**What is Spark architecture?**

* Spark follows master slave architecture where the Driver node acts master and the worker node acts as slave.
* When the spark job is submitted ,spark context is created in the driver node (spark context is the entry point to the cluster
* Now the driver requests the resource manager(cluster manager) for the resources. Once the resource are allocated driver send the tasks to the executors present in the worker node where the computations take place
* Driver is responsible for coordinating jobs,task scheduling, creating DAG’s
* Once the tasks are completed, executors send it to driver and the driver collects and sends it to application

**Cluster Manager-**

**YARN**, **Mesos**, **Kubernetes**, or **Standalone mode**

..

**Cluster configuration - thumb rule for optimal performance**

I would have said, the size of cluster or type of cluster depends on its usage? What purpose its used, who all will be using it? Is it for a single user or shared? Does it need more compute for parallel processing? If it's not for big scale usage then a single node single user cluster would do, if not then keep the size minimal keeping in mind the cost impact of it.

**To process 25 gb of data**

Max partition bytes is by default set to 128 mb

25gb = 25 \* 1024 =25600

Number of partitions = 25600/128 = 200

**How many cpu cores are required**

Cpu cores = number of partitions = 200 cores

**How many executors are required**

To get better job performance researchers found that we can take 2-5 cores for each executor

Let's take 4

No of executors = 200/4 =50

**How much each executor memory is required**

Core memory = 4 \* default partition size = 4\*128 = 512 mb

Each executor has 4 cores → 4 \* 512 =2gb

**How much total memory is required**

50 \* 2gb = 100 gb

**Driver Size**

**…**

**What is DAG And Lazy Evaluation?**

**DAG: Directed acyclic graph(represents execution plan)**:

* when the transformations are applied on RDD or data frame or data set, they are not executed directly instead, they create a logical plan.
* So for each job DAG is created (meaning it represents how the jobs are splitted into stages)but it is triggered only when the action is called.

**Example for code optimization and how it removes unwanted transformations:**

**….**

**Lazy Evaluation :** Transformations are not executed immediately when the transformations are defined. Since the transformations are lazy, spark keeps track of them in the Lineage graph(RDD lineage). When the action is triggered spark uses Lineage graph to execute the transformations

**Advantages:**

**Optimization:** Spark can optimize the entire pipeline before execution

**Reduced Computation**: Unused transformations are ignored

**Fault Tolerance**: fault tolerance thru Lineage Graph

**Lineage Graph(**RDD lineage)It keeps the track of transformations applied to RDD’s

rdd1->rdd2->rdd3

**Catalyst Optimiser**

* It is a scala program for (SQL, Dataframe,dataset) which automatically finds the most efficient execution plan to execute all the operations written by user
* It is there only for Dataframe and dataset but not RDD (because there is no schema for RDD)
* It is a cost based optimiser(Execution plan is selected based on cost) but not rule based optimiser( Execution plan is selected based on rules)

**Example**: use case is to increase the salary for managers by 10%

**User code** - SELECT Salary\*0.1 as Bonus FROM Employee WHERE emp\_type=”Managers”

Now the catalyst performer will give multiple execution plans for one action and does analysis on the performance and give the best execution plan which boost the performance -

Execution plan1

* Increase all the employees salary to 10%
* Filter out managers

Execution plan2

* It filters the managers first
* Then increase their salary by 10%

**Scenarios where catalyst optimiser is used:**

Predicate push down(Rearrange filters): It filters the rows based on the filtering condition

Projection pruning(Removing unwanted columns): Instead of loading the whole dataset, it loads only the columns which are necessary

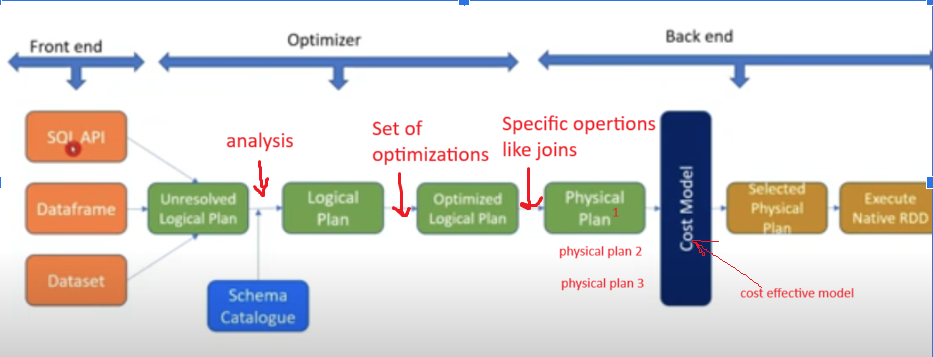
Join type: Selects the most efficient join based o the data sets,

* If one set is big and one set is small it uses broadcast join
* Choosing sort-merge join over hash join or broadcast join

Constant folding: if the user uses constants expressions 5+3 or any concatenations,these are computed at compile time rather than runtime

Cost-Based Optimization: Chooses the most efficient execution plan by evaluating cost models

**ARCHITECTURE**:



* Just the logical plan is written for all the transformations,which is a tree structure to give Unresolved Logical Plan
* It performs semantic analysis**.**It has all the meta data for the files,tables,databases. It is used to check the schema, for example if we are selecting a column which is not in the table we get an “Analysis Exception”. Below file doesn’t have column “name1” and gives Resolved Logical Plan or analyzed logical Plan



