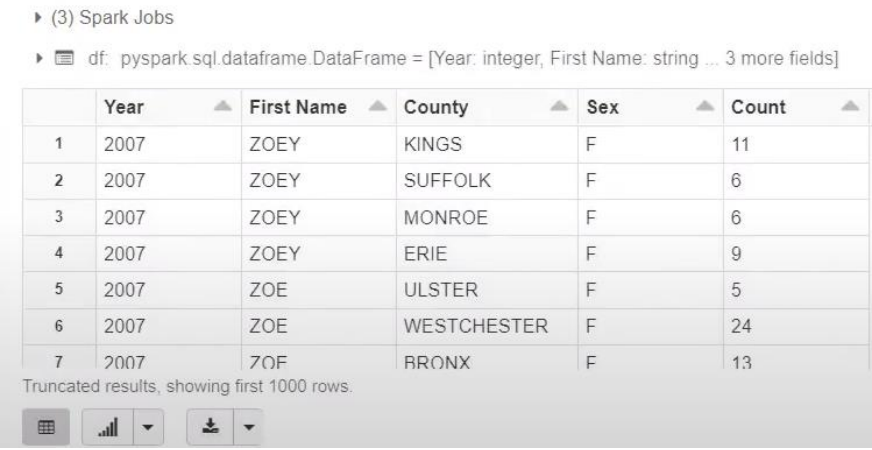
Partition By function used to write dataframe into disc partitioned by specific keys which means in data frame let's say we have a column year, so year is the key. Based

on year we need to partition data . we need to split entire data frame into multiple smaller pieces while writing data into disc then we can go for partition By function. We need to supply key. Based on key, it will split data into multiple partitions. Coming to key, it can be one or more keys. We can give combination of keys also.

Syntax: **df.write.partitionBy(key).csv(Path)**

Eg: #Create Sample Dataframe

```
df=spark.read.format("csv").option("inferSchema",True).option("header",True).option("sep",",").load("/FileStore/tables/baby_names_input/")
display(df)
```

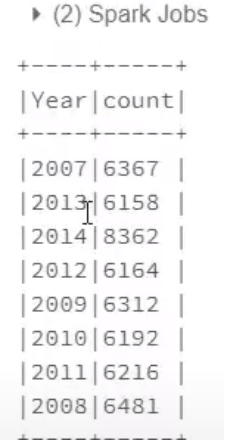


	Year	First Name	County	Sex	Count
1	2007	ZOEY	KINGS	F	11
2	2007	ZOEY	SUFFOLK	F	6
3	2007	ZOEY	MONROE	F	6
4	2007	ZOEY	ERIE	F	9
5	2007	ZOE	ULSTER	F	5
6	2007	ZOE	WESTCHESTER	F	24
7	2007	ZOE	BRONX	F	13

 # Distinct Year List and Count for each Year

If we want to see number of rows for each year then,

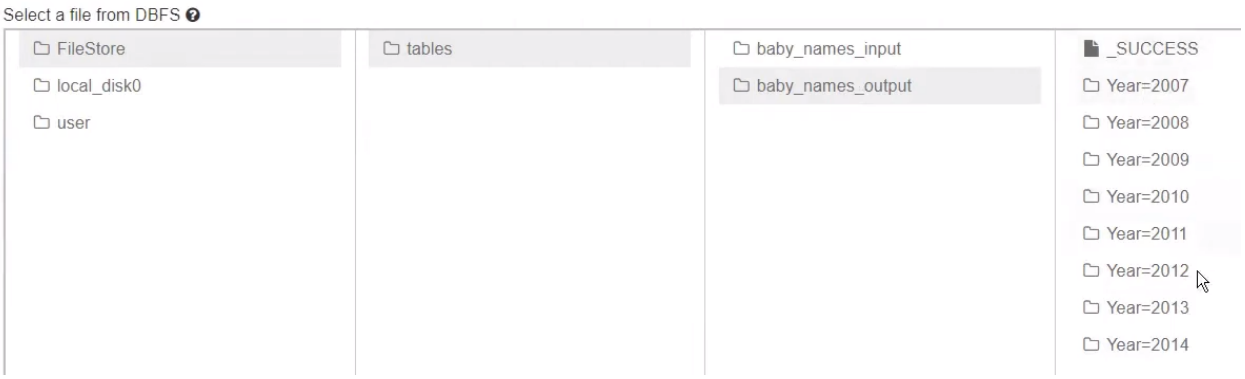
```
df.groupBy("Year").count().show(truncate=False)
```



#Partition by one key column

```
df.write.option("header",True) \
    .partitionBy("Year") \
    .mode("overwrite") \
    .csv("/FileStore/tables/baby_names_output")
```

(1) Spark job is run for above mode.



