

6. What are PySpark serializers?

In Spark, serialization process is used to conduct performance tuning.

PySpark supports custom serializers for transferring data; this can improve performance.

By default, PySpark uses PickleSerializer to serialize objects using Python's cPickle serializer, which can serialize nearly any Python object.

Other serializers, like MarshalSerializer, support fewer datatypes but can be faster.

The data sent or received over the network to the disk or memory should be persisted.

PySpark supports serializers for this purpose. It supports two types of serializers, they are:

PickleSerializer: By using PySpark's Pickle Serializer, it Serializes Objects.

However, its best feature is its supports nearly any Python object, although it is not as fast as more specialized serializers.

This serializes objects using Python's PickleSerializer (class `pyspark.PickleSerializer`).

This supports almost every Python object.

MarshalSerializer: This performs serialization of objects.

We can use it by using class `pyspark.MarshalSerializer`.

This serializer is faster than the PickleSerializer but it supports only limited types.

9. What are the different cluster manager types supported by PySpark?

A cluster manager is a cluster mode platform that helps to run Spark by providing all resources to worker nodes based on the requirements.

*Apache Spark has 4 main open source cluster managers: Mesos, YARN, Standalone, and Kubernetes.

Every cluster manager has its own unique requirements and differences.

PySpark supports the following cluster manager types:

Standalone – This is a simple cluster manager that is included with Spark.

The Resource Manager and Worker are the only Spark Standalone Cluster components that are independent.

There is only one executor that runs tasks on each worker node in Standalone Cluster mode.

Apache Mesos – This manager can run Hadoop MapReduce and PySpark apps.(

The Mesos framework includes three components: Mesos Master,Mesos slave,Mesos frameworks)

Hadoop YARN – This manager is used in Hadoop2

it has resource manager and node manager.

Kubernetes – This is an open-source cluster manager that helps in automated deployment, scaling and automatic management of containerized apps.

A framework for deploying, scaling, and managing containerized applications that are open source.

local – This is simply a mode for running Spark applications on laptops/desktops.

17. What is PySpark Architecture?

PySpark similar to Apache Spark works in master-slave architecture pattern.

Here, the master node is called the Driver and the slave nodes are called the workers.

When a Spark application is run, the Spark Driver creates `SparkContext` which acts as an entry point to the spark application.

All the operations are executed on the worker nodes.

The resources required for executing the operations on the worker nodes are managed by the Cluster Managers

18. What PySpark DAGScheduler?

DAG stands for Direct Acyclic Graph.

DAGScheduler constitutes the scheduling layer of Spark which implements scheduling of tasks in a stage-oriented manner using jobs and stages.

The logical execution plan (Dependencies lineage of transformation actions upon RDDs) is transformed into a physical execution plan consisting of stages.

It computes a DAG of stages needed for each job and keeps track of what stages are RDDs are materialized and finds a minimal schedule for running the jobs.

How DAG works in Spark?

The interpreter is the first layer, using a Scala interpreter, Spark interprets the code with some modifications.

Spark creates an operator graph when you enter your code in Spark console.

When we call an Action on Spark RDD at a high level, Spark submits the operator graph to the DAG Scheduler.

Divide the operators into stages of the task in the DAG Scheduler. A stage contains task based on the partition of the input data. The DAG scheduler pipelines operators together. For example, map operators schedule in a single stage.

The stages pass on to the Task Scheduler. It launches task through cluster manager. The dependencies of stages are unknown to the task scheduler.

The Workers execute the task on the slave.

19. What is the common workflow of a spark program?

The most common workflow followed by the spark program is:

The first step is to create input RDDs depending on the external data. Data can be obtained from different data sources.

after RDD creation, the RDD transformation operations like `filter()` or `map()` are run for creating new RDDs depending on the business logic.

If any intermediate RDDs are required to be reused for later purposes, we can persist those RDDs.

Lastly, if any action operations like first(), count() etc are present then spark launches it to initiate parallel computation.

22. What are the profilers in PySpark?

In PySpark, custom profilers are supported.

These are useful for building predictive models.

Profilers are useful for data review to ensure that it is valid and can be used for consumption.

When we require a custom profiler, it has to define some of the following methods:

profile : This produces a system profile of some sort.

stats : This returns collected stats of profiling.

dump : This dumps the profiles to a specified path.

add : This helps to add profile to existing accumulated profile. The profile class has to be selected at the time of zSparkContext creation.

dump(id, path): This dumps a specific RDD id to the path given

24. What are the different approaches for creating RDD in PySpark?

Basically, there are 3 ways to create Spark RDDs

i. Parallelized collections

By invoking parallelize method in the driver program, we can create parallelized collections.

Using sparkContext.parallelize():

The parallelize() method of the SparkContext can be used for creating RDDs.

This method loads existing collection from the driver and parallelizes it.

This is a basic approach to create RDD and is used when we have data already present in the memory.

This also requires the presence of all data on the Driver before creating RDD.

ii. External datasets

We can create Spark RDDs, by calling a textFile method.

this method takes URL of the file and reads it as a collection of lines.

URL can be a local path on the machine or a hdfs://, s3n://, etc.

Using sparkContext.textFile():

Using this method, we can read .txt file and convert them into RDD.

Syntax: rdd_txt = spark.sparkContext.textFile("/path/to/textFile.txt")

Using sparkContext.wholeTextFiles():

This function returns PairRDD (RDD containing key-value pairs) with file path being the key and the file content is the value.

rdd_whole_text = spark.sparkContext.wholeTextFiles("/path/to/textFile.txt")

iii. Existing RDDs

We can create new RDD in spark, by applying transformation operation on existing RDDs.

Empty RDD with no partition using sparkContext.emptyRDD:

RDD with no data is called empty RDD. We can create such RDDs having no partitions by using emptyRDD() method as shown in the code piece below:

```
empty_rdd = spark.sparkContext.emptyRDD  
# to create empty rdd of string type  
empty_rdd_string = spark.sparkContext.emptyRDD[String]
```

31. What would happen if we lose RDD partitions due to the failure of the worker node?

If any RDD partition is lost, then that partition can be recomputed using operations lineage from the original fault-tolerant dataset.

Que 13. What do mean by Broadcast variables?

Ans.

Broadcast variables are variables that are shared throughout the cluster

broadcast variables area cannot be changed, which means that they can't be modified.

If you want to change or modify, accumulators are needed.

In order to save the copy of data across all nodes, we use it.

*With SparkContext.broadcast(), a broadcast variable is created.

The properties of Broadcast Variable in Apache Spark are:

Immutable

Distributed to the cluster

Fit in memory

Que 14. What are Accumulator variables?

Accumulator is a shared variable in Apache Spark, used to aggregating information across the cluster.

In order to aggregate the information through associative and commutative operations, we use them.

Code: class pyspark.Accumulator(aid, value, accum_param)

Why Accumulator :

When we use a function inside the operation like map(), filter() etc

these functions can use the variables which defined outside these function scope in the driver program.

When we submit the task to cluster,

each task running on the cluster gets a new copy of these variables and updates from these variable do not propagate back to the driver program.
Accumulator lowers this restriction.

Use Cases :

One of the most common use of accumulator is count the events that occur during job execution for debugging purpose.

Meaning count the no. of blank lines from the input file, no. of bad packets from network during session, during Olympic data analysis we have to find age where we said (age != 'NA') in SQL query in short finding bad / corrupted records.

```
*****  
*****
```

spark-case-when-otherwise

Like SQL "case when" statement and "Swith", "if then else" statement from popular programming languages,

Spark SQL Dataframe also supports similar syntax using "when otherwise" or we can also use "case when" statement.

So let's see an example on how to check for multiple conditions and replicate SQL CASE statement.

1. Using "when otherwise" on Spark DataFrame.

when is a Spark function, so to use it first we should import using import org.apache.spark.sql.functions.

when before. Above code snippet replaces the value of gender with new derived value. when value not qualified with the condition, we are assigning "Unknown" as value.

```
val df2 = df.withColumn("new_gender", when(col("gender") === "M", "Male")  
    .when(col("gender") === "F", "Female")  
    .otherwise("Unknown"))
```

2. Using "case when" on Spark DataFrame.

Similar to SQL syntax, we could use "case when" with expression expr() .

3. Using && and || operator

We can also use and (&&) or (||) within when function. To explain this I will use a new set of data to make it simple.

```
*****
```

Reading Parquet file into DataFrame

Spark DataFrameReader provides parquet() function (spark.read.parquet) to read the parquet files and creates a Spark DataFrame.

In this example, we are reading data from an apache parquet.

```
val df = spark.read.parquet("src/main/resources/zipcodes.parquet")
```

```
*****
```

Spark Modules:

Spark Core
Spark SQL
Spark Streaming
Spark MLlib
Spark GraphX

Modules built on Spark:

Spark Core : it is the fundamental unit of the whole Spark project.
It provides all sort of functionalities like task dispatching, scheduling, and input-output operations etc

Spark Streaming : processing real-time data streams

Spark SQL : support for structured data and relational queries

MLlib : built-in machine learning library

GraphX : Spark's new API for graph processing

Bagel (Pregel on Spark): older, simple graph processing model

```
*****
```

Spark Dataframe – Show Full Column Contents?

PySpark Show Full Contents of a DataFrame

In Spark or PySpark by default truncate column content if it is longer than 20 chars when you try to output using show() method of DataFrame, in order to show the full contents without truncating you need to provide a boolean argument false to show(false) method.

Following are some examples.

1.2 PySpark (Spark with Python):

```
# Show full contents of DataFrame (PySpark)  
df.show(truncate=False)
```

```
# Show top 5 rows and full column contents (PySpark)
```

```
df.show(5,truncate=False)

# Shows top 5 rows and only 10 characters of each column (PySpark)
df.show(5,truncate=10)

# Shows rows vertically (one line per column value) (PySpark)
df.show(vertical=True)
```

2. PySpark Show Full Contents of a DataFrame:

Let's assume you have a similar DataFrame mentioned above,]
for PySpark the syntax is slightly different to show the full contents of the columns.
Here you need to specify truncate=False to show() method.
df.show(truncate=False)

```
*****
```

RDD Operations:

On Spark RDD, you can perform two types of operations.
Transformations are lazy in nature i.e., they get execute when we call an action. They are not executed immediately.
Two most basic type of transformations is a map(), filter().
After the transformation, the resultant RDD is always different from its parent RDD.
It can be smaller (e.g. filter, count, distinct, sample),
bigger (e.g. flatMap(), union(), Cartesian()) or the same size (e.g. map).
There are two types of transformations:

Narrow transformation –

In Narrow transformation, all the elements that are required to compute the records in single partition live in the single partition of parent RDD.
A limited subset of partition is used to calculate the result.
Narrow transformations are the result of map(), filter().

Wide transformation –

In wide transformation, all the elements that are required to compute the records in the single partition may live in many partitions of parent RDD.
The partition may live in many partitions of parent RDD.
Wide transformations are the result of groupByKey() and reduceByKey().

Actions:

Transformations create RDDs from each other, but when we want to work with the actual dataset, at that point action is performed.
When the action is triggered after the result, new RDD is not formed like transformation.
Thus, Actions are Spark RDD operations that give non-RDD values.

The values of action are stored to drivers or to the external storage system. It brings laziness of RDD into motion.

An action is one of the ways of sending data from Executer to the driver. Executors are agents that are responsible for executing a task.

While the driver is a JVM process that coordinates workers and execution of the task. Some of the actions of Spark are:

RDD Transformations:

It creates a new Spark RDD from the existing one. Moreover, it passes the dataset to the function and returns new dataset.

Spark RDD Transformations are lazy operations meaning they don't execute until you call an action on RDD.

Since RDD's are immutable, When you run a transformation(for example map()), instead of updating a current RDD, it returns a new RDD.

Some transformations on RDD's are flatMap(), map(), reduceByKey(), filter(), sortByKey() and all these return a new RDD instead of updating the current.

RDD Actions:

In Apache Spark, Action returns final result to driver program or write it to the external data store.

RDD Action operation returns the values from an RDD to a driver node.

In other words, any RDD function that returns non RDD[T] is considered as an action.

RDD operations trigger the computation and return RDD in a List to the driver program.

Some actions on RDD's are count(), collect(), first(), max(), reduce() and more.

How to create an empty DataFrame?

While working with files, some times we may not receive a file for processing, however, we still need to create a DataFrame similar to the DataFrame we create when we receive a file.

If we don't create with the same schema, our operations/transformations on DF fail as we refer to the columns that may not present.

To handle situations similar to these, we always need to create a DataFrame with the same schema,

which means the same column names and datatypes regardless of the file exists or empty file processing

Creating an empty DataFrame (Spark 2.x and above):

SparkSession provides an emptyDataFrame() method, which returns the empty DataFrame with empty schema,

but we wanted to create with the specified StructType schema.

```
df = spark.emptyDataFrame
```

Create empty DataFrame with schema (StructType)
Use createDataFrame() from SparkSession

```
df = spark.createDataFrame(spark.sparkContext  
    .emptyRDD[Row], schema)
```

Spark DataFrame Select First Row of Each Group?

2. Select First Row From a Group

We can select the first row from the group using Spark SQL or DataFrame API, in this section, we will see with DataFrame API using a window function `row_number` and `partitionBy`.

```
simpleData = Seq(("James","Sales",3000),  
    ("Michael","Sales",4600),  
    ("Robert","Sales",4100),  
    ("Maria","Finance",3000),  
    ("Raman","Finance",3000),  
    ("Scott","Finance",3300),  
    ("Jen","Finance",3900),  
    ("Jeff","Marketing",3000),  
    ("Kumar","Marketing",2000)  
)  
import spark.implicits._  
val df = simpleData.toDF("Name","Department","Salary")  
df.show()  
  
w2 = Window.partitionBy("department").orderBy(col("salary"))  
df.withColumn("row",row_number().over(w2))  
    .where($"row" === 1).drop("row")  
    .show()
```

On above snippet, first, we are partitioning on department column which groups all same departments into a group and then apply order on salary column. Now, And will use this window with `row_number` function. This snippet outputs the following.

`row_number` function returns a sequential number starting from 1 within a window partition group.

3. Retrieve Employee who earns the highest salary

To retrieve the highest salary for each department, will use `orderby "salary"` in descending order and retrieve the first element.

```
w3 = Window.partitionBy("department").orderBy(col("salary").desc)
```

```
df.withColumn("row",row_number().over(w3))
  .where($"row" === 1).drop("row")
  .show()
```

4. Select the Highest, Lowest, Average, and Total salary for each department group

Here, we will retrieve the Highest, Average, Total and Lowest salary for each group. Below snippet uses partitionBy and row_number along with aggregation functions avg, sum, min, and max.

```
w4 = Window.partitionBy("department")
val aggDF = df.withColumn("row",row_number().over(w4))
  .withColumn("avg", avg(col("salary")).over(w4))
  .withColumn("sum", sum(col("salary")).over(w4))
  .withColumn("min", min(col("salary")).over(w4))
  .withColumn("max", max(col("salary")).over(w4))
  .where(col("row") === 1).select("department", "avg", "sum", "min", "max")
  .show()
*****
```

Spark Repartition() vs Coalesce():

DataFrame coalesce()

Spark DataFrame coalesce() is used only to decrease the number of partitions.

This is an optimized or improved version of repartition() where the movement of the data across the partitions is fewer using coalesce.

```
df3 = df.coalesce(2)
println(df3.rdd.partitions.length)
```

DataFrame repartition()

Similar to RDD, the Spark DataFrame repartition() method is used to increase or decrease the partitions.

The below example increases the partitions from 5 to 6 by moving data from all partitions.

```
df2 = df.repartition(6)
println(df2.rdd.partitions.length)
```

```
*****
```

Spark SQL Join on multiple columns:

Using Join syntax:

```
join(right: Dataset[_], joinExprs: Column, joinType: String): DataFram
```

This join syntax takes, takes right dataset, joinExprs and joinType as arguments and we use joinExprs to provide join condition on multiple columns.

```
//Using multiple columns on join expression  
empDF.join(deptDF, empDF("dept_id") === deptDF("dept_id") &&  
    empDF("branch_id") === deptDF("branch_id"),"inner")  
    .show(false)
```

Using Where to provide Join condition

Instead of using a join condition with join() operator, we can use where() to provide a join condition.

```
//Using Join with multiple columns on where clause
```

```
empDF.join(deptDF).where(empDF("dept_id") === deptDF("dept_id") &&  
    empDF("branch_id") === deptDF("branch_id"))  
    .show(false)
```

Using Filter to provide Join condition

We can also use filter() to provide Spark Join condition, below example we have provided join with multiple column

```
//Using Join with multiple columns on filter clause
```

```
empDF.join(deptDF).filter(empDF("dept_id") === deptDF("dept_id") &&  
    empDF("branch_id") === deptDF("branch_id"))  
    .show(false)
```

```
*****
```

Spark Window Functions

ranking functions

analytic functions

aggregate functions

ranking functions:

RANK() will assign non-consecutive “ranks” to the values

ranking functions are of 3 types:

row_number window function

rank window function

dense_rank window function

row_number Window Function: row_number() window function is used to give the sequential row number starting from 1 to the result of each window partition.

```
windowSpec = Window.partitionBy("department").orderBy("salary")  
df.withColumn("row_number",row_number.over(windowSpec))  
    .show()
```

rank Window Function: rank() window function is used to provide a rank to the result within a window partition. This function leaves gaps in rank when there are ties.

```
df.withColumn("rank",rank().over(windowSpec))
.show()
```

2.3 dense_rank Window Function

dense_rank() window function is used to get the result with rank of rows within a window partition without any gaps. This is similar to rank() function difference being rank function leaves gaps in rank when there are ties.

```
df.withColumn("dense_rank",dense_rank().over(windowSpec))
.show()
```

analytic functions:

analytic functions are of 2 types:

lag window function

lead window function

lag Window Function: This is the same as the LAG function in SQL.

lead Window Function: This is the same as the LEAD function in SQL.

Aggregate Functions:

using this function calculate sum, min, max for each department using Spark SQL Aggregate window functions and WindowSpec.

When working with Aggregate functions, we don't need to use order by clause.

```
val windowSpecAgg = Window.partitionBy("department")

val aggDF = df.withColumn("row",row_number().over(windowSpec))
  .withColumn("avg", avg(col("salary")).over(windowSpecAgg))
  .withColumn("sum", sum(col("salary")).over(windowSpecAgg))
  .withColumn("min", min(col("salary")).over(windowSpecAgg))
  .withColumn("max", max(col("salary")).over(windowSpecAgg))
  .where(col("row")==1).select("department","avg","sum","min","max")
.show()
```

Link for Window functions:

<https://sparkbyexamples.com/spark/spark-sql-window-functions/#ranking-functions>

Spark SQL map Functions:

Spark SQL map functions are grouped as “collection_funcs” in spark SQL along with several array functions. These map functions are useful when we want to concatenate two or more map columns, convert arrays of StructType entries to map column e.t.c

map Creates a new map column.
map_keys Returns an array containing the keys of the map.
map_values Returns an array containing the values of the map.
map_concat Merges maps specified in arguments.
map_from_entries Returns a map from the given array of StructType entries.
map_entries Returns an array of all StructType in the given map.
explode(e: Column) Creates a new row for every key-value pair in the map by ignoring null & empty. It creates two new columns one for key and one for value.
explode_outer(e: Column) Creates a new row for every key-value pair in the map including null & empty. It creates two new columns one for key and one for value.
posexplode(e: Column) Creates a new row for each key-value pair in a map by ignoring null & empty. It also creates 3 columns “pos” to hold the position of the map element, “key” and “value” columns for every row.
posexplode_outer(e: Column) Creates a new row for each key-value pair in a map including null & empty. It also creates 3 columns “pos” to hold the position of the map element, “key” and “value” columns for every row.
transform_keys(expr: Column, f: (Column, Column) => Column) Transforms map by applying functions to every key-value pair and returns a transformed map.
transform_values(expr: Column, f: (Column, Column) => Column) Transforms map by applying functions to every key-value pair and returns a transformed map.
map_zip_with(left: Column, right: Column, f: (Column, Column, Column) => Column) Merges two maps into a single map.
element_at(column: Column, value: Any) Returns a value of a key in a map.
size(e: Column) Returns length of a map column.

<https://sparkbyexamples.com/spark/spark-sql-map-functions/>

LINK:

<https://sparkbyexamples.com/>

Spark String Functions

Spark Sort Functions

Spark Date and Time Functions

How to convert Parquet file to CSV file

How to process JSON from a CSV file

How to convert Parquet file to CSV file

----->what are the roles and responsibilities on your project?
----->what is hadoop?
----->What is the difference between Internal table and External Table?
----->what is the difference between MR and Spark?
----->Difference between Repartition and coalesce
----->Difference between List and Tuple in Python?

1.I want left join,inner join ,right join,full join counts:

Table A:

1
1
1
1
1

Table B:

1
1
1
1
1

Ans: 25

2.Input: (I want total distance covered by each train between chennai to delhi)

Source city Dest city Distance Train
Chennai Nagpur 15000 A
Nagpur Delhi 12000 A
Chennai UP 17000 B
UP Delhi 6000 B

output:

Source City Dest City Train Distance
Chennai Delhi A 27000
Chennai Delhi B 23000

Ans:

Select city,sum(city) from train group by city
Where souce Chennai and dest delhi

3.Feed File - text.csv.
Read the feed file & load it into the table (Partition: Date)

Ans:
`df = spark.read.csv("text.csv").partitionedby("date")`

4.Student Table :-
Roll_Number | Name | Subject | Marks | Year_passing
Q> Write a query to fetch the Aggregated Marks based on the Year_passing and maximum score of the subject.

Ans:
`select sum(max_marks),year_passing from (select max(marks) as max_marks
,year_passing,subject from student_table
group by year_passing,subject)
group by year_passing0`

What is spark vectorization?

For example, vectorization is used to read multiple rows in a table, or to compute multiple rows at once. So Spark already used vectorization to multiple purposes, which already improves the performance of the Apache Spark program. Vectorized Parquet Reader, Vectorized ORC Reader, Pandas UDF employ Spark

1) Spark architecture ?

The Spark follows the master-slave architecture. Its cluster consists of a single master and multiple slaves.

The Spark architecture depends upon two abstractions:

Resilient Distributed Dataset (RDD)
Directed Acyclic Graph (DAG)

We also have few components in the spark Architecture those are:

Driver node
worker node
Executor
Cluster manager
Task

Resilient Distributed Datasets (RDD)

The Resilient Distributed Datasets are the group of data items that can be stored in-memory on worker nodes. Here,

Resilient: Restore the data on failure.

Distributed: Data is distributed among different nodes.

Dataset: Group of data.

We will learn about RDD later in detail.

Directed Acyclic Graph (DAG)

Directed Acyclic Graph is a finite direct graph that performs a sequence of computations on data. Each node is an RDD partition, and the edge is a transformation on top of data. Here, the graph refers the navigation whereas directed and acyclic refers to how it is done.

Driver Node

The Driver Program is a process that runs the main() function of the application and creates the SparkContext object. The purpose of SparkContext is to coordinate the spark applications, running as independent sets of processes on a cluster.

To run on a cluster, the SparkContext connects to a different type of cluster managers and then perform the following tasks: -

It acquires executors on nodes in the cluster.

Then, it sends your application code to the executors. Here, the application code can be defined by JAR or Python files passed to the SparkContext.

At last, the SparkContext sends tasks to the executors to run.

Cluster Manager

The role of the cluster manager is to allocate resources across applications. The Spark is capable enough of running on a large number of clusters.

It consists of various types of cluster managers such as Hadoop YARN, Apache Mesos and Standalone Scheduler.

Here, the Standalone Scheduler is a standalone spark cluster manager that facilitates to install Spark on an empty set of machines.

Worker Node

The worker node is a slave node

Its role is to run the application code in the cluster.

Executor

An executor is a process launched for an application on a worker node.

It runs tasks and keeps data in memory or disk storage across them.
It reads and writes data to the external sources.
Every application contains its executor.

Task

A unit of work that will be sent to one executor.

2) role of DAG?

DAGScheduler is the scheduling layer of Apache Spark that implements stage-oriented scheduling using Jobs and Stages.

DAGScheduler transforms a logical execution plan (RDD lineage of dependencies built using RDD transformations) to a physical execution plan (using stages).

After an action has been called on an RDD, SparkContext hands over a logical plan to DAGScheduler that it in turn translates to a set of stages that are submitted as TaskSets for execution.

DAGScheduler does three things in Spark:

- >Computes an execution DAG (DAG of stages) for a job
 - >Determines the preferred locations to run each task on
 - >Handles failures due to shuffle output files being lost
-
-

3) partitioning and bucketing

Bucketing is similar to partitioning, but partitioning creates a directory for each partition, whereas bucketing distributes data across a fixed number of buckets by a hash on the bucket value. Tables can be bucketed on more than one value and bucketing can be used with or without partitioning

4) dynamic partitioning vs static partitioning

In static partitioning we need to specify the partition column value in each and every LOAD statement. Dynamic partitioning allows us not to specify partition column value each time.

Data Loading in static partitioning is faster as compare to dynamic partitioning so static partitioning is preferred when we have massive files to load. In static partitioning individual files are loaded as per the partition we want to set.

=====

=====

5) lazy evaluation?

Lazy evaluation means that Spark does not evaluate each transformation as they arrive, but instead queues them together and evaluate all at once, as an Action is called. The benefit of this approach is that Spark can make optimization decisions after it had a chance to look at the DAG in entirety

=====

=====

6) Wide transformations and narrow transformations

Narrow transformations

--> Narrow transformations transform data without any shuffle involved.

--> These transformations transform the data on a per-partition basis; that is to say, each element of the output RDD can be computed without involving any elements from different partitions.

ex: map(), filter()

wide transformation

In wide transformation, all the elements that are required to compute the records in the single partition may live in many partitions of parent RDD

=====

=====

7) on what basis you will decide the column on which partitioning or bucketing can be applied?

Explain with a small example+

Spark/PySpark partitioning is a way to split the data into multiple partitions so that you can execute transformations on multiple partitions in parallel which allows completing the job faster. You can also write partitioned data into a file system (multiple sub-directories) for faster reads by downstream systems.

Spark/PySpark supports partitioning in memory (RDD/DataFrame) and partitioning on the disk (File system).

Partition in memory: You can partition or repartition the DataFrame by calling repartition() or coalesce() transformations.

Example :

Let's assume we have data of software employees in India with state wise
So, we can do partition or bucketing in state wise and region wise.

8) use case for cache or persist

Both caching and persisting are used to save the Spark RDD, Dataframe, and Dataset's. But, the difference is, RDD cache() method default saves it to memory (MEMORY_ONLY) whereas persist() method is used to store it to the user-defined storage level

9) storage levels of persist

Spark has various persistence levels to store the RDDs on disk or in memory or as a combination of both with different replication levels namely:

MEMORY_ONLY
MEMORY_ONLY_SER
MEMORY_AND_DISK
MEMORY_AND_DISK_SER,
DISK_ONLY
OFF_HEAP

10) serialisation

--> Serialization is used for performance tuning on Apache Spark.

--> All data that is sent over the network or written to the disk or persisted in the memory should be serialized.

--> Serialization plays an important role in costly operations.

--> PySpark supports custom serializers for performance tuning.

--> Serialization plays an important role in the performance of any distributed application.

--> Formats that are slow to serialize objects into, or consume a large number of bytes, will greatly slow down the computation

11) optimization techniques of Hive?

However, to run queries on petabytes of data we all know that hive is a query language which is similar to SQL built on Hadoop ecosystem. So, there are several Hive optimization techniques to improve its performance which we can implement when we run our hive queries

Tez-Execution Engine in Hive
Usage of Suitable File Format in Hive
Hive Partitioning
Bucketing in Hive
Vectorization In Hive

Cost-Based Optimization in Hive (CBO) Hive Indexing

<https://data-flair.training/blogs/hive-optimization-techniques/>
follow the link for more details.

12) optimization techniques of spark?

- Data filtering as early as possible
- File format selection
- API Selection
- Use of advance variables
- Parallelism using Coalesce/Repartition
- Data Serialization
- Caching and Persistence

13) Advantages of Spark over Map Reduce?

Spark is a Hadoop enhancement to MapReduce. The primary difference between Spark and MapReduce is that Spark processes and retains data in memory for subsequent steps, whereas MapReduce processes data on disk. As a result, for smaller workloads, Spark's data processing speeds are up to 100x faster than MapReduce

14) client mode vs cluster mode.

In client mode, the driver runs in the client process, and the application master is only used for requesting resources from YARN.

In cluster mode, the Spark driver runs inside an application master process which is managed by YARN on the cluster, and the client can go away after initiating the application.

15) parameters mentioned in spark-submit command

The spark-submit command is a utility to run or submit a Spark or PySpark application program (or job) to the cluster by specifying options and configurations, the application you are submitting can be written in Scala, Java, or Python (PySpark).

we can use Spark-submit command along with .py file

For ex:

spark-submit test.py

16) deploy mode?

Deploy mode specifies the location of where driver executes in the deployment environment.

Deploy mode can be one of the following options: client (default) - the driver runs on the machine that the Spark application was launched. cluster - the driver runs on a random node in a cluster.

17) dynamic resource allocation

Dynamic Resource Allocation. Spark provides a mechanism to dynamically adjust the resources your application occupies based on the workload. This means that your application may give resources back to the cluster if they are no longer used and request them again later when there is demand

18) static resource allocation

Static Allocation – The values are given as part of spark-submit. Dynamic Allocation – The values are picked up based on the requirement (size of data, amount of computations needed) and released after use. This helps the resources to be re-used for other applications

19) which file formats you are using in your project

CSV, Parquet, Json.

If we get data in other formats then we will change it into Csv and we do the transformations on it.

20) avro, orc, parquet file formats

Avro format is a row-based storage format for Hadoop, which is widely used as a serialization platform. Avro format stores the schema in JSON format, making it easy to read and interpret by any program. The data itself is stored in a binary format making it compact and efficient in Avro files

Apache ORC (Optimized Row Columnar) is a free and open-source column-oriented data storage format. It is similar to the other columnar-storage file formats available in the Hadoop ecosystem such as RCFile and Parquet

Apache Parquet is an open source, column-oriented data file format designed for efficient data storage and retrieval. It provides efficient data compression and encoding schemes with enhanced performance to handle complex data in bulk.

21) what is data skewness

Skewness is the statistical term, which refers to the value distribution in a given dataset. When we say that there is highly skewed data, it means that some column values have more rows and some very few, i.e., the data is not properly/evenly distributed

22) use case of data skewness

Usually, in Apache Spark, data skewness is caused by transformations that change data partitioning like join, groupBy, and orderBy. For example, joining on a key that is not evenly

distributed across the cluster, causing some partitions to be very large and not allowing Spark to process data in parallel.

23) salting technique?

In Spark, SALT is a technique that adds random values to push Spark partition data evenly. It's usually good to adopt for wide transformation requires shuffling like join operation. The following image visualizes how SALT is going to change the key distribution.

24) repartition and coalesce?

Spark repartition () is used to increase or decrease the RDD, DataFrame, Dataset partitions whereas the Spark coalesce () is used to only decrease the number of partitions in an efficient way.

coalesce may run faster than repartition, but unequal sized partitions are generally slower to work with than equal sized partitions. One additional point to note here is that, as the basic principle of Spark RDD is immutability. The repartition or coalesce will create new RDD.

data shuffling is main thing

25) how to load data from a file to dataframe

Using the read_csv() function from the pandas package, you can import tabular data from CSV files into pandas dataframe by specifying a parameter value for the file name (e.g. pd.read_csv("filename.csv")). Remember that you gave pandas an alias (pd), so you will use pd to call pandas functions

26) spark context & spark session

sparkContext was used as a channel to access all spark functionality.

The spark driver program uses spark context to connect to the cluster through a resource manager (YARN or Mesos). sparkConf is required to create the spark context object, which stores configuration parameter like appName (to identify your spark driver), application, number of core and memory size of executor running on worker node.

SparkSession provides a single point of entry to interact with underlying Spark functionality and allows programming Spark with Dataframe and Dataset APIs. All the functionality available with sparkContext are also available in sparkSession. Once the SparkSession is instantiated, we can configure Spark's run-time config properties

27) sortby vs order by

`sort()` is more efficient compared to `orderBy()` because the data is sorted on each partition individually and this is why the order in the output data is not guaranteed. On the other hand, `orderBy()` collects all the data into a single executor and then sorts them

28) joins ... Explain with example

join in Spark SQL is the functionality to join two or more datasets that are similar to the table join in SQL based databases. Spark works as the tabular form of datasets and data frames. The Spark SQL supports several types of joins such as inner join, cross join, left outer join, right outer join, full outer join, left semi-join, left anti join

Default is inner join.

1. INNER JOIN

The INNER JOIN returns the dataset which has the rows that have matching values in both the datasets i.e. value of the common field will be the same.

2. CROSS JOIN

The CROSS JOIN returns the dataset which is the number of rows in the first dataset multiplied by the number of rows in the second dataset. Such kind of result is called the Cartesian Product.

Prerequisite: For using a cross join, `spark.sql.crossJoin.enabled` must be set to true. Otherwise, the exception will be thrown.

3. LEFT OUTER JOIN

The LEFT OUTER JOIN returns the dataset that has all rows from the left dataset, and the matched rows from the right dataset.

4. RIGHT OUTER JOIN

The RIGHT OUTER JOIN returns the dataset that has all rows from the right dataset, and the matched rows from the left dataset.

5. FULL OUTER JOIN

The FULL OUTER JOIN returns the dataset that has all rows when there is a match in either the left or right dataset.

6. LEFT SEMI JOIN

The LEFT SEMI JOIN returns the dataset which has all rows from the left dataset having their correspondence in the right dataset. Unlike the LEFT OUTER JOIN, the returned dataset in LEFT SEMI JOIN contains only the columns from the left dataset.

29) window functions

spark window function operate on a group of row and return the single value for every input
spark support 3 kinds of windows funtions:

ranking functions

analytic functions

aggregate functions

ranking functions:

=====

RANK() will assign non-consecutive “ranks” to the values

ranking functions are of 3 types:

dense_rank Window Function:dense_rank() window function is used to get the result with rank of rows within a window partition without any gaps. This is similar to rank() function difference being rank function leaves gaps in rank when there are ties.

analytic functions:

=====

analytic functions are of 2 types:

lag window function

lead window function

lag Window Function: This is the same as the LAG function in SQL.

lead Window Function: This is the same as the LEAD function in SQL.

Aggregate Functions:

using this function calculate sum, min, max for each department using Spark SQL Aggregate window functions and WindowSpec.

When working with Aggregate functions, we don't need to use order by clause.

30) row number, rank, dense rank

RANK() will assign non-consecutive “ranks” to the values

ranking functions are of 3 types:

row_number window function

rank window function

dense_rank window function

row_number Window Function: row_number() window function is used to give the sequential row number starting from 1 to the result of each window partition.

rank Window Function: rank() window function is used to provide a rank to the result within a window partition. This function leaves gaps in rank when there are ties.

dense_rank Window Function:dense_rank() window function is used to get the result with rank of rows within a window partition without any gaps. This is similar to rank() function difference being rank function leaves gaps in rank when there are ties.

31) applying windowing/partition by on aggregate functions

Aggregate functions can be used as window functions; that is, you can use the OVER clause with aggregate functions.

This query computes, for each partition, the aggregate over the rows in that partition.

32) when to use inner join and outer join

If you only want the rows from A that have a match in B, that's an INNER join.

If you want all the rows from A whether or not they match B, that's a LEFT OUTER join.

33) difference between truncate, drop and delete

TRUNCATE which only deletes the data of the tables, the DROP command deletes the data of the table as well as removes the entire schema/structure of the table from the database.

DELETE operations can be rolled back (undone), while DROP and TRUNCATE operations cannot be rolled back

34) how to copy file from local to hdfs

In order to copy a file from the local file system to HDFS, use Hadoop fs -put or hdfs dfs -put, on put command, specify the local-file-path where you wanted to copy from and then HDFS-file-path where you wanted to copy to. If the file already exists on HDFS, you will get an error message saying "File already exists".

35) RDD vs Dataframe vs Datasets

• Spark RDD APIs – An RDD stands for Resilient Distributed Datasets.

--> It is Read-only partition collection of records.

--> RDD is the fundamental data structure of Spark.

--> It allows a programmer to perform in-memory computations on large clusters in a fault-tolerant manner.

• Spark Dataframe APIs – Unlike an RDD, data organized into named columns.

--> For example a table in a relational database.

--> It is an immutable distributed collection of data.

--> DataFrame in Spark allows developers to impose a structure onto a distributed collection of data, allowing higher-level abstraction.

• Spark Dataset APIs – Datasets in Apache Spark are an extension of DataFrame API which provides type-safe, object-oriented programming interface.

36) scenario... How to delete the duplicate records in a table.... By apply rownumber function on table.. we can delete records with rownumber greater than 1. Thus maintaining unique records

The row_number() function returns the sequential row number starting from the 1 to the result of each window partition. The rank() function in PySpark returns the rank to the development within the window partition. So, this function leaves gaps in the class when there are ties.

Example :

```
from pyspark.sql.window import *
from pyspark.sql.functions import row_number
df.withColumn("row_num", row_number().over(Window.partitionBy("Group").orderBy("Date")))
now the row number is unique value and we can delete record where count greater than 1 but it wont maintain unique records once if it delete
```

37) scala program to print word count in scala or fibonicci or palindrome

create several lines of words.

```
scala> val lines = List("Hello world", "This is a hello world example for word count")
lines: List[String] = List(Hello world, This is a hello world example for word count)
```

2) use flatMap to convert to words lists

```
scala> val words = lines.flatMap(_.split(" "))
words: List[String] = List(Hello, world, This, is, a, hello, world, example, for, word, count)
```

3) group words to map

```
scala> words.groupBy((word:String) => word)
res9: scala.collection.immutable.Map[String,List[String]] = Map(for -> List(for), count -> List(count), is -> List(is), This -> List(This), world -> List(world, world), a -> List(a), Hello -> List(Hello), hello -> List(hello), example -> List(example), word -> List(word))
The structure of each map is Map[String,List[String]].
```

4) count length of each words list

```
scala> words.groupBy((word:String) => word).mapValues(_.length)
res12: scala.collection.immutable.Map[String,Int] = Map(for -> 1, count -> 1, i
```

38) types of tables in hive

There are two types of tables in Hive:

- Managed Table (Internal)
 - External
 - . Temporar
- Managed (Internal) Table
-

In the Managed table, Apache Hive is responsible for the table data and metadata, and any action on tables data will affect physical files of data.

External Table (Un-Managed Tables)

In the External table, Apache Hive is responsible ONLY for the table metadata, and any action on the table will affect only table metadata, for example, if you deleted a table's partition actually partition metadata will be deleted from Hive warehouse only, but actual data will still be available on its current location. You can write data to external tables, but actions such as drop, delete, and so on will affect only table metadata, not the actual data.

39) limitations of temporary tables

Temporary tables created with CREATE TEMPORARY TABLE have the following limitations:

TEMPORARY tables are supported only by the InnoDB, MEMORY, MyISAM, and MERGE storage engines.

Temporary tables are not supported for NDB Cluster.

The SHOW TABLES statement does not list TEMPORARY tables.

To rename TEMPORARY tables, RENAME TABLE does not work. Use ALTER TABLE instead:
ALTER TABLE old_name RENAME new_name;

You cannot refer to a TEMPORARY table more than once in the same query. For example, the following does not work:

```
SELECT * FROM temp_table JOIN temp_table AS t2;
```

The statement produces this error:

```
ERROR 1137: Can't reopen table: 'temp_table'
```

You can work around this issue if your query permits use of a common table expression (CTE) rather than a TEMPORARY table. For example, this fails with the Can't reopen table error:

```
CREATE TEMPORARY TABLE t SELECT 1 AS col_a, 2 AS col_b;
```

```
SELECT * FROM t AS t1 JOIN t AS t2;
```

To avoid the error, use a WITH clause that defines a CTE, rather than the TEMPORARY table:

```
WITH cte AS (SELECT 1 AS col_a, 2 AS col_b)
```

```
SELECT * FROM cte AS t1 JOIN cte AS t2;
```

>The Can't reopen table error also occurs if you refer to a temporary table multiple times in a stored function under different aliases, even if the references occur in different statements within the function. It may occur for temporary tables created outside stored functions and referred to across multiple calling and callee functions.

>If a TEMPORARY is created with the same name as an existing non-TEMPORARY table, the non-TEMPORARY table is hidden until the TEMPORARY table is dropped, even if the tables use different storage engines.

>There are known issues in using temporary tables with replication. See Section 17.5.1.31, "Replication and Temporary Tables", for more information.

40) sample code to load a CSV file to dataframe
df = spark.read.csv("path1,path2,path3")

S&P Global - Ankit

Self introduction

1.spark Architecture

The Spark follows the master-slave architecture. Its cluster consists of a single master and multiple slaves.

The Spark architecture depends upon two abstractions:

Resilient Distributed Dataset (RDD)

Directed Acyclic Graph (DAG)

We also have few components in the spark Architecture those are:

Driver node

worker node

Executor

Cluster manager

Task

Resilient Distributed Datasets (RDD)

The Resilient Distributed Datasets are the group of data items that can be stored in-memory on worker nodes. Here,

Resilient: Restore the data on failure.

Distributed: Data is distributed among different nodes.

Dataset: Group of data.

We will learn about RDD later in detail.

Directed Acyclic Graph (DAG)

Directed Acyclic Graph is a finite direct graph that performs a sequence of computations on data. Each node is an RDD partition, and the edge is a transformation on top of data. Here, the graph refers the navigation whereas directed and acyclic refers to how it is done.