* Optimizer applies a set of optimization rules to the resolved logical plan(Analyzed logical plan) to transform it into an optimized logical plan

Ex: predicate push down, projection pruning

* Optimized logical plan is converted into one or more physical plans.The physical plan describes the specific operations like joins
* Best cost effective model is selected, and it is converted to RDD(JAVA byte code)

**RESILIENT DISTRIBUTED DATASET**

* It is a data structure in Spark. It works on distributed data(It works on the partitions on different executors)

**Features of RDD**

* It is immutable, lazy evaluated and fault tolerant
* If a partition of an RDD is lost due to a failure (e.g., a node crashes or a task is lost), Spark can recompute the lost partition using the lineage graph
* Type safe: any data type is given wrong, error is thrown during compilation

**When do we need RDD?**

* For unstructured Data
* User have full control on data

**Why shouldn’t we use RDD:**

* Spark will not optimize automatically with catalyst optimiser
* Need to write complex code since RDDs requires you to manually implement many operations(group by,join), which can be cumbersome and lead to more code and potential bugs.

**RDD VS DATAFRAME VS DATASET**

Data Representation:

* RDD is a distributed collection of data elements without any schema.
* Dataframes is also the distributed collection organized into the named columns
* Datasets is an extension of Dataframes with more features like type-safety and object-oriented interface.

Optimization:

* RDD -> No in-built optimization engine for RDDs. Developers need to write the optimized code themselves.
* Dataframes use a catalyst optimizer for optimization.
* Datasets also uses a catalyst optimizer for optimization purposes.

Projection of Schema:

* RDD -> Here, we need to define the schema manually.
* Dataframes will automatically find out the schema of the dataset.
* Datasets will also automatically find out the schema of the dataset by using the SQL Engine.

Aggregation Operation:

* RDD is slower than both Dataframes and Datasets to perform simple operations like grouping the data.
* Dataframes provide an easy API to perform aggregation operations. It performs aggregation faster than both RDDs and Datasets.
* Dataset is faster than RDDs but a bit slower than Dataframes.

When to use which API ?

If you want rich semantics, high-level abstractions, type-safety then go for Dataframes and datasets.

If you need more control over the pre-processing part(like custom partitioning , you can always use the RDDs.

**Spark Context and Spark Session:**

**Spark Context:** It is the fundamental entry point for low-level Spark operations, typically used for working with RDDs directly.

* Used to create RDDs, accumulators, and broadcast variables.
* Need both SQLContext and HiveContext
* Needs to be created manually

**Spark Session:** It is the entry point for Spark 2.x and later versions for both RDDs and DataFrame-based applications .

* SparkSession replaces both SQLContext and HiveContext. So we can work on RDDs, DataFrames, Datasets, and even SQL queries directly
* We still need Spark context but we are using it under Spark session
* It is initialized automatically in databricks and can be initilized using this in some notebooks

# Initialize Spark Session

spark = SparkSession.builder.appName("ParquetExample").getOrCreate()

**Spark Application,JOBs, Stages, Task:**

**Application:**

* It is the highest-level unit of work and represents the entire Spark program
* One spark submit means 1 application

**JOB:**

* A Job in Spark represents a complete execution of an action on an RDD or DataFrame.
* A job is triggered whenever an action (such as count(), collect(), save(), reduce()) is called on a DataFrame or RDD.
* If there are 5 actions in a application, 5 jobs are triggered

**Stages:**

* Spark splits jobs into multiple stages, and each stage consists of multiple tasks that can be executed independently on different data partitions
* For each job there will be at least 1 stage and at least 1 at task even though shuffle doesn’t happen
* A stage is created when there is a shuffle operation(wide transformations) in the job.

**Tasks:**

* For each job there will be at least 1 stage and at least 1 at task even though shuffle doesn’t happen
* By default group by or join operations will have 200 taks
* Each partition will have 1 task
* Repartition will divide data into 2 parts

Example below is the set of operations done on a dataframe



How many jobs are created? 2

How many stages are created? job1(1 stage) job2(3 stages)

How many tasks are created? job1(1 stage,1job) job2(3 stages,203 tasks)

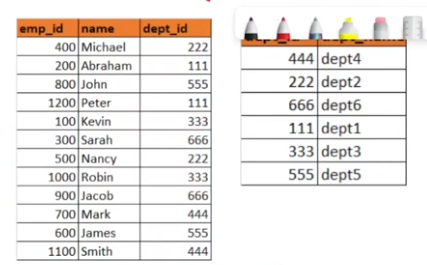
**JOIN STRATEGY:**

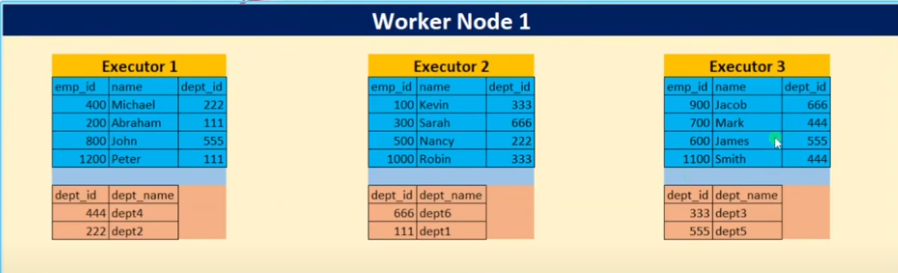
* When we write a join statements, spark can implement join in these below ways
* It depends on the size of data, memory available,skewness in data and characteristics of join keys