Here we can see 8 different folders, one folder per year. If we open folder , there is one partition file as shown below and other files in the folder are logs created as part of this program.so 1 partition file created for each year.

```
#Partition By Multiple Key Columns
df.write.option("header",True) \
    .partitionBy("Year","Sex") \
    .mode("overwrite") \
    .csv("/FileStore/tables/baby_names_output")
```

(1) Spark Jobs is ran→ total 16 partitions are created as shown below.

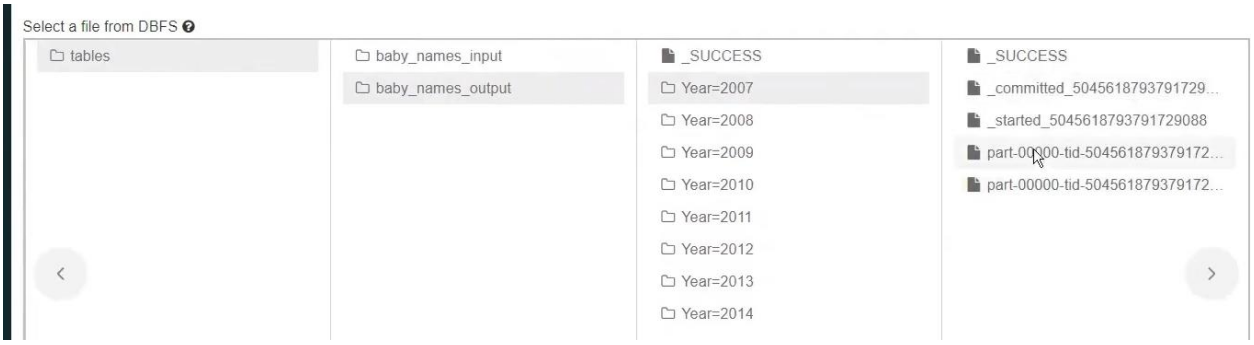
At top level it will be partitioned by year and at sub level it will be partitioned by sex.

#Partition by Key Column along with maximum number of records for each partition

How to limit number of records into particular partition?

```
df.write.option("header",True) \
    .option("maxRecordsPerFile",4200) \
    .partitionBy("Year") \
    .mode("overwrite") \
    .csv("/FileStore/tables/baby_names_output")
```

(1) Spark job is ran.




```
df.groupBy("Year").count().show(truncate=False)
```

▸ (2) Spark Jobs

```
+----+-----+
|Year|count|
+----+-----+
|2007|6367 |
|2013|6158 |
|2014|8362 |
|2012|6164 |
|2009|6312 |
|2010|6192 |
|2011|6216 |
|2008|6481 |
+----+-----+
```

Total 16 partitions created , because each year has 2 partitions and max records per file is 4200 and for 2007 folder, we has 6367 records so 4200 rows in 1 partition file and rest records in another partition file.

HOW TO CALCULATE NUMBER OF RECORDS PER PARTITION IN GIVEN DATAFRAME

```
df=spark.read.format("csv").option("inferSchema",True).option("header",True).option("sep",",").load("/FileStore/tables/baby_names_test/Baby_Names.csv")
display(df)
```

▸ (3) Spark Jobs

	Year	First Name	County	Sex	Count
1	2007	ZOEY	KINGS	F	11
2	2007	ZOEY	SUFFOLK	F	6
3	2007	ZOEY	MONROE	F	6
4	2007	ZOEY	ERIE	F	9
5	2007	ZOE	ULSTER	F	5
6	2007	ZOE	WESTCHESTER	F	24
7	2007	ZOE	BRONX	F	13