Driver Node

The Driver Program is a process that runs the main() function of the application and creates the SparkContext object. The purpose of SparkContext is to coordinate the spark applications, running as independent sets of processes on a cluster.

To run on a cluster, the SparkContext connects to a different type of cluster managers and then perform the following tasks: -

It acquires executors on nodes in the cluster.

Then, it sends your application code to the executors. Here, the application code can be defined by JAR or Python files passed to the SparkContext.

At last, the SparkContext sends tasks to the executors to run.

Cluster Manager

The role of the cluster manager is to allocate resources across applications. The Spark is capable enough of running on a large number of clusters.

It consists of various types of cluster managers such as Hadoop YARN, Apache Mesos and Standalone Scheduler.

Here, the Standalone Scheduler is a standalone spark cluster manager that facilitates to install Spark on an empty set of machines.

Worker Node

The worker node is a slave node

Its role is to run the application code in the cluster.

Executor

An executor is a process launched for an application on a worker node.

It runs tasks and keeps data in memory or disk storage across them.

It read and write data to the external sources.

Every application contains its executor.

Task

A unit of work that will be sent to one executor.

2.what is Spark Driver

It is the master node in a Spark application.

A Spark driver (aka an application's driver process) is a JVM process that hosts SparkContext for a Spark application.

It is the combination of jobs and tasks execution (using DAGScheduler and Task Scheduler).

It splits a Spark application into tasks and schedules them to run on executors.

A driver is where the task scheduler lives and spawns tasks across workers.

A driver coordinates workers and overall execution of tasks.

3.Whom will decide the division of data in the spark driver and worker node

4.how will you filter the records from the particular data frame, like df name is city, He wants the records

We use isin() method for that, here we can left join for better performance.

5.If you have 2df and large in size, and you can not join them in your system, then what will be your approach toward that.

We will

6. Caching techniques in spark and why we use caching techniques?
7. When we have Squit data(key value format) and specific key is in multiple keys then how will you process the data and we can't delete the duplicate data? (salting techniques)
8. Optimization techniques
9. How did you connect to the table using Pyspark.
10. Suppose sale table, with few employee columns emp_name, designation, dept, sales. Now i want top3 emp od every dept who did most sale

```
df.groupby('dept')
```
11. Debugging
12. Class method in python
13. Lambda functions.
14. repartition and coalesce

1.) ways to read a file in pysaprk and what are the file extension that pyspark support to read?

PySpark – Read & Write CSV File

PySpark – Read & Write Parquet File

PySpark – Read & Write JSON file

There are three ways to read text files into PySpark DataFrame.

Using spark.read.text()

Using spark.read.csv()

Using spark.read.format().load()

csv

text

Avro

Parquet

tsv

xml and many more

2.) ways to write a file in pysaprk and what are the file extension that pyspark support to write?

3.) difference between filter and where?

4.) how select is used in pyspark?

5.) what is different types of filter in pyspark?

6.) repartition vs coalesce

with repartition() the number of partitions can be increased/decreased, but

with coalesce() the number of partitions can only be decreased.

7.) hadoop architecture

8.) spark architecture

Apache Spark works in a master-slave architecture where the master is called “Driver” and slaves are called “Workers”. When you run a Spark application, Spark Driver creates a context

that is an entry point to your application, and all operations (transformations and actions) are executed on worker nodes, and the resources are managed by Cluster Manager.

9.) partition vs bucketing

10.) data skewness

Skewness is the statistical term, which refers to the value distribution in a given dataset.

When we say that there is highly skewed data, it means that some column values have more rows and some very few, i.e.,
the data is not properly/evenly distributed

11.) data shuffling

Shuffling is the process of exchanging data between partitions. As a result, data rows can move between worker nodes when their source partition and the target partition reside on a different machine.

Spark doesn't move data between nodes randomly. Shuffling is a time-consuming operation, so it happens only when there is no other option

- 1) How to full data from client location to DBFs
- 2) Dict
- 3) isin
- 4) @class method,@static method
- 5) avro,orc
- 6) Optimization techniques of Hive & Spark
- 7) How to create a new DataFrame
- 8) Join's
- 9) Decorators
- 10) Datawarehouse & Datamart
- 11) Repartition & Coalasce
- 12) Schema & Types
- 13) Fact & Dimension Tables
- 14) Cluster
- 15) Decorators, Datamart, Datalake, Deltatable

Python:

1. Vaiable scopes of LEGB rule

--> The LEGB rule names the possible scopes of a variable in Python: Local, Enclosing, Global, Built-in.

Python searches namespaces in this order to determine the value of an object given its name.

Scopes are created with functions, classes, and modules.

Local : The variables assigned inside a function in python program.

Enclosed : The variables assigned inside inside a function(inner class).
Global : The variables assigned outside a function in python program.
Builtin : If there is no local,enclosed,global variables then python will search for Builtin variables.

2.Data types in python

1.NUMBERS ---> python supports four different data types .
they are int,long,float,complex

2.STRING ---> strings are amongst the most popular type in python.
we can create them simply enclosing characters in quotes
python treats single quotes the same as double quotes.

3.BOOLEAN --->True or False, Yes or No

3. What is decorator,generator& iterator

A Decorator is just a function that takes another function as an argument and extends its behavior without explicitly modifying it.

An iterator is an object that can be iterated upon which means that you can traverse through all the values.

`__iter__` returns the iterator object itself. This is used in for and in statements.
`__next__` method returns the next value from the iterator. If there is no more items to return then it should raise StopIteration exception.

Generator functions act just like regular functions with just one difference that they use the Python yield keyword instead of return . A generator function is a function that returns an iterator. A generator expression is an expression that returns an iterator. Generator objects are used either by calling the next method on the generator object or using the generator object in a “for in” loop.

4. MEMORY MANAGEMENT IN PYTHON:

--> Memory manager inside the PVM allocates memory required for objects created in a Python program.
--> All these objects are stored on separate memory called heap.
--> Heap is the memory which allocated during runtime.
--> The size of the heap memory depends on the Random Access Memory of our computer and it can increase or decrease its size depending on the requirement of the program.

5.Explain break vs continue, pass

--> Break - The break keyword causes program control to exit from the loop when some condition is met.

--> Continue - The continue statement causes program control to skip all the remaining statements in the current iteration of the loop and returns the control to the beginning of the loop.

--> Pass - The Pass statement causes program control to pass by without executing any code. If we want to bypass any code pass statement can be used.

1. What are joins in SQL?

Joins are used to combine rows from two or more tables, based on a related column between them.

Types of Joins:

- INNER JOIN – Returns rows when there is a match in both tables.
- LEFT JOIN – Returns all rows from the left table, even if there are no matches in the right table.
- RIGHT JOIN – Returns all rows from the right table, even if there are no matches in the left table.
- FULL OUTER JOIN – Returns rows when there is a match in one of the tables.
- SELF JOIN – Used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.
- CARTESIAN JOIN (CROSS JOIN) – Returns the Cartesian product of the sets of records from the two or more joined tables.

2.What are the types of SQL Queries?

We have four types of SQL Queries:

DDL (Data Definition Language): the creation of objects

DML (Data Manipulation Language): manipulation of data

DCL (Data Control Language): assignment and removal of permissions

TCL (Transaction Control Language): saving and restoring changes to a database

Let's look at the different commands under DDL:

CREATE Create objects in the database

ALTER Alters the structure of the database object

DROP Delete objects from the database

TRUNCATE Remove all records from a table permanently

COMMENT Add comments to the data dictionary

RENAME Rename an object

3.What is a Stored Procedure?

A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.

So if you have an SQL query that you write over and over again, save it as a stored procedure, and then just call it to execute it.

Stored Procedure With One Parameter

Stored Procedure With Multiple Parameters

4. the difference between Order by and group by clause:-

GROUP BY

Group by statement is used to group the rows that have the same value.

ORDER BY

Whereas Order by statement sort the result-set either in ascending or in descending order.

5.How to Find Duplicate Values in SQL

finding duplicates values in SQL comprises two key steps:

1.Using the GROUP BY clause to group all rows by the target column(s) – i.e. the column(s) you want to check for duplicate values on.

2.Using the COUNT function in the HAVING clause to check if any of the groups have more than 1 entry; those would be the duplicate values.

Todays Spark task :

3. Spark Cluster Managers

4. Introduction to DataBricks

SQL Task:

Try to do in Snowflake On dbeaver

Day 3: primary key, foreign key, unique , not null

We will connect everyday at 6pm to discuss all today tasks & topics

Revision1(13.09.2022):

Spark Task1:

1. Course Introduction: What is Spark?

Apache Spark is a fast and general purpose engine for large scale data processing framework

2. Why Apache Spark?

a. Speed : Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk.

b. easy to use : Write applications quickly in Java, Scala, Python, R

c. generality : Combine SQL, streaming, and complex analytics.

d. runs on everywhere : Spark runs on Hadoop, Mesos, standalone, or in the cloud.

SQL Task1:

Try to do in Snowflake

Day 2: Start & Create a table , insert , update , delete, alter and so on ...

DDL Commands - ALTER COMMENT CREATE DESCRIBE DROP SHOW USE
DML Commands - INSERT MERGE UPDATE DELETE TRUNCATE
DQL Commands - SELECT
TCL Commands - START TRANSACTION COMMIT ROLLBACK
DCL Commands - GRANT REVOKE

- 1.) about yourself?
- 2.) etl vs elt, which one is better?
- 3.) datalake vs datawarehouse
- 4.) fact vs dimension
- 5.) what is fact?
- 6.) what is dimension?
- 7.) how to write etl pipeline?
- 8.) what is the consideration for etl pipeline
- 9.) what kind error you have handled?
- 10.) what is spark context?
- 11.) what is stages in spark?
- 12.) olap vs oltp
- 13.) list vs tuple
- 14.) joins in sql?
- 15.) acid property?
- 16.) have you worked in snowflake?
- 17.) have you handled transaction with comparison of another table?
- 18.) where is destination?
- 19.) primary key vs foreign key vs composite key
- 20.) does foreign key have duplicate value
- 21.) oops in python
- 22.) inheritance , polymorphism
- 23.) batch vs streaming
- 24.) rdd, why rdd is lazy evaluation and what is resilient in rdd?
- 25.) what is transformation and action?
- 26.) Spark driver architecture?
- 27.) star schema vs snowflake schema
- 28.) datalake vs datawarehouse
- 29.) spark sql or pyspark
- 30.) delta lake in databricks is data lake or datawarehouse

- 1.) what is rdd?
- 2.) what is dataframe?
- 3.) what is fault tolerance?
- 4.) what is bigdata?
- 5.) create a udf to reverse a string in pyspark.
- 6.) create a udf to get len of string.

- 1.) ways to read a file in pysaprk and what are the file extension that pyspark support to read?
- 2.) ways to write a file in pysaprk and what are the file extension that pyspark support to write?
- 3.) difference between filter and where?
- 4.) how select is used in pyspark?
- 5.) what is different types of filter in pyspark?
- 6.) what is Caching and Persistence of DataFrame?

- 1.) repartition vs coalesce
- 2.) hadoop architecture
- 3.) spark architecture
- 4.) partition vs bucketing
- 5.) data skewness
- 6.) data shuffling

- 1.) ways to read a file in pysaprk and what are the file extension that pyspark support to read?
- 2.) ways to write a file in pysaprk and what are the file extension that pyspark support to write?
- 3.) difference between filter and where?
- 4.) how select is used in pyspark?
- 5.) what is different types of filter in pyspark?

- 1.) about your self?
- 2.) project problem statement and project architecture?
- 3.) what transformation you did?
- 4.) what kind of data you have handle?
- 5.) how you handle error in pipeline?
- 6.) how you fix pipeline error which is not occurring in local?
- 7.) what is adds gen 2?
- 8.) what is use of data bricks?
- 9.) where you load the data for client?

10.) if you are receiving data from api how you handle?

11.) have you worked in streaming data?

12.) spark architecture?

13.) components in spark?

14.) lazy evaluation?

15.) dag scheduler?

16.) decorator?

17.) tuple?

18.) python datatypes?

19.) repartition and partition?

20.) chache and persist?

21.) optimization tricks in pyspark?

Online Analytical Processing (OLAP):

Online Analytical Processing consists of a type of software tools that are used for data analysis for business decisions. OLAP provides an environment to get insights from the database retrieved from multiple database systems at one time. **Examples** – Any type of Data warehouse system is an OLAP system. The uses of OLAP are as follows:

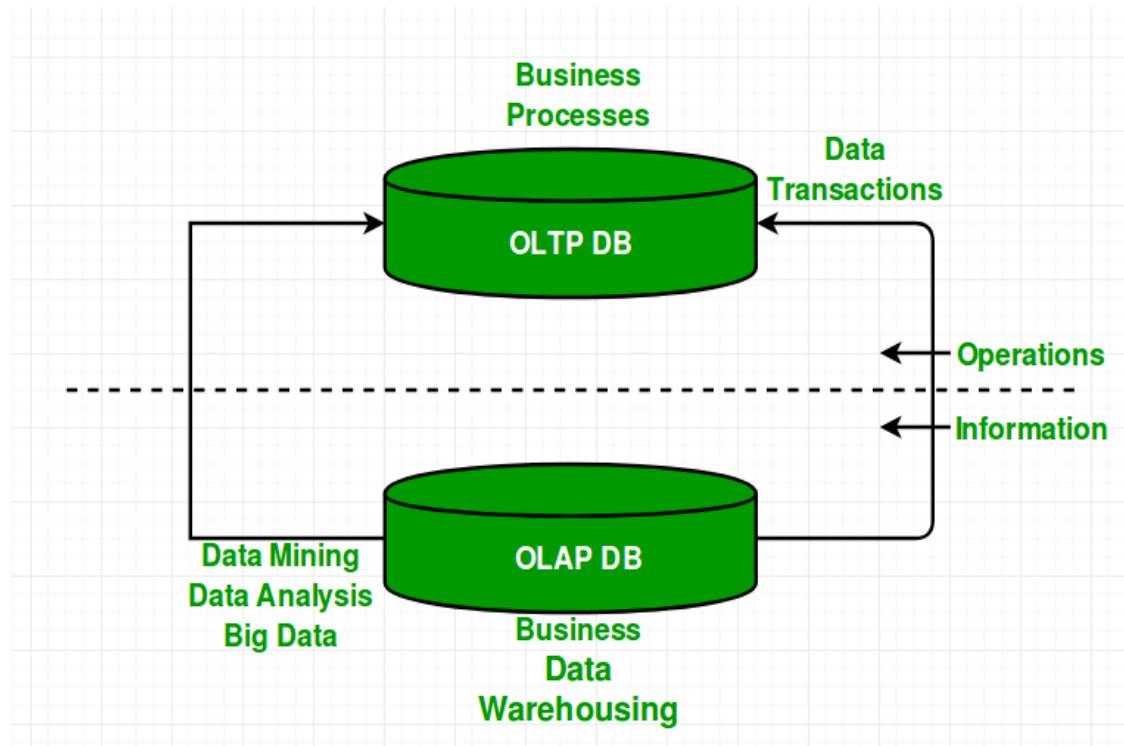
- Spotify analysed songs by users to come up with a personalized homepage of their songs and playlist.
- Netflix movie recommendation system.

Online transaction processing (OLTP):

OLTP or Online Transaction Processing is a type of data processing that consists of executing a number of transactions occurring concurrently

Examples: Uses of OLTP are as follows:

- ATM center is an OLTP application.
- OLTP handles the ACID properties during data transactions via the application.
- It's also used for Online banking, Online airline ticket booking, sending a text message, add a book to the shopping cart.



OLAP	OLTP
✓ Gives a multi-dimensional view of business activities.	✓ Enables a snapshot of ongoing business processes.
✓ Helps with planning, problem solving, and decision support.	✓ Useful for controlling and running fundamental business tasks.
✓ Data source is consolidated data	✓ Data source is the operational data.
✓ Includes Periodic long-running batch jobs that refresh the data.	✓ Has short and fast inserts and updates which are initiated by end users.
✓ OLAP applications are widely used by Data Mining techniques.	✓ Large number of short on-line transactions
✓ Database design is typically de-normalized and contains fewer tables.	✓ Database design in OLTP is highly normalized.
✓ Often involves complex queries along with aggregations, which in turn compels processing speed to be dependent on the amount of data involved; batch data refreshes, etc.	✓ Involves standardized and simple queries that return relatively few records hence is faster.

Database (DB)

Data

Data is a collection of a distinct small unit of information. It can be used in a variety of forms like text, numbers, media, bytes, etc. it can be stored in pieces of paper or electronic memory, etc.

In computing, Data is information that can be translated into a form for efficient movement and processing. Data is interchangeable.

Database

A database is an organized collection of data, so that it can be easily accessed and managed. Between JDK, JRE, and JVM

You can organize data into tables, rows, columns, and index it to make it easier to find relevant information.

Database handlers create a database in such a way that only one set of software program provides access of data to all the users.

VN2 Solutions Pvt. Ltd.

The main purpose of the database is to operate a large amount of information by storing, retrieving, and managing data.

There are many dynamic websites on the World Wide Web nowadays which are handled through databases. For example, a model that checks the availability of rooms in a hotel. It is an example of a dynamic website that uses a database.

There are many databases available like MySQL, Sybase, Oracle, Mongo DB, Informix, PostgreSQL, SQL Server, etc.

Modern databases are managed by the database management system (DBMS).

SQL or Structured Query Language is used to operate on the data stored in a database. SQL depends on relational algebra and tuple relational calculus.

Types of Databases

Here are some popular types of databases.

Distributed databases:

A distributed database is a type of database that has contributions from the common database and information captured by local computers. In this type of database system, the data is not in one place and is distributed at various organizations.

Relational databases:

This type of database defines database relationships in the form of tables. It is also called Relational DBMS, which is the most popular DBMS type in the market. Database example of the RDBMS system includes MySQL, Oracle, and Microsoft SQL Server database.

Object-oriented databases:

This type of computers database supports the storage of all data types. The data is stored in the form of objects. The objects to be held in the database have attributes and methods that define what to do with the data. PostgreSQL is an example of an object-oriented relational DBMS.

Centralized database:

It is a centralized location, and users from different backgrounds can access this data. This type of computers databases store application procedures that help users access the data even from a remote location.

Open-source databases:

This kind of database stored information related to operations. It is mainly used in the field of marketing, employee relations, customer service, of databases.

Cloud databases:

A cloud database is a database which is optimized or built for such a virtualized environment. There are so many advantages of a cloud database, some of which can pay for storage capacity and bandwidth. It also offers scalability on-demand, along with high availability.

Database Access Language:

Database Access language is used to access the data to and from the database, enter new data, update already existing data, or retrieve required data from DBMS. The user writes some specific commands in a database access language and submits these to the database.

Database Management System:

Database Management System (DBMS) is a collection of programs that enable its users to access databases, manipulate data, report, and represent data. It also helps to control access to the database

Advantages of DBMS

- DBMS offers a variety of techniques to store & retrieve data.
- DBMS serves as an efficient handler to balance the needs of multiple applications using the same data.
- Uniform administration procedures for data.
- Application programmers never exposed to details of data representation and storage.
- A DBMS uses various powerful functions to store and retrieve data efficiently.
- Offers Data Integrity and Security.
- The DBMS implies integrity constraints to get a high level of protection against prohibited access to data.
- A DBMS schedules concurrent access to the data in such a manner that only one user can access the same data at a time.
- Reduced Application Development Time.

Disadvantage of DBMS

DBMS may offer plenty of advantages but, it has certain flaws-

- Cost of Hardware and Software of a DBMS is quite high which increases the budget of your organization.
- Most database management systems are often complex systems, so the training for users to use the DBMS is required.
- In some organizations, all data is integrated into a single database which can be damaged because of electric failure or database is corrupted on the storage media.
- Use of the same program at a time by many users sometimes leads to the loss of some data.
- DBMS can't perform sophisticated calculations.

Data Warehouse

VN2 Solutions Pvt. Ltd.

Data Warehouse is a relational database management system (RDBMS) construct to meet the requirement of transaction processing systems. It can be loosely described as any centralized data repository which can be queried for business benefits. It is a database that stores information oriented to satisfy decision-making requests. It is a group of decision support technologies, targets to enabling the knowledge worker (executive, manager, and analyst) to make superior and higher decisions. So, Data Warehousing support architectures and tool for business executives to systematically organize understand and use their information to make strategic decisions.

- Data Warehouse environment contains an extraction, transportation, and loading (ETL) solution, an online analytical processing (OLAP) engine, customer analysis tools, and other applications that handle the process of gathering information and delivering it to business users.

A Data Warehouse can be viewed as a data system with the following attributes:

- It is a database designed for investigative tasks, using data from various applications.
- It supports a relatively small number of clients with relatively long interactions.
- It includes current and historical data to provide a historical perspective of information.
- Its usage is read-intensive.
- It contains a few large tables.

"Data Warehouse is a subject-oriented, integrated, and time-variant store of information in support of management's decisions."

Need for Data Warehouse

1. Business User: Business users require a data warehouse to view summarized data from the past. Since these people are non-technical, the data may be presented to them in an elementary form.
2. Store historical data: Data Warehouse is required to store the time variable data from the past. This input is made to be used for various purposes.
3. Make strategic decisions: Some strategies may be depending upon the data in the data warehouse. So, data warehouse contributes to making strategic decisions.
4. For data consistency and quality: Bringing the data from different sources at a commonplace, the user can effectively undertake to bring the uniformity and consistency in data.
5. High response time: Data warehouse has to be ready for somewhat unexpected loads and types of queries, which demands a significant degree of flexibility and quick response time

VN2 Solutions Pvt. Ltd.

Benefits of Data Warehouse

1. Understand business trends and make better forecasting decisions.
2. Data Warehouses are designed to perform well enormous amounts of data.
3. The structure of data warehouses is more accessible for end-users to navigate, understand, and query.

Data Lakehouse:

A data lakehouse is a new, open data management architecture that combines the flexibility, cost-efficiency, and scale of data lakes with the data management and ACID transactions of data warehouses, enabling business intelligence (BI) and machine learning (ML) on all data.

Data Lakehouse: Simplicity, Flexibility, and Low Cost

Data lakehouses are enabled by a new, open system design: implementing similar data structures and data management features to those in a data warehouse, directly on the kind of low-cost storage used for data lakes. Merging them together into a single system means that data teams can move faster as they are able to use data without needing to access multiple systems. Data lakehouses also ensure that teams have the most complete and up-to-date data available for data science, machine learning, and business analytics projects.

There are a few key technology advancements that have enabled the data lakehouse:

- Metadata layers for data lakes
- New query engine designs providing high-performance SQL execution on data lakes
- Optimized access for data science and machine learning tools.

Difference between Schema and Table:

Schema

A schema is a collection of database objects including tables, views, triggers, stored procedures, indexes, etc. A schema is associated with a username which is known as the schema owner, who is the owner of the logically related database objects.

A schema always belongs to one database. On the other hand, a database may have one or multiple schemas. For example, in our BikeStores sample database, we have two schemas: `sales` and `production`. An object within a schema is qualified using the `schema_name.object_name` format like `sales.orders`. Two tables in two schemas can share the same name so you may have `hr.employees` and `sales.employees`.

Built-in schemas in SQL Server

SQL Server provides us with some pre-defined schemas which have the same names as the built-in database users and roles, for example: `dbo`, `guest`, `sys`, and `INFORMATION_SCHEMA`.

VN2 Solutions Pvt. Ltd.

The default schema for a newly created database is `dbo`, which is owned by the `dbo` user account. By default, when you create a new user with the `CREATE USER` command, the user will take `dbo` as its default schema.

SQL Server CREATE SCHEMA statement overview

The `CREATE SCHEMA` statement allows you to create a new schema in the current database.

The following illustrates the simplified version of the `CREATE SCHEMA` statement:

```
CREATE SCHEMA schema_name  
[AUTHORIZATION owner_name]
```

Code language: SQL (Structured Query Language) (sql)

In this syntax,

- First, specify the name of the schema that you want to create in the `CREATE SCHEMA` clause.
- Second, specify the owner of the schema after the `AUTHORIZATION` keyword.

SQL Server CREATE SCHEMA statement example

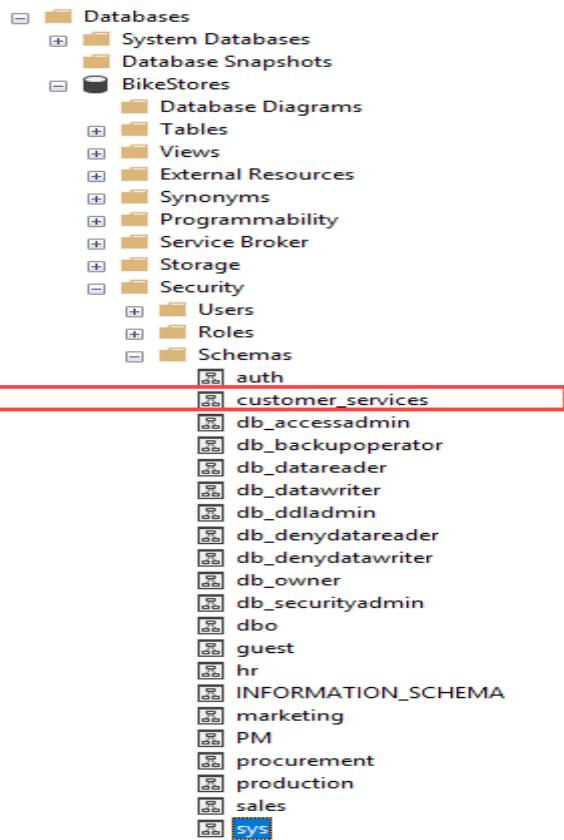
The following example shows how to use the `CREATE SCHEMA` statement to create the `customer_services` schema:

```
CREATE SCHEMA customer_services;  
GO
```

Code language: SQL (Structured Query Language) (sql)

Note that `GO` command instructs the SQL Server Management Studio to send the SQL statements up to the `GO` statement to the server to be executed.

Once you execute the statement, you can find the newly created schema under the Security > Schemas of the database name.



If you want to list all schemas in the current database, you can query schemas from the `sys.schemas` as shown in the following query:

```
SELECT
    s.name AS schema_name,
    u.name AS schema_owner
FROM
    sys.schemas s
INNER JOIN sys.sysusers u ON u.uid = s.principal_id
ORDER BY
    s.name;
```

Code language: SQL (Structured Query Language) (sql)

Here is the output:

schema_name	schema_owner
auth	dbo
customer_services	dbo
db_accessadmin	db_accessadmin
db_backupoperator	db_backupoperator
db_datareader	db_datareader
db_datawriter	db_datawriter
db_ddladmin	db_ddladmin
db_denydatareader	db_denydatareader
db_denydatawriter	db_denydatawriter
db_owner	db_owner
db_securityadmin	db_securityadmin
dbo	dbo
guest	guest
hr	dbo
INFORMATION_SCHEMA	INFORMATION_SCHEMA
marketing	dbo
PM	dbo
procurement	dbo
production	dbo
sales	dbo
sys	sys

After having the `customer_services` schema, you can create objects for the schema. For example, the following statement creates a new table named `jobs` in the `customer_services` schema:

```
CREATE TABLE customer_services.jobs(
job_id INT PRIMARY KEY IDENTITY,
customer_id INT NOT NULL,
description VARCHAR(200),
created_at DATETIME2 NOT NULL
);
```

Code language: SQL (Structured Query Language) (sql)

In this tutorial, you have learned how to use the SQL Server CREATE SCHEMA statement to create a new schema in the current database.

Table

Table is a collection of data, organized in terms of rows and columns. In DBMS term, table is known as relation and row as tuple.

Let's see an example of an employee table:

In the above table, "Employee" is the table name, "EMP_NAME", "ADDRESS" and "SALARY" are the column names. The combination of data of multiple columns forms a row e.g. "Ankit", "Lucknow" and 15000 are the data of one row.

Employee

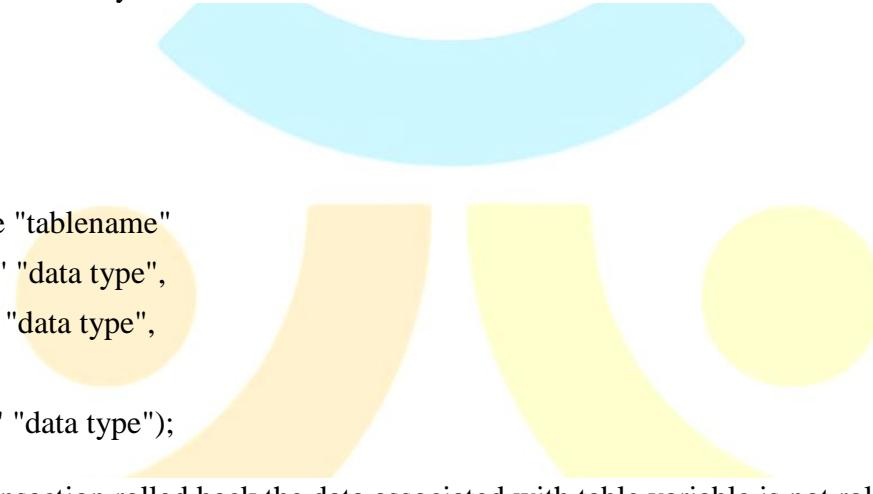
EMP_NAME	ADDRESS	SALARY
Ankit	Lucknow	15000
Raman	Allahabad	18000
Mike	New York	20000

The SQL Table variable is used to create, modify, rename, copy and delete tables. Table variable was introduced by Microsoft.

It was introduced with SQL server 2000 to be an alternative of temporary tables.

It is a variable where we temporary store records and results. This is same like temp table but in the case of temp table we need to explicitly drop it.

Table variables are used to store a set of records. So declaration syntax generally looks like CREATE TABLE syntax.



```
create table "tablename"
("column1" "data type",
"column2" "data type",
...
"columnN" "data type");
```

When a transaction rolled back the data associated with table variable is not rolled back.

A table variable generally uses lesser resources than a temporary variable.

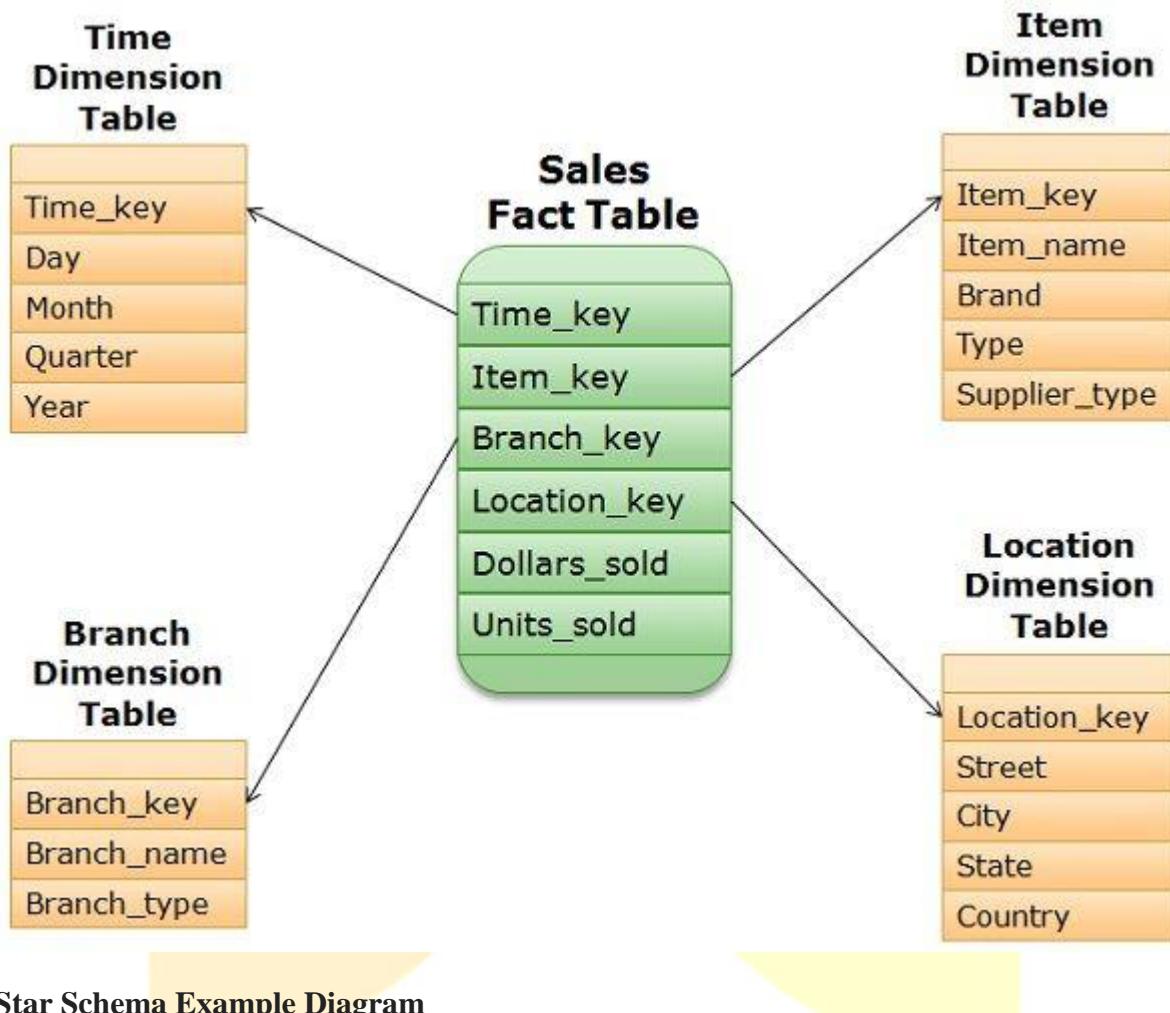
Table variable cannot be used as an input or an output parameter.

Definition of Star Schema

Star schema is the simple and common modeling paradigm where the data warehouse comprises of a fact table with a single table for each dimension. The schema imitates a star, with dimension table presented in an outspread pattern encircling the central fact table. The dimensions in fact table are connected to dimension table through primary key and foreign key.

EXAMPLE:

We are creating a schema which includes the sales of an electronic appliance manufacturing company. Sales are intended along following dimensions: time, item, branch, and location. The schema contains a central fact table for sales that includes keys to each of the four dimensions, along with two measures: dollar-sold and units-sold. The capacity of the fact table is reduced by the generation of dimension identifiers such as time_key and item_key via the system.



Star Schema Example Diagram

Only a single table imitates each dimension, and each table contains a group of attributes as it is shown in the star schema. The location dimension table encompasses the attribute set {location_key, street, city, state and country}. This restriction may introduce some redundancy. For example, two cities can be of same state and country, so entries for such cities in the location dimension table will create redundancy among the state and country attributes.

DEFINITION OF SNOWFLAKE SCHEMA

Snowflake schema is the kind of the star schema which includes the hierarchical form of dimensional tables. In this schema, there is a fact table comprise of various dimension and sub-

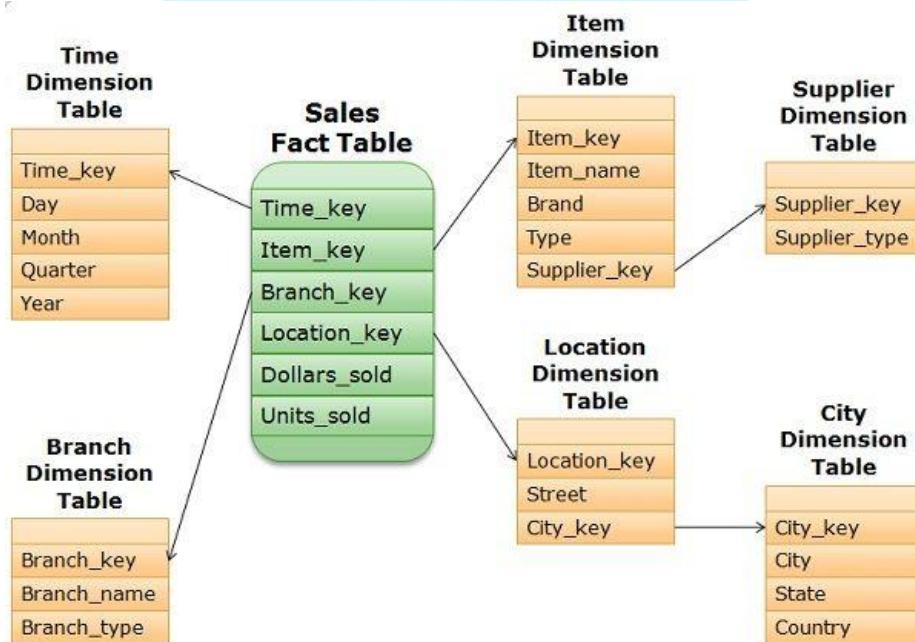
dimension table connected across through primary and foreign key to the fact table. It is named as the snowflake because its structure is similar to a snowflake.

It uses normalization which splits up the data into additional tables. The splitting results in the reduction of redundancy and prevention from memory wastage. A snowflake schema is more easily managed but complex to design and understand. It can also reduce the efficiency of browsing since more joins will be required to execute a query.

Example:

In the snowflake schema, we are taking the same example as we have taken in the star schema. Here the sales fact table is identical to that of the star schema, but the main difference lies in the definition of dimension tables.

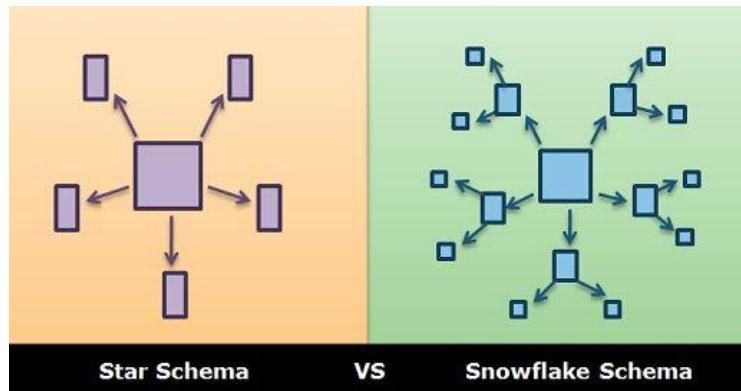
The single dimension table for the item in the star schema is normalized in the snowflake schema, resulting in creation of new item and supplier tables. For instance, the item dimension table comprised of the attributes item_key, brand, item_name, type, and supplier_key, where supplier_key is connected to the supplier dimension table, which holds supplier_key and supplier_type information.



Snowflake Schema Example Diagram

Similarly, the location dimension table involves the attributes location_key, street, and city_key, and city_key is linked to city dimension table containing the city, state and country attribute. Here state attribute can also further normalized.

Difference between Star and Snowflake Schema



Star and snowflake schemas are the most popular multidimensional data models used for a data warehouse. The crucial difference between Star schema and snowflake schema is that star schema does not use normalization whereas snowflake schema uses normalization to eliminate redundancy of data. Fact and dimension tables are essential requisites for creating schema.

Key Differences between Star and Snowflake Schema

1. Star schema contains just one dimension table for one dimension entry while there may exist dimension and sub-dimension table for one entry.
2. Normalization is used in snowflake schema which eliminates the data redundancy. As against, normalization is not performed in star schema which results in data redundancy.
3. Star schema is simple, easy to understand and involves less intricate queries. On the contrary, snowflake schema is hard to understand and involves complex queries.
4. The data model approach used in a star schema is top-down whereas snowflake schema uses bottom-up.
5. Star schema uses a fewer number of joins. On the other hand, snowflake schema uses a large number of joins.
6. The space consumed by star schema is more as compared to snowflake schema.
7. The time consumed for executing a query in a star schema is less. Conversely, snowflake schema consumes more time due to the excessive use of joins.

Star and Snowflake schema is used for designing the data warehouse. Both have certain merits and demerits where snowflake schema is easy to maintain, lessen the redundancy hence consumes less space but complex to design. Whereas star schema is simple to understand and design, uses less number of joins and simple queries but have some issues such as data redundancy and integrity.

However, use of snowflake schema minimizes redundancy, but it is not popular as star schema which is used most of the time in the design of data warehouse.

VN2 Solutions Pvt. Ltd.



VN2 Solutions Pvt. Ltd.

What is spark in big data?

Spark uses Micro-batching for real-time streaming.

Apache **Spark** is open source, general-purpose distributed computing engine **used** for processing and analyzing a large amount of data.

Just like Hadoop MapReduce, it also works with the system to distribute data across the cluster and process the data in parallel.

How is spark different from Hadoop?

Need of Apache Spark

In the industry, there is a need for general purpose cluster computing tool as:

Hadoop MapReduce can only perform batch processing.

Apache Storm / S4 can only perform stream processing.

Apache Impala / Apache Tez can only perform interactive processing

Neo4j / Apache Giraph can only perform to graph processing

Apache Spark Ecosystem

Following are 6 components in Apache Spark Ecosystem which empower to Apache Spark- Spark Core, Spark SQL, Spark Streaming, Spark MLlib, Spark Graphics , and Spark R.

Working of Apache Spark

Apache Spark is open source, general-purpose distributed computing engine used for processing and analyzing a large amount of data.

Just like Hadoop [MapReduce](#),it also works with the system to distribute data across the cluster and process the data in parallel.

Spark uses master/slave architecture i.e. one central coordinator and many distributed workers.

The central coordinator is called the driver. The driver runs in its own java process.

These drivers communicate with a potentially large number of distributed workers called executors.

Each executor is a separate java process.

A Spark Application is a combination of driver and its own executors. With the help of cluster manager

Standalone Cluster Manager is the default built in cluster manager of Spark. Apart from its built-in cluster manager, Spark works with some open source cluster manager like Hadoop Yarn, Apache Mesos etc.