**Different types of join strategies?**

1. Broad-cast hash join →
2. Shuffle Sort-merge →O(nlogn)
3. Shuffle hash join → O(1) —> In-memory utilize
4. Cartesian join →
5. Broadcast nest loop → costly, O(n^2)

**Why join is expensive?**

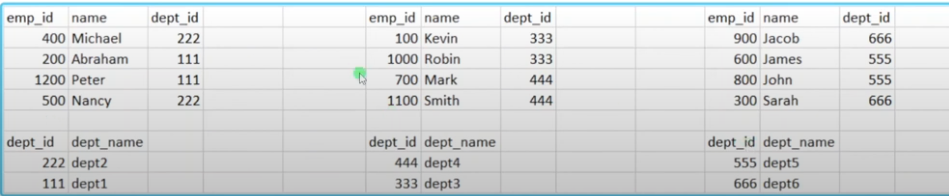
****

****

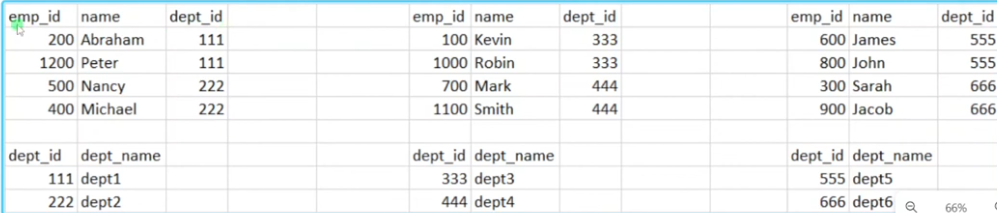
**Shuffle sort-merge join:**

* Default join
* If both datasets are large, Spark uses Sort-Merge Join (SMJ) by default.
* 3 phases(shuffling,sort, merge)

1. Shuffling: Data is shuffled such that it will not look into other executors



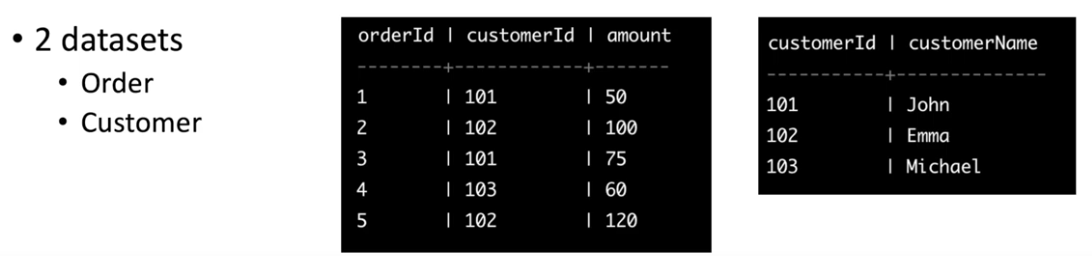
1. Sort: Sorting will take place based on the joining key



1. Merge:



**Shuffle hash join:**

****

1. **Partition**
2. **Hash table**
3. **Shuffle**
4. **Merge**

**Broad-cast join:**

**…**

**Creating RDD:**

There are 3 ways of creating an RDD:

1. Parallelizing an existing collection of data

* one way is to directly call sc variable

rdd1 = sc.parallelize(list1) #type of transformation

rdd1.collect() #action

* other way is to call spark context

sparksc= spark.sparkContext

rdd2 = sparksc.parallelize(list2)

rdd2.collect()

1. Referencing to the external data file stored

* Reading text File

rdd4 = sc.textFile("/FileStore/tables/sample.txt") #type of transformation

rdd4.collect() #action

* For structured text (like CSV), you need to parse it manually

rdd = spark.sparkContext.textFile("path/to/file.csv").map(lambda line: line.split(","))

* **RDDs can only be created directly from text-based files**
* Other formats like Parquet, JSON, or ORC must first be read into a **DataFrame** before converting to an RDD

1. Creating RDD from an already existing RDD

………….

**RDD Transformations:**

**map():** returns 1:1 mapping --> one output element for each input element

* Multiply each element with 2

rdd = sc.parallelize([1, 2, 3])

flat\_mapped\_rdd = rdd.map(lambda x: (x, x \* 2))

o/p [(1,2),(2,4),(3,6)]

* Split

rdd = sc.parallelize(["hello world", "spark is fun"])

rdd\_flatMapped = rdd.map(lambda line: line.split(" "))

# Output: [('hello', 'world)', ('spark', 'is', 'fun')]

**flatMap():**  returns a 1:N or 0:N mapping

* Multiply each element with 2

rdd = sc.parallelize([1, 2, 3])

flat\_mapped\_rdd = rdd.flatMap(lambda x: (x, x \* 2))

o/p [(1,2),(2,4),(3,6)]