#Number of records in data frame

In order to see number of records in data frame for entire data frame, we use df.count()

```
print(df.count())
```

output (2) spark jobs ran

52252

#Default Partition Count

If we want to see number of partitions created for the data frame,

```
print(df.rdd.getNumPartitions())
```

out: 1 partition

#Number of Records per Partition

If we want to see number of records per partition , first we need to import library function.

```
from pyspark.sql.functions import spark_partition_id
df.withColumn("partitionId",spark_partition_id()).groupBy("partitionId").count().show()
```

▸ (2) Spark Jobs

```
+-----------+-----+
|partitionId|count|
+-----------+-----+
|          0|52252|
```

So here in single partition, all 52252 rows we are getting.

#Repartition the data frame to 5

```
df_5 =df.select(df.Year,df.Country,df.Sex,df.Count).repartition(5)
```

df_5 got created with 5 partitions.

In order to ensure it has created 5 partitions or not we can run below command.

```
#Get Number of Partitions in Dataframe
print(df_5.rdd.getNumPartitions())
```

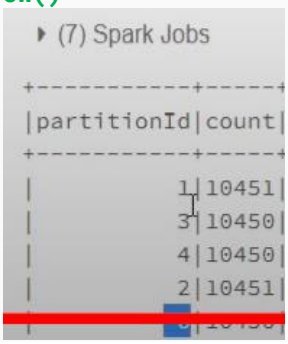
(1) Spark job is ran.

```
Out: 5
```

#Get Number of Records per partition

We don't need to import function again and again already imported at above command.so not importing spark_partition_id.

```
df_5.withColumn("partitionId",spark_partition_id()).groupBy("partitionId").count().show()
```



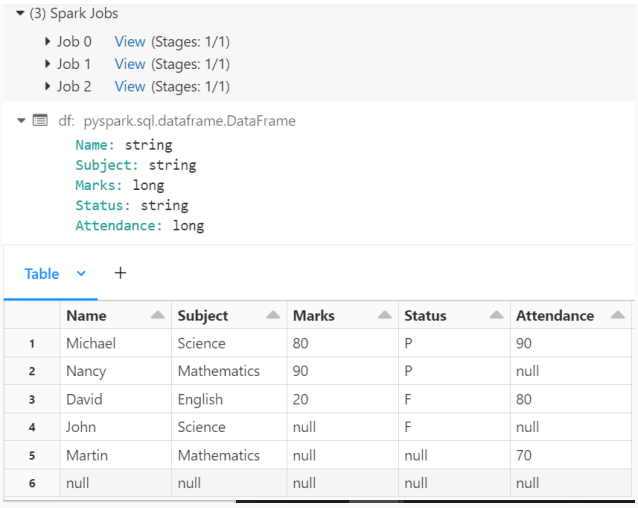
It has created 5 partitions from partitionId 0 to 4.Repartitions will be always creating partitions evenly. It will distribute data across partitions evenly. That is the reason we cannot see huge difference in the number of count. We cannot see big variation. It is almost same