Web UI — Spark Application's Web Console:

Web UI (aka **Application UI** or **webUI** or **Spark UI**) is the web interface of a Spark application to monitor and inspect Spark job executions in a web browser.

The screenshot shows the Apache Spark 2.1.0-SNAPSHOT Web UI. At the top, there is a navigation bar with tabs: Jobs (which is selected), Stages, Storage, Environment, Executors, and SQL. To the right of the tabs, it says "Spark shell application UI".

Spark Jobs (1)

User: jacek
Total Uptime: 35 s
Scheduling Mode: FIFO
Active Jobs: 1
Completed Jobs: 1
Failed Jobs: 1

▶ Event Timeline

Active Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2	show at <console>:24	2016/09/29 14:01:20	5 s	0/1	0/1

Completed Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	show at <console>:24	2016/09/29 14:01:07	0.3 s	1/1	1/1

Failed Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1	show at <console>:24	2016/09/29 14:01:14	87 ms	0/1 (1 failed)	0/1 (1 failed)

web UI comes with the following tabs (which may not all be visible immediately, but only after the respective modules are in use, e.g. the SQL or Streaming tabs):

1. Jobs
2. Stages
3. Storage
4. Environment
5. Executors

Jobs Tab:

Jobs tab in [web UI](#) shows [status of all Spark jobs](#) in a Spark application (i.e. a [SparkContext](#)).

The screenshot shows the Apache Spark 2.1.0-SNAPSHOT web UI interface. At the top, there is a navigation bar with tabs: Jobs (which is selected), Stages, Storage, Environment, Executors, and SQL. To the right of the tabs, it says "Spark shell application UI".

Spark Jobs (?)

User: jacek
Total Uptime: 35 s
Scheduling Mode: FIFO
Active Jobs: 1
Completed Jobs: 1
Failed Jobs: 1

▶ Event Timeline

Active Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2	show at <console>:24	2016/09/29 14:01:20	5 s	0/1	0/1

Completed Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	show at <console>:24	2016/09/29 14:01:07	0.3 s	1/1	1/1

Failed Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1	show at <console>:24	2016/09/29 14:01:14	87 ms	0/1 (1 failed)	0/1 (1 failed)

Stages Tab

Stages tab in [web UI](#) shows... **FIXME**

3 Fair Scheduler Pools

Pool Name	Minimum Share	Pool Weight	Active Stages	Running Tasks	SchedulingMode
production	2	1	0	0	FAIR
test	3	2	0	0	FIFO
default	0	1	1	1	FIFO

Active Stages (1)

Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
2	default	map at <console>:29 +details (kill)	2016/06/02 20:56:36	2 s	7/8	168.0 B			414.0 B

Pending Stages (1)

Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
3		count at <console>:29 +details	Unknown	Unknown	0/8				

Completed Stages (1)

Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
1	default	count at <console>:29 +details	2016/06/02 20:56:05	0.1 s	8/8	192.0 B			

Failed Stages (1)

Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write	Failure Reason
0	default	count at <console>:29 +details	2016/06/02 20:55:45	0.2 s	7/8 (1 failed)					Job aborted due to stage failure: Task 1 in stage 0.0 failed 1 times, most recent failure: Lost task 1.0 in stage 0.0 (TID 1, localhost): java.lang.Exception: failed +details

Executors Tab: Executors tab in web UI shows... **FIXME**

Spark 2.3.1-SNAPSHOT Jobs Stages Storage Environment Executors SQL Spark shell application UI

Executors

Summary

RDD Blocks	Storage Memory	Disk Used	Active Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Blacklisted
Active(1) 8	1 KB / 384.1 MB	0.0 B	8	0	0	9	9	2 s (72 ms)	0.0 B	460 B	460 B	0
Dead(0) 0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	0
Total(1) 8	1 KB / 384.1 MB	0.0 B	8	0	0	9	9	2 s (72 ms)	0.0 B	460 B	460 B	0

Executors

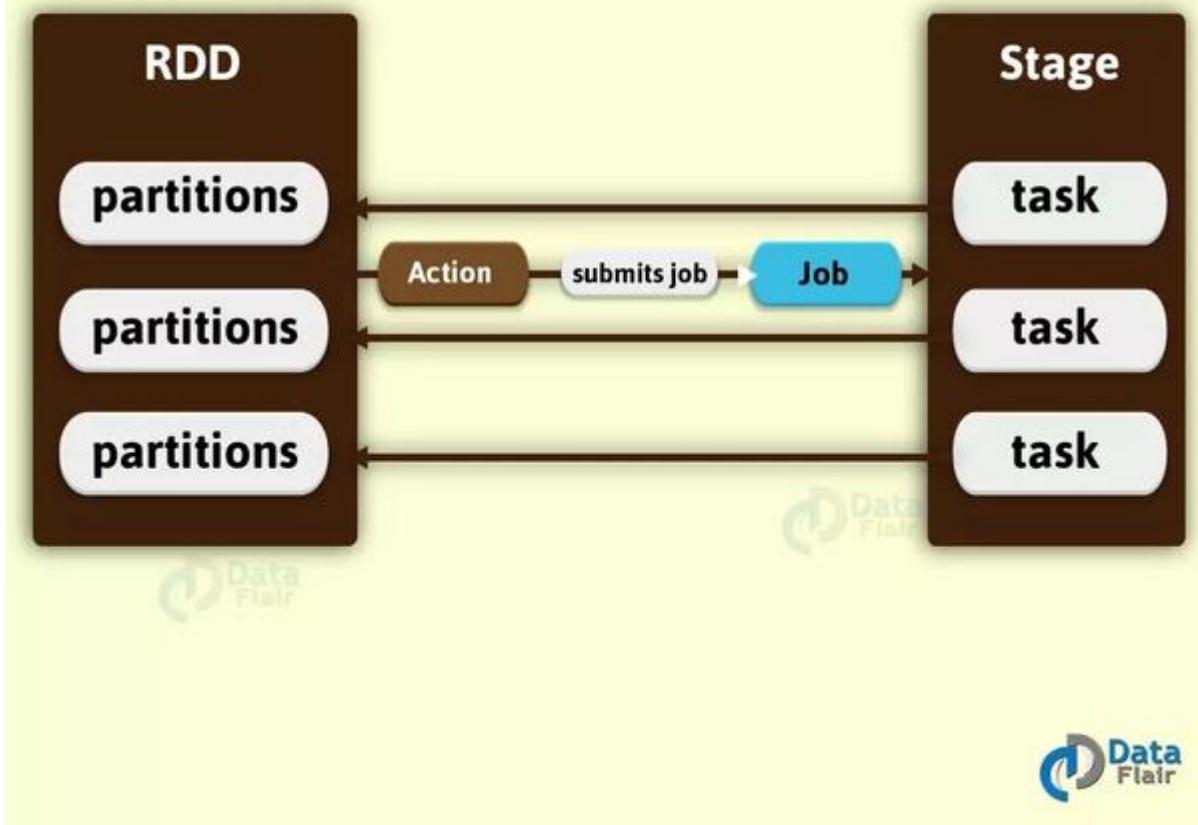
Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Active Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Thread Dump
driver	192.168.0.185:62559	Active	8	1 KB / 384.1 MB	0.0 B	8	0	0	9	9	2 s (72 ms)	0.0 B	460 B	460 B	Thread Dump

Showing 1 to 1 of 1 entries Search:

Previous 1 Next

Introduction to Stages in Spark

Tasks and submitting a job



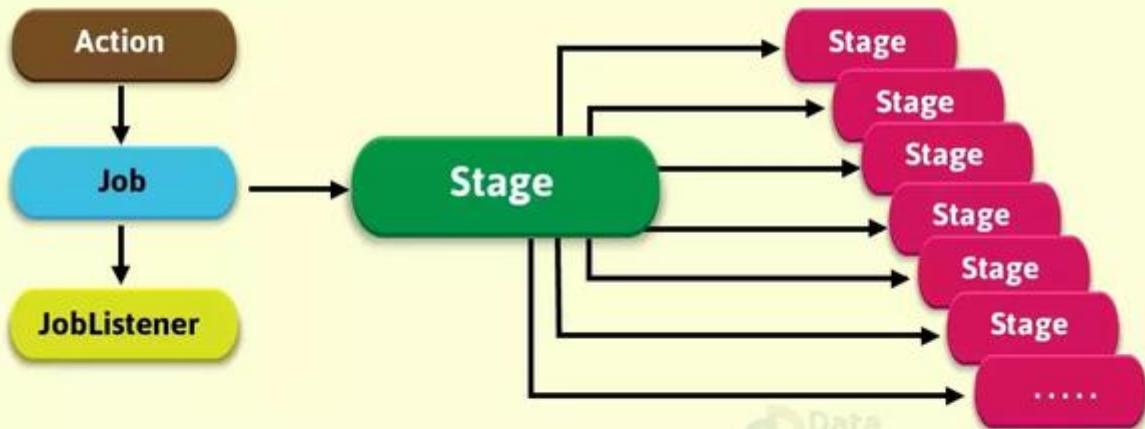
Submitting a job triggers execution of the stage and its parent Spark stages

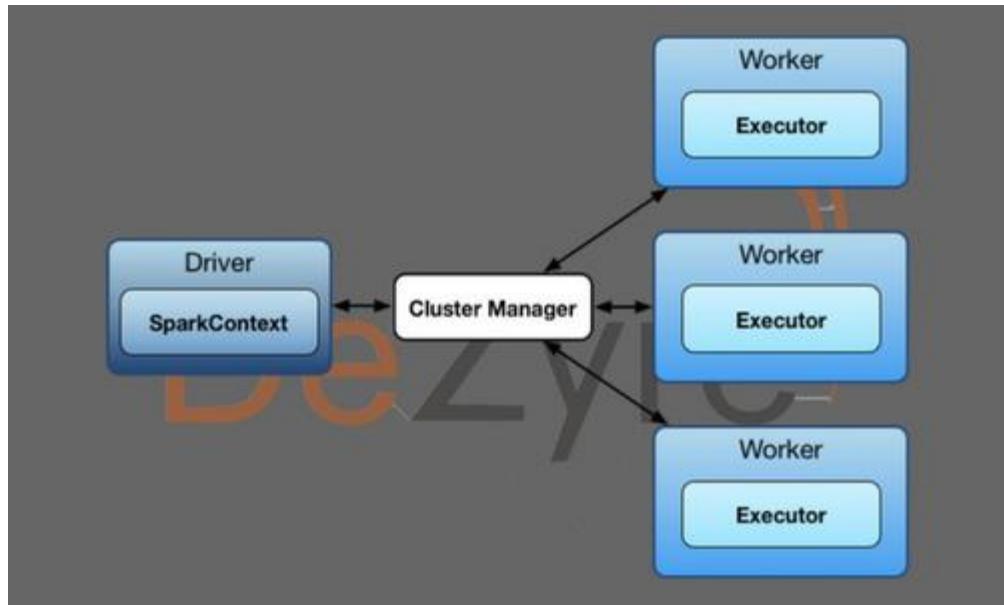
There are two types of Spark Stage

Basically, stages in Apache spark are two categories

- a. ShuffleMapStage in Spark
- b. ResultStage in Spark

Submitting a job triggers execution of the stage and its parent stages





Spark Architecture Overview

Apache Spark follows a master/slave architecture with two main daemons and a cluster manager –

1. Master Daemon – (Master/Driver Process)
2. Worker Daemon –(Slave Process)

A spark cluster has a single Master and any number of Slaves/Workers. The driver and the executors run their individual Java processes.

Role of Driver in Spark Architecture

Spark Driver – Master Node of a Spark Application

It is the central point and the entry point of the Spark Shell (Scala, Python, and R).

The driver program runs the main () function of the application and is the place where the Spark Context is created.

Spark Driver contains various components – DAGScheduler, TaskScheduler, BackendScheduler and BlockManager responsible for the translation of spark user code into actual spark jobs executed on the cluster.

- The driver program that runs on the master node of the spark cluster schedules the job execution and negotiates with the cluster manager.

- It translates the RDD's into the execution graph and splits the graph into multiple stages.
- Driver stores the metadata about all the Resilient Distributed Databases and their partitions.
- Cockpits of Jobs and Tasks Execution -Driver program converts a user application into smaller execution units known as tasks. Tasks are then executed by the executors i.e. the worker processes which run individual tasks.
- Driver exposes the information about the running spark application through a Web UI at port 4040.

Role of Executor in Spark Architecture

Executor is a distributed agent responsible for the execution of tasks.

Every spark applications has its own executor process. Executors usually run for the entire lifetime of a Spark application and this phenomenon is known as “Static Allocation of Executors”.

However, users can also opt for dynamic allocations of executors wherein they can add or remove spark executors dynamically to match with the overall workload.

- Executor performs all the data processing.
- Reads from and Writes data to external sources.
- Executor stores the computation results data in-memory, cache or on hard disk drives.
- Interacts with the storage systems.

Role of Cluster Manager in Spark Architecture

An external service responsible for acquiring resources on the spark cluster and allocating them to a spark job. There are 3 different types of cluster managers a Spark application can leverage for the allocation and deallocation of various physical resources such as memory for client spark jobs, CPU memory, etc. Hadoop YARN, Apache Mesos or the simple standalone spark cluster manager either of them can be launched on-premise or in the cloud for a spark application to run.

Choosing a cluster manager for any spark application depends on the goals of the application because all cluster managers provide different set of scheduling capabilities. To get started with apache spark, the standalone cluster manager is the easiest one to use when developing a new spark application.

Understanding the Run Time Architecture of a Spark Application

What happens when a Spark Job is submitted?

When a client submits a spark user application code, the driver implicitly converts the code containing transformations and actions into a logical directed acyclic graph (DAG).

At this stage, the driver program also performs certain optimizations like pipelining transformations and then it converts the logical

DAG into physical execution plan with set of stages. After creating the physical execution plan, it creates small physical execution units referred to as tasks under each stage.

Then tasks are bundled to be sent to the Spark Cluster.

The driver program then talks to the cluster manager and negotiates for resources. The cluster manager then launches executors on the worker nodes on behalf of the driver.

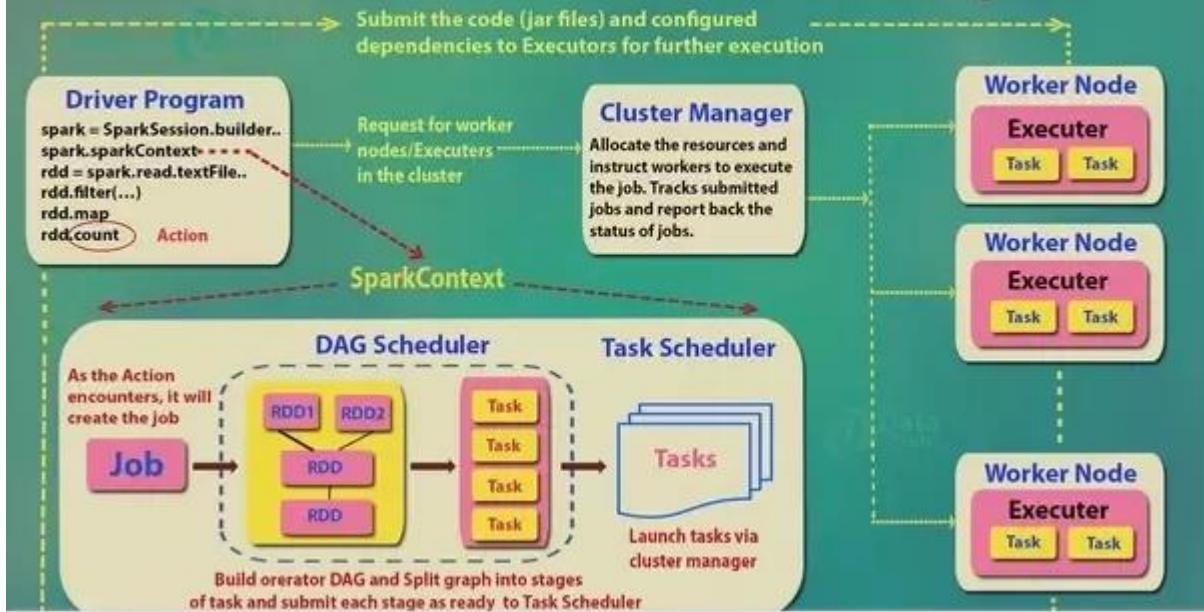
At this point the driver sends tasks to the cluster manager based on data placement. Before executors begin execution, they register themselves with the driver program so that the driver has holistic view of all the executors. Now executors start executing the various tasks assigned by the driver program.

At any point of time when the spark application is running, the driver program will monitor the set of executors that run. Driver program in the spark architecture also schedules future tasks based on data placement by tracking the location of cached data. When driver programs main () method exits or when it call the stop () method of the Spark Context, it will terminate all the executors and release the resources from the cluster manager.

The structure of a Spark program at higher level is - RDD's are created from the input data and new RDD's are derived from the existing RDD's using different transformations, after which an action is performed on the data. In any spark program, the DAG operations are created by default and whenever the driver runs the Spark DAG will be converted into a physical execution plan.



Internals of Job Execution In Spark



RDD:

At a high level, every Spark application consists of a *driver program* that runs the user's main function and executes various *parallel operations* on a cluster. The main abstraction Spark provides is a *resilient distributed dataset* (RDD), which is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel. RDDs are created by starting with a file in the Hadoop file system (or any other Hadoop-supported file system), or an existing Scala collection in the driver program, and transforming it. Users may also ask Spark to *persist* an RDD in memory, allowing it to be reused efficiently across parallel operations. Finally, RDDs automatically recover from node failures.

SparkContext—Entry Point to Spark Core

SparkContext (aka **Spark context**) is the heart of a Spark application.

You could also assume that a SparkContext instance *is* a Spark application.

Once a [SparkContext is created](#) you can use it to [create RDDs](#), [accumulators](#) and [broadcast variables](#), access Spark services and [run jobs](#) (until [SparkContext is stopped](#))

A Spark context is essentially a client of Spark's execution environment and acts as the *master of your Spark application*

Creating `SparkContext` Instance

You can create a `SparkContext` instance with or without creating a [SparkConf](#) object first.

Getting Existing or Creating New `SparkContext` — `getOrCreate` Methods

`getOrCreate(): SparkContext`

`getOrCreate(conf: SparkConf): SparkContext`

`getOrCreate` methods allow you to get the existing `SparkContext` or create a new one.

```
import org.apache.spark.SparkContext  
val sc = SparkContext.getOrCreate()
```

```
// Using an explicit SparkConf object  
import org.apache.spark.SparkConf  
val conf = new SparkConf()  
  .setMaster("local[*]")  
  .setAppName("SparkMe App")  
val sc = SparkContext.getOrCreate(conf)
```

`SparkContext()`

`SparkContext(conf: SparkConf)`

`SparkContext(master: String, appName: String, conf: SparkConf)`

`SparkContext(`

`master: String,`

`appName: String,`

`sparkHome: String = null,`

`jars: Seq[String] = Nil,`

`environment: Map[String, String] = Map()`

Units of Physical Execution

Jobs: Work required to compute RDD in runJob.

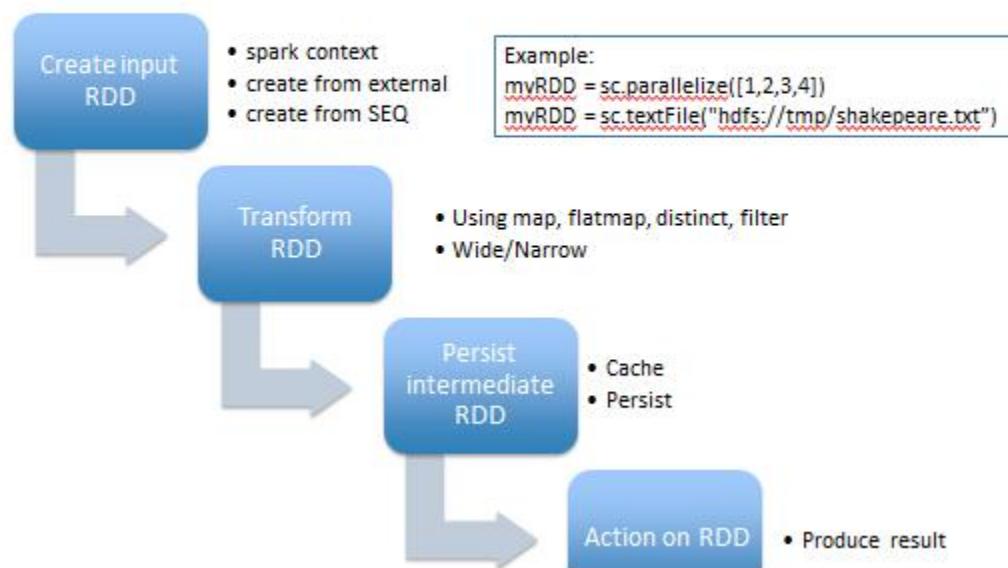
Stages: A wave of work within a job, corresponding to one or more pipelined RDD's.

Tasks: A unit of work within a stage, corresponding to one RDD partition.

Shuffle: The transfer of data between stages.

DAG(Directed Acyclic Graph) in [Apache Spark](#) is a set of Vertices and Edges, where *vertices* represent the RDDs and the *edges* represent the Operation to be applied on RDD. In Spark DAG, every edge directs from earlier to later in the sequence. On the calling of *Action*, the created DAG submits to DAG Scheduler which further splits the graph into the stages of the task

Spark Program Flow by RDD



The difference between `cache` and `persist` operations is purely syntactic. `cache` is a synonym of `persist` or `persist(MEMORY_ONLY)`, i.e. `cache` is merely `persist` with the default storage level `MEMORY_ONLY`

Caching or persistence are optimization techniques for (iterative and interactive) Spark computations. They help saving interim partial results so they can be reused in subsequent stages. These interim results as `RDDs` are thus kept in memory (default) or more solid storage like disk and/or replicated. `RDDs` can be cached using `cache` operation. They can also be persisted using `persist` operation.

`persist, cache`

These functions can be used to adjust the storage level of a `RDD`. When freeing up memory, Spark will use the storage level identifier to decide which partitions should be kept. The parameter less variants `persist()` and `cache()` are just abbreviations for `persist(StorageLevel.MEMORY_ONLY)`.

Warning: Once the storage level has been changed, it cannot be changed again!

Warning -Cache judiciously... see ([\(Why\) do we need to call cache or persist on a RDD](#))

Just because you can `cache` a `RDD` in memory doesn't mean you should blindly do so. Depending on how many times the dataset is accessed and the amount of work involved in doing so, recomputation can be faster than the price paid by the increased memory pressure.

It should go without saying that if you only read a dataset once there is no point in caching it, it will actually make your job slower. The size of cached datasets can be seen from the Spark Shell..

Listing Variants...

```
def cache(): RDD[T] def persist(): RDD[T] def persist(newLevel: StorageLevel):  
RDD[T]
```

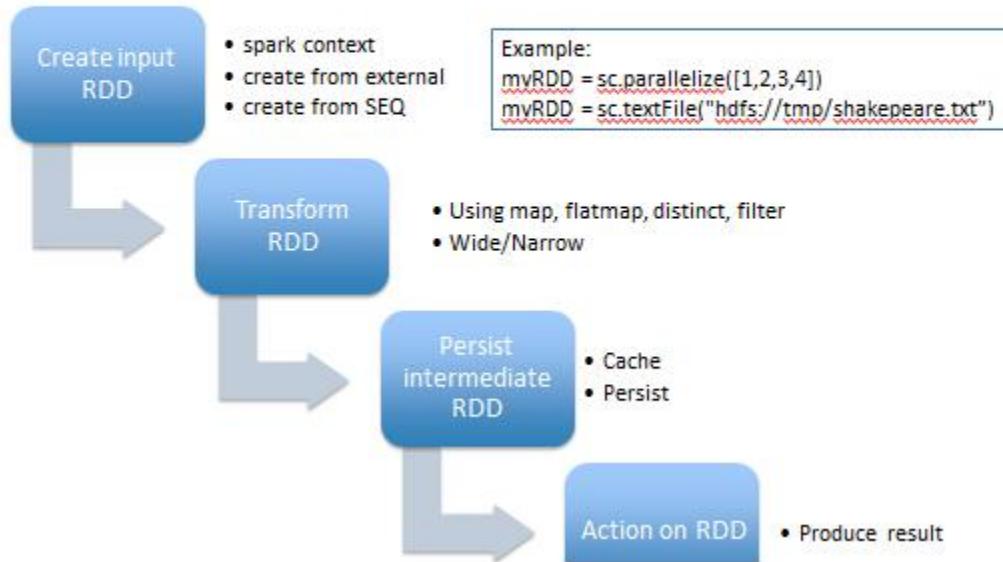
***See below example : ***

```

val c = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog", "Gnu", "Rat"), 2)
c.getStorageLevel res0: org.apache.spark.storage.StorageLevel =
StorageLevel(false, false, false, false, 1) c.cache c.getStorageLevel res2:
org.apache.spark.storage.StorageLevel = StorageLevel(false, true, false, true,
1)

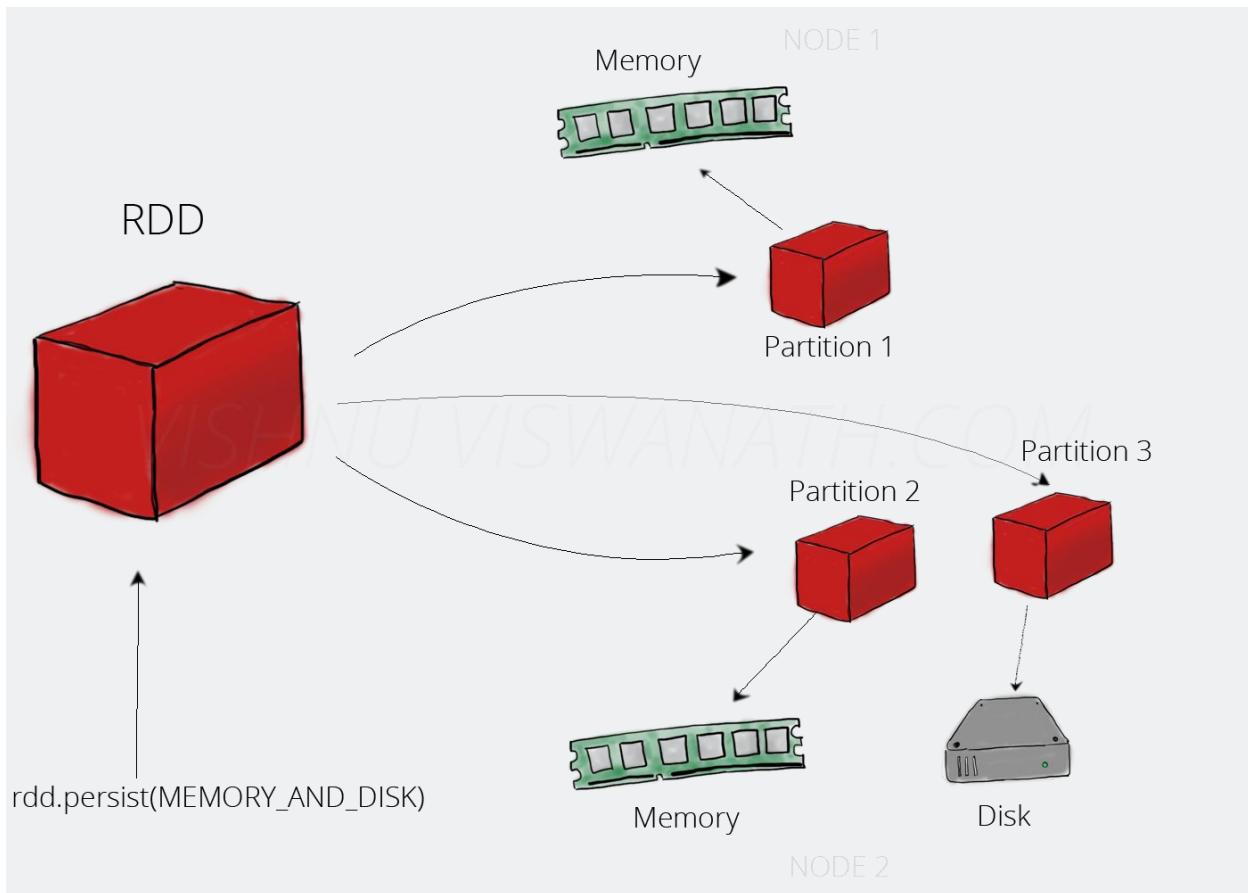
```

Spark Program Flow by RDD



The difference between `cache` and `persist` operations is purely syntactic. `cache` is a synonym of `persist` or `persist(MEMORY_ONLY)`, i.e. `cache` is merely `persist` with the default storage level `MEMORY_ONLY`.

Persist in memory and disk:



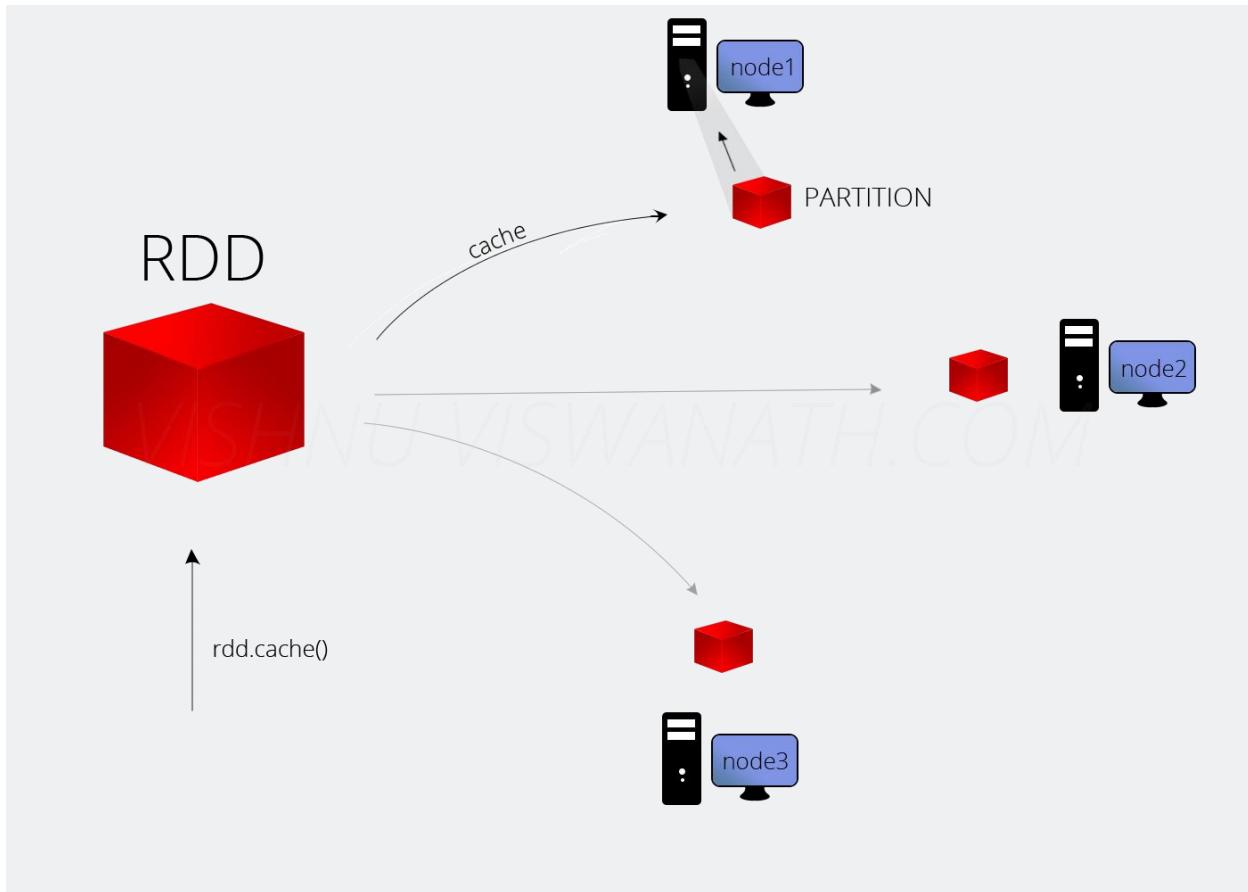
Spark gives 5 types of Storage level

- `MEMORY_ONLY`
- `MEMORY_ONLY_SER`
- `MEMORY_AND_DISK`
- `MEMORY_AND_DISK_SER`
- `DISK_ONLY`

`cache()` will use `MEMORY_ONLY`. If you want to use something else, use
`persist(StorageLevel.<*type*>)`

Cache

Caching can improve the performance of your application to a great extent.



1. Broadcast Variables
2. Accumulators

2.1. Broadcast Variables in Spark

Broadcast Variables despite shipping a copy of it with tasks. We can use them, for example, to give a copy of a large input dataset in an efficient manner to every node. In [Spark](#), by using efficient algorithms it is possible to distribute broadcast variables. It helps to reduce communication cost.

Spark can broadcast the common data automatically, needed by tasks within each stage. The data broadcasted this way then cached in serialized form and also deserialized before running each task. Hence, creating broadcast

variables explicitly is useful in some cases, like while tasks across multiple stages need the same data. While caching the data in the deserialized form is important.

It is also very important that no modification can take place on the object v after it is broadcast. It will help ensure that all nodes get the same value of the broadcast variable.

```
1. scala> val broadcastVar1 = sc.broadcast(Array(1, 2, 3))
2. broadcastVar1:
   org.apache.spark.broadcast.Broadcast[Array[Int]] =
   Broadcast(0)
3.
4. scala> broadcastVar1.value
5. res0: Array[Int] = Array(1, 2, 3)
```

2.2. Accumulators

The variables which are only “added” through a commutative and associative operation. Also, can efficiently support in parallel. We can use *Accumulators* to implement counters or sums. Spark natively supports programmers for new types and accumulators of numeric types.

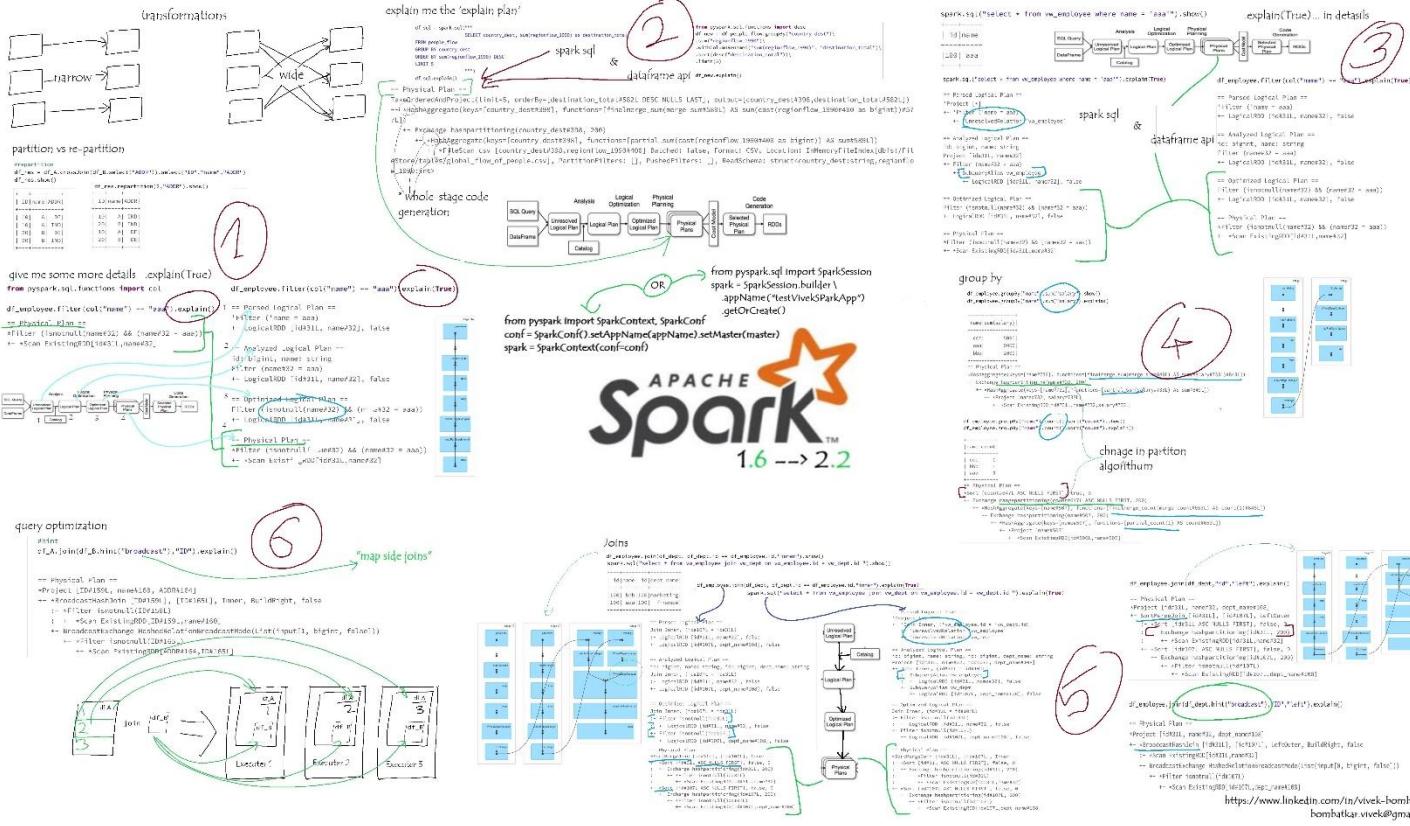
By calling `SparkContext.longAccumulator()`, we can create a numeric accumulator and by `SparkContext.doubleAccumulator()`, we can accumulate values of type long or double.

My Spark practice notes.

Learning is a continuous process. Though I am using Spark from quite a long time now, I never noted down my practice exercise. With this repo, I am documenting it!

I have used databricks free community cloude for this excercises,
link: <https://community.cloud.databricks.com/login.html>





spark_explain_plan

spark_explain_plan notebook

DAG and explain plan

<https://www.tutorialkart.com/apache-spark/dag-and-physical-execution-plan/>

***How Apache Spark builds a DAG and Physical Execution Plan ? ***

1. User submits a spark application to the Apache Spark.
 2. Driver is the module that takes in the application from Spark side.
 3. Driver identifies transformations and actions present in the spark application. These identifications are the tasks.
 4. Based on the flow of program, these tasks are arranged in a graph like structure with directed flow of execution from task to task forming no loops in the graph (also called DAG). DAG is pure logical.
 5. This logical DAG is converted to Physical Execution Plan. Physical Execution Plan contains stages.

6. Some of the subsequent tasks in DAG could be combined together in a single stage.
Based on the nature of transformations, Driver sets stage boundaries.
7. There are two transformations, namely
 - a. narrow transformations : Transformations like Map and Filter that does not require the data to be shuffled across the partitions.
 - b. wide transformations : Transformations like ReduceByKey that does require the data to be shuffled across the partitions.
8. Transformation that requires data shuffling between partitions, i.e., a wide transformation results in stage boundary.
9. DAG Scheduler creates a Physical Execution Plan from the logical DAG. Physical Execution Plan contains tasks and are bundled to be sent to nodes of cluster.

Catalyst optimizer

<http://data-informed.com/6-steps-to-get-top-performance-from-the-changes-in-spark-2-0/>

What is Catalyst? Catalyst is the name of Spark's integral query optimizer and execution planner for Dataset/DataFrame.

Catalyst is where most of the "magic" happens to improve the execution speed of your code. But in any complex system, "magic" is unfortunately not good enough to always guarantee optimal performance. Just as with relational databases, it is valuable to learn a bit about exactly how the optimizer works in order to understand its planning and tune your applications.

In particular, Catalyst can perform sophisticated refactors of complex queries. However, almost all of its optimizations are qualitative and rule-based rather than quantitative and statistics-based. For example, Spark knows how and when to do things like combine filters, or move filters before joins. Spark 2.0 even allows you to define, add, and test out your own additional optimization rules at runtime. [1][2]

On the other hand, Catalyst is not designed to perform many of the common optimizations that RDBMSs have performed for decades, and that takes some understanding and getting used to.

For example, Spark doesn't "own" any storage, so it does not build on-disk indexes, B-Trees, etc. (although its parquet file support, if used well, can get you some related features). Spark has been optimized for the volume, variety, etc. of big data – so, traditionally, it has not been designed to maintain and use statistics about a stable dataset. E.g., where an RDBMS might know that a specific filter will eliminate most

records, and apply it early in the query, Spark 2.0 does not know this fact and won't perform that optimization

Catalyst, the optimizer and Tungsten, the execution engine!

<https://db-blog.web.cern.ch/blog/luca-canali/2016-09-spark-20-performance-improvements-investigated-flame-graphs>

*** Note in particular the steps marked with (*), they are optimized with who-stage code generation

Code generation is the key The key to understand the improved performance is with the new features in Spark 2.0 for whole-stage code generation.

Deep dive into the new Tungsten execution engine

<https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html>

1. The explain() function in the expression below has been extended for whole-stage code generation. In the explain output, when an operator has a star around it (*), whole-stage code generation is enabled. In the following case, Range, Filter, and the two Aggregates are both running with whole-stage code generation. Exchange, however, does not implement whole-stage code generation because it is sending data across the network.

```
spark.range(1000).filter("id > 100").selectExpr("sum(id)").explain()

== Physical Plan ==
*Aggregate(functions=[sum(id#201L)])
+- Exchange SinglePartition, None
  +- *Aggregate(functions=[sum(id#201L)])
    +- *Filter (id#201L > 100)
      +- *Range 0, 1, 3, 1000, [id#201L]
```

2. Vectorization The idea here is that instead of processing data one row at a time, the engine batches multiples rows together in a columnar format, and each operator uses simple loops to iterate over data within a batch. Each next() call would thus return a batch of tuples, amortizing the cost of virtual function dispatches. These simple loops would also enable compilers and CPUs to execute more efficiently with the benefits mentioned earlier.