* Split

rdd = sc.parallelize(["hello world", "spark is fun"])

rdd\_flatMapped = rdd.flatMap(lambda line: line.split(" "))

# Output: ['hello', 'world', 'spark', 'is', 'fun']

**First()**

**take()**

**keys()**

**values()**

**reduceby()**

**sortBy()**

**Creating a data frame and read data into DataFrame**

1. Creating from RDD and creating directly:

ToDF is used to create it from RDD and spark.createDataFrame is used to create it directly

With our schema:

rdd1 = [(1,"India"),(2,"US"),(3,"UK"),(4,"Italy")]

rdd2 =[(1,"Mango"),(2,"banana"),(3,"apple"),(4,"pears")]

cdf=rdd1.toDF()

cdf1=spark.createDataFrame(rdd2)

* With giving schema explicitly

Rdd1 = [(1,"India"),(2,"US"),(3,"UK"),(4,"Italy")]

myschema1=["id","country"]

cdf3=rdd1.toDF(myschema1)

cdf4=rdd1.toDF("id long, country string")

* giving schema struct types

#we need to import these explicitly --> StructType,StructField, IntegerType,StringType,DoubleType, DateType, TimestampType

from pyspark.sql.types import \*

mydfstruct = StructType([

StructField("id",IntegerType(),True), # take column name, type , True if it is nullable else False

StructField("country",StringType(),True)]

)

cdf5=rdd1.toDF(mydfstruct)

1. Creating directly:
2. Reading from File:

df6 = spark.read.format("csv") \

.option("path","/FileStore/tables/014\_Data.csv") \

.option("header","True") \

.option("inferSchema","True") \

.option(“mode”,”FAILFAST)

.load()

df8 = (spark.read.format("csv")

.option("path","/FileStore/tables/014\_Data.csv")

.option("header","True")

.option("inferSchema","True")

.option(“mode”,”FAILFAST)

.load())

Modes available when reading the data to handle bad records

| Keeps all the records, but Set null values to the corrupted column | PERMISSIVE (Default) |
| --- | --- |
| You want to drop bad records and keep only valid data | DROPMALFORMED |
| You want strict validation and to stop processing on the first error | FAILFAST |

spark will automatically detect the schema if we use inferschema but it is not the optimized way, since extra spark jobs will run; instead we can explicitly mention schema using schema parameter

Creating Schema Manually bu giving infer schema as false:

2 types

1. Struct type and struct field

StructType: defines the structure of DF. it is the list of struct Field

StructField: it contains column name, column type and if it nullable or not

StructType,StructField can be imported from pysark.sql.types

EX:

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

Schema\_ddf =StructType([

StructField("DEST\_COUNTRY\_NAME",IntegerType(),True),

StructField("ORIGIN\_COUNTRY\_NAME",StringType(),True),

StructField("count",IntegerType(),True),

])

myschema\_df = spark.read.format("csv")\

.option("path","/FileStore/SparkData/2010\_summary.csv")\

.option("inferschema",False)\

.schema(Schema\_ddf )\

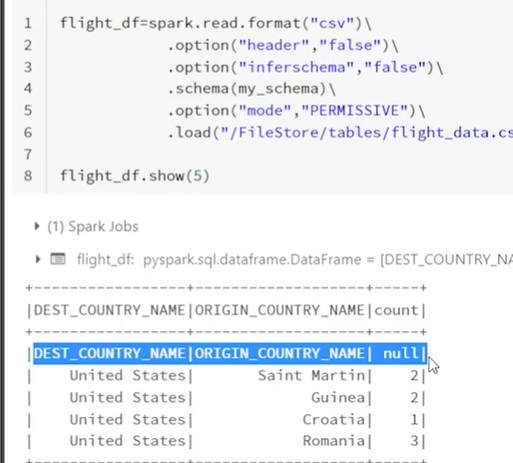
.option("header",False)\

.option("mode","permissive")\

.load()

myschema\_df.show()

#here the schema we wrote results in the header and the file is also having the header,so we need to skip the first row(header row)



From pyspark.sql.types import StructType,StructType,IntegerType,StringType

Flight\_df = spark.read.format(“csv)\

.option(‘path’,”/filesStoe/tables/flightdata.csv”)\

.option(“inferschema”,”False”)\

.schema(my\_schema)\

.option("header",”False”)\

.option(“skiprows’,1)\

.option(“mode”,”FAILFAST)\

.load()

1. DDL

Ex:

Schema\_ddl = "id INT, name STRING, age INT, salary DOUBLE"

Flight\_df\_new = spark.read.format(“csv)\

.option(‘path’,”/filesStoe/tables/flightdata.csv”)\

.option(“inferschema”,False)\

.schema(Schema\_ddl )\

.option("header",False)\

.option(“skiprows’,1)\

.option(“mode”,”FAILFAST)\

.load()

**Handle corrupted Data in Spark:**

How can you say it is a corrupted record?