---

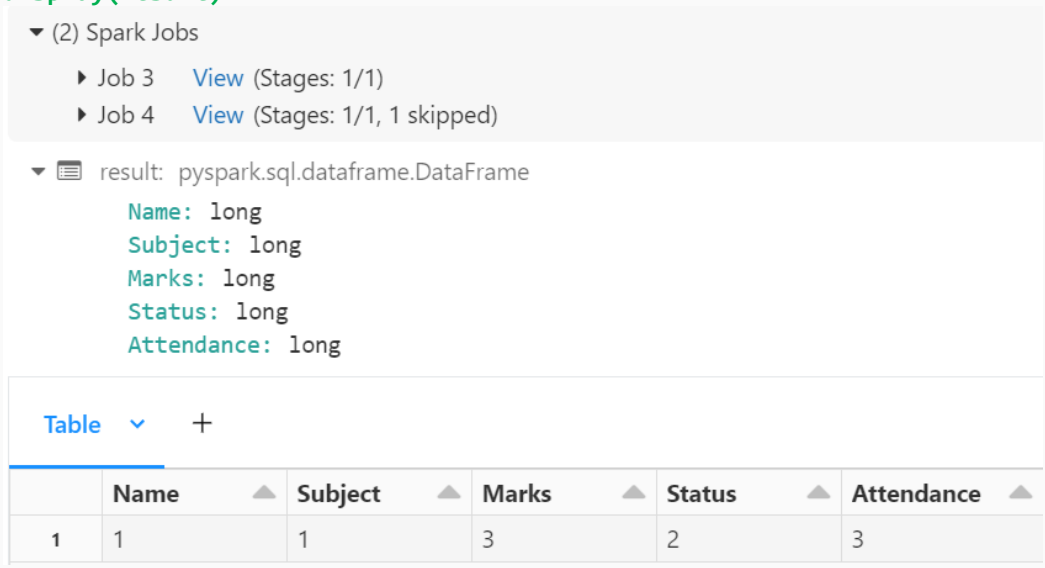
NULL COUNT OF ALL COLUMNS IN DATA FRAME

While performing ETL operation in staging layer ,it is important to perform number of sanity checks such as dropping duplicates, replacing null values, dropping null values etc. This statistics is key indicator of quality check in most of the projects.

```
data_student =[("Michael","Science",80,"P",90),
               ("Nancy","Mathematics",90,"P",None),
               ("David","English",20,"F",80),
               ("John","Science",None,"F",None),
               ("Martin","Mathematics",None,None,70),
               (None,None,None,None,None)]

schema =["Name","Subject","Marks","Status","Attendance"]

df= spark.createDataFrame(data_student,schema)

display(df)
```




```
#Find Null occurrences of each column in data frame

from pyspark.sql.functions import col,count,when
result = df.select([count(when(col(c).isNull(),c)).alias(c) for c in df.columns])
display(result)
```




```
                    #FIND TOP AND BOTTOM N ROWS PER GROUP

data_student=[("Michael","Physics",80,"P",90),
              ("Michael","Chemistry",67,"P",90),
              ("Michael","Mathematics",78,"P",90),
              ("Nancy","Physics",30,"F",80),
              ("Nancy","Chemistry",59,"P",80),
              ("Nancy","Mathematics",75,"P",80),
              ("David","Physics",90,"P",70),
              ("David","Chemistry",87,"P",70),
```

```
            ("David","Mathematics",97,"P",70),
            ("John","Physics",33,"F",60),
            ("John","Chemistry",28,"F",60),
            ("John","Mathematics",52,"P",60),
            ("Blessy","Physics",89,"P",75),
            ("Blessy","Chemistry",76,"P",75),
            ("Blessy","Mathematics",63,"P",75)]
schema =["Name","Subject","Mark","Status","Attendance"]
df= spark.createDataFrame(data_student,schema)
display(df)
```

(3) Spark Jobs

- Job 10 View (Stages: 1/1)
- Job 11 View (Stages: 1/1)
- Job 12 View (Stages: 1/1)

df: pyspark.sql.dataframe.DataFrame

```
Name: string
Subject: string
Mark: long
Status: string
Attendance: long
```

Table +

	Name	Subject	Mark	Status	Attendance
1	Michael	Physics	80	P	90
2	Michael	Chemistry	67	P	90
3	Michael	Mathematics	78	P	90
4	Nancy	Physics	30	F	80
5	Nancy	Chemistry	59	P	80
6	Nancy	Mathematics	75	P	80
7	David	Phvsics	90	P	70

15 rows | 1.10 seconds runtime

```
#create rank within each group of Name

from pyspark.sql.window import Window

from pyspark.sql.functions import col,row_number

windowDept =Window.partitionBy("name").orderBy(col("Mark").desc())

df2=df.withColumn("row",row_number().over(windowDept)).orderBy("name","row")
```

```
display(df2)
```

(2) Spark Jobs

- Job 13 View (Stages: 1/1)
- Job 14 View (Stages: 1/1, 1 skipped)

df2: pyspark.sql.dataframe.DataFrame

```
Name: string
Subject: string
Mark: long
Status: string
Attendance: long
row: integer
```

	Name	Subject	Mark	Status	Attendance	row
1	Blessy	Physics	89	P	75	1
2	Blessy	Chemistry	76	P	75	2
3	Blessy	Mathematics	63	P	75	3
4	David	Mathematics	97	P	70	1
5	David	Physics	90	P	70	2
6	David	Chemistry	87	P	70	3
7	John	Mathematics	52	P	60	1