Catalyst Optimizer

<https://data-flair.training/blogs/spark-sql-optimization-catalyst-optimizer/>

1. Fundamentals of Catalyst Optimizer

In the depth, Catalyst contains the tree and the set of rules to manipulate the tree.

Trees

A tree is the main data type in the catalyst. A tree contains node object. For each node, there is a node

Rules

We can manipulate tree using rules. We can define rules as a function from one tree to another tree.

2.

a. Analysis - Spark SQL Optimization starts from relation to be computed. It is computed either from abstract syntax tree (AST) returned by SQL parser or dataframe object created using API.

b. Logical Optimization - In this phase of Spark SQL optimization, the standard rule-based optimization is applied to the logical plan. It includes constant folding, predicate pushdown, projection pruning and other rules.

c. In this phase, one or more physical plan is formed from the logical plan, using physical operator matches the Spark execution engine. And it selects the plan using the cost model.

d. Code Generation - It involves generating Java bytecode to run on each machine. Catalyst uses the special feature of Scala language, "Quasiquotes" to make code generation easier because it is very tough to build code generation engines.

cost based optimization

<https://databricks.com/blog/2017/08/31/cost-based-optimizer-in-apache-spark-2-2.html>

Query Benchmark and Analysis We took a non-intrusive approach while adding these cost-based optimizations to Spark by adding a global config spark.sql.cbo.enabled to enable/disable this feature. In Spark 2.2, this parameter is set to false by default.

1. At its core, Spark's Catalyst optimizer is a general library for representing query plans as trees and sequentially applying a number of optimization rules to manipulate them.
2. A majority of these optimization rules are based on heuristics, i.e., they only account for a query's structure and ignore the properties of the data being processed,

3. ANALYZE TABLE command

CBO relies on detailed statistics to optimize a query plan. To collect these statistics, users can issue these new SQL commands described below:

```
ANALYZE TABLE table_name COMPUTE STATISTICS
```

sigmod_spark_sql

http://people.csail.mit.edu/matei/papers/2015/sigmod_spark_sql.pdf

working-with-udfs-in-apache-spark

<https://blog.cloudera.com/blog/2017/02/working-with-udfs-in-apache-spark/>

- It's important to understand the performance implications of Apache Spark's UDF features. Python UDFs for example (such as our CTOF function) result in data being serialized between the executor JVM and the Python interpreter running the UDF logic – this significantly reduces performance as compared to UDF implementations in Java or Scala. Potential solutions to alleviate this serialization bottleneck include:
 - Accessing a Hive UDF from PySpark as discussed in the previous section. The Java UDF implementation is accessible directly by the executor JVM. Note again that this approach only provides access to the UDF from the Apache Spark's SQL query language. Making use of the approach also shown to access UDFs implemented in Java or Scala from PySpark, as we demonstrated using the previously defined Scala UDAF example.
 - Another important component of Spark SQL to be aware of is the Catalyst query optimizer. Its capabilities are expanding with every release and can often provide dramatic performance improvements to Spark SQL queries; however, arbitrary UDF implementation code may not be well understood by Catalyst (although future features[3] which analyze bytecode are being considered to address this). As such, using Apache Spark's built-in SQL query functions will often lead to the best performance and should be the first approach considered whenever introducing a UDF can be avoided

spark-functions-vs-udf-performance

<https://stackoverflow.com/questions/38296609/spark-functions-vs-udf-performance>

```

df_employee.show()
+---+---+
| id|name|
+---+---+
|100| aaa|
|120| bbb|
|150| ccc|
+---+---+

from pyspark.sql.functions import col
df_employee.filter(col("name") == "aaa").show()
+---+---+
| id|name|
+---+---+
|100| aaa|
+---+---+

```

SQL Query → DataFrame

1 Unresolved Logical Plan

2 Analysis

Logical Optimization

3 Optimized Logical Plan

Physical Planning

4 Physical Plans

Cost Model

Selected Physical Plan

Code Generation

RDDs

df_employee.filter(col("name") == "aaa").explain()

df_employee.filter(col("name") == "aaa").explain(True)

1 == Parsed Logical Plan ==
'Filter ('name = aaa)
+- LogicalRDD [id#31L, name#32], false

2 == Analyzed Logical Plan ==
id: bigint, name: string
Filter (name#32 = aaa)
+- LogicalRDD [id#31L, name#32], false

3 == Optimized Logical Plan ==
Filter (isnotnull(name#32) && (name#32 = aaa))
+- LogicalRDD [id#31L, name#32], false

4 == Physical Plan ==
*Filter (isnotnull(name#32) && (name#32 = aaa))
+- *Scan ExistingRDD[id#31L, name#32]

```

spark.sql("select * from vw_employee where name = 'aaa').show()
+---+---+
| id|name|
+---+---+
|100| aaa|
+---+---+

```

SQL Query → DataFrame

Catalog

1 Unresolved Logical Plan

Analysis

Logical Optimization

Physical Planning

2 Logical Plan

Optimized Logical Plan

Physical Plans

Cost Model

Selected Physical Plan

Code Generation

RDDs

df_employee.filter(col("name") == "aaa").explain(True)

df_employee.filter(col("name") == "aaa").explain()

== Parsed Logical Plan ==
'Project [*]
+- 'Filter ('name = aaa)
 +- 'UnresolvedRelation` vw_employee'

== Analyzed Logical Plan ==
id: bigint, name: string
Project [id#31L, name#32]
+- Filter (name#32 = aaa)
 +- SubqueryAlias vw_employee
 +- LogicalRDD [id#31L, name#32], false

== Optimized Logical Plan ==
Filter (isnotnull(name#32) && (name#32 = aaa))
+- LogicalRDD [id#31L, name#32], false

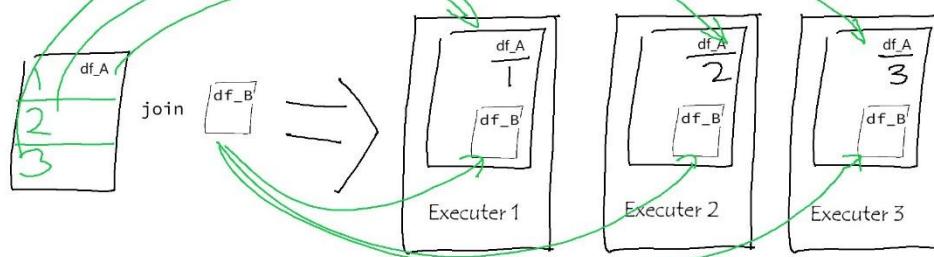
== Physical Plan ==
*Filter (isnotnull(name#32) && (name#32 = aaa))
+- *Scan ExistingRDD[id#31L, name#32]

```

#hint
df_A.join(df_B.hint("broadcast"), "ID").explain()

== Physical Plan ==
*Project [ID#159L, name#160, ADDR#164]
+- *BroadcastHashJoin [ID#159L], [ID#165L], Inner, BuildRight, false
  :- *Filter isnotnull(ID#159L)
  :  +- *Scan ExistingRDD[ID#159L, name#160]
+- BroadcastExchange HashedRelationBroadcastMode(List(input[1, bigint, false]))
  +- *Filter isnotnull(ID#165L)
    +- *Scan ExistingRDD[ADDR#164, ID#165L]

```



```

df_employee.join(df_dept, df_dept.id == df_employee.id, "inner").show()
spark.sql("select * from vw_employee join vw_dept on vw_employee.id = vw_dept.id ").show()

+---+-----+
| id|name| id|dept_name|
+---+-----+
|120| bbb|120|marketing|
|100| aaa|100| finance|
+---+-----+

== Parsed Logical Plan ==
Join Inner, (id#107L = id#31L)
:- LogicalRDD [id#31L, name#32], false
+- LogicalRDD [id#107L, dept_name#108], false

== Analyzed Logical Plan ==
id: bigint, name: string, id: bigint, dept_name: string
Join Inner, (id#107L = id#31L)
:- LogicalRDD [id#31L, name#32], false
+- LogicalRDD [id#107L, dept_name#108], false

== Optimized Logical Plan ==
Join Inner, (id#107L = id#31L)
:- Filter isnotnull(id#31L)
  +- LogicalRDD [id#31L, name#32], false
  +- Filter isnotnull(id#107L)
    +- LogicalRDD [id#107L, dept_name#108], false

== Physical Plan ==
*SortMergeJoin [id#31L], [id#107L], Inner
  :- *Sort [id#31L ASC NULLS FIRST], false, 0
  :  +- Exchange hashpartitioning(id#31L, 200)
  :     +- *Filter isnotnull(id#31L)
  :       +- *Scan ExistingRDD[id#31L, name#32]
  +- *Sort [id#107L ASC NULLS FIRST], false, 0
    +- Exchange hashpartitioning(id#107L, 200)
      +- *Filter isnotnull(id#107L)
        +- *Scan ExistingRDD[id#107L, dept_name#108]

== Parsed Logical Plan ==
'Project []
+- 'In Inner, ('vw_employee.id = 'vw_dept.id')
  :- 'UnresolvedRelation vw_employee'
    :- 'UnresolvedRelation vw_dept'

== Analyzed Logical Plan ==
id: bigint, name: string, id: bigint, dept_name: string
Project [id#31L, name#32, id#107L, dept_name#108]
+- Join Inner, (id#31L = id#107L)
  :- SubqueryAlias vw_employee
    :- +- LogicalRDD [id#31L, name#32], false
  +- SubqueryAlias vw_dept
    +- LogicalRDD [id#107L, dept_name#108], false

== Optimized Logical Plan ==
Join Inner, (id#31L = id#107L)
:- Filter isnotnull(id#31L)
  :- LogicalRDD [id#31L, name#32], false
+- Filter isnotnull(id#107L)
  +- LogicalRDD [id#107L, dept_name#108], false

== Physical Plan ==
*SortMergeJoin [id#31L], [id#107L], Inner
  :- *Sort [id#31L ASC NULLS FIRST], false, 0
  :  +- Exchange hashpartitioning(id#31L, 200)
  :     +- *Filter isnotnull(id#31L)
  :       +- *Scan ExistingRDD[id#31L, name#32]
  +- *Sort [id#107L ASC NULLS FIRST], false, 0
    +- Exchange hashpartitioning(id#107L, 200)
      +- *Filter isnotnull(id#107L)
        +- *Scan ExistingRDD[id#107L, dept_name#108]

```

```

df_employee.join(df_dept,"id","left").explain()

== Physical Plan ==
*Project [id#31L, name#32, dept_name#108]
+- SortMergeJoin [id#31L], [id#107L], LeftOuter
  :- *Sort [id#31L ASC NULLS FIRST], false, 0
  :+- Exchange hashpartitioning(id#31L, 200)
  :  +- *Scan ExistingRDD[id#31L,name#32]
+- *Sort [id#107L ASC NULLS FIRST], false, 0
  +- Exchange hashpartitioning(id#107L, 200)
    +- *Filter isnotnull(id#107L)
      +- *Scan ExistingRDD[id#107L,dept_name#108]

df_employee.join(df_dept.hint("broadcast"),"ID","left").explain()

== Physical Plan ==
*Project [id#31L, name#32, dept_name#108]
+- BroadcastHashJoin [id#31L], [id#107L], LeftOuter, BuildRight, false
  :- *Scan ExistingRDD[id#31L,name#32]
  +- BroadcastExchange HashedRelationBroadcastMode(List(input[0, bigint, false]))
    +- *Filter isnotnull(id#107L)
      +- *Scan ExistingRDD[id#107L,dept_name#108]

df_employee.groupBy("name").sum("salary").show()
df_employee.groupBy("name").sum("salary").explain()

+---+-----+
|name|sum(salary)|
+---+-----+
|ccc| 5000|
|aaa| 9000|
|bbb| 2000|
+---+-----+
== Physical Plan ==
*HashAggregate(keys=[name#732], functions=[finalmerge_sum(merge sum#849L) AS sum(salary#733L)#843L])
+- Exchange hashpartitioning(name#732, 200)
  +- *HashAggregate(keys=[name#732], functions=[partial_sum#733L] AS sum#849L)
    +- *Project [name#732, salary#733L]
      +- *Scan ExistingRDD[id#731L,name#732,salary#733L]

df_employee.groupBy("name").count().sort("count").show()
df_employee.groupBy("name").count().sort("count").explain()

+---+-----+
|name|count|
+---+-----+
|ccc| 1|
|bbb| 1|
|aaa| 3|
+---+-----+
== Physical Plan ==
*Sort [count#647L ASC NULLS FIRST], true, 0
+- Exchange rangepartitioning(count#647L ASC NULLS FIRST, 200)
  +- *HashAggregate(keys=[name#507], functions=[finalmerge_count(merge count#653L) AS count(1)#646L])
    +- Exchange hashpartitioning(name#507, 200)
      +- *HashAggregate(keys=[name#507], functions=[partial_count(1) AS count#653L])
        +- *Project [name#507]
          +- *Scan ExistingRDD[id#506L,name#507]

```

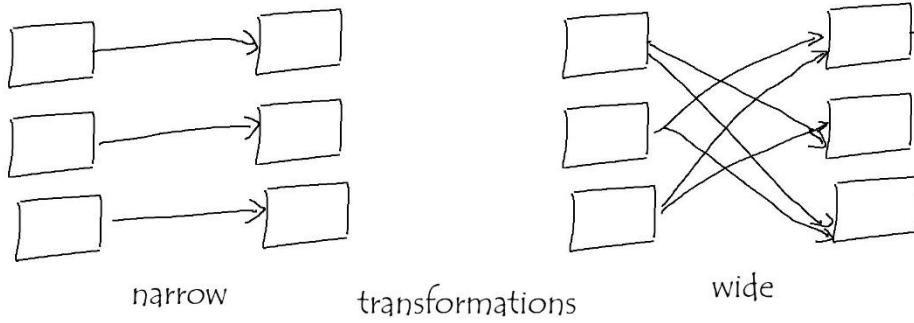
```

#repartition
df_res = df_A.crossJoin(df_B.select("ADDR")).select("ID", "name", "ADDR")
df_res.show()

+---+-----+
| ID|name|ADDR|
+---+-----+
| 10|   A|  DE|
| 10|   A| IND|
| 20|   B|  DE|
| 20|   B| IND|
+---+-----+

df_res.repartition(2,"ADDR").show()
+---+-----+
| ID|name|ADDR|
+---+-----+
| 10|   A| IND|
| 20|   B| IND|
| 10|   A|  DE|
| 20|   B|  DE|
+---+-----+
df_res.repartition("ADDR").show()
+---+-----+
| ID|name|ADDR|
+---+-----+
| 10|   A|  DE|
| 20|   B|  DE|
| 10|   A| IND|
| 20|   B| IND|
+---+-----+

```



spark-submit

<https://spark.apache.org/docs/2.2.0/submitting-applications.html> https://www.cloudera.com/documentation/enterprise/5-4-x/topics/cdh_ig_running_spark_on_yarn.html <https://jaceklaskowski.gitbooks.io/masterin>

g-apache-spark/yarn/ <https://blog.cloudera.com/blog/2014/05/apache-spark-resource-management-and-yarn-app-models/>

```
./bin/spark-submit \
--class <main-class> \
--master <master-url> \
--deploy-mode <deploy-mode> \
--conf <key>=<value> \
... # other options
<application-jar> \
[application-arguments]

--class: The entry point for your application (e.g. org.apache.spark.examples.SparkPi)
--master: The master URL for the cluster (e.g. spark://23.195.26.187:7077)
--deploy-mode: Whether to deploy your driver on the worker nodes (cluster) or locally as an external client (client) (default: client) +
--conf: Arbitrary Spark configuration property in key=value format. For values that contain spaces wrap "key=value" in quotes (as shown).
application-jar: Path to a bundled jar including your application and all dependencies. The URL must be globally visible inside of your cluster, for instance, an hdfs:// path or a file:// path that is present on all nodes.
application-arguments: Arguments passed to the main method of your main class, if any
```

java vs python code execution

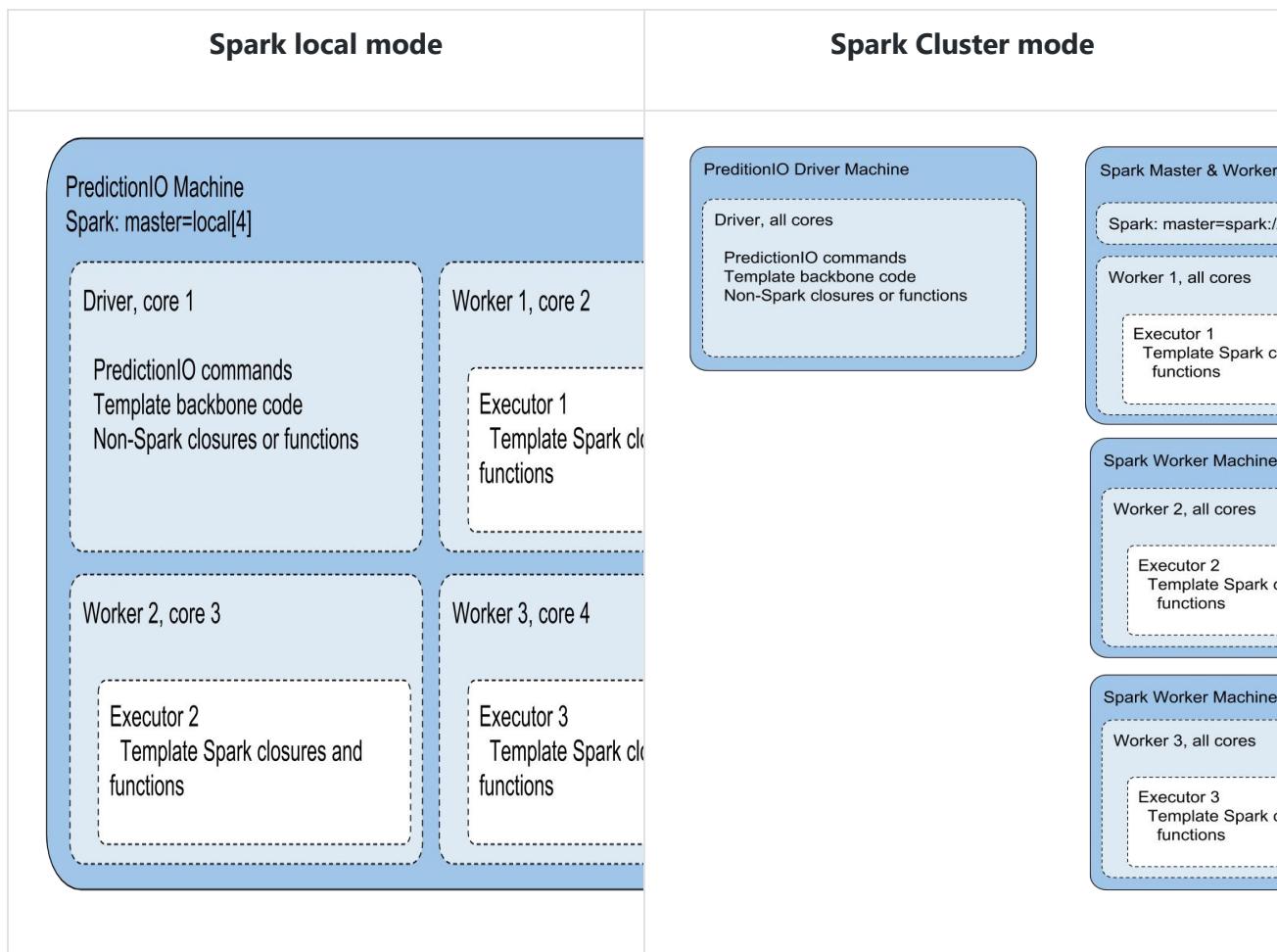
java class	python script		
--class 'class path of java main application'	(at the end of spark-submit) 'fully qualified path of the main python script'		
ex. --class com.abc.project1.Main	/opt/src/project1/module1/main.py 'pass the parameters'		
--jars 'assembly jar (or "uber" jar) containing your code and its dependencies, to be distributed with your application'	--py-files 'add .py, .zip or .egg files to be distributed with your application.'		
local	local[n]	local[n,f]	yarn
Run locally with one worker thread, no	Run locally with K worker threads , set this to the number of cores.	Run Spark locally with n worker threads and F	Connect to a YARN cluster in client or cluster mode depending on the value of --deploy-mode. The cluster location will be found based

local	local[n]	local[n,f]	yarn
parallelism	local[*] Run with as many worker threads as logical cores	maxFailures	on the HADOOP_CONF_DIR or YARN_CONF_DIR variable

YARN client vs cluster

Deploy modes are all about where the Spark driver runs.

YARN client	YARN cluster
driver runs on the host where the job is submitted	the driver runs in the ApplicationMaster on a cluster host chosen by YARN.
client that launches the application needs to be alive	client doesn't need to continue running for the entire lifetime of the application
Spark local mode	Spark Cluster mode



	YARN Cluster	YARN Client	Spark Standalone
Driver runs in:	Application Master	Client	Client
Who requests resources?	Application Master	Application Master	Client
Who starts executor processes?	YARN NodeManager	YARN NodeManager	Spark Slave
Persistent services	YARN ResourceManager and NodeManagers	YARN ResourceManager and NodeManagers	Spark Master and Workers
Supports Spark Shell?	No	Yes	Yes

Drivers and Executors

IMP Concepts	
Application	single job, a sequence of jobs, a long-running service issuing new commands as needed or an interactive exploration session.
Spark Driver	driver is the process running the spark context. This driver is responsible for converting the application to a directed graph of individual steps to execute on the cluster. There is one driver per application.
Spark Application Master	responsible for negotiating resource requests made by the driver with YARN and finding a suitable set of hosts/containers in which to run the Spark applications. There is one Application Master per application.
Spark Executor	A single JVM instance on a node that serves a single Spark application. An executor runs multiple tasks over its lifetime, and multiple tasks concurrently. A node may have several Spark executors and there are many nodes running Spark Executors for each client application.
Spark Task	represents a unit of work on a partition of a distributed dataset.

Dataframe operation on multiple columns

<https://medium.com/@mrpowers/performing-operations-on-multiple-columns-in-a-pyspark-dataframe-36e97896c378>

'Parsed Logical Plan' --> 'Analyzed Logical Plan' --> 'Optimized Logical Plan' --> 'Physical Plan'

Spark is smart enough to optimized (in Physical Plan) the multiple operation done in for kind of loop on dataframe

Below 2 code snipped will produce similler Physical Plan

```
for col in data_frame.columns:  
    df_res= data_frame.withColumn() \  
        .withColumn()
```

```
df_res= data_frame.select(*(when(col(c) ... ,...).otherwise(col(c)).alias(c) for c in data_frame.columns ))
```

Spark job monitoring

<https://databricks.com/blog/2015/06/22/understanding-your-spark-application-through-visualization.html>

Spark History Server web UI

a. Event timeline of spark events

The ability to view Spark events in a timeline is useful for identifying the bottlenecks in an application.

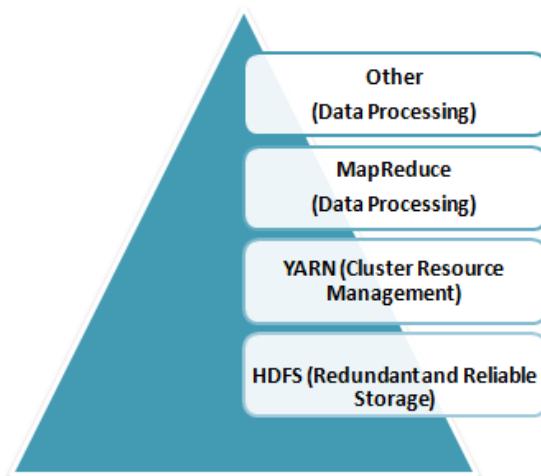
- Event timeline available in three levels
 - across all jobs
 - within one job
 - within one stage.

b. DAG

Explain the core components of Hadoop.

Answer: Hadoop is an open source framework that is meant for storage and processing of big data in a distributed manner. The core components of Hadoop are –

- **HDFS (Hadoop Distributed File System)** – HDFS is the basic storage system of Hadoop. The large data files running on a cluster of commodity hardware are stored in HDFS. It can store data in a reliable manner even when hardware fails.



- **Hadoop MapReduce** – MapReduce is the Hadoop layer that is responsible for data processing. It writes an application to process unstructured and structured data stored in HDFS. It is responsible for the parallel processing of high volume of data by dividing data into independent tasks. The processing is done in two phases Map and Reduce. The Map is the first phase of processing that specifies complex logic code and the Reduce is the second phase of processing that specifies light-weight operations.
- **YARN** – The processing framework in Hadoop is YARN. It is used for resource management and provides multiple data processing engines i.e. data science, real-time streaming, and batch processing.

Define respective components of HDFS and YARN

The two main components of HDFS are-

- Name Node – This is the master node for processing metadata information for data blocks within the HDFS
- Data Node/Slave node – This is the node which acts as slave node to store the data, for processing and use by the Name Node

The two main components of YARN are–

- Resource Manager– This component receives processing requests and accordingly allocates to respective Node Managers depending on processing needs.
- Node Manager– It executes tasks on each single Data Node

Write the command used to copy data from the local system onto HDFS?

The command used for copying data from the Local system to HDFS is:

hadoop fs –copyFromLocal [source][destination]

What is partitioning in Hive?

In general partitioning in Hive is a logical division of tables into related columns such as date, city, and department based on the values of partitioned columns. Then these partitions are subdivided into buckets so that they provide extra structure to the data that may be used for more efficient querying.

Bucketing Hive

Partitions are sub-divided into **buckets**, to provide extra structure to the data that may be used for more efficient querying. Bucketing works based on the value of hash function of some column of a table.

OLTP

Online transaction processing (OLTP) captures, stores, and processes data from transactions in real time.

OLAP

Online analytical processing (OLAP) uses complex queries to analyze aggregated historical data from OLTP systems.

The basic difference between OLTP and OLAP is that OLTP is an online database modifying system, whereas, OLAP is an online database query system

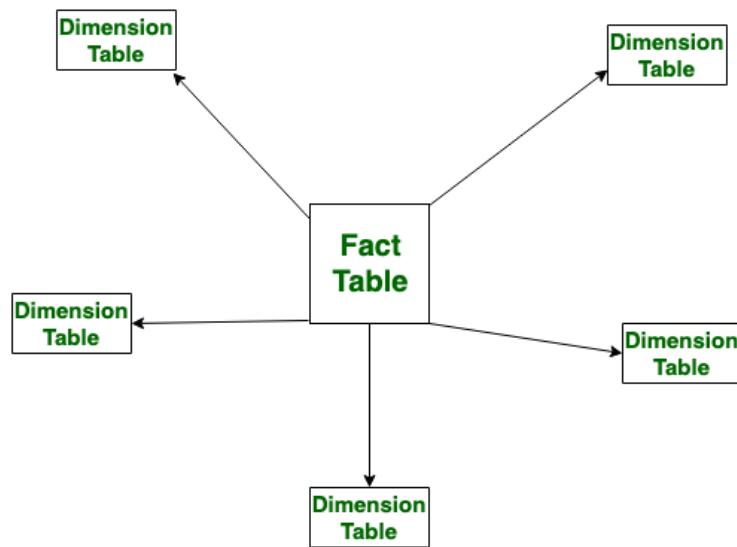
ONLINE TRANSACTION PROCESSING	ONLINE ANALYTICAL PROCESSING
Handles recent operational data	Handles all historical data
Size is smaller, typically ranging from 100 Mb to 10 Gb	Size is larger, typically ranging from 1 Tb to 100 Pb
Goal is to perform day-to-day operations	Goal is to make decisions from large data sources
Uses simple queries	Uses complex queries
Faster processing speeds	Slower processing speeds
Requires read/write operations	Requires only read operations

Fact and Dimension Table

A fact table holds the data to be analyzed, and a dimension table stores data about the ways in which the data in the fact table can be analyzed.

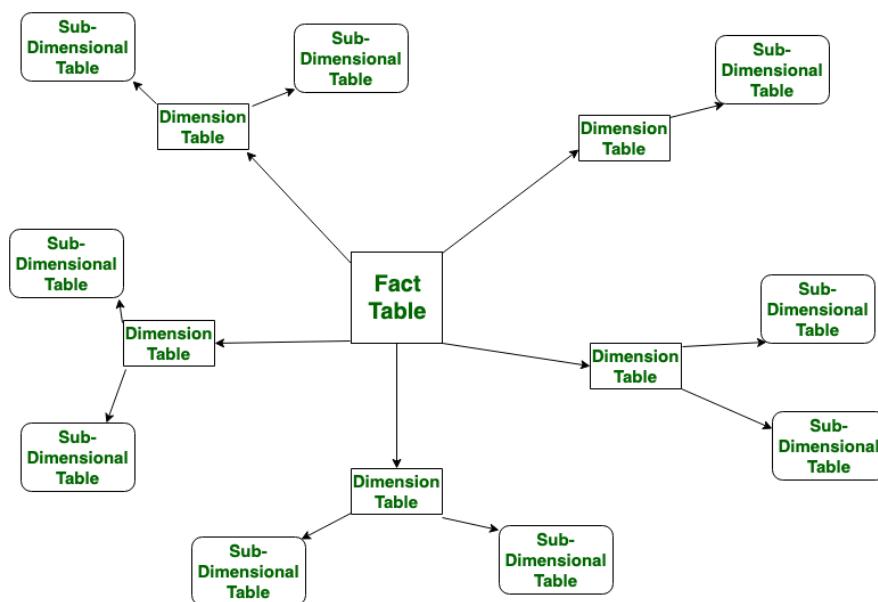
Star Schema

Star schema is the type of multidimensional model which is used for data warehouse. In star schema, the fact tables and the dimension tables are contained. In this schema fewer foreign-key join is used. This schema forms a star with fact table and dimension tables



Snowflake Schema

Snowflake Schema is also the type of multidimensional model which is used for data warehouse. In snowflake schema, the fact tables, dimension tables as well as sub dimension tables are contained. This schema forms a snowflake with fact tables, dimension tables as well as sub-dimension tables.



S.NO	Star Schema	Snowflake Schema
1.	In star schema, The fact tables and the dimension tables are contained.	While in snowflake schema, The fact tables, dimension tables as well as sub dimension tables are contained.
2.	Star schema is a top-down model.	While it is a bottom-up model.
3.	In star schema, Normalization is not used.	While in this, Both normalization and demoralization are used.
4.	It has less number of foreign keys.	While it has more number of foreign keys.
5.	It has high data redundancy.	While it has low data redundancy.

What is SCD in data warehouse?

A Slowly Changing Dimension (SCD) is a **dimension that stores and manages both current and historical data over time in a data warehouse**

- Type 1 – This model involves overwriting the old current value with the new current value. Overwrite the changes
- Type 2 – The current and the historical records are kept and maintained in the same file or table. History will be added as a new row.
- Type 3 – The current data and historical data are kept in the same record. History will be added as a new column.

File Formats:

AVRO is a **row-based storage format**. Writing operations in AVRO are better than in PARQUET

PARQUET is a **columnar-based storage format**. PARQUET is much better for analytical querying, i.e., reads and querying are much more efficient than writing. Parquet is more efficient in terms of storage and performance

ORC is Optimized Row Columnar, and it is a free and open-source columnar storage format designed for Hadoop workloads.

ORC supports ACID properties **ORC reduces the size of the original data up to 75%**. As a result the speed of data processing also increases and shows better performance than Text

CSV is a **comma-separated values file**, which allows data to be saved in a tabular format.

JSON file is a file that stores simple data structures and objects in JavaScript Object Notation (**JSON**) format, which is a standard data interchange format. It is primarily used for transmitting data between a web application and a server

Spark:

Open source distributed computing engine, you can store and process huge volume of data 100 time faster than hadoop. Its uses in memory and parallel processing which makes spark faster

Spark architecture:

Master / slave nodes, it contains three layers driver cluster manager and worker node, between driver and worker layer is cluster manager. All the components are loosely coupled within the boundary

In **master node**, you have the *driver program*, which drives our application. The code you are writing behaves as a driver program or if you are using the interactive shell, the shell acts as the driver program. Inside the driver program, the first thing you do is, you *create a **Spark Context***. Assume that the Spark context is a gateway to all the Spark functionalities. It is similar to our database connection.

How does Spark work?

STEP 1: The client submits spark user application code. When an application code is submitted, the driver implicitly converts user code that contains transformations and actions into a logically *directed acyclic graph* called **DAG**. At this stage, it also performs optimizations such as pipelining transformations.

STEP 2: After that, it converts the logical graph called DAG into physical execution plan with many stages. After converting into a physical execution plan, it creates physical execution units called tasks under each stage. Then the tasks are bundled and sent to the cluster.

STEP 3: Now the driver talks to the cluster manager and negotiates the resources. Cluster manager launches executors in worker nodes on behalf of the driver. At this point, the driver will send the tasks to the executors based on

data placement. When executors start, they register themselves with drivers. So, the driver will have a complete view of executors that are executing the task

STEP 4: During the course of execution of tasks, driver program will monitor the set of executors that runs. Driver node also schedules future tasks based on data placement.

What is meant by Lazy evaluation in spark?

Execution will not start until action is called. That means Data is not loaded until the point where action is called which helps spark engine to have better optimization

Terminologies of Spark

Driver and worker Process:

These are nothing but JVM process. Within one worker node, there could be multiple executors. Each executor runs its own JVM process.

Application:

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 an application is submitted to Spark, driver process converts the code into job.

Stage:

Jobs are divided into stages. If the application code demands shuffling the data across nodes, new stage is created. Number of stages is determined by number of shuffling operations. Join is example of shuffling operation

Tasks:

Stages are further divided into multiple tasks. In a stage, all the tasks would execute same logic. Each task will process 1 partition at a time. So number of partition in the distributed cluster determines the number of tasks in each stage

Transformation:

Transforms the input RDD and creates new RDD. Until action is called, transformations are evaluated lazily. Some of the transformations are

(map,filter,flatMap,mapPartitions,mapPartitionsWithIndex,groupBy,sortBy,union,intersection,subtract,distinct,Cartesian,zip,sample,randomSplit,keyBy,coalesce,repartition)

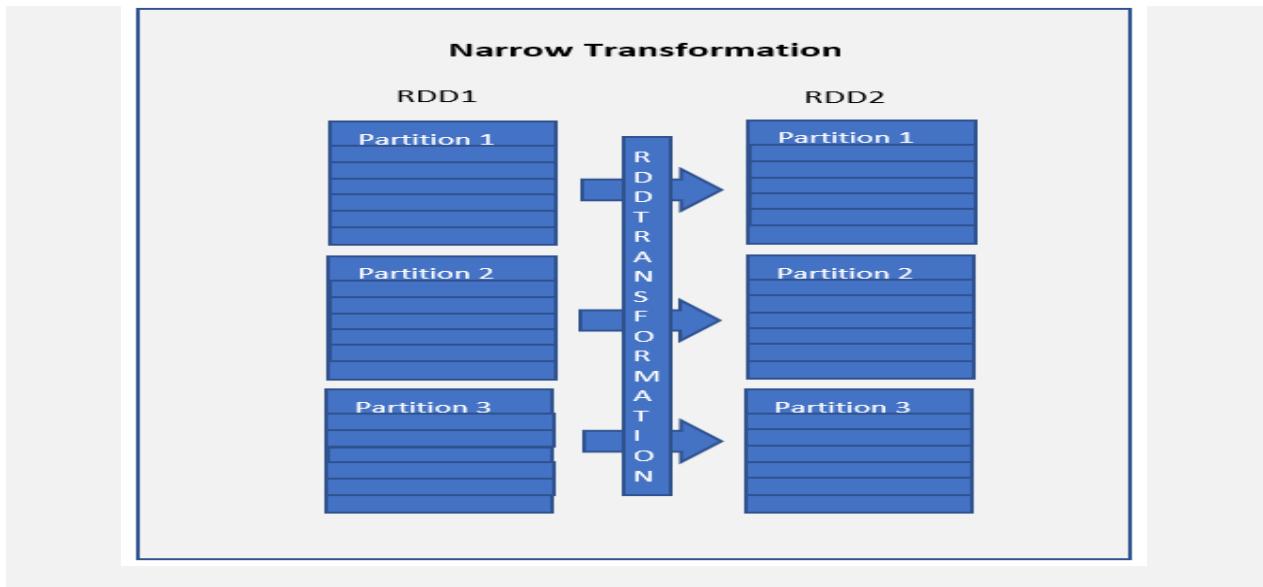
Two types of transformations in SPARK:

- Wide Transformations
- Narrow Transformations

Narrow Transformations:

These types of transformations convert each input partition to only one output partition. When each partition at the parent RDD is used by at most one partition of the child RDD or when each partition from child produced or dependent on single parent RDD.

- This kind of transformation is basically fast.
- Does not require any data shuffling over the cluster network or no data movement.
- Operation of map () and filter () belongs to this transformation.

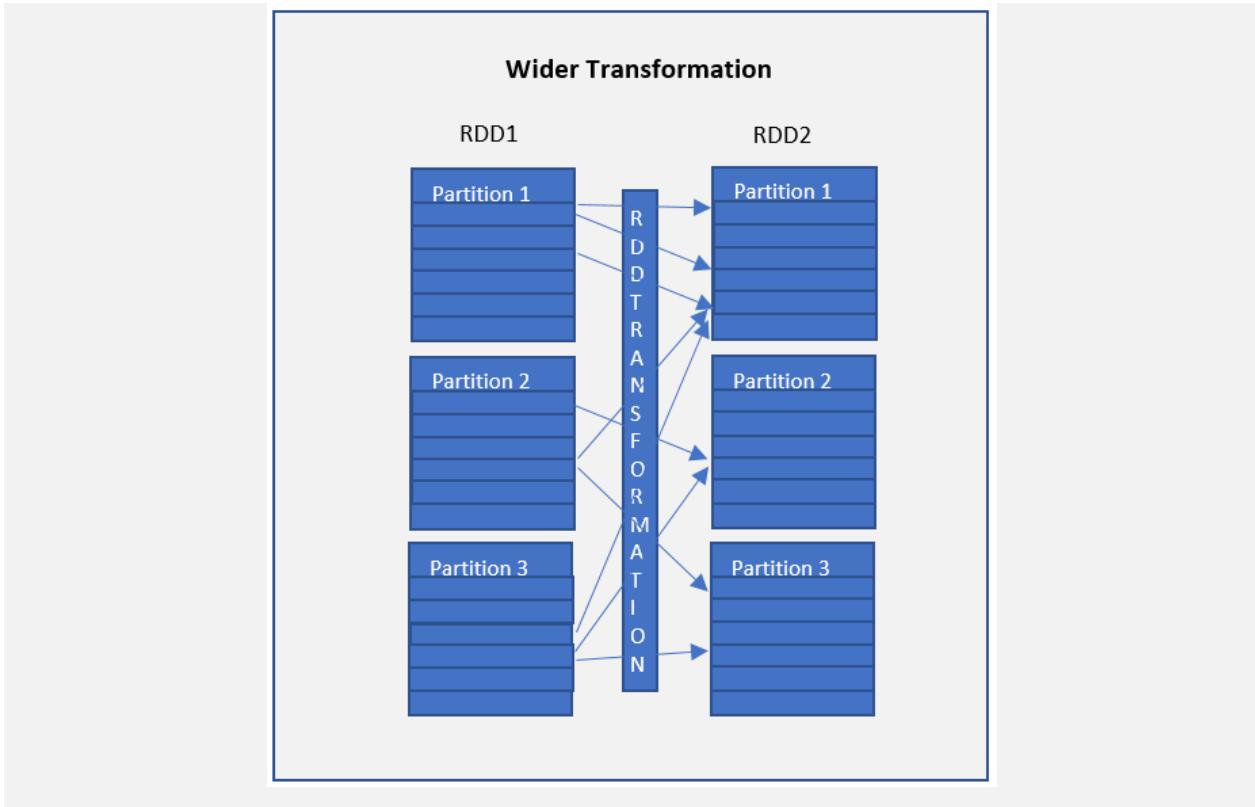


Wide Transformations:

This type of transformation will have input partitions contributing to many output partitions. When each partition at the parent RDD is used by multiple partitions of the child RDD or when each partition from child produced or dependent on multiple parents RDD.

- Slow as compare to narrow dependencies speed might be significantly affected as it might be required to shuffle data around different nodes when creating new partitions.

- Might Require data shuffling over the cluster network or no data movement.
- Functions such as `groupByKey()`, `aggregateByKey()`, `aggregate()`, `join()`, `repartition()` are some examples of wider transformations.



When working with Spark, it is always good to keep in mind all operations or transformations which might require data shuffling and hence slow down the process. Try to optimize and reduce the usage of wide dependencies as much as you can.

Pesris

:

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.

Some of the actions are

(reduce, collect, aggregate, foldfirst, take, foreach, top, treeAggregate, treeReduce, Partitioncount, takeSample, max, min, sum, histogram, mean, variance, Save)

RDD:

RDDs expand to Resilient Distributed Datasets. These are the elements that are used for running and operating on multiple nodes to perform parallel processing on a cluster. Since RDDs are suited for parallel processing, they are immutable elements. This means that once we create RDD, we cannot modify it. RDDs are also fault-tolerant which means that whenever failure happens, they can be recovered automatically. Multiple operations can be performed on RDDs to perform a certain task. The operations can be of 2 types:

- **Resilient:** Fault tolerant and is capable of quickly recover from failure
- **Distributed:** Distributed data among the multiple nodes in a cluster
- **FFDataset:** Collection of partitioned data with values

What are the advantages of PySpark RDD?