JSON:

* Open braces or closing braces are missing, incorrect JSON structure
* Special characters or unreadable symbols.
* Strings in numeric columns, invalid dates

CSV

* Missing values, extra/missing columns, wrong delimiters.
* Special characters or unreadable symbols.
* Strings in numeric columns, invalid dates

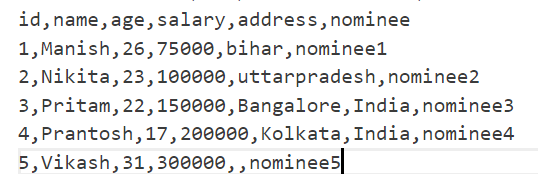
What happens when we encounter with corrupted records in different modes

| Keeps all the records, but Set null values to the corrupted column | PERMISSIVE (Default) | o/p records- same num of records |
| --- | --- | --- |
| You want to drop bad records and keep only valid data | DROPMALFORMED | o/p records- less number of records sense drops corrupted rows |
| You want strict validation and to stop processing on the first error | FAILFAST | Zero columns, fails immediately |

How can we print bad records?

We can just put the bad records in column by using schema and print them directly(in CSV)

This is the csv file we are using



EX:

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

schema\_corrupt =StructType([

StructField("id",IntegerType(),True),

StructField("name",StringType(),True),

StructField("age",IntegerType(),True),

StructField("salary",IntegerType(),True),

StructField("address",StringType(),True),

StructField("nomminee",StringType(),True),

StructField("\_corrupt\_record",StringType(),True),

])

df\_corrupt = spark.read.format("csv")\

.option("path","dbfs:/FileStore/SparkData/corruptedData.csv")\

.option("inferschema","False")\

.schema(schema\_corrupt )\

.option("header","False")\

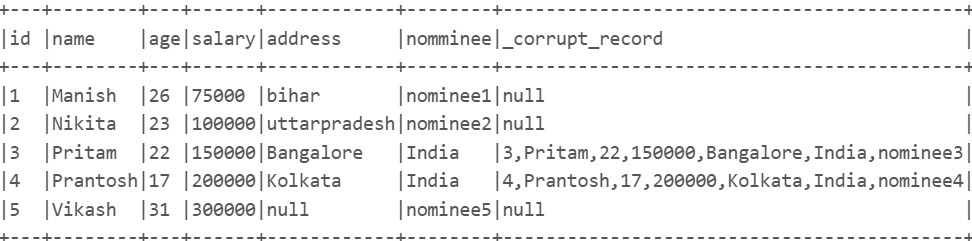
.option("skiprows",1)\

.option("mode","permissive")\

.load()

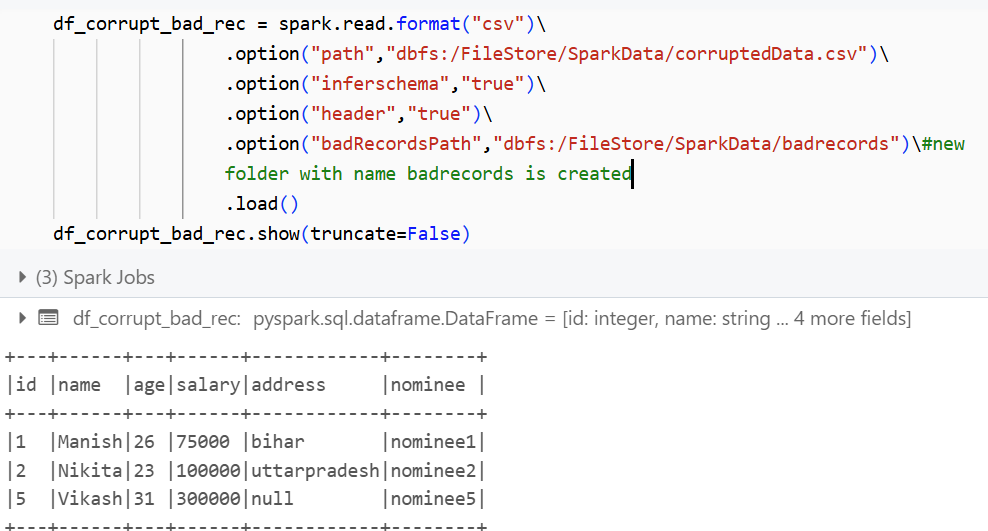
df\_corrupt.show(truncate=False)

Output:



Can we store corrupted records and can we access them later?

We can store them in separate file and use them later using badRecordsPath



We can read that file and print the DF



**How to read Json file**

**JSON - javascript object notation**

What is json data and how to read it ?

* Semi structured data
* EX:

[

{"id": 1, "name": "Alice", "age": 25},

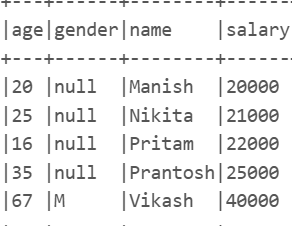
{"id": 2, "name": "Bob", "age": 30}

]

* JSON supports key-value pairs, arrays, and nested objects.

What if i have 3 keys in all lines and 4 keys in oneline?

* Means 1 rows is having extra key value pair
* So null values are created for the other columns



What is mutiline json and delimited json?

Delimited JSON (One JSON Object Per Line)

* Each JSON record is in a single line (no [ ] brackets).
* Commonly used in logs and streaming data.
* Spark automatically detects and processes one record per line.