15 rows | 3.11 seconds runtime

```
#Get Top N rows per Group of Name
df3=df2.filter(col("row")<=1)  → If I want to get top 2 records, we can keep 2 here.
we can get N rows per group
df3.display()
```

(2) Spark Jobs

- Job 18 View (Stages: 1/1)
- Job 19 View (Stages: 1/1, 1 skipped)

df3: pyspark.sql.dataframe.DataFrame

```
Name: string
Subject: string
Mark: long
Status: string
Attendance: long
row: integer
```

	Name	Subject	Mark	Status	Attendance	row
1	Blessy	Physics	89	P	75	1
2	David	Mathematics	97	P	70	1
3	John	Mathematics	52	P	60	1
4	Michael	Physics	80	P	90	1
5	Nancy	Mathematics	75	P	80	1

5 rows | 1.48 seconds runtime

```
#Create rank within each group of Subject
windowDept =Window.partitionBy("Subject").orderBy(col("Mark").desc())
df4=df.withColumn("row",row_number().over(windowDept)).orderBy("name","row")
display(df4)
```

(2) Spark Jobs

Job 20 View (Stages: 1/1)

Job 21 View (Stages: 1/1, 1 skipped)

df4: pyspark.sql.dataframe.DataFrame = [Name: string, Subject: string ... 4 more fields]

Table +

	Name	Subject	Mark	Status	Attendance	row
1	Blessy	Chemistry	76	P	75	2
2	Blessy	Physics	89	P	75	2
3	Blessy	Mathematics	63	P	75	4
4	David	Chemistry	87	P	70	1
5	David	Mathematics	97	P	70	1
6	David	Physics	90	P	70	1
7	John	Physics	33	F	60	4

15 rows | 1.76 seconds runtime

```
#Get Top N rows per Group of Subject
df5=df4.filter(col("row")<=1)
df5.display()
```

(2) Spark Jobs

df5: pyspark.sql.dataframe.DataFrame = [Name: string, Subject: string ... 4 more fields]

Table +

	Name	Subject	Mark	Status	Attendance	row
1	David	Chemistry	87	P	70	1
2	David	Mathematics	97	P	70	1
3	David	Physics	90	P	70	1

3 rows | 1.25 seconds runtime

```
#Reverse Rank to get Bottom N Rows Per Group
windowDept =Window.partitionBy("Subject").orderBy(col("Mark"))
df6=df.withColumn("row",row_number().over(windowDept)).orderBy("name","row")
display(df6)
```

(2) Spark Jobs

df6: pyspark.sql.dataframe.DataFrame = [Name: string, Subject: string ... 4 more fields]

Table +

	Name	Subject	Mark	Status	Attendance	row
1	Blessy	Mathematics	63	P	75	2
2	Blessy	Chemistry	76	P	75	4
3	Blessy	Physics	89	P	75	4
4	David	Chemistry	87	P	70	5
5	David	Mathematics	97	P	70	5
6	David	Physics	90	P	70	5
7	John	Chemistry	28	F	60	1

15 rows | 1.27 seconds runtime

```
#Get Bottom N Rows Per Group
df7=df6.filter(col("row")<=1)
df7.display()
```

(2) Spark Jobs

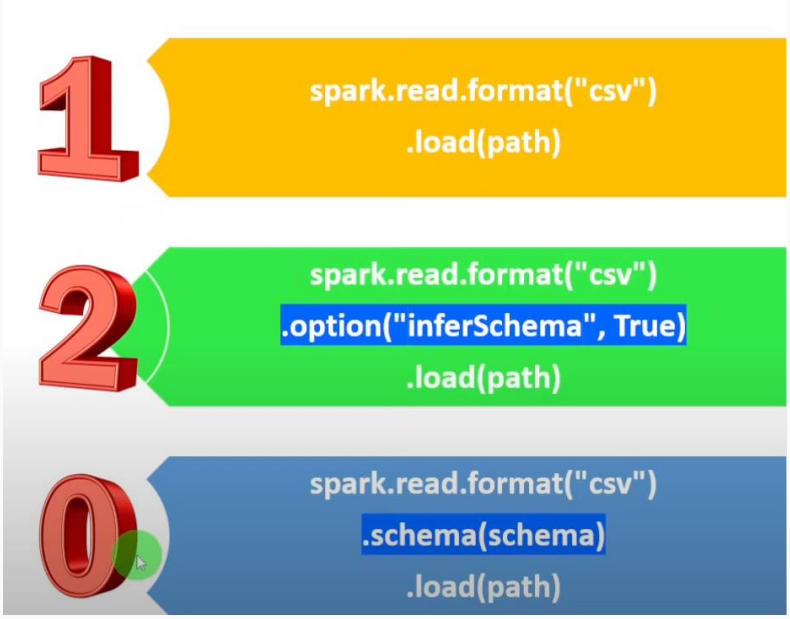
df7: pyspark.sql.dataframe.DataFrame = [Name: string, Subject: string ... 4 more fields]