PySpark RDDs have the following advantages:

- **In-Memory Processing:** Spark's RDD helps in loading data from the disk to the memory. The RDDs can even be persisted in the memory for reusing the computations.
- **Immutability:** The RDDs are immutable which means that once created, they cannot be modified. While applying any transformation operations on the RDDs, a new RDD would be created.
- **Fault Tolerance:** The RDDs are fault-tolerant. This means that whenever an operation fails, the data gets automatically reloaded from other available partitions. This results in seamless execution of the Spark applications.
- **Lazy Evolution:** The Spark transformation operations are not performed as soon as they are encountered. The operations would be stored in the DAG and are evaluated once it finds the first RDD action.
- **Partitioning:** Whenever RDD is created from any data, the elements in the RDD are partitioned to the cores available by default

Executor:

Each worker node consist of many executors.it can be configure by spark setting

Core:

Each executor can consist of multiple cores. This is configurable by spark settings. Each core can process on task at a time

Important:

A Spark application can have many jobs. A job can have many stages. A stage can have many tasks. A task executes a series of instructions.

Different betyouen RDD vs. Dataframes vs. Datasets

	RDDs	Dataframes	Datasets
Data Representation	RDD is a distributed collection of data elements without any schema.	It is also the distributed collection organized into the named columns	It is an extension of Dataframes with more features like type-safety and object-oriented interface.
Optimization	No in-built optimization engine for RDDs. Developers need to write the optimized code themselves.	It uses a catalyst optimizer for optimization.	It also uses a catalyst optimizer for optimization purposes.
Projection of Schema	Here, we need to define the schema manually.	It will automatically find out the schema of the dataset.	It 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.	It provides 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.

What is DAG and how it works in Fault Tolerance?

DAG (Directed Acyclic Graph) in Apache Spark is an alternative to the MapReduce. It is a programming style used in distributed systems. In MapReduce, you just have two functions (map and reduce), while DAG has multiple levels that form a tree structure. Hence, DAG execution is faster than MapReduce because intermediate results do not write to disk.

Advantage of DAG:

- The lost RDD can recover using the Directed Acyclic Graph.
- Map Reduce has just two queries the map, and reduce but in DAG you have multiple levels. So to execute SQL query, DAG is more flexible.
- DAG helps to achieve fault tolerance. Thus you can recover the lost data.
- It can do a better global optimization than a system like Hadoop Map Reduce.

How spark achieves fault tolerance?

- Spark provides fault tolerance through lineage graph. Lineage graph keeps the track of the transformations to be executed once the action has been called. It helps in recomputing any missing RDD in case of any node failure.
- Immutability of RDD and lineage graph helps in recreating missing RDD in case of failure making it fault tolerant

```
from pyspark.sql import SparkSession  
spark=SparkSession.builder.appName("Name").getOrCreate()
```

Which one do you prefer? Either groupByKey() or ReduseByKey?

groupByKey

The groupByKey can cause out of disk problems as data is sent over the network and collected on the reduced workers. You can see the below example.

```
sparkContext.textFile("hdfs://")  
    .flatMap(line => line.split(" "))  
    .map(word => (word, 1))  
    .groupByKey()  
    .map((x, y) => (x, sum(y)))
```

reducebykey

Whereas in reducebykey, Data are combined at each partition, only one output for one key at each partition to send over the network. reduceByKey required combining all our values into another value with the exact same type.

```
sparkContext.textFile("hdfs://")
    .flatMap(line => line.split(" "))
    .map(word => (word,1))
    .reduceByKey((x,y)=> (x+y))
```

Broadcast Variable & Accumulator Variable

The main difference between these two is Broadcast variable is primarily used for reading some data across worker node .Accumulator is used for writing some data across worker node.

Broadcast Variable

Broadcast variables are used to save the copy of data across all nodes. This variable is cached on all the machines and not sent on machines with tasks. The following code block has the details of a Broadcast class for PySpark.

```
class pyspark.Broadcast():
    sc = None,
    value = None,
    pickle_registry = None,
    path = None
)|
```

Rdd=sc.broadcast([“raju”,”tharan”])

To submit broadcast variable:

Spark-submit broadcast.py

OUTPUT:

Stored data →[‘raju’,’tharun’]

Accumulator

Accumulator variables are used for aggregating the information through associative and commutative operations. For example, you can use an accumulator for a sum operation or counters (in MapReduce).

```
from pyspark import SparkContext
sc = SparkContext("local", "Accumulator app")
num = sc.accumulator(10)
def f(x):
    global num
    num+=x
rdd = sc.parallelize([20,30,40,50])
rdd.foreach(f)
final = num.value
print "Accumulated value is -> %i" %(final)
```

Num=sc.accumulator(10)

Rdd=sc.parallelize([20,30,40,50])

When you submit accumulator variable:

Spark-submit accumulator.py

OUTPUT:

Accumulated value is: 150

Broadcasting Join

Broadcast Join is a type of join operation in PySpark that is used to join data frames by broadcasting it in PySpark application. This join can be used for the data frame that is smaller in size which can be broadcasted with the PySpark application to be used further. The data is sent and broadcasted to all nodes in

the cluster. This is an optimal and cost-efficient join model that can be used in the PySpark application.

What is cluster mode and client mode in Spark?

Cluster mode puts the Spark driver in an application master process managed by YARN on the cluster. In client mode, the driver can run in the client process without an application master, which simplifies deployment and makes the driver easier to test.

What is Spark context in Spark?

A **SparkContext** represents the connection to a Spark cluster, and can be used to create RDDs, accumulators and broadcast variables on that cluster. Only one SparkContext should be active per JVM. You must stop() the active SparkContext before creating a new one.

What are PySpark serializers?

The serialization process is used to conduct performance tuning on Spark. The data sent or received over the network to the disk or memory should be persisted. PySpark supports serializers for this purpose. It supports two types of serializers, they are:

- **PickleSerializer:** This serializes objects using Python's PickleSerializer (`class pyspark.PickleSerializer`). This supports almost every Python object.
- **MarshalSerializer:** This performs serialization of objects. We can use it by using `class pyspark.MarshalSerializer`. This serializer is faster than the PickleSerializer but it supports only limited types.

Consider an example of serialization which makes use of MarshalSerializer:

```
# --serializing.py----  
from pyspark.context import SparkContext
```

```
from pyspark.serializers import MarshalSerializer
sc = SparkContext("local", "Marshal Serialization", serializer =
MarshalSerializer()) #Initialize spark context and serializer
print(sc.parallelize(list(range(1000))).map(lambda x: 3 * x).take(5))
sc.stop()
```

When we run the file using the command:

```
$SPARK_HOME/bin/spark-submit serializing.py
```

The output of the code would be the list of size 5 of numbers multiplied by 3:

```
[0, 3, 6, 9, 12]
```

What is the difference between union and union all?

Union and Union All are similar except that Union only selects the rows specified in the query, while Union All selects all the rows including duplicates (repeated values) from both queries

What is a Spark session?

SparkSession is the entry point to Spark SQL. It is one of the very first objects we need to create while developing a Spark SQL application. we **create a SparkSession using the SparkSession. builder method** (that gives you access to Builder API that you use to configure the session).

cache () and persist ()?

Both persist () and cache () are the Spark optimization technique, used to store the data, but only difference is cache () method by default stores the data in-memory (MEMORY_ONLY) whereas in persist () method developer can define the storage level to in-memory or in-disk.

In cache() - default storage level is **MEMORY_ONLY**

Persist() -default storage level is **MEMORY_AND_DISK** .

Class Method

Static Method

We have many option of storage levels that can be used with persist()

- **MEMORY_ONLY,**
- **MEMORY_AND_DISK,**
- **MEMORY_ONLY_SERIALIZED**
- **MEMORY_AND_DISK_SER,**
- **DISK_ONLY,**
- **MEMORY_ONLY_2,**
- **MEMORY_AND_DISK_2,**
- **DISK_ONLY_2**
- **MEMORY_ONLY_SER_2,**
- **MEMORY_AND_DISK_SER_2**

To check the storage level of the dataframe or RDD, we can use **rdd.getStorageLevel** or **df.storageLevel**

The class method takes cls (class) as first argument.	The static method does not take any specific parameter.
Class method can access and modify the class state.	Static Method cannot access or modify the class state.
The class method takes the class as parameter to know about the state of that class.	Static methods do not know about class state. These methods are used to do some utility tasks by taking some parameters.
@classmethod decorator is used here.	@staticmethod decorator is used here.

Class vs static method:

Decorators:

A decorator is a **design pattern** in Python that allows a user to add new functionality to an existing object without modifying its structure.

Decorators are usually called before the definition of a function

we use a decorator **when you need to change the behaviour of a function without modifying the function itself**. A few good examples are when you want to add logging, test performance, perform caching, verify permissions

iterators and generators?

Iterators are the objects that use the next() method to get the next value of the sequence. A generator is a function that produces or yields a sequence of values using a yield statement. Classes are used to Implement the iterators. Functions are used to implement the generator.

Lambda function:

A lambda function is **a small anonymous function**. A lambda function can take any number of arguments, but can only have one expression.

Lambda functions are used **when you need a function for a short period of time**

Polymorphism in python defines methods in the child class that have the same name as the methods in the parent class

Isin():

The `isin()` method **checks if the Dataframe contains the specified value(s)**. It returns a DataFrame similar to the original DataFrame, but the original values have been replaced with True if the value was one of the specified values, otherwise False

Partition and Bucketing:

Both Partitioning and Bucketing in Hive are used to improve performance by eliminating table scans when dealing with a large set of data on a Hadoop file system (HDFS). The major difference between Partitioning vs Bucketing lives in the way how they split the data.

Partition is a way to organize large tables into smaller logical tables based on values of columns; one logical table (partition) for each distinct value.

Bucketing is a technique to split the data into more manageable files, (By specifying the number of buckets to create). The value of the bucketing column will be hashed by a user-defined number into buckets

Below are some of the differences between Partitioning vs bucketing

PARTITIONING	BUCKETING
Directory is created on HDFS for each partition.	File is created on HDFS for each bucket.
You can have one or more Partition columns	You can have only one Bucketing column
You can't manage the number of partitions to create	You can manage the number of buckets to create by specifying the count
NA	Bucketing can be created on a partitioned table
Uses PARTITIONED BY	Uses CLUSTERED BY

How to connect Hive through Spark SQL?



Solution to this is to copy your `hive-site.xml` and `core-site.xml` in spark conf folder which will give Spark job all the required metadata about Hive metastore and you have to enable Hive Support along with specifying your warehouse directory location of Hive in configuration while starting your Spark Session as given below:

```
spark = SparkSession \
    .builder \
    .appName("Python Spark SQL Hive integration example") \
    .config("spark.sql.warehouse.dir", warehouse_location) \
    .enableHiveSupport() \
    .getOrCreate()
```

Difference between Rank and Dense Rank?

This a sql question but I included it because we can expect this question if we go in window-partition section. Suppose, we have a dataset as given below:

Name	Salary	Rank	Dense_rank
Abid	1000	1	1
Ron	1500	2	2
Joy	1500	2	2
Aly	2000	4	3
Raj	3000	5	4

Here salary is in increasing order and we are getting rank() an dense_rank() for the dataset. As Ron and Joy have same salary they get the same rank, but rank() will leave hole and keep “3” as empty whereas dense_rank() will fill all the gaps even though same values are encountered

Databricks

Cluster types

1. All-purpose cluster
 - Also known as interactive cluster because
 - All-purpose cluster used for mainly used for developing purpose. While developing you should see the intermediate result.
 - All-purpose cluster also can be used for job.
 - Can be paused ,stop ,started and multiple user can share this cluster
2. Job cluster
 - Mainly used for schedule jobs
 - While scheduling jobs you need to configure the cluster parameter based on the cluster would be created during runtime and once the job got completed it will terminate automatically

- You couldn't control manually. Job cluster are visible during job runtime

3. Pool cluster

- When you have multiple cluster you want to combine then you can create pool
- Advantage of pool while creating you can set parameters such as this many number of instances should be active always and ready to use
- Suitable for larger teams
- It will be costly

Cluster Modes

- Standard
- High concurrency
- Single

What is auto scaling?

Databricks chooses dynamically the appropriate number of workers required to run the job based on range of number of workers.

It is one of the performance optimization technique

It is also one of cost saving technique

Auto scaling has two types

1. Standard
2. optimized

Sqoop

Apache Sqoop in Hadoop is used to fetch structured data from RDBMS systems like Teradata, Oracle, MySQL, MSSQL, PostgreSQL and on the other hand

Flume

Apache Flume is used to fetch data that is stored on various sources as like the log files on a Web Server or an Application Server.

Different Types of SQL JOINS

Here are the different types of the JOINs in SQL:

- **(INNER) JOIN**: Returns records that have matching values in both tables
- **LEFT (OUTER) JOIN**: Returns all records from the left table, and the matched records from the right table
- **RIGHT (OUTER) JOIN**: Returns all records from the right table, and the matched records from the left table
- **FULL (OUTER) JOIN**: Returns all records when there is a match in either left or right table

Set Operators:

Set operators are used to combine results from two or more SELECT statements. They combine the same type of data from two or more tables. This looks similar to SQL joins although there is a difference. SQL joins are used to combine columns whereas Set

operators are used to join rows from multiple SELECT queries. They return only one result set.

These operators work on complete rows of the queries, so the results of the queries must have the same column name, same column order and the types of columns must be compatible.

There are the following 4 set operators in SQL Server: union, unionall, intersect and except

UNION

The UNION operator combines two or more result sets into a single result set, without duplications

UNION ALL

Like the UNION operator the UNION ALL operator also combines two or more result sets into a single result set. The only difference between a UNION and UNION ALL is that the UNION ALL allows duplicate rows.

INTERSECT

INTERSECT operator returns only the rows present in all the result sets. The intersection of two queries gives the rows that are present in both result sets

EXCEPT

EXCEPT operator returns all distinct the rows that are present in the result set of the first query, but not in the result set of the second query. It means it returns the difference between the two result sets.

Execution order of SQL

- FROM
- WHERE
- GROUP BY
- HAVING
- SELECT
- ORDER BY
- LIMIT

Primary Key vs Foreign Key

Primary Key	Foreign Key
Primary key uniquely identify a record in the table.	Foreign key is a field in the table that is primary key in another table.
Primary Key can't accept null values.	Foreign key can accept multiple null value.
By default, Primary key is clustered index and data in the database table is physically organized in the sequence of clustered index.	Foreign key do not automatically create an index, clustered or non-clustered. You can manually create an index on foreign key.
We can have only one Primary key in a table.	We can have more than one foreign key in a table.
The primary key of a particular table is the attribute which uniquely identifies every record and does not contain any null value.	The foreign key of a particular table is simply the primary key of some other table which is used as a reference key in the second table.
A primary key attribute in a table can never contain a null value.	A foreign key attribute may have null values as well.
Not more than one primary key is permitted in a table.	A table can have one or more than one foreign key for referential purposes.
Duplicity is strictly prohibited in the primary key; there cannot be any duplicate values.	Duplicity is permitted in the foreign key attribute, hence duplicate values are permitted.

JOINS IN SQL

- **INNER JOIN:** return all the rows from multiple tables where the join condition is satisfied.
- **LEFT JOIN:** return all the rows from the left table but only the matching rows from the right table where the join condition is fulfilled.
- **RIGHT JOIN:** return all the rows from the right table but only the matching rows from the left table where the join condition is fulfilled.

- **FULL JOIN:** returns all the records when there is a match in any of the tables. Therefore, it returns all the rows from the left-hand side table and all the rows from the right-hand side table.
- **SELF JOIN** – is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.
- **CARTESIAN JOIN** – returns the Cartesian product of the sets of records from the two or more joined tables.

Common clauses used with SELECT query in SQL?

The following are some frequent SQL clauses used in conjunction with a SELECT query:

WHERE clause: In SQL, the WHERE clause is used to filter records that are required depending on certain criteria.

```
Example: SELECT ID, NAME, SALARY
FROM CUSTOMERS
WHERE SALARY > 2000;
```

ORDER BY clause: The ORDER BY clause in SQL is used to sort data in ascending (ASC) or descending (DESC) order depending on specified field(s) (DESC).

```
SELECT * FROM CUSTOMERS
ORDER BY NAME DESC;
```

GROUP BY clause: GROUP BY clause in SQL is used to group entries with identical data and may be used with aggregation methods to obtain summarized database results.

```
SELECT DEPT, SUM(SALARY) FROM CUSTOMERS GROUP BY DEPT;
SELECT DEPT, min(SALARY) FROM CUSTOMERS GROUP BY DEPT ;
```

```
SELECT DEPT, MAX(SALARY) FROM CUSTOMERS GROUP BY DEPT ;  
SELECT DEPT, AVG(SALARY) FROM CUSTOMERS GROUP BY DEPT ;
```

HAVING clause in SQL is used to filter records in combination with the GROUP BY clause. It is different from WHERE, since the WHERE clause cannot filter aggregated records Syntax:

SELECT FROM WHERE GROUP BY

HAVING
ORDER BY

Example

```
SELECT ID, NAME, AGE, ADDRESS, SALARY  
FROM CUSTOMERS  
GROUP BY age  
HAVING COUNT(age) >= 2;
```

How to remove duplicate rows in SQL?

A.) DISTINCT

```
SELECT DISTINCT SALARY FROM CUSTOMERS  
ORDER BY SALARY;
```

B.) DELETE BY ROW

- If the SQL table has duplicate rows, the duplicate rows must be removed.
- Let's assume the following table as our dataset:

ID	Name	Age
1	A	21
2	B	23
2	B	23
4	D	22

5	E	25
6	G	26
5	E	25

```
DELETE FROM table WHERE ID IN (
SELECT
ID, COUNT (ID)
FROM  table
GROUP BY ID
HAVING
COUNT (ID) > 1);
```

How to find the nth highest salary in SQL?

```
select salary AS SecondHighestSalary from ( select row_number () over ( order by
salary desc ) row_ , salary from CUSTOMERS ) as emp where emp.row_ = 2
```

```
select salary AS ThirdHighestSalary from ( select row_number () over ( order by
salary desc ) row_ , salary from CUSTOMERS ) as emp where emp.row_ = 3
```

What is the ACID property in a database?

ACID stands for Atomicity, Consistency, Isolation, and Durability. It is used to ensure that the data transactions are processed reliably in a database system.

- **Atomicity** - each statement in a transaction (to read, write, update or delete data) is treated as a single unit. Either the entire statement is executed, or none of it is executed. This property prevents data loss and corruption from occurring if, for example, if you're streaming data source fails mid-stream.
- **Consistency** - ensures that transactions only make changes to tables in predefined, predictable ways. Transactional consistency ensures that corruption or errors in your data do not create unintended consequences for the integrity of your table.
- **Isolation** - when multiple users are reading and writing from the same table all at once, isolation of their transactions ensures that the concurrent transactions don't interfere with or affect one another. Each request can

occur as though they were occurring one by one, even though they're actually occurring simultaneously.

- **Durability** - ensures that changes to your data made by successfully executed transactions will be saved, even in the event of system failure

Rank vs Dense Rank vs Row _Number:

RANK function skips the next N-1 ranks if there is a tie between N previous ranks.

DENSE_RANK function does not skip ranks if there is a tie between ranks

ROW_NUMBER function has no concern with ranking. It simply returns the row number of the sorted records

	name	company	power	Rank	Dense Rank	Row Number
1	800	BMW	8000	1	1	1
2	110	Bugatti	8000	1	1	2
3	208	Peugeot	5400	3	2	3
4	Atlas	Volkswagen	5000	4	3	4
5	Mustang	Ford	5000	4	3	5
6	C500	Mercedez	5000	4	3	6
7	Prius	Toyota	3200	7	4	7
8	Landcruiser	Toyota	3000	8	5	8
9	Accord	Honda	2000	9	6	9
10	C200	Mercedez	2000	9	6	10
11	Corolla	Toyota	1800	11	7	11

AZURE

What is Activity in Azure Data Factory?

The activity is the task we performed on our data. We use activity inside the Azure Data Factory pipelines. ADF pipelines are a group of one or more activities. For ex: When you create an ADF pipeline to perform ETL you can use multiple activities to extract data, transform data and load data to your data warehouse. Activity uses Input and output datasets. Dataset represents

your data if it is tables, files, folders etc. Below diagram shows the relationship between Activity, dataset and pipeline:



An Input dataset simply tells you about the input data and its schema. And an Output dataset will tell you about the output data and its schema. You can attach zero or more Input datasets and one or more Output datasets. Activities in Azure Data Factory can be broadly categorized as:

- 1- Data Movement Activities
- 2- Data Transformation Activities
- 3- Control Activities

DATA MOVEMENT ACTIVITIES:

1- Copy Activity: It simply copies the data from Source location to destination location. Azure supports multiple data store locations such as Azure Storage, Azure DBs, NoSQL, Files, etc.

DATA TRANSFORMATION ACTIVITIES:

1- Data Flow: In data flow, First, you need to design data transformation workflow to transform or move data. Then you can call Data Flow activity inside the ADF pipeline. It runs on Scaled out Apache Spark Clusters. There are two types of DataFlows: Mapping and Wrangling DataFlows

MAPPING DATA FLOW: It provides a platform to graphically design data transformation logic. You don't need to write code. Once your data flow is complete, you can use it as an Activity in ADF pipelines.

WRANGLING DATA FLOW: It provides a platform to use power query in Azure Data Factory which is available on Ms excel. You can use power query M functions also on the cloud.

2- Hive Activity: This is a HD insight activity that executes Hive queries on windows/linux based HDInsight cluster. It is used to process and analyze structured data.

3- Pig activity: This is a HD insight activity that executes Pig queries on windows/linux based HDInsight cluster. It is used to analyze large datasets.

4- MapReduce: This is a HD insight activity that executes MapReduce programs on windows/linux based HDInsight cluster. It is used for processing and generating large datasets with a parallel distributed algorithm on a cluster.

5- Hadoop Streaming: This is a HD Insight activity that executes Hadoop streaming program on windows/linux based HDInsight cluster. It is used to write mappers and reducers with any executable script in any language like Python, C++ etc.

6- Spark: This is a HD Insight activity that executes Spark program on windows/linux based HDInsight cluster. It is used for large scale data processing.

7- Stored Procedure: In Data Factory pipeline, you can use execute Stored procedure activity to invoke a SQL Server Stored procedure. You can use the following data stores: Azure SQL Database, Azure Synapse Analytics, SQL Server Database, etc.

8- U-SQL: It executes U-SQL script on Azure Data Lake Analytics cluster. It is a big data query language that provides benefits of SQL.

9- Custom Activity: In custom activity, you can create your own data processing logic that is not provided by Azure. You can configure .Net

activity or R activity that will run on Azure Batch service or an Azure HDInsight cluster.

10- Databricks Notebook: It runs your databricks notebook on Azure databricks workspace. It runs on Apache spark.

11- Databricks Python Activity: This activity will run your python files on Azure Databricks cluster.

12- Azure Functions: It is Azure Compute service that allows us to write code logic and use it based on events without installing any infrastructure. It stores your code into Storage and keep the logs in application Insights.Key points of Azure Functions are :

1- It is a Serverless service.

2- It has Multiple languages available : C#, Java, Javascript, Python and PowerShell

3- It is a Pay as you go Model.

3- Control Flow Activities:

1- Append Variable Activity: It assigns a value to the array variable.

2- Execute Pipeline Activity: It allows you to call Azure Data Factory pipelines.

3- Filter Activity: It allows you to apply different filters on your input dataset.

4- For Each Activity: It provides the functionality of a for each loop that executes for multiple iterations.

5- Get Metadata Activity: It is used to get metadata of files/folders. You need to provide the type of metadata you require: childItems,

columnCount, contentMDS, exists, itemName, itemType, lastModified, size, structure, created etc.

6- If condition Activity: It provides the same functionality as If statement, it executes the set of expressions based on if the condition evaluates to true or false.

7- Lookup Activity: It reads and returns the content of multiple data sources such as files or tables or databases. It could also return the result set of a query or stored procedures.

8- Set Variable Activity: It is used to set the value to a variable of type String, Array, etc.

9- Switch Activity: It is a Switch statement that executes the set of activities based on matching cases.

10- Until Activity: It is same as do until loop. It executes a set of activities until the condition is set to true.

11- Validation Activity: It is used to validate the input dataset.

12- Wait Activity: It just waits for the given interval of time before moving ahead to the next activity. You can specify the number of seconds.

13- Web Activity: It is used to make a call to REST APIs. You can use it for different use cases such as ADF pipeline execution.

14- Webhook Activity: It is used to call the endpoint URLs to start/stop the execution of the pipelines. You can call external URLs also.

PYSPARK DATAFRAME

PySpark – Create DataFrame with Examples:

You can manually **create a PySpark**

DataFrame using `toDF()` and `createDataFrame()` methods, both these function takes different signatures in order to create DataFrame from existing RDD, list, and DataFrame. You can also create PySpark DataFrame from data sources like TXT, CSV, JSON, ORV, Avro, Parquet, XML formats by reading from HDFS, S3, DBFS, Azure Blob file systems e.t.c.

SPARKSESSION	RDD	DATAFRAME
<code>createDataFrame(rdd)</code>	<code>toDF()</code>	<code>toDF(*cols)</code>
<code>createDataFrame(dataList)</code>		<code>toDF(*cols)</code>
<code>createDataFrame(rowData,columns)</code>		
<code>createDataFrame(dataList,schema)</code>		

In order to create a DataFrame from a list we need the data hence, first, let's create the data and the columns that are needed.

```
columns = ["language","users_count"]
data = [("Java", "20000"), ("Python", "100000"), ("Scala", "3000")]
```

1. Create DataFrame from RDD

One easy way to manually create PySpark DataFrame is from an existing RDD. first, let's create a Spark RDD from a collection List by calling `parallelize()` function from `SparkContext`. We would need this `rdd` object for all our examples below.

```
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()
rdd = spark.sparkContext.parallelize(data)
```

1.1 Using `toDF()` function

PySpark RDD's `toDF()` method is used to create a DataFrame from the existing RDD. Since RDD doesn't have columns, the DataFrame is created with default column names "`_1`" and "`_2`" as we have two columns.

```
dfFromRDD1 = rdd.toDF()
dfFromRDD1.printSchema()
```

PySpark printschema() yields the schema of the DataFrame to console.

```
root
|-- _1: string (nullable = true)
|-- _2: string (nullable = true)
```

If you wanted to provide column names to the DataFrame use toDF() method with column names as arguments as shown below.

```
columns = ["language","users_count"]
dfFromRDD1 = rdd.toDF(columns)
dfFromRDD1.printSchema()
```

This yields the schema of the DataFrame with column names. use the show() method on PySpark DataFrame to show the DataFrame

```
root
|-- language: string (nullable = true)
|-- users: string (nullable = true)
```

By default, the datatype of these columns infers to the type of data. We can change this behavior by supplying schema, where we can specify a column name, data type, and nullable for each field/column.

1.2 Using createDataFrame () from SparkSession

Using createDataFrame () from SparkSession is another way to create manually and it takes rdd object as an argument. and chain with toDF () to specify name to the columns.

```
dfFromRDD2 = spark.createDataFrame(rdd).toDF(*columns)
```

2. Create DataFrame from List Collection

In this section, we will see how to create PySpark DataFrame from a list. These examples would be similar to what we have seen in the above section with RDD, but we use the list data object instead of “rdd” object to create DataFrame.

2.1 Using createDataFrame() from SparkSession

Calling createDataFrame() from SparkSession is another way to create PySpark DataFrame manually, it takes a list object as an argument. and chain with toDF() to specify names to the columns.

```
dfFromData2 = spark.createDataFrame(data).toDF(*columns)
```

2.2 Using `createDataFrame()` with the Row type

`createDataFrame()` has another signature in PySpark which takes the collection of Row type and schema for column names as arguments. To use this first we need to convert our “data” object from the list to list of Row.

```
rowData = map(lambda x: Row(*x), data)
dfFromData3 = spark.createDataFrame(rowData,columns)
```

2.3 Create DataFrame with schema

If you wanted to specify the column names along with their data types, you should create the StructType schema first and then assign this while creating a DataFrame.

```
from pyspark.sql.types import StructType,StructField, StringType, IntegerType
data2 = [("James","","Smith","36636","M",3000),
         ("Michael","Rose","","40288","M",4000),
         ("Robert","","Williams","42114","M",4000),
         ("Maria","Anne","Jones","39192","F",4000),
         ("Jen","Mary","Brown","","F",-1)
        ]

schema = StructType([ \
    StructField("firstname",StringType(),True), \
    StructField("middlename",StringType(),True), \
    StructField("lastname",StringType(),True), \
    StructField("id", StringType(), True), \
    StructField("gender", StringType(), True), \
    StructField("salary", IntegerType(), True) \
])

df = spark.createDataFrame(data=data2,schema=schema)
df.printSchema()
df.show(truncate=False)
```

This yields below output.

```
root
|-- firstname: string (nullable = true)
|-- middlename: string (nullable = true)
|-- lastname: string (nullable = true)
|-- id: string (nullable = true)
|-- gender: string (nullable = true)
|-- salary: integer (nullable = true)

+---+---+---+---+---+
|firstname|middlename|lastname|id    |gender|salary|
+---+---+---+---+---+
|James      |          |Smith     |36636|M    |3000   |
|Michael    |Rose      |          |40288|M    |4000   |
|Robert     |          |Williams  |42114|M    |4000   |
|Maria      |Anne      |Jones     |39192|F    |4000   |
|Jen        |Mary      |Brown     |          |F    |-1     |
+---+---+---+---+---+
```

3. Create DataFrame from Data sources

In real-time mostly you create DataFrame from data source files like CSV, Text, JSON, XML e.t.c.

PySpark by default supports many data formats out of the box without importing any libraries and to create DataFrame you need to use the appropriate method available in `DataFrameReader` class.

3.1 Creating DataFrame from CSV

Use `csv()` method of the `DataFrameReader` object to create a DataFrame from CSV file. you can also provide options like what delimiter to use, whether you have quoted data, date formats, infer schema, and many more. Please refer PySpark Read CSV into DataFrame

```
df2 = spark.read.csv("/src/resources/file.csv")
```

3.2. Creating from text (TXT) file

Similarly you can also create a DataFrame by reading a from Text file, use `text()` method of the `DataFrameReader` to do so.

```
df2 = spark.read.text("/src/resources/file.txt")
```

3.3. Creating from JSON file

PySpark is also used to process semi-structured data files like JSON format. you can use `json()` method of the `DataFrameReader` to read JSON file into DataFrame. Below is a simple example.

```
df2 = spark.read.json("/src/resources/file.json")
```

Similarly, we can create DataFrame in PySpark from most of the relational databases which I've not covered here and I will leave this to you to explore.

PySpark – Create an Empty DataFrame & RDD

While working with files, sometimes we may not receive a file for processing, however, we still need to create a DataFrame manually with the same schema we expect. If we don't create with the same schema, our operations/transformations (like union's) on DataFrame fail as we refer to the columns that may not present.

To handle situations similar to these, we always need to create a DataFrame with the same schema, which means the same column names and datatypes regardless of the file exists or empty file processing.

1. Create Empty RDD in PySpark

Create an empty RDD by using `emptyRDD()` of SparkContext for example `spark.sparkContext.emptyRDD()`.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

#Creates Empty RDD
emptyRDD = spark.sparkContext.emptyRDD()
print(emptyRDD)

#Displays
#EmptyRDD[188] at emptyRDD
```

Alternatively you can also get empty RDD by using `spark.sparkContext.parallelize([])`.

```
#Creates Empty RDD using parallelize
rdd2= spark.sparkContext.parallelize([])
```

VN2 Solutions Pvt. Ltd.

```
print(rdd2)

#EmptyRDD[205] at emptyRDD at NativeMethodAccessorImpl.java:0
#ParallelCollectionRDD[206] at readRDDFromFile at PythonRDD.scala:262
```

Note: If you try to perform operations on empty RDD you going to get `ValueError("RDD is empty")`.

2. Create Empty DataFrame with Schema (StructType)

In order to create an empty PySpark DataFrame manually with schema (column names & data types) first, Create a schema using StructType and StructField .

```
#Create Schema
from pyspark.sql.types import StructType,StructField, StringType
schema = StructType([
    StructField('firstname', StringType(), True),
    StructField('middlename', StringType(), True),
    StructField('lastname', StringType(), True)
])
```

Now use the empty RDD created above and pass it to `createDataFrame()` of SparkSession along with the schema for column names & data types.

```
#Create empty DataFrame from empty RDD
df = spark.createDataFrame(emptyRDD,schema)
df.printSchema()
```

This yields below schema of the empty DataFrame.

```
root
 |-- firstname: string (nullable = true)
 |-- middlename: string (nullable = true)
 |-- lastname: string (nullable = true)
```

3. Convert Empty RDD to DataFrame

You can also create empty DataFrame by converting empty RDD to DataFrame using `toDF()`.

```
#Convert empty RDD to Dataframe
df1 = emptyRDD.toDF(schema)
df1.printSchema()
```

4. Create Empty DataFrame with Schema.

So far I have covered creating an empty DataFrame from RDD, but here will create it manually with schema and without RDD.

```
#Create empty DataFrame directly.  
df2 = spark.createDataFrame([], schema)  
df2.printSchema()
```

5. Create Empty DataFrame without Schema (no columns)

To create empty DataFrame with out schema (no columns) just create a empty schema and use it while creating PySpark DataFrame.

```
#Create empty DataFrame with no schema (no columns)  
df3 = spark.createDataFrame([], StructType([]))  
df3.printSchema()  
  
#print below empty schema  
#root
```

Convert PySpark RDD to DataFrame

In PySpark, `toDF()` function of the RDD is used to convert RDD to DataFrame. We would need to convert RDD to DataFrame as DataFrame provides more advantages over RDD. For instance, DataFrame is a distributed collection of data organized into named columns similar to Database tables and provides optimization and performance improvements.

- Create PySpark RDD
- Convert PySpark RDD to DataFrame
 - using `toDF()`
 - using `createDataFrame()`
 - using RDD row type & schema

1. Create PySpark RDD

First, let's create an RDD by passing Python list object to `sparkContext.parallelize()` function. We would need this `rdd` object for all our examples below.

VN2 Solutions Pvt. Ltd.

In PySpark, when you have data in a list meaning you have a collection of data in a PySpark driver memory when you create an RDD, this collection is going to be parallelized.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()
dept = [("Finance",10),("Marketing",20),("Sales",30),("IT",40)]
rdd = spark.sparkContext.parallelize(dept)
```

2. Convert PySpark RDD to DataFrame

Converting PySpark RDD to DataFrame can be done using `toDF()`, `createDataFrame()`. In this section, I will explain these two methods.

2.1 Using `rdd.toDF()` function

PySpark provides `toDF()` function in RDD which can be used to convert RDD into Dataframe

```
df = rdd.toDF()
df.printSchema()
df.show(truncate=False)
```

By default, `toDF()` function creates column names as “`_1`” and “`_2`”. This snippet yields below schema.

```
root
 |-- _1: string (nullable = true)
 |-- _2: long (nullable = true)

+-----+---+
|_1    |_2 |
+-----+---+
|Finance|10 |
|Marketing|20 |
|Sales   |30 |
|IT     |40 |
+-----+---+
```

`toDF()` has another signature that takes arguments to define column names as shown below.

VN2 Solutions Pvt. Ltd.

```
deptColumns = ["dept_name","dept_id"]
df2 = rdd.toDF(deptColumns)
df2.printSchema()
df2.show(truncate=False)
```

Outputs below schema.

```
root
|-- dept_name: string (nullable = true)
|-- dept_id: long (nullable = true)

+-----+-----+
|dept_name|dept_id|
+-----+-----+
|Finance |10    |
|Marketing|20    |
|Sales   |30    |
|IT      |40    |
+-----+-----+
```

2.2 Using PySpark `createDataFrame()` function

SparkSession class provides `createDataFrame()` method to create DataFrame and it takes rdd object as an argument.

```
deptDF = spark.createDataFrame(rdd, schema = deptColumns)
deptDF.printSchema()
deptDF.show(truncate=False)
```

This yields the same output as above.

2.3 Using `createDataFrame()` with `StructType` schema

When you infer the schema, by default the datatype of the columns is derived from the data and set's nullable to true for all columns. We can change this behavior by supplying schema using `StructType` – where we can specify a column name, data type and nullable for each field/column.

If you wanted to know more about `StructType`, please go through [how to use StructType and StructField to define the custom schema](#).

```
from pyspark.sql.types import StructType,StructField, StringType
deptSchema = StructType([
    StructField('dept_name', StringType(), True),
    StructField('dept_id', StringType(), True)
])

deptDF1 = spark.createDataFrame(rdd, schema = deptSchema)
```

VN2 Solutions Pvt. Ltd.

```
deptDF1.printSchema()  
deptDF1.show(truncate=False)
```

3. Complete Example

```
import pyspark  
from pyspark.sql import SparkSession  
  
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()  
  
dept = [("Finance",10),("Marketing",20),("Sales",30),("IT",40)]  
rdd = spark.sparkContext.parallelize(dept)  
  
df = rdd.toDF()  
df.printSchema()  
df.show(truncate=False)  
  
deptColumns = ["dept_name","dept_id"]  
df2 = rdd.toDF(deptColumns)  
df2.printSchema()  
df2.show(truncate=False)  
  
deptDF = spark.createDataFrame(rdd, schema = deptColumns)  
deptDF.printSchema()  
deptDF.show(truncate=False)  
  
from pyspark.sql.types import StructType,StructField, StringType  
deptSchema = StructType([  
    StructField('dept_name', StringType(), True),  
    StructField('dept_id', StringType(), True)  
])  
  
deptDF1 = spark.createDataFrame(rdd, schema = deptSchema)  
deptDF1.printSchema()  
deptDF1.show(truncate=False)
```

4. Conclusion:

In this article, you have learned how to convert PySpark RDD to DataFrame, we would need these frequently while working in PySpark as these provides optimization and performance over RDD.

Convert PySpark DataFrame to Pandas

- Post author:[NNK](#)

- Post category:[Pandas](#) / [PySpark](#) / [Python](#)

(Spark with Python) PySpark DataFrame can be converted to [Python pandas DataFrame](#) using a function `toPandas()`, In this article, I will explain how to create Pandas DataFrame from PySpark (Spark) DataFrame with examples.

Before we start first understand the main differences between the Pandas & PySpark, operations on Pyspark run faster than Pandas due to its distributed nature and parallel execution on multiple cores and machines.

In other words, pandas run operations on a single node whereas PySpark runs on multiple machines. If you are working on a Machine Learning application where you are dealing with larger datasets, PySpark processes operations many times faster than pandas. [Refer to pandas DataFrame Tutorial beginners guide with examples](#)

After processing data in PySpark we would need to convert it back to Pandas DataFrame for a further procession with Machine Learning application or any Python applications.

Prepare PySpark DataFrame

In order to explain with an example first let's [create a PySpark DataFrame](#).

```
import pyspark
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

data = [("James","","Smith","36636","M",60000),
        ("Michael","Rose","","40288","M",70000),
        ("Robert","","Williams","42114","",400000),
        ("Maria","Anne","Jones","39192","F",500000),
        ("Jen","Mary","Brown","","F",0)]
```

```
columns = ["first_name","middle_name","last_name","dob","gender","salary"]
pysparkDF = spark.createDataFrame(data = data, schema = columns)
pysparkDF.printSchema()
pysparkDF.show(truncate=False)
```

This yields below schema and result of the DataFrame.

```
root
 |-- first_name: string (nullable = true)
 |-- middle_name: string (nullable = true)
```

VN2 Solutions Pvt. Ltd.

```
-- last_name: string (nullable = true)
-- dob: string (nullable = true)
-- gender: string (nullable = true)
-- salary: long (nullable = true)

+-----+-----+-----+-----+
|first_name|middle_name|last_name|dob |gender|salary|
+-----+-----+-----+-----+
|James    |          |Smith    |36636|M   |60000 |
|Michael  |Rose      |          |40288|M   |70000 |
|Robert   |          |Williams |42114|     |400000|
|Maria    |Anne      |Jones    |39192|F   |500000|
|Jen     |Mary      |Brown    |      |F   |0    |
+-----+-----+-----+-----+
```

Convert PySpark Dataframe to Pandas DataFrame

PySpark DataFrame provides a method `toPandas()` to convert it to Python Pandas DataFrame.

`toPandas()` results in the collection of all records in the PySpark DataFrame to the driver program and should be done only on a small subset of the data. running on larger dataset's results in memory error and crashes the application. To deal with a larger dataset, you can also try increasing memory on the driver.

```
pandasDF = pysparkDF.toPandas()
print(pandasDF)
```

This yields the below panda's DataFrame. Note that pandas add a sequence number to the result as a row Index. You can rename pandas columns by using `rename()` function.

```
first_name middle_name last_name   dob gender  salary
0   James           Smith  36636    M  60000
1 Michael         Rose   40288    M  70000
2 Robert          Williams 42114    400000
3 Maria           Anne   Jones  39192    F 500000
4 Jen             Mary   Brown    F   0
```

I have dedicated [Python pandas Tutorial with Examples](#) where I explained pandas concepts in detail.