Ex:



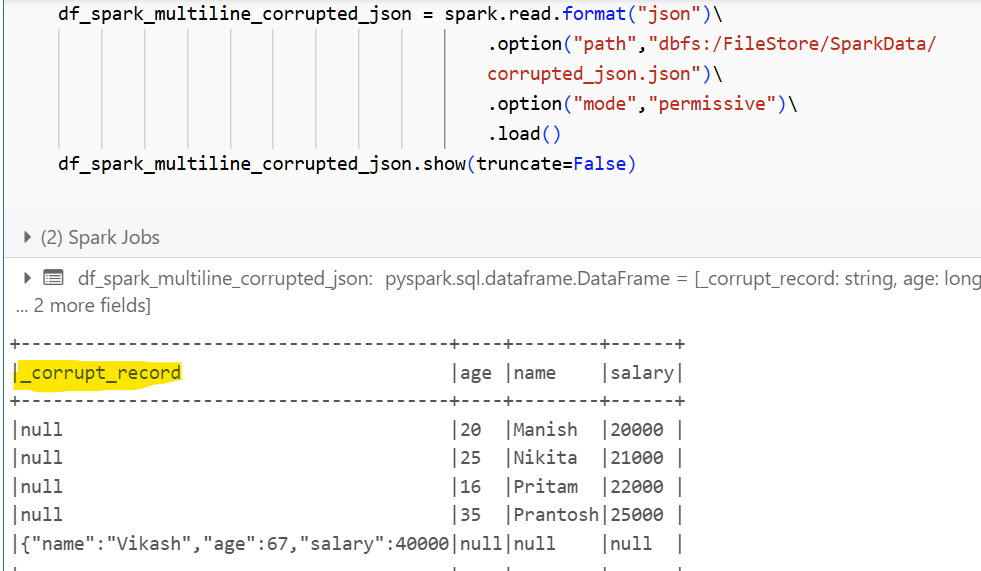
Multiline JSON (Nested JSON):

* JSON records span multiple lines and start with [ and end with ].
* Used for structured objects and arrays.
* Spark reads the entire JSON block as a single DataFrame. So we need to use multiline option



What will happen if i have a corrupted json file or invalid format json file

Spark stores bad records in \_corrupt\_record column instead of failing the job. It is automatically created unlike CSV file where we need to create the column manually

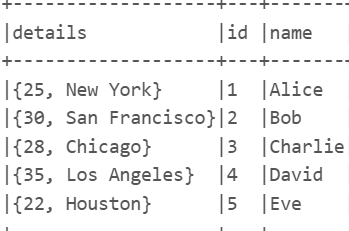


How to convert nested json into spark data frame?

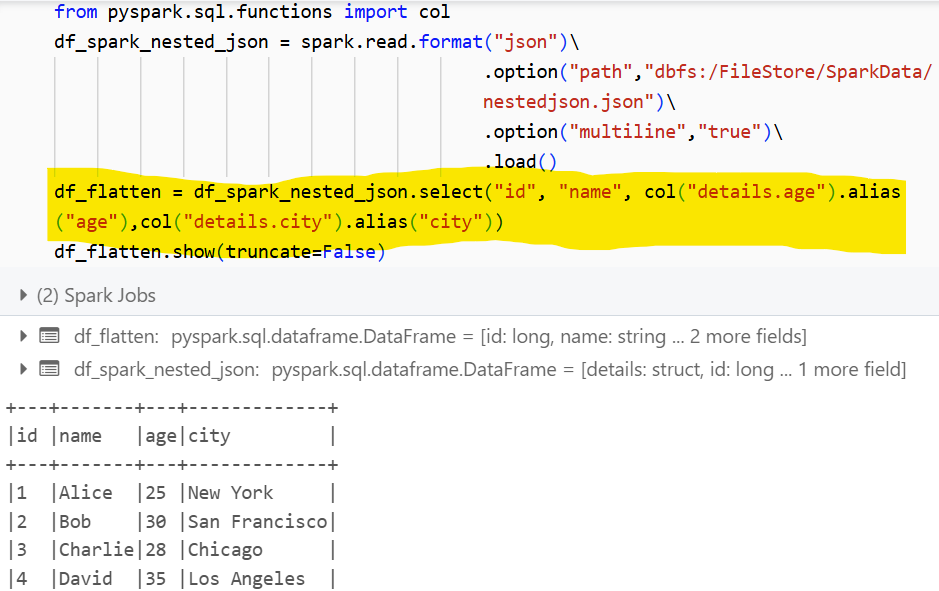


JSON often contains nested objects, which we need to flatten

If we directly reads-



So we need to flatten it first



**PARQUET FILE**

What is parquet file format? And why do we need it ??

* It is a Columnar based file format

| name | age | profession |
| --- | --- | --- |
| jags | 28 | DE |
| jaggu | 21 | student |

Row based:

name,age,salary

Jags, 28, DE

jaggu,21,student

Column based:

Name:jags,jaggu

Age:28,21

profession:DE,student

* Structured file format
* Data is stored in binary format
* In big data, we follow write once, read many times → Reads faster→Read performance is more since if we want just profession, it reads only profession but not the entire data. So reading is much faster and I/o is less so cost is optimized
* Used for OLAP (online Analytical process)
* Less storage→ since all the rows of column have similar data type, and they are stored together compression algorithm works better
* Write performance is less → all the since the data needs to be compressed
* Schema Evolution exists in parquet → means, if we add a new column no need to change the previous data, previous records will have null (for ex if we add new column in csv file, data before will not be read properly throws parsing errors)
* Partitioning is supported in parquet→ large file can be partitioned into two and can be stored in separate directories based on column values

How to read it



* Spark automatically infers the schema, no need to mention inferschema

What makes parquet difficult choice?