Table +

	Name	Subject	Mark	Status	Attendance	row
1	John	Chemistry	28	F	60	1
2	John	Mathematics	52	P	60	1
3	Nancy	Physics	30	F	80	1

3 rows | 1.07 seconds runtime

HOW MANY SPARK JOBS ARE GETTING CREATED?



As per spark definition, spark engine will create a job whenever it has to touch the data which is residing in the file.

In the first method as shown above, when we have to read csv file, spark engine needs to know number of columns in order to define the dataframe, so it will scan the file and it will fetch only 1 record out of that and it will calculate the number of columns . so, in this method it has to touch data file only once in order to get number of columns, so it will create 1 job.

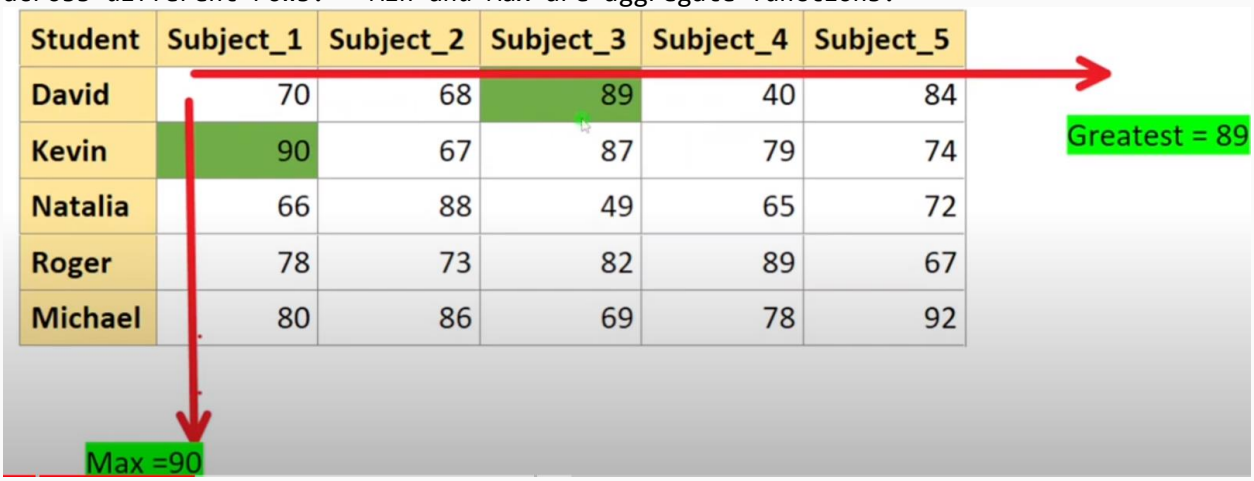
Coming to 2nd method, we are specifying inferSchema.in this method first it has to determine number of columns for the data frame.so for that it will scan the file only once and it will fetch only one record out of that, and it will calculate the number of columns , that is 1 job . once that is done, it has to perform inferSchema which means it has to get data type for each of the column so for that purpose it will scan the entire data once again and it will get the data type, it will infer the schema. So, for that reason it will touch the data file 2nd time.so it will create 2 different jobs.

Third method is we are explicitly specifying the schema. From schema spark engine will get number of columns along with its data type. So that's the reason it is no necessity for this method to scan the data until some action is called.This particular method will create 0 jobs.