Convert Spark Nested Struct DataFrame to Pandas

Most of the time data in PySpark DataFrame will be in a structured format meaning one column contains other columns so let's see how it convert to Pandas. Here is an example with nested struct where we have `firstname`, `middlename` and `lastname` are part of the `name` column.

```
# Nested structure elements
from pyspark.sql.types import StructType, StructField, StringType, IntegerType
dataStruct = [(("James","","Smith"),"36636","M","3000"), \
    (("Michael","Rose","",), "40288","M","4000"), \
    (("Robert","","Williams"), "42114","M","4000"), \
    (("Maria","Anne","Jones"), "39192","F","4000"), \
    (("Jen","Mary","Brown"), "", "F", "-1") \
]
schemaStruct = StructType([
    StructField('name', StructType([
        StructField('firstname', StringType(), True),
        StructField('middlename', StringType(), True),
        StructField('lastname', StringType(), True)
    ])),
    StructField('dob', StringType(), True),
    StructField('gender', StringType(), True),
    StructField('salary', StringType(), True)
])
df = spark.createDataFrame(data=dataStruct, schema = schemaStruct)
df.printSchema()

pandasDF2 = df.toPandas()
print(pandasDF2)
```

Converting structured DataFrame to Pandas DataFrame results below output.

	name	dob	gender	salary
0	(James, , Smith)	36636	M	3000
1	(Michael, Rose,)	40288	M	4000
2	(Robert, , Williams)	42114	M	4000
3	(Maria, Anne, Jones)	39192	F	4000
4	(Jen, Mary, Brown)		F	-1

Conclusion

In this simple article, you have learned to convert Spark DataFrame to pandas using `toPandas()` function of the Spark DataFrame. also have seen a similar example with complex nested structure elements. `toPandas()` results in the collection of all records in the DataFrame to the driver program and should be done on a small subset of the data.

PySpark show() – Display DataFrame Contents in Table

PySpark DataFrame show() is used to display the contents of the DataFrame in a Table Row & Column Format. By default, it shows only 20 Rows, and the column values are truncated at 20 characters.

2. PySpark DataFrame show () Syntax & Example

1.1 Syntax

```
def show (self, n=20, truncate=True, vertical=False):
```

1.2 Example

Use show() method to display the contents of the DataFrame and use [pyspark printSchema\(\)](#) method to print the schema.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()
columns = ["Seqno","Quote"]
data = [("1", "Be the change that you wish to see in the world"),
       ("2", "Everyone thinks of changing the world, but no one thinks of changing
himself."),
       ("3", "The purpose of our lives is to be happy."),
       ("4", "Be cool.")]
df = spark.createDataFrame(data,columns)
df.show()

# Output
#+---+-----+
#|Seqno|      Quote|
#+---+-----+
#|   1|Be the change tha...|
#|   2|Everyone thinks o...|
#|   3|The purpose of ou...|
#|   4|      Be cool.|
```

As you see above, values in the `Quote` column are truncated at 20 characters, Let's see how to display the full column contents.

VN2 Solutions Pvt. Ltd.

```
#Display full column contents
df.show(truncate=False)

#-----+-----+
#|Seqno|Quote
#-----+-----+
#|1  |Be the change that you wish to see in the world      |
#|2  |Everyone thinks of changing the world, but no one thinks of changing himself.| |
#|3  |The purpose of our lives is to be happy.           |
#|4  |Be cool.                                         |
#-----+-----+
```

By default `show()` method displays only 20 rows from PySpark DataFrame. The below example limit the rows to 2 and full column contents. Our DataFrame has just 4 rows hence I can't demonstrate with more than 4 rows. If you have a DataFrame with thousands of rows try changing the value from 2 to 100 to display more than 20 rows.

```
# Display 2 rows and full column contents
df.show(2,truncate=False)

#-----+-----+
#|Seqno|Quote
#-----+-----+
#|1  |Be the change that you wish to see in the world      |
#|2  |Everyone thinks of changing the world, but no one thinks of changing himself.| |
#-----+-----+
```

You can also truncate the column value at the desired length.

```
# Display 2 rows & column values 25 characters
df.show(2,truncate=25)

#-----+-----+
#|Seqno|      Quote|
#-----+-----+
#| 1|Be the change that you...|
#| 2|Everyone thinks of cha...|
#-----+-----+
#only showing top 2 rows
```

Finally, let's see how to display the DataFrame vertically record by record.

```
# Display DataFrame rows & columns vertically
```

```
df.show(n=3,truncate=25,vertical=True)

#-RECORD 0-----
# Seqno | 1
# Quote | Be the change that you...
#-RECORD 1-----
# Seqno | 2
# Quote | Everyone thinks of cha...
#-RECORD 2-----
# Seqno | 3
# Quote | The purpose of our liv...
```

PySpark StructType & StructField Explained with Example
PySpark StructType & StructField classes are used to programmatically specify the schema to the DataFrame and create complex columns like nested struct, array, and map columns. StructType is a collection of StructField's that defines column name, column data type, boolean to specify if the field can be nullable or not and metadata.

In this article, I will explain different ways to define the structure of DataFrame using StructType with PySpark examples. Though PySpark infers a schema from data, sometimes we may need to define our own column names and data types and this article explains how to define simple, nested, and complex schemas.

- [Using PySpark StructType & StructField with DataFrame](#)
- [Defining Nested StructType or struct](#)
- [Adding & Changing columns of the DataFrame](#)
- [Using SQL ArrayType and MapType](#)
- [Creating StructType or struct from Json file](#)
- [Creating StructType object from DDL string](#)
- [Check if a field exists in a StructType](#)

1. StructType – Defines the structure of the Dataframe

PySpark provides from `pyspark.sql.types import StructType` class to define the structure of the DataFrame. StructType is a collection or list of StructField objects. PySpark `printSchema()` method on the DataFrame shows StructType columns as struct.

2. StructField – Defines the metadata of the DataFrame column

PySpark provides `pyspark.sql.types import StructField` class to define the columns which include column name(String), column type (`DataType`), nullable column (Boolean) and metadata (MetaDataTable).

3. Using PySpark StructType & StructField with DataFrame

While creating a PySpark DataFrame we can specify the structure using StructType and StructField classes. As specified in the introduction, StructType is a collection of StructField's which is used to define the column name, data type, and a flag for nullable or not. Using StructField we can also add nested struct schema, ArrayType for arrays, and MapType for key-value pairs which we will discuss in detail in later sections.

The below example demonstrates a very simple example of how to create a StructType & StructField on DataFrame and it's usage with sample data to support it.

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

spark = SparkSession.builder.master("local[1]") \
    .appName('SparkByExamples.com') \
    .getOrCreate()

data = [("James","","Smith","36636","M",3000),
       ("Michael","Rose","","40288","M",4000),
       ("Robert","","Williams","42114","M",4000),
       ("Maria","Anne","Jones","39192","F",4000),
       ("Jen","Mary","Brown","","F",-1)
      ]

schema = StructType([ \
    StructField("firstname",StringType(),True), \
    StructField("middlename",StringType(),True), \
    StructField("lastname",StringType(),True), \
    StructField("id", StringType(), True), \
    StructField("gender", StringType(), True), \
    StructField("salary", IntegerType(), True) \
  ])

df = spark.createDataFrame(data=data,schema=schema)
df.printSchema()
df.show(truncate=False)
```

By running the above snippet, it displays below outputs.

VN2 Solutions Pvt. Ltd.

```
root
|-- firstname: string (nullable = true)
|-- middlename: string (nullable = true)
|-- lastname: string (nullable = true)
|-- id: string (nullable = true)
|-- gender: string (nullable = true)
|-- salary: integer (nullable = true)

+-----+-----+-----+-----+
|firstname|middlename|lastname|id   |gender|salary|
+-----+-----+-----+-----+
|James    |          |Smith   |36636|M    |3000  |
|Michael |Rose     |        |40288|M    |4000  |
|Robert  |          |Williams|42114|M    |4000  |
|Maria   |Anne     |Jones   |39192|F    |4000  |
|Jen     |Mary     |Brown   |      |F    |-1    |
+-----+-----+-----+-----+
```

4. Defining Nested StructType object struct

While working on DataFrame we often need to work with the nested struct column and this can be defined using StructType.

In the below example column “name” data type is StructType which is nested.

```
structureData = [
    ("James","","Smith"), "36636", "M", 3100),
    ("Michael","Rose"), "40288", "M", 4300),
    ("Robert","","Williams"), "42114", "M", 1400),
    ("Maria","Anne","Jones"), "39192", "F", 5500),
    ("Jen","Mary","Brown"), "", "F", -1)
]

structureSchema = StructType([
    StructField('name', StructType([
        StructField('firstname', StringType(), True),
        StructField('middlename', StringType(), True),
        StructField('lastname', StringType(), True)
    ])),
    StructField('id', StringType(), True),
    StructField('gender', StringType(), True),
    StructField('salary', IntegerType(), True)
])

df2 = spark.createDataFrame(data=structureData,schema=structureSchema)
df2.printSchema()
df2.show(truncate=False)
```

VN2 Solutions Pvt. Ltd.

Outputs below schema and the DataFrame

```
root
|-- name: struct (nullable = true)
|   |-- firstname: string (nullable = true)
|   |-- middlename: string (nullable = true)
|   |-- lastname: string (nullable = true)
|-- id: string (nullable = true)
|-- gender: string (nullable = true)
|-- salary: integer (nullable = true)

+-----+-----+-----+
|name      |id    |gender|salary|
+-----+-----+-----+
|[James, , Smith]  |36636|M    |3100  |
|[Michael, Rose, ] |40288|M    |4300  |
|[Robert, , Williams]|42114|M    |1400  |
|[Maria, Anne, Jones]|39192|F    |5500  |
|[Jen, Mary, Brown]  |    |F    |-1   |
+-----+-----+-----+
```

5. Adding & Changing struct of the DataFrame

Using [PySpark SQL function struct\(\)](#), we can change the struct of the existing DataFrame and add a new StructType to it. The below example demonstrates how to copy the columns from one structure to another and adding a new column. [PySpark Column Class](#) also provides some functions to work with the StructType column.

```
from pyspark.sql.functions import col,struct,when
updatedDF = df2.withColumn("OtherInfo",
    struct(col("id").alias("identifier"),
    col("gender").alias("gender"),
    col("salary").alias("salary"),
    when(col("salary").cast(IntegerType()) < 2000,"Low")
    .when(col("salary").cast(IntegerType()) < 4000,"Medium")
    .otherwise("High").alias("Salary_Grade"))
)).drop("id","gender","salary")

updatedDF.printSchema()
updatedDF.show(truncate=False)
```

Here, it copies “gender”, “salary” and “id” to the new struct “otherInfo” and add’s a new column “Salary_Grade”.

```
root
|-- name: struct (nullable = true)
|   |-- firstname: string (nullable = true)
|   |-- middlename: string (nullable = true)
|   |-- lastname: string (nullable = true)
|-- OtherInfo: struct (nullable = false)
|   |-- identifier: string (nullable = true)
|   |-- gender: string (nullable = true)
|   |-- salary: integer (nullable = true)
|   |-- Salary_Grade: string (nullable = false)
```

6. Using SQL ArrayType and MapType

SQL StructType also supports [ArrayType](#) and [MapType](#) to define the DataFrame columns for array and map collections respectively. On the below example, column `hobbies` defined as `ArrayType(StringType())` and `properties` defined as `MapType(StringType(),StringType())` meaning both key and value as String.

```
arrayStructureSchema = StructType([
    StructField('name', StructType([
        StructField('firstname', StringType(), True),
        StructField('middlename', StringType(), True),
        StructField('lastname', StringType(), True)
    ])),
    StructField('hobbies', ArrayType(StringType()), True),
    StructField('properties', MapType(StringType(),StringType()), True)
])
```

Outputs the below schema. Note that field `Hobbies` is array type and `properties` is map type.

```
root
|-- name: struct (nullable = true)
|   |-- firstname: string (nullable = true)
|   |-- middlename: string (nullable = true)
|   |-- lastname: string (nullable = true)
|-- hobbies: array (nullable = true)
|   |-- element: string (containsNull = true)
|-- properties: map (nullable = true)
|   |-- key: string
|   |-- value: string (valueContainsNull = true)
```

7. Creating StructType object struct from JSON file

If you have too many columns and the structure of the DataFrame changes now and then, it's a good practice to load the SQL StructType schema from JSON file. You can

VN2 Solutions Pvt. Ltd.

get the schema by using `df2.schema.json()` , store this in a file and will use it to create a the schema from this file.

```
print(df2.schema.json())

{
  "type" : "struct",
  "fields" : [ {
    "name" : "name",
    "type" : {
      "type" : "struct",
      "fields" : [ {
        "name" : "firstname",
        "type" : "string",
        "nullable" : true,
        "metadata" : { }
      }, {
        "name" : "middlename",
        "type" : "string",
        "nullable" : true,
        "metadata" : { }
      }, {
        "name" : "lastname",
        "type" : "string",
        "nullable" : true,
        "metadata" : { }
      } ]
    },
    "nullable" : true,
    "metadata" : { }
  }, {
    "name" : "dob",
    "type" : "string",
    "nullable" : true,
    "metadata" : { }
  }, {
    "name" : "gender",
    "type" : "string",
    "nullable" : true,
    "metadata" : { }
  }, {
    "name" : "salary",
    "type" : "integer",
    "nullable" : true,
    "metadata" : { }
  }
}
```

```
 } ]  
 }
```

Alternatively, you could also use `df.schema.simpleString()`, this will return an relatively simpler schema format.

Now let's load the json file and use it to create a DataFrame.

```
import json  
schemaFromJson = StructType.fromJson(json.loads(schema.json))  
df3 = spark.createDataFrame(  
    spark.sparkContext.parallelize(structureData), schemaFromJson)  
df3.printSchema()
```

This prints the same output as the previous section. You can also, have a name, type, and flag for nullable in a comma-separated file and we can use these to create a StructType programmatically, I will leave this to you to explore.

8. Creating StructType object struct from DDL String

Like loading structure from JSON string, we can also create it from DDL (by using `fromDDL()` static function on SQL StructType class `StructType.fromDDL`). You can also generate DDL from a schema using `toDDL()`. `printTreeString()` on struct object prints the schema similar to `printSchema` function returns.

```
ddlSchemaStr = ``fullName` STRUCT<`first`: STRING, `last`: STRING,  
`middle`: STRING>, `age` INT, `gender` STRING"  
ddlSchema = StructType.fromDDL(ddlSchemaStr)  
ddlSchema.printTreeString()
```

9. Checking if a Column Exists in a DataFrame

If you want to perform some checks on metadata of the DataFrame, for example, if a column or field exists in a DataFrame or data type of column; we can easily do this using several functions on SQL StructType and StructField.

```
print(df.schema.fieldNames.contains("firstname"))  
print(df.schema.contains(StructField("firstname", StringType, true)))
```

This example returns “true” for both scenarios. And for the second one if you have IntegerType instead of StringType it returns false as the datatype for first name column is String, as it checks every property in a field. Similarly, you can also check if two schemas are equal and more.

10. Complete Example of PySpark StructType & StructField

VN2 Solutions Pvt. Ltd.

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType,StructField, StringType,
IntegerType,ArrayType,MapType
from pyspark.sql.functions import col,struct,when

spark = SparkSession.builder.master("local[1]") \
    .appName('SparkByExamples.com') \
    .getOrCreate()

data = [("James","","Smith","36636","M",3000),
        ("Michael","Rose","","40288","M",4000),
        ("Robert","","Williams","42114","M",4000),
        ("Maria","Anne","Jones","39192","F",4000),
        ("Jen","Mary","Brown","","F",-1)
       ]

schema = StructType([
    StructField("firstname",StringType(),True),
    StructField("middlename",StringType(),True),
    StructField("lastname",StringType(),True),
    StructField("id", StringType(), True),
    StructField("gender", StringType(), True),
    StructField("salary", IntegerType(), True)
])

df = spark.createDataFrame(data=schema)
df.printSchema()
df.show(truncate=False)

structureData = [
    ("James","","Smith"), "36636", "M", 3100),
    ("Michael","Rose","","40288", "M", 4300),
    ("Robert","","Williams"), "42114", "M", 1400),
    ("Maria","Anne","Jones"), "39192", "F", 5500),
    ("Jen","Mary","Brown"), "", "F", -1)
]

structureSchema = StructType([
    StructField('name', StructType([
        StructField('firstname', StringType(), True),
        StructField('middlename', StringType(), True),
        StructField('lastname', StringType(), True)
    ])),
    StructField('id', StringType(), True),
    StructField('gender', StringType(), True),
```

VN2 Solutions Pvt. Ltd.

```
StructField('salary', IntegerType(), True)
])

df2 = spark.createDataFrame(data=structureData,schema=structureSchema)
df2.printSchema()
df2.show(truncate=False)

updatedDF = df2.withColumn("OtherInfo",
    struct(col("id").alias("identifier"),
    col("gender").alias("gender"),
    col("salary").alias("salary"),
    when(col("salary").cast(IntegerType()) < 2000,"Low")
    .when(col("salary").cast(IntegerType()) < 4000,"Medium")
    .otherwise("High").alias("Salary_Grade")
)).drop("id","gender","salary")

updatedDF.printSchema()
updatedDF.show(truncate=False)

""" Array & Map"""

arrayStructureSchema = StructType([
    StructField('name', StructType([
        StructField('firstname', StringType(), True),
        StructField('middlename', StringType(), True),
        StructField('lastname', StringType(), True)
    ])),
    StructField('hobbies', ArrayType(StringType()), True),
    StructField('properties', MapType(StringType(),StringType()), True)
])
```

PySpark Row using on DataFrame and RDD

In PySpark Row class is available by importing `pyspark.sql.Row` which is represented as a record/row in DataFrame, one can create a Row object by using named arguments, or create a custom Row like class. In this article I will explain how to use Row class on RDD, DataFrame and its functions.

Before we start using it on RDD & DataFrame, let's understand some basics of Row class.

VN2 Solutions Pvt. Ltd.

Related Article: [PySpark Column Class Usage & Functions with Examples](#)

Key Points of Row Class:

- Earlier to Spark 3.0, when used Row class with named arguments, the fields are sorted by name.
- Since 3.0, Rows created from named arguments are not sorted alphabetically instead they will be ordered in the position entered.
- To enable sorting by names, set the environment variable `PYSPARK_ROW_FIELD_SORTING_ENABLED` to true.
- Row class provides a way to create a struct-type column as well.
-

1. Create a Row Object

Row class extends the tuple hence it takes variable number of arguments, `Row()` is used to create the row object. Once the row object created, we can retrieve the data from Row using index similar to tuple.

```
from pyspark.sql import Row
row=Row("James",40)
print(row[0] + ","+str(row[1]))
```

This outputs James,40. Alternatively you can also write with named arguments.

Benefits with the named argument is you can access with field name `row.name`. Below example print "Alice".

```
row=Row(name="Alice", age=11)
print(row.name)
```

2. Create Custom Class from Row

We can also create a Row like class, for example "Person" and use it similar to Row object. This would be helpful when you wanted to create real time object and refer its properties. On below example, we have created a Person class and used similar to Row.

```
Person = Row("name", "age")
p1=Person("James", 40)
p2=Person("Alice", 35)
print(p1.name + ","+p2.name)
```

This outputs James,Alice

3. Using Row class on PySpark RDD

We can use Row class on PySpark RDD. When you use Row to create an RDD, after collecting the data you will get the result back in Row.

```
from pyspark.sql import SparkSession, Row
```

VN2 Solutions Pvt. Ltd.

```
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

data = [Row(name="James,,Smith",lang=["Java","Scala","C++"],state="CA"),
        Row(name="Michael,Rose,",lang=["Spark","Java","C++"],state="NJ"),
        Row(name="Robert,,Williams",lang=["CSharp","VB"],state="NV")]
rdd=spark.sparkContext.parallelize(data)
print(rdd.collect())
```

This yields below output.

```
[Row(name='James,,Smith', lang=['Java', 'Scala', 'C++'], state='CA'),
Row(name='Michael,Rose,', lang=['Spark', 'Java', 'C++'], state='NJ'),
Row(name='Robert,,Williams', lang=['CSharp', 'VB'], state='NV')]
```

Now, let's collect the data and access the data using its properties.

```
collData=rdd.collect()
for row in collData:
    print(row.name + "," +str(row.lang))
```

This yields below output.

```
James,,Smith,['Java', 'Scala', 'C++']
Michael,Rose,['Spark', 'Java', 'C++']
Robert,,Williams,['CSharp', 'VB']
```

Alternatively, you can also do by creating a Row like class "Person"

```
Person=Row("name","lang","state")
data = [Person("James,,Smith",["Java","Scala","C++"],"CA"),
        Person("Michael,Rose,",["Spark","Java","C++"],"NJ"),
        Person("Robert,,Williams",["CSharp","VB"],"NV")]
```

4. Using Row class on PySpark DataFrame

Similarly, Row class also can be used with PySpark DataFrame, By default data in DataFrame represent as Row. To demonstrate, I will use the same data that was created for RDD.

Note that Row on DataFrame is not allowed to omit a named argument to represent that the value is None or missing. This should be explicitly set to None in this case.

```
df=spark.createDataFrame(data)
df.printSchema()
df.show()
```

VN2 Solutions Pvt. Ltd.

This yields below output. Note that DataFrame able to take the column names from Row object.

```
root
|-- name: string (nullable = true)
|-- lang: array (nullable = true)
|   |-- element: string (containsNull = true)
|-- state: string (nullable = true)

+-----+-----+----+
|      name |      lang | state |
+-----+-----+----+
| James,,Smith|[Java, Scala, C++]| CA |
| Michael,Rose,|[Spark, Java, C++]| NJ |
| Robert,,Williams| [CSharp, VB] | NV |
+-----+-----+----+
```

You can also change the column names by using `toDF()` function

```
columns = ["name","languagesAtSchool","currentState"]
df=spark.createDataFrame(data).toDF(*columns)
df.printSchema()
```

This yields below output, note the column name “languagesAtSchool” from the previous example.

```
root
|-- name: string (nullable = true)
|-- languagesAtSchool: array (nullable = true)
|   |-- element: string (containsNull = true)
|-- currentState: string (nullable = true)
```

5. Create Nested Struct Using Row Class

The below example provides a way to create a struct type using the Row class.

Alternatively, you can also create struct type using By [Providing Schema using PySpark StructType & StructFields](#)

```
#Create DataFrame with struct using Row class
from pyspark.sql import Row
data=[Row(name="James",prop=Row(hair="black",eye="blue")),
      Row(name="Ann",prop=Row(hair="grey",eye="black"))]
df=spark.createDataFrame(data)
df.printSchema()
```

Yields below schema

```
root
|-- name: string (nullable = true)
|-- prop: struct (nullable = true)
|   |-- hair: string (nullable = true)
|   |-- eye: string (nullable = true)
```

6. Complete Example of PySpark Row usage on RDD & DataFrame

Below is complete example for reference.

```
from pyspark.sql import SparkSession, Row

row=Row("James",40)
print(row[0] + "," +str(row[1]))
row2=Row(name="Alice", age=11)
print(row2.name)

Person = Row("name", "age")
p1=Person("James", 40)
p2=Person("Alice", 35)
print(p1.name +"," +p2.name)

#PySpark Example
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

data = [Row(name="James,,Smith",lang=["Java","Scala","C++"],state="CA"),
       Row(name="Michael,Rose,",lang=["Spark","Java","C++"],state="NJ"),
       Row(name="Robert,,Williams",lang=["CSharp","VB"],state="NV")]

#RDD Example 1
rdd=spark.sparkContext.parallelize(data)
collData=rdd.collect()
print(collData)
for row in collData:
    print(row.name + "," +str(row.lang))

# RDD Example 2
Person=Row("name","lang","state")
data = [Person("James,,Smith",["Java","Scala","C++"],"CA"),
        Person("Michael,Rose,",["Spark","Java","C++"],"NJ"),
        Person("Robert,,Williams",["CSharp","VB"],"NV")]
rdd=spark.sparkContext.parallelize(data)
collData=rdd.collect()
print(collData)
```

```
for person in collData:  
    print(person.name + "," +str(person.lang))  
  
#DataFrame Example 1  
columns = ["name","languagesAtSchool","currentState"]  
df=spark.createDataFrame(data)  
df.printSchema()  
df.show()  
  
collData=df.collect()  
print(collData)  
for row in collData:  
    print(row.name + "," +str(row.lang))  
  
#DataFrame Example 2  
columns = ["name","languagesAtSchool","currentState"]  
df=spark.createDataFrame(data).toDF(*columns)  
df.printSchema()
```

PySpark Column Class | Operators & Functions

`pyspark.sql.Column` class provides several functions to work with DataFrame to manipulate the Column values, evaluate the boolean expression to filter rows, retrieve a value or part of a value from a DataFrame column, and to work with list, map & struct columns.

In this article, I will cover how to create Column object, access them to perform operations, and finally most used PySpark Column Functions with Examples.

Related Article: [PySpark Row Class with Examples](#)

Key Points:

- PySpark Column class represents a single Column in a DataFrame.
- It provides functions that are most used to manipulate DataFrame Columns & Rows.
- Some of these Column functions evaluate a Boolean expression that can be used with `filter ()` transformation to [filter the DataFrame Rows](#).
- Provides functions to get a value from a list column by index, map value by key & index, and finally struct nested column.
- PySpark also provides additional functions `pyspark.sql.functions` that take Column object and return a Column type.

Note: Most of the `pyspark.sql.functions` return Column type hence it is very important to know the operation you can perform with Column type.

1. Create Column Class Object

One of the simplest ways to create a Column class object is by using [PySpark lit\(\) SQL function](#), this takes a literal value and returns a Column object.

```
from pyspark.sql.functions import lit  
colObj = lit("sparkbyexamples.com")
```

You can also access the Column from DataFrame by multiple ways.

```
data=[("James",23),("Ann",40)]  
df=spark.createDataFrame(data).toDF("name.fname","gender")  
df.printSchema()  
#root  
# |-- name.fname: string (nullable = true)  
# |-- gender: long (nullable = true)  
  
# Using DataFrame object (df)  
df.select(df.gender).show()  
df.select(df["gender"]).show()  
#Accessing column name with dot (with backticks)  
df.select(df[`\name.fname`]).show()  
  
#Using SQL col() function  
from pyspark.sql.functions import col  
df.select(col("gender")).show()  
#Accessing column name with dot (with backticks)  
df.select(col(`name.fname`)).show()
```

Below example demonstrates accessing struct type columns. Here I have use [PySpark Row class](#) to create a struct type. Alternatively you can also create it by using [PySpark StructType & StructField classes](#)

```
#Create DataFrame with struct using Row class  
from pyspark.sql import Row  
data=[Row(name="James",prop=Row(hair="black",eye="blue")),  
      Row(name="Ann",prop=Row(hair="grey",eye="black"))]  
df=spark.createDataFrame(data)  
df.printSchema()  
#root  
# |-- name: string (nullable = true)  
# |-- prop: struct (nullable = true)  
# |   |-- hair: string (nullable = true)  
# |   |-- eye: string (nullable = true)  
  
#Access struct column  
df.select(df.prop.hair).show()  
df.select(df["prop.hair"]).show()  
df.select(col("prop.hair")).show()
```

VN2 Solutions Pvt. Ltd.

```
#Access all columns from struct  
df.select(col("prop.*")).show()
```

2. PySpark Column Operators

PySpark column also provides a way to do arithmetic operations on columns using operators.

```
data=[(100,2,1),(200,3,4),(300,4,4)]  
df=spark.createDataFrame(data).toDF("col1","col2","col3")  
  
#Arthmetic operations  
df.select(df.col1 + df.col2).show()  
df.select(df.col1 - df.col2).show()  
df.select(df.col1 * df.col2).show()  
df.select(df.col1 / df.col2).show()  
df.select(df.col1 % df.col2).show()  
  
df.select(df.col2 > df.col3).show()  
df.select(df.col2 < df.col3).show()  
df.select(df.col2 == df.col3).show()
```

3. PySpark Column Functions

Let's see some of the most used Column Functions, on below table, I have grouped related functions together to make it easy, click on the link for examples.

COLUMN FUNCTION

FUNCTION DESCRIPTION

alias(*alias, **kwargs) name(*alias, **kwargs)	Provides alias to the column or expressions name() returns same as alias().
---	--

asc() asc_nulls_first() asc_nulls_last()	Returns ascending order of the column. asc_nulls_first() Returns null values first then non-null values. asc_nulls_last() – Returns null values after non-null values.
--	--

astype(dataType) cast(dataType)	Used to cast the data type to another type. astype() returns same as cast().
------------------------------------	---

between(lowerBound, upperBound)	Checks if the columns values are between lower and upper bound. Returns boolean value.
---------------------------------	---

COLUMN FUNCTION	FUNCTION DESCRIPTION
bitwiseAND(other) bitwiseOR(other) bitwiseXOR(other)	Compute bitwise AND, OR & XOR of this expression with another expression respectively.
contains(other)	Check if String contains in another string.
desc() desc_nulls_first() desc_nulls_last()	Returns descending order of the column. desc_nulls_first() -null values appear before non-null values. desc_nulls_last() – null values appear after non-null values.
startswith(other) endswith(other)	String starts with. Returns boolean expression String ends with. Returns boolean expression
eqNullSafe(other)	Equality test that is safe for null values.
getField(name)	Returns a field by name in a StructField and by key in Map.
getItem(key)	Returns a values from Map/Key at the provided position.
isNotNull() isNull()	isNotNull() – Returns True if the current expression is NOT null. isNull() – Returns True if the current expression is null.
isin(*cols)	A boolean expression that is evaluated to true if the value of this expression is contained by the evaluated values of the arguments.
like(other) rlike(other)	Similar to SQL like expression. Similar to SQL RLIKE expression (LIKE with Regex).
over(window)	Used with window column
substr(startPos, length)	Return a Column which is a substring of the column.
when(condition, value) otherwise(value)	Similar to SQL CASE WHEN, Executes a list of conditions and returns one of multiple possible result expressions.

COLUMN FUNCTION	FUNCTION DESCRIPTION
dropFields(*fieldNames)	Used to drops fields in StructType by name.
withField(fieldName, col)	An expression that adds/replaces a field in StructType by name.

4. PySpark Column Functions Examples

Let's create a simple DataFrame to work with PySpark SQL Column examples. For most of the examples below, I will be referring DataFrame object name (df.) to get the column.

```
data=[("James","Bond","100",None),
      ("Ann","Varsa","200",'F'),
      ("Tom Cruise","XXX","400",''),
      ("Tom Brand",None,"400",'M')]
columns=["fname","lname","id","gender"]
df=spark.createDataFrame(data,columns)
```

4.1 alias() – Set's name to Column

On below example df.fname refers to Column object and alias() is a function of the Column to give alternate name. Here, fname column has been changed to first_name & lname to last_name.

On second example I have use PySpark expr() function to concatenate columns and named column as fullName.

```
#alias
from pyspark.sql.functions import expr
df.select(df.fname.alias("first_name"), \
          df.lname.alias("last_name"))
.show()

#Another example
df.select(expr(" fname ||','|| lname").alias("fullName")) \
.show()
```

4.2 asc() & desc() – Sort the DataFrame columns by Ascending or Descending order.

```
#asc, desc to sort ascending and descending order repsectively.
df.sort(df.fname.asc()).show()
df.sort(df.fname.desc()).show()
```

4.3 cast() & astype() – Used to convert the data Type.

```
#cast  
df.select(df.fname,df.id.cast("int")).printSchema()
```

4.4 between() – Returns a Boolean expression when a column values in between lower and upper bound.

```
#between  
df.filter(df.id.between(100,300)).show()
```

4.5 contains() – Checks if a DataFrame column value contains a a value specified in this function.

```
#contains  
df.filter(df.fname.contains("Cruise")).show()
```

4.6 startswith() & endswith() – Checks if the value of the DataFrame Column starts and ends with a String respectively.

```
#startswith, endswith()  
df.filter(df.fname.startswith("T")).show()  
df.filter(df.fname.endswith("Cruise")).show()
```

4.7 eqNullSafe() –

4.8 isNull & isNotNull() – Checks if the DataFrame column has NULL or non NULL values.

```
#isNull & isNotNull  
df.filter(df.lname.isNull()).show()  
df.filter(df.lname.isNotNull()).show()
```

4.9 like() & rlike() – Similar to SQL LIKE expression

VN2 Solutions Pvt. Ltd.

```
#like , rlike  
df.select(df.fname,df.lname,df.id) \  
.filter(df.fname.like("%om"))
```

4.10 substr() – Returns a Column after getting sub string from the Column

```
df.select(df.fname.substr(1,2).alias("substr")).show()
```

4.11 when() & otherwise() – It is similar to SQL Case When, executes sequence of expressions until it matches the condition and returns a value when match.

```
#when & otherwise  
from pyspark.sql.functions import when  
df.select(df.fname,df.lname,when(df.gender=="M","Male") \  
.when(df.gender=="F","Female") \  
.when(df.gender==None , "") \  
.otherwise(df.gender).alias("new_gender") \  
).show()
```

4.12 isin() – Check if value presents in a List.

```
#isin  
li=["100","200"]  
df.select(df.fname,df.lname,df.id) \  
.filter(df.id.isin(li)) \  
.show()
```

4.13 getField() – To get the value by key from MapType column and by struct child name from StructType column

Rest of the below functions operates on List, Map & Struct data structures hence to demonstrate these I will use another DataFrame with list, map and struct columns. For more explanation how to use Arrays refer to [PySpark ArrayType Column on DataFrame Examples](#) & for map refer to [PySpark MapType Examples](#)

```
#Create DataFrame with struct, array & map  
from pyspark.sql.types import StructType,StructField,StringType,ArrayType,MapType  
data=[(("James","Bond"),["Java","C#"],{'hair':'black','eye':'brown'}),  
      ("Ann","Varsa"),[".NET","Python"],{'hair':'brown','eye':'black'}),  
      ("Tom Cruise","",["Python","Scala"],{'hair':'red','eye':'grey'}),  
      ("Tom Brand",None),["Perl","Ruby"],{'hair':'black','eye':'blue'})]
```

VN2 Solutions Pvt. Ltd.

```
schema = StructType([
    StructField('name', StructType([
        StructField('fname', StringType(), True),
        StructField('lname', StringType(), True)])),
    StructField('languages', ArrayType(StringType()),True),
    StructField('properties', MapType(StringType(),StringType()),True)
])
df=spark.createDataFrame(data,schema)
df.printSchema()

#Display's to console
root
|-- name: struct (nullable = true)
|   |-- fname: string (nullable = true)
|   |-- lname: string (nullable = true)
|-- languages: array (nullable = true)
|   |-- element: string (containsNull = true)
|-- properties: map (nullable = true)
|   |-- key: string
|   |-- value: string (valueContainsNull = true)
getField Example
```

```
#getField from MapType
df.select(df.properties.getField("hair")).show()
```

```
#getField from Struct
df.select(df.name.getField("fname")).show()
4.14 getItem() – To get the value by index from MapType or ArrayType & my key for MapType column.
```

```
#getItem() used with ArrayType
df.select(df.languages.getItem(1)).show()

#getItem() used with MapType
df.select(df.properties.getItem("hair")).show()
```

4.15 dropFields –

```
# TO-DO, getting runtime error
```

4.16 withField() –

```
# TO-DO getting runtime error
```

4.17 over() – Used with Window Functions

```
TO-DO
```

PySpark Select Columns From DataFrame

In PySpark, `select()` function is used to select single, multiple, column by index, all columns from the list and the nested columns from a DataFrame, PySpark `select()` is a transformation function hence it returns a new DataFrame with the selected columns.

- [Select a Single & Multiple Columns from PySpark](#)
- [Select All Columns From List](#)
- [Select Columns By Index](#)
- [Select a Nested Column](#)
- [Other Ways to Select Columns](#)

First, let's [create a Dataframe](#).

```
import pyspark
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()
data = [("James","Smith","USA","CA"),
       ("Michael","Rose","USA","NY"),
       ("Robert","Williams","USA","CA"),
       ("Maria","Jones","USA","FL")]
columns = ["firstname","lastname","country","state"]
df = spark.createDataFrame(data = data, schema = columns)
df.show(truncate=False)
```

1. Select Single & Multiple Columns From PySpark

You can select the single or multiple columns of the DataFrame by passing the column names you wanted to select to the `select()` function. Since DataFrame is immutable, this creates a new DataFrame with selected columns. `show()` function is used to show the Dataframe contents.

Below are ways to select single, multiple or all columns.

VN2 Solutions Pvt. Ltd.

```
df.select("firstname","lastname").show()
df.select(df.firstname,df.lastname).show()
df.select(df["firstname"],df["lastname"]).show()

#By using col() function
from pyspark.sql.functions import col
df.select(col("firstname"),col("lastname")).show()

#Select columns by regular expression
df.select(df.colRegex("^.+name$")).show()
```

2. Select All Columns From List

Sometimes you may need to select all DataFrame columns from a Python list. In the below example, we have all columns in the `columns` list object.

```
# Select All columns from List
df.select(*columns).show()

# Select All columns
df.select([col for col in df.columns]).show()
df.select("*").show()
```

3. Select Columns by Index

Using a python list features, you can select the columns by index.

```
#Selects first 3 columns and top 3 rows
df.select(df.columns[:3]).show(3)
```

```
#Selects columns 2 to 4 and top 3 rows
df.select(df.columns[2:4]).show(3)
```

4. Select Nested Struct Columns from PySpark

If you have a nested struct (StructType) column on PySpark DataFrame, you need to use an explicit column qualifier in order to select. If you are new to PySpark and you have not learned StructType yet, I would recommend skipping the rest of the section or first [Understand PySpark StructType](#) before you proceed.

First, let's create a new DataFrame with a struct type.

```
data = [
    ("James",None,"Smith"),"OH","M"),
    ("Anna","Rose","","NY","F"),
    ("Julia","","Williams"),"OH","F"),
    ("Maria","Anne","Jones"),"NY","M"),
```

VN2 Solutions Pvt. Ltd.

```
((("Jen","Mary","Brown"),"NY","M"),
(("Mike","Mary","Williams"),"OH","M")
]

from pyspark.sql.types import StructType,StructField, StringType
schema = StructType([
    StructField('name', StructType([
        StructField('firstname', StringType(), True),
        StructField('middlename', StringType(), True),
        StructField('lastname', StringType(), True)
    ])),
    StructField('state', StringType(), True),
    StructField('gender', StringType(), True)
])
df2 = spark.createDataFrame(data = data, schema = schema)
df2.printSchema()
df2.show(truncate=False) # shows all columns
```

Yields below schema output. If you notice the `column name` is a `struct type` which consists of `columns firstname, middlename, lastname`.

```
root
|-- name: struct (nullable = true)
|   |-- firstname: string (nullable = true)
|   |-- middlename: string (nullable = true)
|   |-- lastname: string (nullable = true)
|-- state: string (nullable = true)
|-- gender: string (nullable = true)

+-----+-----+
|name      |state|gender|
+-----+-----+
|[James,, Smith] |OH  |M   |
|[Anna, Rose, ] |NY  |F   |
|[Julia, , Williams] |OH  |F   |
|[Maria, Anne, Jones] |NY  |M   |
|[Jen, Mary, Brown] |NY  |M   |
|[Mike, Mary, Williams]|OH  |M   |
+-----+-----+
```

Now, let's select struct column.

VN2 Solutions Pvt. Ltd.

```
df2.select("name").show(truncate=False)
```

This returns struct column name as is.

```
+-----+  
|name      |  
+-----+  
|[James,, Smith]   |  
|[Anna, Rose, ]    |  
|[Julia, , Williams] |  
|[Maria, Anne, Jones] |  
|[Jen, Mary, Brown]  |  
|[Mike, Mary, Williams]|  
+-----+
```

In order to select the specific column from a nested struct, you need to explicitly qualify the nested struct column name.

```
df2.select("name.firstname","name.lastname").show(truncate=False)
```

This outputs `firstname` and `lastname` from the name struct column.

```
+-----+-----+  
|firstname|lastname|  
+-----+-----+  
|James   |Smith   |  
|Anna    |         |  
|Julia   |Williams|  
|Maria   |Jones   |  
|Jen     |Brown   |  
|Mike    |Williams|  
+-----+-----+
```

In order to get all columns from struct column.

```
df2.select("name.*").show(truncate=False)
```

This yields below output.

```
+-----+-----+-----+  
|firstname|middlename|lastname|  
+-----+-----+-----+  
|James   |null      |Smith   |  
|Anna    |Rose      |         |  
|Julia   |          |Williams|
```

VN2 Solutions Pvt. Ltd.

Maria	Anne	Jones	
Jen	Mary	Brown	
Mike	Mary	Williams	
+-----+	+-----+	+-----+	

5. Complete Example

```
import pyspark
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

data = [("James","Smith","USA","CA"),
        ("Michael","Rose","USA","NY"),
        ("Robert","Williams","USA","CA"),
        ("Maria","Jones","USA","FL")
       ]

columns = ["firstname","lastname","country","state"]
df = spark.createDataFrame(data = data, schema = columns)
df.show(truncate=False)

df.select("firstname").show()

df.select("firstname","lastname").show()

#Using Dataframe object name
df.select(df.firstname,df.lastname).show()

# Using col function
from pyspark.sql.functions import col
df.select(col("firstname"),col("lastname")).show()

data = [((("James",None,"Smith"),"OH","M"),
          ((("Anna","Rose","","NY","F"),
            (("Julia","","Williams"),"OH","F"),
            (("Maria","Anne","Jones"),"NY","M"),
            (("Jen","Mary","Brown"),"NY","M"),
            (("Mike","Mary","Williams"),"OH","M")
           )
         ]
         )

from pyspark.sql.types import StructType,StructField, StringType
schema = StructType([
    StructField('name', StructType([

```

```
StructField('firstname', StringType(), True),
StructField('middlename', StringType(), True),
StructField('lastname', StringType(), True)
]),
StructField('state', StringType(), True),
StructField('gender', StringType(), True)
])

df2 = spark.createDataFrame(data = data, schema = schema)
df2.printSchema()
df2.show(truncate=False) # shows all columns

df2.select("name").show(truncate=False)
df2.select("name.firstname","name.lastname").show(truncate=False)
df2.select("name.*").show(truncate=False)
```

PySpark Collect() – Retrieve data from DataFrame

PySpark RDD/DataFrame `collect()` is an action operation that is used to retrieve all the elements of the dataset (from all nodes) to the driver node. We should use the `collect()` on smaller dataset usually after `filter()`, `group()` e.t.c. Retrieving larger datasets results in `OutOfMemory` error.

In this PySpark article, I will explain the usage of `collect()` with DataFrame example, when to avoid it, and the difference between `collect()` and `select()`.

Related Articles:

- [How to Iterate PySpark DataFrame through Loop](#)
- [How to Convert PySpark DataFrame Column to Python List](#)

In order to explain with example, first, let's [create a DataFrame](#).