* Limited Write Performance → Writing Parquet is slower than CSV/JSON due to compression.
* Complex File Structure → More challenging to debug or inspect compared to text files.
* Inefficient for Small Files → Works best with large datasets.

What encoding is done on data ?

| **Encoding Type** | **Description** | **When Used?** |
| --- | --- | --- |
| PLAIN | Default encoding, stores data as-is | Used for non-repeating values |
| DICTIONARY | Stores a dictionary of unique values & replaces actual values with indexes | Best for categorical data with low cardinality (e.g., Country column)  If we have columns as “usa”,”india”,“usa”,"uk",”india”“usa”,"uk"  Unique Values (Dictionary): {1: "usa", 2: "India", 3: "uk"}  Column Stored As: [1, 2, 1, 3, 2, 1, 3] |
| RUN-LENGTH ENCODING (RLE) | Stores sequences of repeated values as (value, count) pairs | Best for columns with many repeated values (e.g., Gender)  Stored As: [("usa", 3), ("india", 2), ("uk", 2)] |
| BIT-PACKING | Packs multiple values into a smaller bit representation | Used for integer and boolean values  Original: [1, 0, 1, 1, 0, 0, 1, 0]  Stored As: 10110010 (bit-packed format) |
| DELTA ENCODING | Stores differences between consecutive values instead of full values | Used for increasing sequences (e.g., timestamps)  Original: [100, 110, 120, 150]  Stored As: [100, 10, 10, 30] |

What compression techniques are used?

| **Compression Type** | **Desc** | **performance** |
| --- | --- | --- |
| SNAPPY | Fast compression and decompression | Fastest but less compression |
| GZIP | High compression ratio but slower read/write | Better compression but slower |
| LZO | Medium compression, very fast decompression | Good for real time analytics |
| ZSTD | Advanced high-ratio compression | Best for large scale data storage |

* Snappy is default because it's fast and balances size & speed well.
* Use GZIP if storage savings are more important than speed.

What is row group,column and pages?

* Parquet stores the data in hierarchical format

row group: total data is divided into a large number of row groups. If a file with 500gb is given, it is divided into 4 row groups. Default size of each row group is 128mb

Column chunks: each column is stored separately in a rowgroup

Pages: Smallest unit in Parquet (contains data + metadata)

Row Group 1 (128MB)

├── Column Chunk: id

│ ├── Page 1

│ │ │\_\_\_ metaData

│ │ │ │—count

│ │ │ │—max

│ │ │ │—min

│ │ │ │–compression

│ │ │ │–encodings

│ ├── Page 2

├── Column Chunk: name

│ ├── Page 1

│ ├── Page 2

How to optimize the parquet file?

* Partitioning
* bucketinng

How projection pruning and predicate pushdown works?

How to write data in spark(on disk)?

df8.write.format("csv")/

.option(“header” true”)/

.option(“mode”,”overwrite”)/

.option(“path”,"/FileStore/tables/savedcsv")

.save()

df8.write.format("csv")/

.option(“header” true”)/

.option(“mode”,”overwrite”)/

.option(“path”,"/FileStore/tables/savedcsv")

.repartition(3)

.save()

Modes available?

Append: if we give same path, it will create a new copy of the file but will not delete the old one

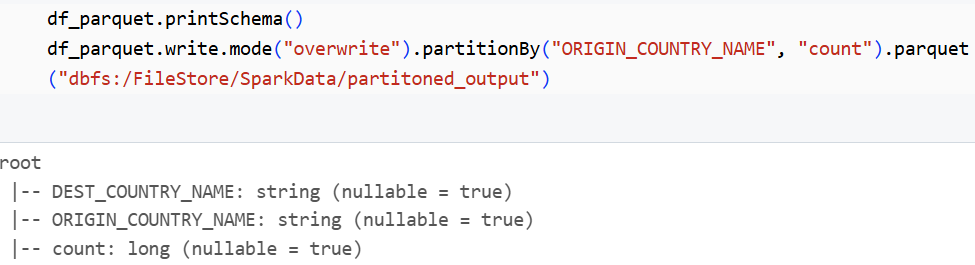
overwrite:if we give same path, it will create a new copy of the file byt deleting the old one