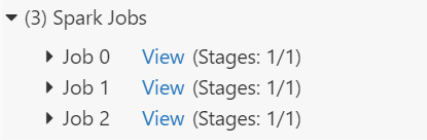
DIFFERENCE BETWEEN GREATEST vs MAX:

Greatest is used to find maximum value across the columns. When we are giving columns as an input then greatest will find maximum value across those columns. Max is function which can be applied across multiple rows .Max will take multiple rows as an input and it will find maximum among rows. Greatest is used to find maximum values across columns whereas Max is used to find maximum value across different rows. Least function used

to find minimum value across columns whereas Min function used to find minimum value across different rows. Min and Max are aggregate functions.




```
input_data =[("David",70,68,89,40,84),
             ("Kevin",90,67,87,79,74),
             ("Natalia",66,88,49,65,72),
             ("Roger",78,73,82,89,67),
             ("Michael",80,86,69,78,92)]
schema=["Student","Subject1","Subject2","Subject3","Subject4","Subject5"]
df= spark.createDataFrame(input_data,schema)
df.display()
```



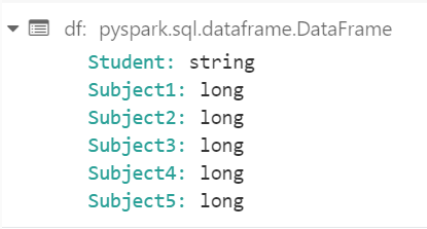


Table +

	Student	Subject1	Subject2	Subject3	Subject4	Subject5
1	David	70	68	89	40	84
2	Kevin	90	67	87	79	74
3	Natalia	66	88	49	65	72
4	Roger	78	73	82	89	67
5	Michael	80	86	69	78	92

5 rows | 15.25 seconds runtime

```
#Greatest of Columns
from pyspark.sql.functions import greatest
greatDF=df.withColumn("Greatest",greatest("Subject1","Subject2","Subject3","Subject4"
,"Subject5"))
greatDF.display()
```

(3) Spark Jobs

- Job 3 View (Stages: 1/1)
- Job 4 View (Stages: 1/1)
- Job 5 View (Stages: 1/1)

greatDF: pyspark.sql.dataframe.DataFrame = [Student: string, Subject1: long ... 5 more fields]

Table +

	Student	Subject1	Subject2	Subject3	Subject4	Subject5	Greatest
1	David	70	68	89	40	84	89
2	Kevin	90	67	87	79	74	90
3	Natalia	66	88	49	65	72	88
4	Roger	78	73	82	89	67	89
5	Michael	80	86	69	78	92	92

5 rows | 1.77 seconds runtime

```
#Least of Columns
from pyspark.sql.functions import least
leastDF=df.withColumn("Least",least("Subject1","Subject2","Subject3","Subject4","Subj
ect5"))
leastDF.display()
```

(3) Spark Jobs

- Job 6 View (Stages: 1/1)
- Job 7 View (Stages: 1/1)
- Job 8 View (Stages: 1/1)

leastDF: pyspark.sql.dataframe.DataFrame = [Student: string, Subject1: long ... 5 more fields]

Table +

	Student	Subject1	Subject2	Subject3	Subject4	Subject5	Least
1	David	70	68	89	40	84	40
2	Kevin	90	67	87	79	74	67
3	Natalia	66	88	49	65	72	49
4	Roger	78	73	82	89	67	67
5	Michael	80	86	69	78	92	69

```
#Max of Rows
df.agg({'Subject1':'max'}).display()
```

(2) Spark Jobs

- Job 9 View (Stages: 1/1)
- Job 10 View (Stages: 1/1, 1 skipped)

Table +

	max(Subject1)
1	90

1 row | 2.83 seconds runtime

```
#Min of Rows

df.agg({'Subject1':'min','Subject2':'min','Subject3':'min','Subject4':'min','Subject5':'min'}).display()
```

(2) Spark Jobs

- Job 11 View (Stages: 1/1)
- Job 12 View (Stages: 1/1, 1 skipped)

Table +

	min(Subject3)	min(Subject4)	min(Subject1)	min(Subject5)	min(Subject2)
1	49	40	66	67	67

1 row | 1.25 seconds runtime