```
import pyspark
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

dept = [("Finance",10), \
         ("Marketing",20), \
         ("Sales",30), \
         ("IT",40) \
        ]
deptColumns = ["dept_name", "dept_id"]
```

VN2 Solutions Pvt. Ltd.

```
deptDF = spark.createDataFrame(data=dept, schema = deptColumns)
deptDF.show(truncate=False)
```

show() function on DataFrame prints the result of DataFrame in a table format. By default, it shows only 20 rows. The above snippet returns the data in a table.

```
+-----+-----+
|dept_name|dept_id|
+-----+-----+
|Finance |10    |
|Marketing|20    |
|Sales   |30    |
|IT      |40    |
+-----+-----+
```

Now, let's use the `collect()` to retrieve the data.

```
dataCollect = deptDF.collect()
print(dataCollect)
```

`deptDF.collect()` retrieves all elements in a DataFrame as an Array of Row type to the driver node. printing a resultant array yields the below output.

```
[Row(dept_name='Finance', dept_id=10),
Row(dept_name='Marketing', dept_id=20),
Row(dept_name='Sales', dept_id=30),
Row(dept_name='IT', dept_id=40)]
```

Note that `collect()` is an action hence it does not return a DataFrame instead, it returns data in an Array to the driver. Once the data is in an array, you can use `python for loop` to process it further.

```
for row in dataCollect:
    print(row['dept_name'] + "," +str(row['dept_id']))
```

If you wanted to get first row and first column from a DataFrame.

```
#Returns value of First Row, First Column which is "Finance"
deptDF.collect()[0][0]
```

VN2 Solutions Pvt. Ltd.

Let's understand what's happening on above statement.

- `deptDF.collect()` returns Array of Row type.
- `deptDF.collect()[0]` returns the first element in an array (1st row).
- `deptDF.collect[0][0]` returns the value of the first row & first column.

In case you want to just return certain elements of a DataFrame, you should call PySpark select() transformation first.

```
dataCollect = deptDF.select("dept_name").collect()
```

When to avoid Collect()

Usually, `collect()` is used to retrieve the action output when you have very small result set and calling `collect()` on an RDD/DataFrame with a bigger result set causes out of memory as it returns the entire dataset (from all workers) to the driver hence we should avoid calling `collect()` on a larger dataset.

collect () vs select ()

`select()` is a transformation that returns a new DataFrame and holds the columns that are selected whereas `collect()` is an action that returns the entire data set in an Array to the driver.

Complete Example of PySpark collect()

Below is complete PySpark example of using `collect()` on DataFrame, similarly you can also create a program using `collect()` with RDD.

```
import pyspark
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

dept = [("Finance",10), \
        ("Marketing",20), \
        ("Sales",30), \
        ("IT",40) \
      ]
```

VN2 Solutions Pvt. Ltd.

```
deptColumns = ["dept_name","dept_id"]
deptDF = spark.createDataFrame(data=dept, schema = deptColumns)
deptDF.printSchema()
deptDF.show(truncate=False)

dataCollect = deptDF.collect()

print(dataCollect)

dataCollect2 = deptDF.select("dept_name").collect()
print(dataCollect2)

for row in dataCollect:
    print(row['dept_name'] + "," +str(row['dept_id']))
```

PySpark withColumn() Usage with Examples

PySpark withColumn() is a transformation function of DataFrame which is used to change the value, convert the datatype of an existing column, create a new column, and many more. In this post, I will walk you through commonly used PySpark DataFrame column operations using withColumn() examples.

- [PySpark withColumn – To change column DataType](#)
- [Transform/change value of an existing column](#)
- [Derive new column from an existing column](#)
- [Add a column with the literal value](#)
- [Rename column name](#)
- [Drop DataFrame column](#)

First, let's create a DataFrame to work with.

```
data = [('James','','Smith','1991-04-01','M',3000),
        ('Michael','Rose','','2000-05-19','M',4000),
        ('Robert','','Williams','1978-09-05','M',4000),
        ('Maria','Anne','Jones','1967-12-01','F',4000),
        ('Jen','Mary','Brown','1980-02-17','F",-1)
       ]

columns = ["firstname","middlename","lastname","dob","gender","salary"]
```

VN2 Solutions Pvt. Ltd.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()
df = spark.createDataFrame(data=data, schema = columns)
```

1. Change DataType using PySpark withColumn()

By using PySpark `withColumn()` on a DataFrame, we can cast or change the data type of a column. In order to change data type, you would also need to use `cast()` function along with `withColumn()`. The below statement changes the datatype from String to Integer for the `salary` column.

```
df.withColumn("salary",col("salary").cast("Integer")).show()
```

2. Update The Value of an Existing Column

PySpark `withColumn()` function of DataFrame can also be used to change the value of an existing column. In order to change the value, pass an existing column name as a first argument and a value to be assigned as a second argument to the `withColumn()` function. Note that the second argument should be `Column` type . Also, see Different Ways to Update PySpark DataFrame Column.

```
df.withColumn("salary",col("salary")*100).show()
```

This snippet multiplies the value of “salary” with 100 and updates the value back to “salary” column.

3. Create a Column from an Existing

To add/create a new column, specify the first argument with a name you want your new column to be and use the second argument to assign a value by applying an operation on an existing column. Also, see Different Ways to Add New Column to PySpark DataFrame.

```
df.withColumn("CopiedColumn",col("salary")* -1).show()
```

This snippet creates a new column “CopiedColumn” by multiplying “salary” column with value -1.

4. Add a New Column using withColumn()

In order to create a new column, pass the column name you wanted to the first argument of `withColumn()` transformation function. Make sure this new column not already present on DataFrame, if it presents it updates the value of that column.

On below snippet, PySpark lit() function is used to add a constant value to a DataFrame column. We can also chain in order to add multiple columns.

```
df.withColumn("Country", lit("USA")).show()
df.withColumn("Country", lit("USA")) \
```

```
.withColumn("anotherColumn",lit("anotherValue")) \  
.show()
```

5. Rename Column Name

Though you cannot rename a column using withColumn, still I wanted to cover this as renaming is one of the common operations we perform on DataFrame. To rename an existing column use withColumnRenamed() function on DataFrame.

```
df.withColumnRenamed("gender","sex") \  
.show(truncate=False)
```

6. Drop Column From PySpark DataFrame

Use “drop” function to drop a specific column from the DataFrame.

```
df.drop("salary") \  
.show()
```

Note: Note that all of these functions return the new DataFrame after applying the functions instead of updating DataFrame.

7. PySpark withColumn() Complete Example

```
import pyspark  
from pyspark.sql import SparkSession  
from pyspark.sql.functions import col, lit  
from pyspark.sql.types import StructType, StructField, StringType, IntegerType  
  
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()  
  
data = [('James','','Smith','1991-04-01','M',3000),  
        ('Michael','Rose','','2000-05-19','M',4000),  
        ('Robert','','Williams','1978-09-05','M',4000),  
        ('Maria','Anne','Jones','1967-12-01','F',4000),  
        ('Jen','Mary','Brown','1980-02-17','F",-1)  
]  
  
columns = ["firstname","middlename","lastname","dob","gender","salary"]  
df = spark.createDataFrame(data=data, schema = columns)  
df.printSchema()  
df.show(truncate=False)  
  
df2 = df.withColumn("salary",col("salary").cast("Integer"))  
df2.printSchema()  
df2.show(truncate=False)  
  
df3 = df.withColumn("salary",col("salary")*100)
```

VN2 Solutions Pvt. Ltd.

```
df3.printSchema()
df3.show(truncate=False)

df4 = df.withColumn("CopiedColumn", col("salary") * -1)
df4.printSchema()

df5 = df.withColumn("Country", lit("USA"))
df5.printSchema()

df6 = df.withColumn("Country", lit("USA")) \
    .withColumn("anotherColumn", lit("anotherValue"))
df6.printSchema()

df.withColumnRenamed("gender", "sex") \
    .show(truncate=False)

df4.drop("CopiedColumn") \
    .show(truncate=False)
```

PySpark withColumnRenamed to Rename Column on DataFrame

Use PySpark `withColumnRenamed()` to rename a DataFrame column, we often need to rename one column or multiple (or all) columns on PySpark DataFrame, you can do this in several ways. When columns are nested it becomes complicated.

Since DataFrame's are an immutable collection, you can't rename or update a column instead when using `withColumnRenamed()` it creates a new DataFrame with updated column names, In this PySpark article, I will cover different ways to rename columns with several use cases like rename nested column, all columns, selected multiple columns with Python/PySpark examples.

Refer to this page, If you are looking for a [Spark with Scala example](#) and [rename pandas column with examples](#)

1. [PySpark withColumnRenamed – To rename DataFrame column name](#)
2. [PySpark withColumnRenamed – To rename multiple columns](#)
3. [Using StructType – To rename nested column on PySpark DataFrame](#)
4. [Using Select – To rename nested columns](#)
5. [Using withColumn – To rename nested columns](#)
6. [Using col\(\) function – To Dynamically rename all or multiple columns](#)
7. [Using toDF\(\) – To rename all or multiple columns](#)

First, let's create our data set to work with.

```
dataDF = [(['James','','Smith'),'1991-04-01','M',3000),
          ('Michael','Rose',''),'2000-05-19','M',4000),
```

VN2 Solutions Pvt. Ltd.

```
(('Robert','','Williams'),'1978-09-05','M',4000),  
  (('Maria','Anne','Jones'),'1967-12-01','F',4000),  
  (('Jen','Mary','Brown'),'1980-02-17','F',-1)  
]
```

Our base schema with nested structure.

```
from pyspark.sql.types import StructType,StructField, StringType, IntegerType  
schema = StructType([  
    StructField('name', StructType([  
        StructField('firstname', StringType(), True),  
        StructField('middlename', StringType(), True),  
        StructField('lastname', StringType(), True)  
    ])),  
    StructField('dob', StringType(), True),  
    StructField('gender', StringType(), True),  
    StructField('salary', IntegerType(), True)  
])
```

Let's create the DataFrame by using parallelize and provide the above schema.

```
from pyspark.sql import SparkSession  
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()  
df = spark.createDataFrame(data = dataDF, schema = schema)  
df.printSchema()
```

Below is our schema structure. I am not printing data here as it is not necessary for our examples. This schema has a nested structure.

```
root  
  |-- name: struct (nullable = true)  
  |    |-- firstname: string (nullable = true)  
  |    |-- middlename: string (nullable = true)  
  |    |-- lastname: string (nullable = true)  
  |-- dob: string (nullable = true)  
  |-- gender: string (nullable = true)  
  |-- salary: integer (nullable = true)
```

1. PySpark withColumnRenamed – To rename DataFrame column name

PySpark has a `withColumnRenamed()` function on DataFrame to change a column name. This is the most straight forward approach; this function takes two parameters; the first is your existing column name and the second is the new column name you wish for.

PySpark withColumnRenamed() Syntax:

```
withColumnRenamed(existingName, newName)
```

`existingName` – The existing column name you want to change

`newName` – New name of the column

Returns a new DataFrame with a column renamed.

Example

```
df.withColumnRenamed("dob","DateOfBirth").printSchema()
```

The above statement changes column “dob” to “DateOfBirth” on PySpark DataFrame.

Note that `withColumnRenamed` function returns a new DataFrame and doesn’t modify the current DataFrame.

```
root
|-- name: struct (nullable = true)
|   |-- firstname: string (nullable = true)
|   |-- middlename: string (nullable = true)
|   |-- lastname: string (nullable = true)
|-- DateOfBirth: string (nullable = true)
|-- gender: string (nullable = true)
|-- salary: integer (nullable = true)
```

2. PySpark withColumnRenamed – To rename multiple columns

To change multiple column names, we should chain `withColumnRenamed` functions as shown below. You can also store all columns to rename in a list and loop through to rename all columns, I will leave this to you to explore.

```
df2 = df.withColumnRenamed("dob","DateOfBirth") \
    .withColumnRenamed("salary","salary_amount")
df2.printSchema()
```

This creates a new DataFrame “df2” after renaming dob and salary columns.

VN2 Solutions Pvt. Ltd.

3. Using PySpark StructType – To rename a nested column in Dataframe

Changing a column name on nested data is not straight forward and we can do this by creating a new schema with new DataFrame columns using StructType and use it using cast function as shown below.

```
schema2 = StructType([
    StructField("fname",StringType()),
    StructField("middlename",StringType()),
    StructField("lname",StringType())])

df.select(col("name").cast(schema2), \
    col("dob"), col("gender"),col("salary")) \
    .printSchema()
```

This statement renames `firstname` to `fname` and `lastname` to `lname` within name structure.

```
root
| -- name: struct (nullable = true)
|   | -- fname: string (nullable = true)
|   | -- middlename: string (nullable = true)
|   | -- lname: string (nullable = true)
| -- dob: string (nullable = true)
| -- gender: string (nullable = true)
| -- salary: integer (nullable = true)
```

4. Using Select – To rename nested elements.

Let's see another way to change nested columns by transposing the structure to flat.

```
from pyspark.sql.functions import *
df.select(col("name.firstname").alias("fname"), \
    col("name.middlename").alias("mname"), \
    col("name.lastname").alias("lname"), \
    col("dob"),col("gender"),col("salary")) \
    .printSchema()
```

5. Using PySpark DataFrame withColumn – To rename nested columns

When you have nested columns on PySpark DataFrame and if you want to rename it, use `withColumn` on a data frame object to create a new column from an existing and we will need to drop the existing column. Below example creates a “fname” column from “name.firstname” and drops the “name” column

```
from pyspark.sql.functions import *
df4 = df.withColumn("fname", col("name.firstname")) \
    .withColumn("mname", col("name.middlename")) \
    .withColumn("lname", col("name.lastname")) \
    .drop("name")
df4.printSchema()
```

6. Using col() function – To Dynamically rename all or multiple columns

Another way to change all column names on Dataframe is to use `col()` function.

IN progress

7. Using toDF() – To change all columns in a PySpark DataFrame

When we have data in a flat structure (without nested) , use `toDF()` with a new schema to change all column names.

```
newColumns = ["newCol1", "newCol2", "newCol3", "newCol4"]
df.toDF(*newColumns).printSchema()
```

[Source code](#)

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

spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

dataDF = [(['James','','Smith'),'1991-04-01','M',3000),
          ('Michael','Rose',''), '2000-05-19', 'M', 4000),
          ('Robert','','Williams'), '1978-09-05', 'M', 4000),
          ('Maria','Anne','Jones'), '1967-12-01', 'F', 4000),
          ('Jen','Mary','Brown'), '1980-02-17', 'F', -1)
]
```

VN2 Solutions Pvt. Ltd.

```
schema = StructType([
    StructField('name', StructType([
        StructField('firstname', StringType(), True),
        StructField('middlename', StringType(), True),
        StructField('lastname', StringType(), True)
    ])),
    StructField('dob', StringType(), True),
    StructField('gender', StringType(), True),
    StructField('salary', IntegerType(), True)
])

df = spark.createDataFrame(data = dataDF, schema = schema)
df.printSchema()

# Example 1
df.withColumnRenamed("dob","DateOfBirth").printSchema()
# Example 2
df2 = df.withColumnRenamed("dob","DateOfBirth") \
    .withColumnRenamed("salary","salary_amount")
df2.printSchema()

# Example 3
schema2 = StructType([
    StructField("fname",StringType()),
    StructField("middlename",StringType()),
    StructField("lname",StringType())])

df.select(col("name").cast(schema2),
    col("dob"),
    col("gender"),
    col("salary")) \
    .printSchema()

# Example 4
df.select(col("name.firstname").alias("fname"),
    col("name.middlename").alias("mname"),
    col("name.lastname").alias("lname"),
    col("dob"),col("gender"),col("salary")) \
    .printSchema()

# Example 5
df4 = df.withColumn("fname",col("name.firstname")) \
    .withColumn("mname",col("name.middlename")) \
    .withColumn("lname",col("name.lastname")) \
    .drop("name")
```

```
df4.printSchema()

#Example 7
newColumns = ["newCol1","newCol2","newCol3","newCol4"]
df.toDF(*newColumns).printSchema()

# Example 6
...
not working
old_columns = Seq("dob","gender","salary","fname","mname","lname")
new_columns =
Seq("DateOfBirth","Sex","salary","firstName","middleName","lastName")
columnsList = old_columns.zip(new_columns).map(f=>{col(f._1).as(f._2)})
df5 = df4.select(columnsList:_*)
df5.printSchema()
...
```

PySpark Where Filter Function | Multiple Conditions

PySpark `filter()` function is used to filter the rows from RDD/DataFrame based on the given condition or SQL expression, you can also use `where()` clause instead of the `filter()` if you are coming from an SQL background, both these functions operate exactly the same.

In this PySpark article, you will learn how to apply a filter on DataFrame columns of string, arrays, struct types by using single and multiple conditions and also applying filter using `isin()` with PySpark (Python Spark) examples.

Related Article:

- [How to Filter Rows with NULL/NONE \(IS NULL & IS NOT NULL\) in PySpark](#)
- [Spark Filter – startsWith\(\), endsWith\(\) Examples](#)
- [Spark Filter – contains\(\), like\(\), rlike\(\) Examples](#)

Note: [PySpark Column Functions](#) provides several options that can be used with `filter()`.

1. PySpark DataFrame filter() Syntax

Below is syntax of the filter function. condition would be an expression you wanted to filter.

```
filter(condition)
```

VN2 Solutions Pvt. Ltd.

Before we start with examples, first let's [create a DataFrame](#). Here, I am using a [DataFrame with StructType](#) and [ArrayType](#) columns as I will also be covering examples with struct and array types as-well.

```
from pyspark.sql.types import StructType,StructField
from pyspark.sql.types import StringType, IntegerType, ArrayType
data = [
    ("James","","Smith"),["Java","Scala","C++"],"OH","M"),
    ("Anna","Rose","",["Spark","Java","C++"],"NY","F"),
    ("Julia","","Williams"),["CSharp","VB"],"OH","F"),
    ("Maria","Anne","Jones"),["CSharp","VB"],"NY","M"),
    ("Jen","Mary","Brown"),["CSharp","VB"],"NY","M"),
    ("Mike","Mary","Williams"),["Python","VB"],"OH","M")
]

schema = StructType([
    StructField('name', StructType([
        StructField('firstname', StringType(), True),
        StructField('middlename', StringType(), True),
        StructField('lastname', StringType(), True)
    ])),
    StructField('languages', ArrayType(StringType()), True),
    StructField('state', StringType(), True),
    StructField('gender', StringType(), True)
])

df = spark.createDataFrame(data = data, schema = schema)
df.printSchema()
df.show(truncate=False)
```

This yields below schema and DataFrame results.

```
root
|-- name: struct (nullable = true)
|   |-- firstname: string (nullable = true)
|   |-- middlename: string (nullable = true)
|   |-- lastname: string (nullable = true)
|-- languages: array (nullable = true)
|   |-- element: string (containsNull = true)
|-- state: string (nullable = true)
|-- gender: string (nullable = true)

+-----+-----+-----+
```

VN2 Solutions Pvt. Ltd.

```
+-----+-----+-----+
|name      |languages    |state|gender|
+-----+-----+-----+
|[James, , Smith]  |[Java, Scala, C++]|OH  |M   | |
|[Anna, Rose, ]   |[Spark, Java, C++]|NY  |F   |
|[Julia, , Williams]|[CSharp, VB]  |OH  |F   |
|[Maria, Anne, Jones]|[CSharp, VB]  |NY  |M   |
|[Jen, Mary, Brown]|[CSharp, VB]  |NY  |M   |
|[Mike, Mary, Williams]||[Python, VB] |OH  |M   |
+-----+-----+-----+
```

2. DataFrame filter() with Column Condition

Use Column with the condition to filter the rows from DataFrame, using this you can express complex condition by referring column names using dfObject.colname

```
# Using equals condition
df.filter(df.state == "OH").show(truncate=False)

+-----+-----+-----+
|name      |languages    |state|gender|
+-----+-----+-----+
|[James, , Smith]  |[Java, Scala, C++]|OH  |M   | |
|[Julia, , Williams]|[CSharp, VB]  |OH  |F   |
|[Mike, Mary, Williams]||[Python, VB] |OH  |M   |
+-----+-----+-----+

# not equals condition
df.filter(df.state != "OH") \
.show(truncate=False)
df.filter(~(df.state == "OH")) \
.show(truncate=False)
```

Same example can also written as below. In order to use this first you need to import from pyspark.sql.functions import col

```
#Using SQL col() function
from pyspark.sql.functions import col
df.filter(col("state") == "OH") \
.show(truncate=False)
```

3. DataFrame filter() with SQL Expression

If you are coming from SQL background, you can use that knowledge in PySpark to filter DataFrame rows with SQL expressions.

```
#Using SQL Expression
df.filter("gender == 'M'").show()
```

VN2 Solutions Pvt. Ltd.

```
#For not equal  
df.filter("gender != 'M'").show()  
df.filter("gender <> 'M'").show()
```

4. PySpark Filter with Multiple Conditions

In PySpark, to filter() rows on DataFrame based on multiple conditions, you can use either `Column` with a condition or SQL expression. Below is just a simple example using AND (&) condition, you can extend this with OR(||), and NOT(!) conditional expressions as needed.

```
//Filter multiple condition  
df.filter( (df.state == "OH") & (df.gender == "M") ) \  
.show(truncate=False)
```

This yields below DataFrame results.

```
+-----+-----+-----+  
|name      |languages     |state|gender|  
+-----+-----+-----+  
|[James, , Smith] |[Java, Scala, C++]|OH  |M   |  
|[Mike, Mary, Williams]|[Python, VB]    |OH  |M   |  
+-----+-----+-----+
```

5. Filter Based on List Values

If you have a list of elements and you wanted to filter that is not in the list or in the list, use `isin()` function of `Column class` and it doesn't have `isnotin()` function but you do the same using not operator (~)

```
#Filter IS IN List values  
li=["OH","CA","DE"]  
df.filter(df.state.isin(li)).show()  
+-----+-----+-----+  
|       name|     languages|state|gender|  
+-----+-----+-----+  
| [James, , Smith]|[Java, Scala, C++]| OH| M|  
| [Julia, , Williams]| [CSharp, VB]| OH| F|  
|[Mike, Mary, Will...]| [Python, VB]| OH| M|  
+-----+-----+-----+
```

```
# Filter NOT IS IN List values
#These show all records with NY (NY is not part of the list)
df.filter(~df.state.isin(li)).show()
df.filter(df.state.isin(li)==False).show()
```

6. Filter Based on Starts With, Ends With, Contains

You can also filter DataFrame rows by using `startswith()`, `endswith()` and `contains()` methods of Column class. For more examples on Column class, refer to [PySpark Column Functions](#).

```
# Using startswith
df.filter(df.state.startswith("N")).show()
+-----+-----+-----+
|      name|languages|state|gender|
+-----+-----+-----+
| [Anna, Rose, ]|[Spark, Java, C++]|  NY|   F|
|[Maria, Anne, Jones]| [CSharp, VB]|  NY|   M|
| [Jen, Mary, Brown]| [CSharp, VB]|  NY|   M|
+-----+-----+-----+

#using endswith
df.filter(df.state.endswith("H")).show()

#contains
df.filter(df.state.contains("H")).show()
```

7. PySpark Filter like and rlike

If you have SQL background you must be familiar with `like` and `rlike` (regex like), PySpark also provides similar methods in Column class to filter similar values using wildcard characters. You can use `rlike()` to filter by checking values case insensitive.

```
data2 = [(2,"Michael Rose"),(3,"Robert Williams"),
        (4,"Rames Rose"),(5,"Rames rose")]
df2 = spark.createDataFrame(data = data2, schema = ["id","name"])

# like - SQL LIKE pattern
df2.filter(df2.name.like("%rose%")).show()
+---+-----+
| id|    name|
+---+-----+
|  5|Rames rose|
+---+-----+
```

VN2 Solutions Pvt. Ltd.

```
# rlike - SQL RLIKE pattern (LIKE with Regex)
#This check case insensitive
df2.filter(df2.name.rlike("(?i)^*rose$")).show()
+---+-----+
| id |      name |
+---+-----+
| 2 |Michael Rose|
| 4 | Rames Rose|
| 5 | Rames rose|
```

8. Filter on an Array column

When you want to filter rows from DataFrame based on value present in an array collection column, you can use the first syntax. The below example uses `array_contains()` from [Pyspark SQL functions](#) which checks if a value contains in an array if present it returns true otherwise false.

```
from pyspark.sql.functions import array_contains
df.filter(array_contains(df.languages,"Java")) \
.show(truncate=False)
```

This yields below DataFrame results.

```
+-----+-----+-----+
|name      |languages      |state|gender|
+-----+-----+-----+
|[James, , Smith]||[Java, Scala, C++]|OH  |M   |
|[Anna, Rose, ] ||[Spark, Java, C++]|NY  |F   |
+-----+-----+-----+
```

9. Filtering on Nested Struct columns

If your DataFrame consists of nested struct columns, you can use any of the above syntaxes to filter the rows based on the nested column.

```
//Struct condition
df.filter(df.name.lastname == "Williams") \
.show(truncate=False)
```

This yields below DataFrame results

```
+-----+-----+-----+
|name          |languages  |state|gender|
+-----+-----+-----+
|[Julia, , Williams] ||[CSharp, VB]|OH  |F   |
|[Mike, Mary, Williams]||[Python, VB]|OH  |M   |
+-----+-----+-----+
```

10. Source code of PySpark where filter

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType,StructField, StringType, IntegerType,
ArrayType
from pyspark.sql.functions import col,array_contains

spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

arrayStructureData = [
    (("James","","Smith"),["Java","Scala","C++"],"OH","M"),
    (("Anna","Rose","",""),["Spark","Java","C++"],"NY","F"),
    (("Julia","","Williams"),["CSharp","VB"],"OH","F"),
    (("Maria","Anne","Jones"),["CSharp","VB"],"NY","M"),
    (("Jen","Mary","Brown"),["CSharp","VB"],"NY","M"),
    (("Mike","Mary","Williams"),["Python","VB"],"OH","M")
]

arrayStructureSchema = StructType([
    StructField('name', StructType([
        StructField('firstname', StringType(), True),
        StructField('middlename', StringType(), True),
        StructField('lastname', StringType(), True)
    ])),
    StructField('languages', ArrayType(StringType()), True),
    StructField('state', StringType(), True),
    StructField('gender', StringType(), True)
])

df = spark.createDataFrame(data = arrayStructureData, schema =
arrayStructureSchema)
df.printSchema()
df.show(truncate=False)

df.filter(df.state == "OH") \
.show(truncate=False)

df.filter(col("state") == "OH") \
.show(truncate=False)
```

```
df.filter("gender == 'M'") \  
.show(truncate=False)  
  
df.filter( (df.state == "OH") & (df.gender == "M") ) \  
.show(truncate=False)  
  
df.filter(array_contains(df.languages,"Java")) \  
.show(truncate=False)  
  
df.filter(df.name.lastname == "Williams") \  
.show(truncate=False)
```

Examples explained here are also available at [PySpark examples GitHub](#) project for reference.

PySpark – Distinct to Drop Duplicate Rows

PySpark `distinct()` function is used to drop/remove the duplicate rows (all columns) from DataFrame and `dropDuplicates()` is used to drop rows based on selected (one or multiple) columns. In this article, you will learn how to use `distinct()` and `dropDuplicates()` functions with PySpark example.

Before we start, first let's [create a DataFrame](#) with some duplicate rows and values on a few columns. We use this DataFrame to demonstrate how to get distinct multiple columns.

```
import pyspark  
from pyspark.sql import SparkSession  
from pyspark.sql.functions import expr  
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()  
  
data = [("James", "Sales", 3000), \  
       ("Michael", "Sales", 4600), \  
       ("Robert", "Sales", 4100), \  
       ("Maria", "Finance", 3000), \  
       ("James", "Sales", 3000), \  
       ("Scott", "Finance", 3300), \  
       ("Jen", "Finance", 3900), \  
       ("Jeff", "Marketing", 3000), \  
       ("Kumar", "Marketing", 2000), \  
       ("Saif", "Sales", 4100) \  
       ]  
columns= ["employee_name", "department", "salary"]  
df = spark.createDataFrame(data = data, schema = columns)  
df.printSchema()  
df.show(truncate=False)
```

VN2 Solutions Pvt. Ltd.

Yields below output

```
+-----+-----+-----+
|employee_name|department|salary|
+-----+-----+-----+
|James      |Sales     |3000   |
|Michael    |Sales     |4600   |
|Robert     |Sales     |4100   |
|Maria      |Finance   |3000   |
|James      |Sales     |3000   |
|Scott      |Finance   |3300   |
|Jen        |Finance   |3900   |
|Jeff       |Marketing |3000   |
|Kumar      |Marketing |2000   |
|Saif       |Sales     |4100   |
+-----+-----+-----+
```

On the above table, record with employer name Robert has duplicate rows, As you notice we have 2 rows that have duplicate values on all columns and we have 4 rows that have duplicate values on department and salary columns.

1. Get Distinct Rows (By Comparing All Columns)

On the above DataFrame, we have a total of 10 rows with 2 rows having all values duplicated, performing distinct on this DataFrame should get us 9 after removing 1 duplicate row.

```
distinctDF = df.distinct()
print("Distinct count: "+str(distinctDF.count()))
distinctDF.show(truncate=False)
```

distinct() function on DataFrame returns a new DataFrame after removing the duplicate records. This example yields the below output.

```
Distinct count: 9
+-----+-----+-----+
|employee_name|department|salary|
+-----+-----+-----+
|James      |Sales     |3000   |
|Michael    |Sales     |4600   |
|Maria      |Finance   |3000   |
|Robert     |Sales     |4100   |
|Saif       |Sales     |4100   |
|Scott      |Finance   |3300   |
```

VN2 Solutions Pvt. Ltd.

```
|Jeff      |Marketing |3000 |
|Jen       |Finance  |3900 |
|Kumar     |Marketing |2000 |
+-----+-----+-----+
```

Alternatively, you can also run `dropDuplicates()` function which returns a new DataFrame after removing duplicate rows.

```
df2 = df.dropDuplicates()
print("Distinct count: "+str(df2.count()))
df2.show(truncate=False)
```

2. PySpark Distinct of Selected Multiple Columns

PySpark doesn't have a `distinct` method which takes columns that should run `distinct` on (drop duplicate rows on selected multiple columns) however, it provides another signature of `dropDuplicates()` function which takes multiple columns to eliminate duplicates.

Note that calling `dropDuplicates()` on DataFrame returns a new DataFrame with duplicate rows removed.

```
dropDisDF = df.dropDuplicates(["department","salary"])
print("Distinct count of department & salary : "+str(dropDisDF.count()))
dropDisDF.show(truncate=False)
```

Yields below output. If you notice the output, It dropped 2 records that are duplicate.

```
Distinct count of department & salary : 8
+-----+-----+-----+
|employee_name|department|salary|
+-----+-----+-----+
|Jen       |Finance  |3900 |
|Maria     |Finance  |3000 |
|Scott     |Finance  |3300 |
|Michael   |Sales    |4600 |
|Kumar     |Marketing |2000 |
|Robert    |Sales    |4100 |
|James     |Sales    |3000 |
|Jeff      |Marketing |3000 |
+-----+-----+-----+
```

3. Source Code to Get Distinct Rows

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import expr
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

data = [("James", "Sales", 3000), \
        ("Michael", "Sales", 4600), \
        ("Robert", "Sales", 4100), \
        ("Maria", "Finance", 3000), \
        ("James", "Sales", 3000), \
        ("Scott", "Finance", 3300), \
        ("Jen", "Finance", 3900), \
        ("Jeff", "Marketing", 3000), \
        ("Kumar", "Marketing", 2000), \
        ("Saif", "Sales", 4100) \
    ]
columns= ["employee_name", "department", "salary"]
df = spark.createDataFrame(data = data, schema = columns)
df.printSchema()
df.show(truncate=False)

#Distinct
distinctDF = df.distinct()
print("Distinct count: "+str(distinctDF.count()))
distinctDF.show(truncate=False)

#Drop duplicates
df2 = df.dropDuplicates()
print("Distinct count: "+str(df2.count()))
df2.show(truncate=False)

#Drop duplicates on selected columns
dropDisDF = df.dropDuplicates(["department","salary"])
print("Distinct count of department salary : "+str(dropDisDF.count()))
dropDisDF.show(truncate=False)
}
```

VN2 Solutions Pvt. Ltd.

PySpark orderBy() and sort() explained

You can use either `sort()` or `orderBy()` function of PySpark DataFrame to sort DataFrame by ascending or descending order based on single or multiple columns, you can also do sorting using PySpark SQL sorting functions, In this article, I will explain all these different ways using PySpark examples.

- [Using sort\(\) function](#)
- [Using orderBy\(\) function](#)
- [Ascending order](#)
- [Descending order](#)
- [SQL Sort functions](#)

Related: [How to sort DataFrame by using Scala](#)

Before we start, first let's [create a DataFrame](#).

```
simpleData = [("James","Sales","NY",90000,34,10000), \
              ("Michael","Sales","NY",86000,56,20000), \
              ("Robert","Sales","CA",81000,30,23000), \
              ("Maria","Finance","CA",90000,24,23000), \
              ("Raman","Finance","CA",99000,40,24000), \
              ("Scott","Finance","NY",83000,36,19000), \
              ("Jen","Finance","NY",79000,53,15000), \
              ("Jeff","Marketing","CA",80000,25,18000), \
              ("Kumar","Marketing","NY",91000,50,21000) \
            ]
columns= ["employee_name", "department", "state", "salary", "age", "bonus"]
df = spark.createDataFrame(data = simpleData, schema = columns)
df.printSchema()
df.show(truncate=False)
```

This Yields below output.

```
root
|-- employee_name: string (nullable = true)
|-- department: string (nullable = true)
|-- state: string (nullable = true)
|-- salary: integer (nullable = false)
|-- age: integer (nullable = false)
|-- bonus: integer (nullable = false)

+-----+-----+-----+-----+-----+
|employee_name|department|state|salary|age|bonus|
+-----+-----+-----+-----+-----+
|      James|    Sales|   NY| 90000| 34|10000|
```

VN2 Solutions Pvt. Ltd.

	Michael	Sales	NY	86000	56	20000
	Robert	Sales	CA	81000	30	23000
	Maria	Finance	CA	90000	24	23000
	Raman	Finance	CA	99000	40	24000
	Scott	Finance	NY	83000	36	19000
	Jen	Finance	NY	79000	53	15000
	Jeff	Marketing	CA	80000	25	18000
	Kumar	Marketing	NY	91000	50	21000
+	-----	-----	-----	-----	-----	-----

DataFrame sorting using the sort() function

PySpark DataFrame class provides `sort()` function to sort on one or more columns. By default, it sorts by ascending order.

Syntax

```
sort(self, *cols, **kwargs):
```

Example

```
df.sort("department","state").show(truncate=False)
df.sort(col("department"),col("state")).show(truncate=False)
```

The above two examples return the same below output, the first one takes the DataFrame column name as a string and the next takes columns in Column type. This table sorted by the first `department` column and then the `state` column.

+	-----	-----	-----	-----	-----	-----
	employee_name	department	state	salary	age	bonus
+	-----	-----	-----	-----	-----	-----
	Maria	Finance	CA	90000	24	23000
	Raman	Finance	CA	99000	40	24000
	Jen	Finance	NY	79000	53	15000
	Scott	Finance	NY	83000	36	19000
	Jeff	Marketing	CA	80000	25	18000
	Kumar	Marketing	NY	91000	50	21000
	Robert	Sales	CA	81000	30	23000
	James	Sales	NY	90000	34	10000
	Michael	Sales	NY	86000	56	20000
+	-----	-----	-----	-----	-----	-----

DataFrame sorting using orderBy() function

PySpark DataFrame also provides `orderBy()` function to sort on one or more columns. By default, it orders by ascending.

Example

```
df.orderBy("department","state").show(truncate=False)
df.orderBy(col("department"),col("state")).show(truncate=False)
```

This returns the same output as the previous section.

Sort by Ascending (ASC)

If you wanted to specify the ascending order/sort explicitly on DataFrame, you can use the `asc` method of the `Column` function. for example

```
df.sort(df.department.asc(),df.state.asc()).show(truncate=False)
df.sort(col("department").asc(),col("state").asc()).show(truncate=False)
df.orderBy(col("department").asc(),col("state").asc()).show(truncate=False)
```

The above three examples return the same output.

```
+-----+-----+-----+-----+
|employee_name|department|state|salary|age|bonus|
+-----+-----+-----+-----+
|Maria      |Finance   |CA   |90000 |24 |23000|
|Raman      |Finance   |CA   |99000 |40 |24000|
|Jen        |Finance   |NY   |79000 |53 |15000|
|Scott      |Finance   |NY   |83000 |36 |19000|
|Jeff        |Marketing |CA   |80000 |25 |18000|
|Kumar      |Marketing |NY   |91000 |50 |21000|
|Robert     |Sales     |CA   |81000 |30 |23000|
|James      |Sales     |NY   |90000 |34 |10000|
|Michael    |Sales     |NY   |86000 |56 |20000|
+-----+-----+-----+-----+
```

Sort by Descending (DESC)

If you wanted to specify the sorting by descending order on DataFrame, you can use the `desc` method of the `Column` function. for example. From our example, let's use `desc` on the state column.

```
df.sort(df.department.asc(),df.state.desc()).show(truncate=False)
df.sort(col("department").asc(),col("state").desc()).show(truncate=False)
df.orderBy(col("department").asc(),col("state").desc()).show(truncate=False)
```

This yields the below output for all three examples.

```
+-----+-----+-----+-----+
|employee_name|department|state|salary|age|bonus|
+-----+-----+-----+-----+
|Scott      |Finance   |NY    |83000 |36 |19000|
|Jen        |Finance   |NY    |79000 |53 |15000|
|Raman      |Finance   |CA    |99000 |40 |24000|
|Maria      |Finance   |CA    |90000 |24 |23000|
|Kumar      |Marketing |NY    |91000 |50 |21000|
|Jeff       |Marketing |CA    |80000 |25 |18000|
|James      |Sales     |NY    |90000 |34 |10000|
|Michael    |Sales     |NY    |86000 |56 |20000|
|Robert     |Sales     |CA    |81000 |30 |23000|
+-----+-----+-----+-----+
```

Besides `asc()` and `desc()` functions, PySpark also provides `asc_nulls_first()` and `asc_nulls_last()` and equivalent descending functions.

Using Raw SQL

Below is an example of how to sort DataFrame using raw SQL syntax.

```
df.createOrReplaceTempView("EMP")
spark.sql("select employee_name,department,state,salary,age,bonus from EMP ORDER
BY department asc").show(truncate=False)
```

The above two examples return the same output as above.

Dataframe Sorting Complete Example

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, asc,desc

spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

simpleData = [("James","Sales","NY",90000,34,10000), \
 ("Michael","Sales","NY",86000,56,20000), \
 ("Robert","Sales","CA",81000,30,23000), \
 ("Maria","Finance","CA",90000,24,23000), \
 ("Raman","Finance","CA",99000,40,24000), \
```

VN2 Solutions Pvt. Ltd.

```
("Scott","Finance","NY",83000,36,19000), \
("Jen","Finance","NY",79000,53,15000), \
("Jeff","Marketing","CA",80000,25,18000), \
("Kumar","Marketing","NY",91000,50,21000) \
]
columns= ["employee_name","department","state","salary","age","bonus"]

df = spark.createDataFrame(data = simpleData, schema = columns)

df.printSchema()
df.show(truncate=False)

df.sort("department","state").show(truncate=False)
df.sort(col("department"),col("state")).show(truncate=False)

df.orderBy("department","state").show(truncate=False)
df.orderBy(col("department"),col("state")).show(truncate=False)

df.sort(df.department.asc(),df.state.asc()).show(truncate=False)
df.sort(col("department").asc(),col("state").asc()).show(truncate=False)
df.orderBy(col("department").asc(),col("state").asc()).show(truncate=False)

df.sort(df.department.asc(),df.state.desc()).show(truncate=False)
df.sort(col("department").asc(),col("state").desc()).show(truncate=False)
df.orderBy(col("department").asc(),col("state").desc()).show(truncate=False)

df.createOrReplaceTempView("EMP")
spark.sql("select employee_name,department,state,salary,age,bonus from E
```

PySpark Groupby Explained with Example

Similar to SQL GROUP BY clause, PySpark `groupBy()` function is used to collect the identical data into groups on DataFrame and perform aggregate functions on the grouped data. In this article, I will explain several `groupBy()` examples using PySpark (Spark with Python).

Related: [How to group and aggregate data using Spark and Scala](#)

Syntax:

```
groupBy(col1 : scala.Predef.String, cols : scala.Predef.String*) :
    org.apache.spark.sql.RelationalGroupedDataset
```

When we perform `groupBy()` on PySpark Dataframe, it returns `GroupedData` object which contains below aggregate functions.