errorIfExists: gives error when the path already exists

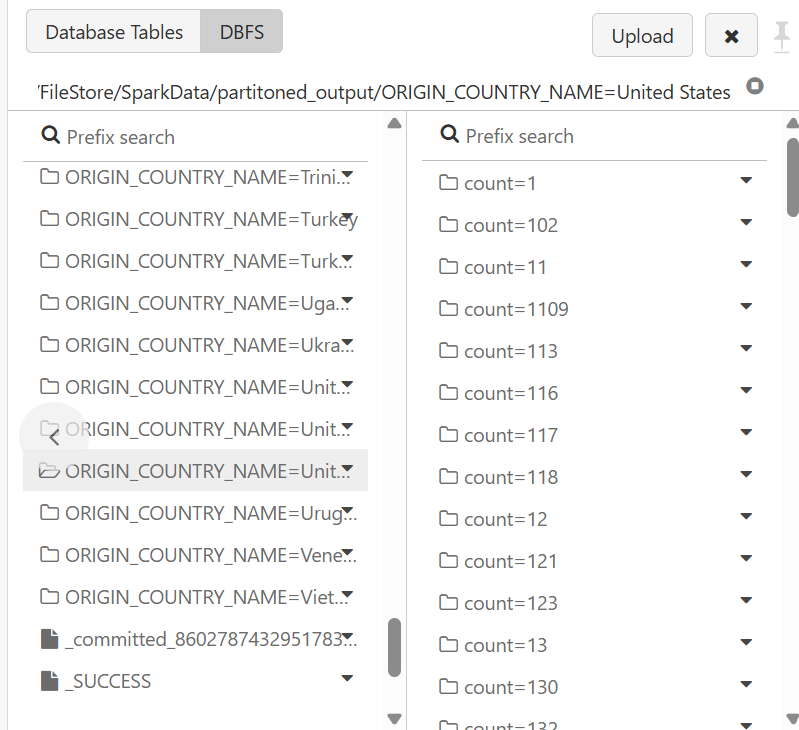
ignore: Ignores error when the path already exists

PARTITION BY AND BUCKETING

Partition By: we can partition the large size data by the columns we frequently use. This technique is used for the column based files(not files like csv)



* New folder with small parquet files are created
* While partitioning , we can give the compression techniques - default is snappy compression
* We can read the data from the partitioned parquet file. Here the data is partitioned based on the origin country and again on the count





Bucketing:

* bucketBy() only works with tables (e.g., Hive tables, Parquet, ORC, Delta Lake).
* CSV does not support bucketing because it lacks metadata for managing buckets.
* We need to bucket the column and save them as tables

Example:

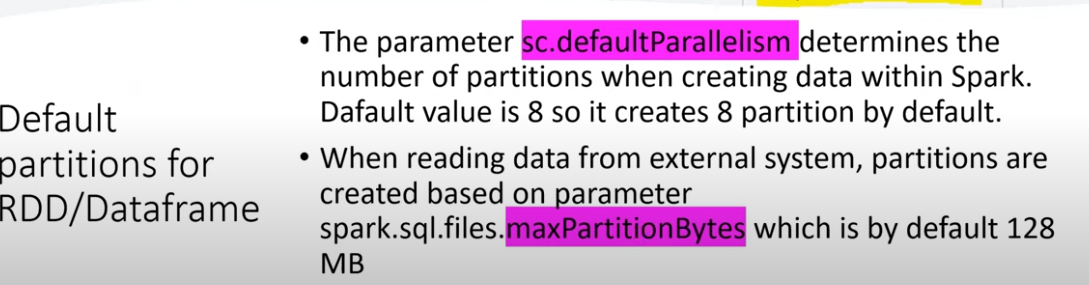
….

**REPARTITION and COALESCE**

* For the best performance spark application, partition plays a key role
* Consider 4 cores, below is the data with different partition sizes. If 1 core is processing 100 gb of data, 1 core is idle & remaining 2 cores process data for some and and then they wait for core which process more data
* So, we need to create right number of partitions
* Evenly distribute partitions increases the performance of spark application
* Repartition and coalesce are used to manually increase or decrease the number of partitions



Parameters for default partition size



**Repartition vs Coalesce:**

**Repartition:**

* It is used to increase or decrease the number of partitions
* All the partitions are equal sized - performance is good in some cases
* Shuffling(hash based) of data is done and new partitions are build from scratch - doesn’t have good performance in some cases
* Repartition can be done on column as well

ADV:

* Prevents data skew when writing large datasets.
* Ensures balanced partition sizes for parallel writing.

DISADV:

* Shuffling is done, so more cost

**Coalesce:**

* Used to decrease the number of partitions
* It merges the partitions, no shuffling takes place
* So, the partitions are unevenly distributed
* Since no partitions, coalesce is better performant than repartition

ADV:

* Avoids generating too many small files when writing to disk.
* Faster than repartition() because it avoids a full shuffle

DISADV:

* It is only used to decrease the number of partitions
* Can create skewed partitions (unequal sizes) because it avoids shuffle.

| Feature | used for | Best for | Dis adv |
| --- | --- | --- | --- |
| repartition(n) | Used for reading | Adjusting partitions before writing to ensure load balancing | Expensive shuffle operation |
| coalesce(n) | Used for reading | Reducing small files before writing (avoids shuffle) | # Works only for reducing partitions, not increasing.  # Can create skewed partitions (unequal sizes) because it avoids shuffle. |
| partitionBy(col) | Used for writing | Physically partitioning data on disk (faster queries) | #Can create too many small partitions if column has high cardinality(.  #Requires predicate pushdown to benefit from faster reads. |
| bucketBy(n, col) | Used for writing | Optimizing joins & aggregations | #Works only with tables (not just files).  # Needs a pre-specified number of buckets (not auto-scalable).  #Joins must be on bucketed columns to see performance benefits. |

What is adaptive query execution?

What is speculative execution