`count()` - Returns the count of rows for each group.

VN2 Solutions Pvt. Ltd.

`mean()` - Returns the mean of values for each group.
`max()` - Returns the maximum of values for each group.
`min()` - Returns the minimum of values for each group.
`sum()` - Returns the total for values for each group.
`avg()` - Returns the average for values for each group.
`agg()` - Using `agg()` function, we can calculate more than one aggregate at a time.
`pivot()` - This function is used to Pivot the DataFrame which I will not be covered in this article as I already have a dedicated article for [Pivot & Unpivot DataFrame](#).

Preparing Data & creating DataFrame

Before we start, let's create the DataFrame from a sequence of the data to work with. This DataFrame contains columns "employee_name", "department", "state", "salary", "age" and "bonus" columns.

We will use this PySpark DataFrame to run `groupBy()` on "department" columns and calculate aggregates like minimum, maximum, average, total salary for each group using `min()`, `max()` and `sum()` aggregate functions respectively. and finally, we will also see how to do group and aggregate on multiple columns.

```
simpleData = [("James","Sales","NY",90000,34,10000),
              ("Michael","Sales","NY",86000,56,20000),
              ("Robert","Sales","CA",81000,30,23000),
              ("Maria","Finance","CA",90000,24,23000),
              ("Raman","Finance","CA",99000,40,24000),
              ("Scott","Finance","NY",83000,36,19000),
              ("Jen","Finance","NY",79000,53,15000),
              ("Jeff","Marketing","CA",80000,25,18000),
              ("Kumar","Marketing","NY",91000,50,21000)
            ]

schema = ["employee_name","department","state","salary","age","bonus"]
df = spark.createDataFrame(data=simpleData, schema = schema)
df.printSchema()
df.show(truncate=False)
```

Yields below output.

```
+-----+-----+-----+-----+-----+
|employee_name|department|state|salary|age|bonus|
+-----+-----+-----+-----+-----+
|      James|    Sales|   NY| 90000| 34|10000|
|    Michael|    Sales|   NY| 86000| 56|20000|
|     Robert|    Sales|   CA| 81000| 30|23000|
|      Maria|  Finance|   CA| 90000| 24|23000|
|     Raman|  Finance|   CA| 99000| 40|24000|
```

VN2 Solutions Pvt. Ltd.

```
| Scott| Finance| NY| 83000| 36|19000|
| Jen| Finance| NY| 79000| 53|15000|
| Jeff| Marketing| CA| 80000| 25|18000|
| Kumar| Marketing| NY| 91000| 50|21000|
+-----+-----+-----+-----+
```

PySpark groupBy and aggregate on DataFrame columns

Let's do the `groupBy()` on `department` column of DataFrame and then find the sum of salary for each department using `sum()` aggregate function.

```
df.groupBy("department").sum("salary").show(truncate=False)
+-----+-----+
|department|sum(salary)|
+-----+-----+
|Sales    |257000    |
|Finance  |351000    |
|Marketing|171000    |
+-----+-----+
```

Similarly, we can calculate the number of employee in each department using `count()`

```
df.groupBy("department").count()
```

Calculate the minimum salary of each department using `min()`

```
df.groupBy("department").min("salary")
```

Calculate the maximum salary of each department using `max()`

Calculate the average salary of each department using `avg()`

```
df.groupBy("department").avg( "salary")
```

Calculate the mean salary of each department using `mean()`

```
df.groupBy("department").mean( "salary")
```

PySpark groupBy and aggregate on multiple columns

Similarly, we can also run `groupBy` and `aggregate` on two or more DataFrame columns, below example does group by on `department,state` and does `sum()` on `salary` and `bonus` columns.

```
//GroupBy on multiple columns
df.groupBy("department","state") \
.sum("salary","bonus") \
.show(false)
```

This yields the below output.

VN2 Solutions Pvt. Ltd.

```
+-----+-----+-----+
|department|state|sum(salary)|sum(bonus)|
+-----+-----+-----+
|Finance |NY   |162000   |34000   |
|Marketing|NY   |91000    |21000   |
|Sales   |CA   |81000    |23000   |
|Marketing|CA   |80000    |18000   |
|Finance |CA   |189000   |47000   |
|Sales   |NY   |176000   |30000   |
+-----+-----+-----+
```

similarly, we can run group by and aggregate on tow or more columns for other aggregate functions, please refer below source code for example.

Running more aggregates at a time

Using `agg()` aggregate function we can calculate many aggregations at a time on a single statement using PySpark SQL aggregate functions `sum()`, `avg()`, `min()`, `max()` `mean()` e.t.c. In order to use these, we should import "from pyspark.sql.functions import sum,avg,max,min,mean,count"

```
df.groupBy("department") \
  .agg(sum("salary").alias("sum_salary"), \
        avg("salary").alias("avg_salary"), \
        sum("bonus").alias("sum_bonus"), \
        max("bonus").alias("max_bonus")) \
  ) \
  .show(truncate=False)
```

This example does group on `department` column and calculates `sum()` and `avg()` of `salary` for each department and calculates `sum()` and `max()` of `bonus` for each department.

```
+-----+-----+-----+-----+
|department|sum_salary|avg_salary      |sum_bonus|max_bonus|
+-----+-----+-----+-----+
|Sales   |257000   |85666.66666666667|53000   |23000   |
|Finance |351000   |87750.0       |81000   |24000   |
|Marketing|171000   |85500.0       |39000   |21000   |
+-----+-----+-----+-----+
```

Using filter on aggregate data

Similar to SQL “HAVING” clause, On PySpark DataFrame we can use either `where()` or `filter()` function to filter the rows of aggregated data.

```
df.groupBy("department") \
  .agg(sum("salary").alias("sum_salary"), \
        avg("salary").alias("avg_salary"), \
```

VN2 Solutions Pvt. Ltd.

```
sum("bonus").alias("sum_bonus"), \
max("bonus").alias("max_bonus")) \
.where(col("sum_bonus") >= 50000) \
.show(truncate=False)
```

This removes the sum of a bonus that has less than 50000 and yields below output.

```
+-----+-----+-----+-----+
|department|sum_salary|avg_salary      |sum_bonus|max_bonus|
+-----+-----+-----+-----+
|Sales    |257000   |85666.66666666667|53000    |23000    |
|Finance  |351000   |87750.0        |81000    |24000    |
+-----+-----+-----+-----+
```

[PySpark groupBy Example](#) [Source code](#)

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import col,sum,avg,max

spark = SparkSession.builder.appName('SparkByExample.com').getOrCreate()

simpleData = [("James","Sales","NY",90000,34,10000),
              ("Michael","Sales","NY",86000,56,20000),
              ("Robert","Sales","CA",81000,30,23000),
              ("Maria","Finance","CA",90000,24,23000),
              ("Raman","Finance","CA",99000,40,24000),
              ("Scott","Finance","NY",83000,36,19000),
              ("Jen","Finance","NY",79000,53,15000),
              ("Jeff","Marketing","CA",80000,25,18000),
              ("Kumar","Marketing","NY",91000,50,21000)
            ]

schema = ["employee_name","department","state","salary","age","bonus"]
df = spark.createDataFrame(data=simpleData, schema = schema)
df.printSchema()
df.show(truncate=False)

df.groupBy("department").sum("salary").show(truncate=False)

df.groupBy("department").count().show(truncate=False)

df.groupBy("department","state") \
.sum("salary","bonus") \
```

```
.show(truncate=False)

df.groupBy("department") \
    .agg(sum("salary").alias("sum_salary"), \
        avg("salary").alias("avg_salary"), \
        sum("bonus").alias("sum_bonus"), \
        max("bonus").alias("max_bonus")) \
    ) \
.show(truncate=False)

df.groupBy("department") \
    .agg(sum("salary").alias("sum_salary"), \
        avg("salary").alias("avg_salary"), \
        sum("bonus").alias("sum_bonus"), \
        max("bonus").alias("max_bonus")) \
    .where(col("sum_bonus") >= 50000) \
    .show(truncate=False)
```

PySpark Join Types | Join Two DataFrames

PySpark Join is used to combine two DataFrames and by chaining these you can join multiple DataFrames; it supports all basic join type operations available in traditional SQL like INNER, LEFT OUTER, RIGHT OUTER, LEFT ANTI, LEFT SEMI, CROSS, SELF JOIN. PySpark Joins are wider transformations that involve data shuffling across the network.

PySpark SQL Joins comes with more optimization by default (thanks to DataFrames) however still there would be some performance issues to consider while using.

In this **PySpark SQL Join** tutorial, you will learn different Join syntaxes and using different Join types on two or more DataFrames and Datasets using examples.

- [PySpark Join Syntax](#)
- [PySpark Join Types](#)
- [Inner Join DataFrame](#)
- [Full Outer Join DataFrame](#)
- [Left Outer Join DataFrame](#)
- [Right Outer Join DataFrame](#)
- [Left Anti Join DataFrame](#)
- [Left Semi Join DataFrame](#)
- [Self Join DataFrame](#)
- [Using SQL Expression](#)

1. PySpark Join Syntax

PySpark SQL join has a below syntax and it can be accessed directly from DataFrame.

```
join(self, other, on=None, how=None)
```

join() operation takes parameters as below and returns DataFrame.

- param other: Right side of the join
- param on: a string for the join column name
- param how: default inner. Must be one
of inner, cross, outer, full, full_outer, left, left_outer, right, right_outer, left_semi, and left_anti.

You can also write Join expression by adding `where()` and `filter()` methods on DataFrame and can have Join on multiple columns.

2. PySpark Join Types

Below are the different Join Types PySpark supports.

Join String	Equivalent SQL Join
inner	INNER JOIN
outer, full, fullouter, full_outer	FULL OUTER JOIN
left, leftouter, left_outer	LEFT JOIN
right, rightouter, right_outer	RIGHT JOIN
cross	
anti, leftanti, left_anti	
semi, leftsemi, left_semi	

PySpark Join Types

Before we jump into PySpark SQL Join examples, first, let's create an "emp" and "dept" DataFrames. here, column "emp_id" is unique on emp and "dept_id" is unique on the dept dataset's and emp_dept_id from emp has a reference to dept_id on dept dataset.

VN2 Solutions Pvt. Ltd.

```
emp = [(1,"Smith",-1,"2018","10","M",3000), \
        (2,"Rose",1,"2010","20","M",4000), \
        (3,"Williams",1,"2010","10","M",1000), \
        (4,"Jones",2,"2005","10","F",2000), \
        (5,"Brown",2,"2010","40","", -1), \
        (6,"Brown",2,"2010","50","", -1) \
    ]
empColumns = ["emp_id","name","superior_emp_id","year_joined", \
              "emp_dept_id","gender","salary"]

empDF = spark.createDataFrame(data=emp, schema = empColumns)
empDF.printSchema()
empDF.show(truncate=False)

dept = [("Finance",10), \
        ("Marketing",20), \
        ("Sales",30), \
        ("IT",40) \
    ]
deptColumns = ["dept_name","dept_id"]
deptDF = spark.createDataFrame(data=dept, schema = deptColumns)
deptDF.printSchema()
deptDF.show(truncate=False)
```

This prints “emp” and “dept” DataFrame to the console. Refer complete example below on how to create `spark` object.

```
Emp Dataset
+-----+-----+-----+-----+-----+
|emp_id|name  |superior_emp_id|year_joined|emp_dept_id|gender|salary|
+-----+-----+-----+-----+-----+
|1     |Smith  |-1           |2018       |10          |M      |3000  |
|2     |Rose   |1           |2010       |20          |M      |4000  |
|3     |Williams|1           |2010       |10          |M      |1000  |
|4     |Jones  |2           |2005       |10          |F      |2000  |
|5     |Brown  |2           |2010       |40          |        |-1    |
|6     |Brown  |2           |2010       |50          |        |-1    |
+-----+-----+-----+-----+-----+
```

```
Dept Dataset
+-----+-----+
|dept_name|dept_id|
+-----+-----+
|Finance  |10    |
|Marketing|20    |
|Sales    |30    |
```

VN2 Solutions Pvt. Ltd.

```
|IT    |40   |  
+-----+-----+
```

3. PySpark Inner Join DataFrame

Inner join is the default join in PySpark and it's mostly used. This joins two datasets on key columns, where keys don't match the rows get dropped from both datasets (emp & dept).

```
empDF.join(deptDF,empDF.emp_dept_id == deptDF.dept_id,"inner") \  
.show(truncate=False)
```

When we apply Inner join on our datasets, It drops "emp_dept_id" 50 from "emp" and "dept_id" 30 from "dept" datasets. Below is the result of the above Join expression.

```
+----+-----+-----+-----+-----+-----+-----+  
|emp_id|name  
|superior_emp_id|year_joined|emp_dept_id|gender|salary|dept_name|dept_id|  
+----+-----+-----+-----+-----+-----+-----+  
|1    |Smith  |-1        |2018      |10       |M     |3000  |Finance |10    |  
|2    |Rose   |1        |2010      |20       |M     |4000  |Marketing|20    |  
|3    |Williams|1        |2010      |10       |M     |1000  |Finance |10    |  
|4    |Jones  |2        |2005      |10       |F     |2000  |Finance |10    |  
|5    |Brown  |2        |2010      |40       |-1    |IT     |40    |  
+----+-----+-----+-----+-----+-----+-----+
```

4. PySpark Full Outer Join

Outer a.k.a full, fullouter join returns all rows from both datasets, where join expression doesn't match it returns null on respective record columns.

```
empDF.join(deptDF,empDF.emp_dept_id == deptDF.dept_id,"outer") \  
.show(truncate=False)  
empDF.join(deptDF,empDF.emp_dept_id == deptDF.dept_id,"full") \  
.show(truncate=False)  
empDF.join(deptDF,empDF.emp_dept_id == deptDF.dept_id,"fullouter") \  
.show(truncate=False)
```

From our "emp" dataset's "emp_dept_id" with value 50 doesn't have a record on "dept" hence dept columns have null and "dept_id" 30 doesn't have a record in "emp" hence you see null's on emp columns. Below is the result of the above Join expression.

```
+----+-----+-----+-----+-----+-----+-----+  
|emp_id|name  
|superior_emp_id|year_joined|emp_dept_id|gender|salary|dept_name|dept_id|  
+----+-----+-----+-----+-----+-----+-----+  
|2    |Rose   |1        |2010      |20       |M     |4000  |Marketing|20    |  
|5    |Brown  |2        |2010      |40       |-1    |IT     |40    |  
|1    |Smith  |-1        |2018      |10       |M     |3000  |Finance |10    |  
+----+-----+-----+-----+-----+-----+-----+
```

VN2 Solutions Pvt. Ltd.

3	Williams	1	2010	10	M	1000	Finance	10	
4	Jones	2	2005	10	F	2000	Finance	10	
6	Brown	2	2010	50		-1	null	null	
null	null	null	null	null	null	null	Sales	30	
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+									

5. PySpark Left Outer Join

Left a.k.a `Leftouter` join returns all rows from the left dataset regardless of match found on the right dataset when join expression doesn't match, it assigns null for that record and drops records from right where match not found.

```
empDF.join(deptDF,empDF("emp_dept_id") == deptDF("dept_id"),"left")
      .show(false)
empDF.join(deptDF,empDF("emp_dept_id") == deptDF("dept_id"),"leftouter")
      .show(false)
```

From our dataset, “`emp_dept_id`” 50 doesn't have a record on “`dept`” dataset hence, this record contains null on “`dept`” columns (`dept_name` & `dept_id`). and “`dept_id`” 30 from “`dept`” dataset dropped from the results. Below is the result of the above Join expression.

+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+									
emp_id name									
superior_emp_id year_joined emp_dept_id gender salary dept_name dept_id									
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+									
1	Smith	-1	2018	10	M	3000	Finance	10	
2	Rose	1	2010	20	M	4000	Marketing	20	
3	Williams	1	2010	10	M	1000	Finance	10	
4	Jones	2	2005	10	F	2000	Finance	10	
5	Brown	2	2010	40		-1	IT	40	
6	Brown	2	2010	50		-1	null	null	
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+									

6. Right Outer Join

Right a.k.a `Rightouter` join is opposite of `left` join, here it returns all rows from the right dataset regardless of math found on the left dataset, when join expression doesn't match, it assigns null for that record and drops records from left where match not found.

```
empDF.join(deptDF,empDF.emp_dept_id == deptDF.dept_id,"right") \
      .show(truncate=False)
empDF.join(deptDF,empDF.emp_dept_id == deptDF.dept_id,"rightouter") \
      .show(truncate=False)
```

VN2 Solutions Pvt. Ltd.

From our example, the right dataset “dept_id” 30 doesn’t have it on the left dataset “emp” hence, this record contains null on “emp” columns. and “emp_dept_id” 50 dropped as a match not found on left. Below is the result of the above Join expression.

emp_id	name	superior_emp_id	year_joined	emp_dept_id	gender	salary	dept_name	dept_id
4	Jones	2	2005	10	F	2000	Finance	10
3	Williams	1	2010	10	M	1000	Finance	10
1	Smith	-1	2018	10	M	3000	Finance	10
2	Rose	1	2010	20	M	4000	Marketing	20
null	null	null	null	null	null	null	Sales	30
5	Brown	2	2010	40		-1	IT	40

7. Left Semi Join

`leftsemi` join is similar to `inner` join difference being `leftsemi` join returns all columns from the left dataset and ignores all columns from the right dataset. In other words, this join returns columns from the only left dataset for the records match in the right dataset on join expression, records not matched on join expression are ignored from both left and right datasets.

The same result can be achieved using select on the result of the inner join however, using this join would be efficient.

```
empDF.join(deptDF,empDF.emp_dept_id == deptDF.dept_id,"leftsemi") \
.show(truncate=False)
```

Below is the result of the above join expression.

leftsemi join						
emp_id	name	superior_emp_id	year_joined	emp_dept_id	gender	salary
1	Smith	-1	2018	10	M	3000
2	Rose	1	2010	20	M	4000
3	Williams	1	2010	10	M	1000
4	Jones	2	2005	10	F	2000
5	Brown	2	2010	40		-1

8. Left Anti Join

`leftanti` join does the exact opposite of the `leftsemi`, `leftanti` join returns only columns from the left dataset for non-matched records.

```
empDF.join(deptDF,empDF.emp_dept_id == deptDF.dept_id,"leftanti") \  
.show(truncate=False)
```

Yields below output

```
+----+-----+-----+-----+-----+  
|emp_id|name |superior_emp_id|year_joined|emp_dept_id|gender|salary|  
+----+-----+-----+-----+-----+  
|6    |Brown|2          |2010      |50       |-1     |  
+----+-----+-----+-----+-----+
```

9. PySpark Self Join

Joins are not complete without a self join, Though there is no self-join type available, we can use any of the above-explained join types to join DataFrame to itself. below example use `inner` self join.

```
empDF.alias("emp1").join(empDF.alias("emp2"), \  
  col("emp1.superior_emp_id") == col("emp2.emp_id"),"inner") \  
.select(col("emp1.emp_id"),col("emp1.name"), \  
  col("emp2.emp_id").alias("superior_emp_id"), \  
  col("emp2.name").alias("superior_emp_name")) \  
.show(truncate=False)
```

Here, we are joining `emp` dataset with itself to find out superior `emp_id` and `name` for all employees.

```
+----+-----+-----+  
|emp_id|name   |superior_emp_id|superior_emp_name|  
+----+-----+-----+  
|2    |Rose   |1          |Smith           |  
|3    |Williams|1          |Smith           |  
|4    |Jones   |2          |Rose            |  
|5    |Brown   |2          |Rose            |  
|6    |Brown   |2          |Rose            |  
+----+-----+-----+
```

4. Using SQL Expression

Since PySpark SQL support native SQL syntax, we can also write join operations after creating temporary tables on DataFrames and use these tables on `spark.sql()`.

```
empDF.createOrReplaceTempView("EMP")
```

VN2 Solutions Pvt. Ltd.

```
deptDF.createOrReplaceTempView("DEPT")

joinDF = spark.sql("select * from EMP e, DEPT d where e.emp_dept_id == d.dept_id") \
.show(truncate=False)

joinDF2 = spark.sql("select * from EMP e INNER JOIN DEPT d ON e.emp_dept_id == \
d.dept_id") \
.show(truncate=False)
```

5. PySpark SQL Join on multiple DataFrames

When you need to join more than two tables, you either use SQL expression after creating a temporary view on the DataFrame or use the result of join operation to join with another DataFrame like chaining them. for example

```
df1.join(df2,df1.id1 == df2.id2,"inner") \
.join(df3,df1.id1 == df3.id3,"inner")
```

6. PySpark SQL Join Complete Example

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import col

spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

emp = [(1,"Smith",-1,"2018","10","M",3000), \
(2,"Rose",1,"2010","20","M",4000), \
(3,"Williams",1,"2010","10","M",1000), \
(4,"Jones",2,"2005","10","F",2000), \
(5,"Brown",2,"2010","40","","-1), \
(6,"Brown",2,"2010","50","","-1) \
]
empColumns = ["emp_id","name","superior_emp_id","year_joined", \
"emp_dept_id","gender","salary"]

empDF = spark.createDataFrame(data=emp, schema = empColumns)
empDF.printSchema()
empDF.show(truncate=False)

dept = [("Finance",10), \
("Marketing",20), \
("Sales",30), \
("IT",40) \
```

VN2 Solutions Pvt. Ltd.

```
]

deptColumns = ["dept_name","dept_id"]
deptDF = spark.createDataFrame(data=dept, schema = deptColumns)
deptDF.printSchema()
deptDF.show(truncate=False)

empDF.join(deptDF,empDF.emp_dept_id ==  deptDF.dept_id,"inner") \
    .show(truncate=False)

empDF.join(deptDF,empDF.emp_dept_id ==  deptDF.dept_id,"outer") \
    .show(truncate=False)
empDF.join(deptDF,empDF.emp_dept_id ==  deptDF.dept_id,"full") \
    .show(truncate=False)
empDF.join(deptDF,empDF.emp_dept_id ==  deptDF.dept_id,"fullouter") \
    .show(truncate=False)

empDF.join(deptDF,empDF.emp_dept_id ==  deptDF.dept_id,"left") \
    .show(truncate=False)
empDF.join(deptDF,empDF.emp_dept_id ==  deptDF.dept_id,"leftouter") \
    .show(truncate=False)

empDF.join(deptDF,empDF.emp_dept_id ==  deptDF.dept_id,"right") \
    .show(truncate=False)
empDF.join(deptDF,empDF.emp_dept_id ==  deptDF.dept_id,"rightouter") \
    .show(truncate=False)

empDF.join(deptDF,empDF.emp_dept_id ==  deptDF.dept_id,"leftsemi") \
    .show(truncate=False)

empDF.join(deptDF,empDF.emp_dept_id ==  deptDF.dept_id,"leftanti") \
    .show(truncate=False)

empDF.alias("emp1").join(empDF.alias("emp2"), \
    col("emp1.superior_emp_id") == col("emp2.emp_id"),"inner") \
    .select(col("emp1.emp_id"),col("emp1.name"), \
        col("emp2.emp_id").alias("superior_emp_id"), \
        col("emp2.name").alias("superior_emp_name")) \
    .show(truncate=False)

empDF.createOrReplaceTempView("EMP")
deptDF.createOrReplaceTempView("DEPT")

joinDF = spark.sql("select * from EMP e, DEPT d where e.emp_dept_id == d.dept_id") \
    .show(truncate=False)
```

VN2 Solutions Pvt. Ltd.

```
joinDF2 = spark.sql("select * from EMP e INNER JOIN DEPT d ON e.emp_dept_id ==  
d.dept_id") \  
.show(truncate=False)
```

PySpark Union and UnionAll Explained

PySpark union() and unionAll() transformations are used to merge two or more DataFrame's of the same schema or structure. In this PySpark article, I will explain both union transformations with PySpark examples.

Dataframe union() – union() method of the DataFrame is used to merge two DataFrame's of the same structure/schema. If schemas are not the same it returns an error.

DataFrame unionAll() – unionAll() is deprecated since Spark “2.0.0” version and replaced with union().

Note: In other SQL languages, Union eliminates the duplicates but UnionAll merges two datasets including duplicate records. But, in PySpark both behave the same and recommend using [DataFrame duplicate\(\) function to remove duplicate rows](#).

First, let's create two DataFrame with the same schema.

First DataFrame

```
import pyspark  
from pyspark.sql import SparkSession  
  
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()  
  
simpleData = [("James","Sales","NY",90000,34,10000), \  
 ("Michael","Sales","NY",86000,56,20000), \  
 ("Robert","Sales","CA",81000,30,23000), \  
 ("Maria","Finance","CA",90000,24,23000) \  
 ]  
  
columns= ["employee_name","department","state","salary","age","bonus"]  
df = spark.createDataFrame(data = simpleData, schema = columns)  
df.printSchema()  
df.show(truncate=False)
```

This yields the below schema and DataFrame output.

```
root  
 |-- employee_name: string (nullable = true)  
 |-- department: string (nullable = true)  
 |-- state: string (nullable = true)  
 |-- salary: long (nullable = true)
```

VN2 Solutions Pvt. Ltd.

```
|-- age: long (nullable = true)
|-- bonus: long (nullable = true)

+-----+-----+-----+-----+
|employee_name|department|state|salary|age|bonus|
+-----+-----+-----+-----+
|James      |Sales     |NY   |90000 |34 |10000|
|Michael    |Sales     |NY   |86000 |56 |20000|
|Robert     |Sales     |CA   |81000 |30 |23000|
|Maria      |Finance   |CA   |90000 |24 |23000|
+-----+-----+-----+-----+
```

Second DataFrame

Now, let's create a second Dataframe with the new records and some records from the above Dataframe but with the same schema.

```
simpleData2 = [("James","Sales","NY",90000,34,10000), \
    ("Maria","Finance","CA",90000,24,23000), \
    ("Jen","Finance","NY",79000,53,15000), \
    ("Jeff","Marketing","CA",80000,25,18000), \
    ("Kumar","Marketing","NY",91000,50,21000) \
]
columns2= ["employee_name","department","state","salary","age","bonus"]

df2 = spark.createDataFrame(data = simpleData2, schema = columns2)

df2.printSchema()
df2.show(truncate=False)
```

This yields below output

```
+-----+-----+-----+-----+
|employee_name|department|state|salary|age|bonus|
+-----+-----+-----+-----+
|James      |Sales     |NY   |90000 |34 |10000|
|Maria      |Finance   |CA   |90000 |24 |23000|
|Jen        |Finance   |NY   |79000 |53 |15000|
|Jeff       |Marketing |CA   |80000 |25 |18000|
|Kumar      |Marketing |NY   |91000 |50 |21000|
+-----+-----+-----+-----+
```

Merge two or more DataFrames using union

DataFrame `union()` method merges two DataFrames and returns the new DataFrame with all rows from two Dataframes regardless of duplicate data.

VN2 Solutions Pvt. Ltd.

```
unionDF = df.union(df2)
unionDF.show(truncate=False)
```

As you see below it returns all records.

```
+-----+-----+-----+-----+
|employee_name|department|state|salary|age|bonus|
+-----+-----+-----+-----+
|James      |Sales     |NY   |90000 |34  |10000|
|Michael    |Sales     |NY   |86000 |56  |20000|
|Robert     |Sales     |CA   |81000 |30  |23000|
|Maria      |Finance   |CA   |90000 |24  |23000|
|James      |Sales     |NY   |90000 |34  |10000|
|Maria      |Finance   |CA   |90000 |24  |23000|
|Jen        |Finance   |NY   |79000 |53  |15000|
|Jeff       |Marketing |CA   |80000 |25  |18000|
|Kumar      |Marketing |NY   |91000 |50  |21000|
+-----+-----+-----+-----+
```

Merge DataFrames using unionAll

DataFrame `unionAll()` method is deprecated since PySpark “2.0.0” version and recommends using the `union()` method.

```
unionAllDF = df.unionAll(df2)
unionAllDF.show(truncate=False)
```

Returns the same output as above.

Merge without Duplicates

Since the `union()` method returns all rows without distinct records, we will use the `distinct()` function to return just one record when duplicate exists.

```
disDF = df.union(df2).distinct()
disDF.show(truncate=False)
```

Yields below output. As you see, this returns only distinct rows.

```
+-----+-----+-----+-----+
|employee_name|department|state|salary|age|bonus|
+-----+-----+-----+-----+
|James      |Sales     |NY   |90000 |34  |10000|
|Maria      |Finance   |CA   |90000 |24  |23000|
|Kumar      |Marketing |NY   |91000 |50  |21000|
|Michael    |Sales     |NY   |86000 |56  |20000|
|Jen        |Finance   |NY   |79000 |53  |15000|
+-----+-----+-----+-----+
```

VN2 Solutions Pvt. Ltd.

```
|Jeff      |Marketing |CA   |80000 |25 |18000|
|Robert    |Sales    |CA   |81000 |30 |23000|
+-----+-----+-----+-----+-----+
```

Python

Complete Example of DataFrame Union

```
import pyspark
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

simpleData = [("James","Sales","NY",90000,34,10000), \
             ("Michael","Sales","NY",86000,56,20000), \
             ("Robert","Sales","CA",81000,30,23000), \
             ("Maria","Finance","CA",90000,24,23000) \
            ]

columns= ["employee_name","department","state","salary","age","bonus"]
df = spark.createDataFrame(data = simpleData, schema = columns)
df.printSchema()
df.show(truncate=False)

simpleData2 = [("James","Sales","NY",90000,34,10000), \
              ("Maria","Finance","CA",90000,24,23000), \
              ("Jen","Finance","NY",79000,53,15000), \
              ("Jeff","Marketing","CA",80000,25,18000), \
              ("Kumar","Marketing","NY",91000,50,21000) \
             ]
columns2= ["employee_name","department","state","salary","age","bonus"]

df2 = spark.createDataFrame(data = simpleData2, schema = columns2)

df2.printSchema()
df2.show(truncate=False)

unionDF = df.union(df2)
unionDF.show(truncate=False)
disDF = df.union(df2).distinct()
disDF.show(truncate=False)

unionAllDF = df.unionAll(df2)
unionAllDF.show(truncate=False)
```

Spark Merge Two DataFrames with Different Columns or Schema

In Spark or PySpark let's see how to merge/union two DataFrames with a different number of columns (different schema). In Spark 3.1, you can easily achieve this using `unionByName()` transformation by passing `allowMissingColumns` with the value true. In older versions, this property is not available.

```
//Scala  
merged_df = df1.unionByName(df2, true)  
  
#PySpark  
merged_df = df1.unionByName(df2, allowMissingColumns=True)
```

The difference between `unionByName()` function and `union()` is that this function resolves columns by name (not by position). In other words, `unionByName()` is used to merge two DataFrame's by column names instead of by position.

In case if you are using older than Spark 3.1 version, use below approach to merge DataFrame's with different column names.

- [Spark Merge DataFrames with Different Columns \(Scala Example\)](#)
- [PySpark Merge DataFrames with Different Columns \(Python Example\)](#)

Spark Merge Two DataFrames with Different Columns

In this section I will cover Spark with Scala example of how to merge two different DataFrames, first let's create DataFrames with different number of columns. DataFrame `df1` missing column `state` and `salary` and `df2` missing column `age`.

```
//Create DataFrame df1 with columns name,dept & age  
val data = Seq(("James","Sales",34), ("Michael","Sales",56),  
               ("Robert","Sales",30), ("Maria","Finance",24) )  
import spark.implicits._  
val df1 = data.toDF("name","dept","age")  
df1.printSchema()  
  
//root  
// |-- name: string (nullable = true)  
// |-- dept: string (nullable = true)  
// |-- age: long (nullable = true)  
Second DataFrame
```

```
//Create DataFrame df1 with columns name,dep,state & salary  
val data2=Seq(("James","Sales","NY",9000),("Maria","Finance","CA",9000),  
               ("Jen","Finance","NY",7900),("Jeff","Marketing","CA",8000))
```

VN2 Solutions Pvt. Ltd.

```
val df2 = data2.toDF("name","dept","state","salary")
df2.printSchema()

//root
// |-- name: string (nullable = true)
// |-- dept: string (nullable = true)
// |-- state: string (nullable = true)
// |-- salary: long (nullable = true)
```

Now create a new DataFrames from existing after adding missing columns. newly added columns contains `null` values and we add constant column using `lit()` function.

```
val merged_cols = df1.columns.toSet ++ df2.columns.toSet
import org.apache.spark.sql.functions.{col,lit}
def getNewColumns(column: Set[String], merged_cols: Set[String]) = {
  merged_cols.toList.map(x => x match {
    case x if column.contains(x) => col(x)
    case _ => lit(null).as(x)
  })
}
val new_df1=df1.select(getNewColumns(df1.columns.toSet, merged_cols):_*)
val new_df2=df2.select(getNewColumns(df2.columns.toSet, merged_cols):_*)
```

Finally merge two DataFrame's by using column names

```
//Finally join two dataframe's df1 & df2 by name
val merged_df=new_df1.unionByName(new_df2)
merged_df.show()

//+-----+-----+-----+
//|  name|  dept| age|state|salary|
//+-----+-----+-----+
//| James|  Sales|  34| null|  null|
//|Michael|  Sales|  56| null|  null|
//| Robert|  Sales|  30| null|  null|
//| Maria| Finance|  24| null|  null|
//| James|  Sales| null|  NY| 9000|
//| Maria| Finance| null|  CA| 9000|
//| Jen| Finance| null|  NY| 7900|
//| Jeff| Marketing| null|  CA| 8000|
//+-----+-----+-----+
```

PySpark Merge Two DataFrames with Different Columns

In PySpark to merge two DataFrames with different columns, will use the similar approach explain above and uses `unionByName()` transformation. First let's create DataFrame's with different number of columns.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

#Create DataFrame df1 with columns name,dept & age
data = [("James","Sales",34), ("Michael","Sales",56), \
        ("Robert","Sales",30), ("Maria","Finance",24) ]
columns= ["name","dept","age"]
df1 = spark.createDataFrame(data = data, schema = columns)
df1.printSchema()

#Create DataFrame df1 with columns name,dep,state & salary
data2=[("James","Sales","NY",9000),("Maria","Finance","CA",9000), \
       ("Jen","Finance","NY",7900),("Jeff","Marketing","CA",8000)]
columns2= ["name","dept","state","salary"]
df2 = spark.createDataFrame(data = data2, schema = columns2)
df2.printSchema()
```

Now add missing columns 'state' and 'salary' to `df1` and 'age' to `df2` with null values.

```
#Add missing columns 'state' & 'salary' to df1
from pyspark.sql.functions import lit
for column in [column for column in df2.columns if column not in df1.columns]:
    df1 = df1.withColumn(column, lit(None))

#Add missing column 'age' to df2
for column in [column for column in df1.columns if column not in df2.columns]:
    df2 = df2.withColumn(column, lit(None))
```

Now merge/union the DataFrames using `unionByName()`. The difference between `unionByName()` function and `union()` is that this function resolves columns by name (not by position). In other words, `unionByName()` is used to merge two DataFrame's by column names instead of by position.

```
#Finally join two dataframe's df1 & df2 by name
merged_df=df1.unionByName(df2)
merged_df.show()
```

PySpark UDF Example

PySpark UDF (a.k.a User Defined Function) is the most useful feature of Spark SQL & DataFrame that is used to extend the PySpark build in capabilities. In this article, I will explain what is UDF? why do we need it and how to create and use it on DataFrame `select()`, `withColumn()` and SQL using PySpark (Spark with Python) examples.

Note: UDF's are the most expensive operations hence use them only you have no choice and when essential. In the later section of the article, I will explain why using UDF's is an expensive operation in detail.

Table of contents

- [PySpark UDF Introduction](#)
 - [What is UDF?](#)
 - [Why do we need it?](#)
- [Create PySpark UDF \(User Defined Function\)](#)
 - [Create a DataFrame](#)
 - [Create a Python function](#)
 - [Convert python function to UDF](#)
- [Using UDF with DataFrame](#)
 - [Using UDF with DataFrame select\(\)](#)
 - [Using UDF with DataFrame withColumn\(\)](#)
 - [Registering UDF & Using it on SQL query](#)
- [Create UDF using annotation](#)
- [Special handling](#)
 - [Null check](#)
 - [Performance concern](#)
- [Complete Example](#)

1. PySpark UDF Introduction

1.1 What is UDF?

UDF's a.k.a User Defined Functions, If you are coming from SQL background, UDF's are nothing new to you as most of the traditional RDBMS databases support User Defined Functions, these functions need to register in the database library and use them on SQL as regular functions.

PySpark UDF's are similar to UDF on traditional databases. In PySpark, you create a function in a Python syntax and wrap it with PySpark SQL `udf()` or register it as `udf` and use it on DataFrame and SQL respectively.

1.2 Why do we need a UDF?

UDF's are used to extend the functions of the framework and re-use these functions on multiple DataFrame's. For example, you wanted to convert every first letter of a word in a name string to a capital case; PySpark build-in features don't have this function hence you can create it a UDF and reuse this as needed on many Data

VN2 Solutions Pvt. Ltd.

Frames. UDF's are once created they can be re-used on several DataFrame's and SQL expressions.

Before you create any UDF, do your research to check if the similar function you wanted is already available in [Spark SQL Functions](#). PySpark SQL provides several predefined common functions and many more new functions are added with every release. hence, It is best to check before you reinventing the wheel.

When you creating UDF's you need to design them very carefully otherwise you will come across optimization & performance issues.

2. Create PySpark UDF

2.1 Create a DataFrame

Before we jump in creating a UDF, first let's create a PySpark DataFrame.

```
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

columns = ["Seqno","Name"]
data = [("1", "john jones"),
       ("2", "tracey smith"),
       ("3", "amy sanders")]

df = spark.createDataFrame(data=schema=columns)

df.show(truncate=False)
```

Yields below output.

```
+----+-----+
|Seqno|Names    |
+----+-----+
| 1  |john jones |
| 2  |tracey smith|
| 3  |amy sanders |
+----+-----+
```

2.2 Create a Python Function

The first step in creating a UDF is creating a Python function. Below snippet creates a function `convertCase()` which takes a string parameter and converts the first letter of every word to capital letter. UDF's take parameters of your choice and returns a value.

```
def convertCase(str):
    resStr=""
    arr = str.split(" ")
    for x in arr:
        resStr= resStr + x[0:1].upper() + x[1:len(x)] + " "
```

```
    return resStr
```

2.3 Convert a Python function to PySpark UDF

Now convert this function `convertCase()` to UDF by passing the function to PySpark SQL `udf()`, this function is available at `org.apache.spark.sql.functions.udf` package.

Make sure you import this package before using it.

PySpark SQL `udf()` function

returns `org.apache.spark.sql.expressions.UserDefinedFunction` class object.

```
""" Converting function to UDF """
```

```
convertUDF = udf(lambda z: convertCase(z),StringType())
```

Note: The default type of the `udf()` is `StringType` hence, you can also write the above statement without return type.

```
""" Converting function to UDF
```

```
StringType() is by default hence not required """
```

```
convertUDF = udf(lambda z: convertCase(z))
```

3. Using UDF with DataFrame

3.1 Using UDF with PySpark DataFrame select()

Now you can use `convertUDF()` on a DataFrame column as a regular build-in function.

```
df.select(col("Seqno"), \
    convertUDF(col("Name")).alias("Name") ) \
    .show(truncate=False)
```

This results below output.

```
+----+-----+
|Seqno|Name      |
+----+-----+
| 1  |John Jones  |
| 2  |Tracey Smith|
| 3  |Amy Sanders |
+----+-----+
```

3.2 Using UDF with PySpark DataFrame withColumn()

You could also use `udf` on DataFrame `withColumn()` function, to explain this I will create another `upperCase()` function which converts the input string to upper case.

```
def upperCase(str):
    return str.upper()
```

Let's convert `upperCase()` python function to UDF and then use it with DataFrame `withColumn()`. Below example converts the values of "Name" column to upper case and creates a new column "Curated Name"

```
upperCaseUDF = udf(lambda z:upperCase(z),StringType())
```

VN2 Solutions Pvt. Ltd.

```
df.withColumn("Cureated Name", upperCaseUDF(col("Name"))) \
.show(truncate=False)
```

This yields below output.

```
+----+-----+-----+
|Seqno|Name      |Cureated Name|
+----+-----+-----+
| 1   |john jones |JOHN JONES  |
| 2   |tracey smith|TRACEY SMITH |
| 3   |amy sanders |AMY SANDERS |
+----+-----+-----+
```

3.3 Registering PySpark UDF & use it on SQL

In order to use `convertCase()` function on PySpark SQL, you need to register the function with PySpark by using `spark.udf.register()`.

```
""" Using UDF on SQL """
spark.udf.register("convertUDF", convertCase, StringType())
df.createOrReplaceTempView("NAME_TABLE")
spark.sql("select Seqno, convertUDF(Name) as Name from NAME_TABLE") \
.show(truncate=False)
```

This yields the same output as 3.1 example.

4. Creating UDF using annotation

In the previous sections, you have learned creating a UDF is a 2 step process, first, you need to create a Python function, second convert function to UDF using SQL `udf()` function, however, you can avoid these two steps and create it with just a single step by using annotations.

```
@udf(returnType=StringType())
def upperCase(str):
    return str.upper()

df.withColumn("Cureated Name", upperCase(col("Name"))) \
.show(truncate=False)
```

This results same output as section 3.2

5. Special Handling

5.1 Execution order

One thing to aware is in PySpark/Spark does not guarantee the order of evaluation of subexpressions meaning expressions are not guarantee to evaluated left-to-right or in any other fixed order. PySpark reorders the execution for query optimization and planning hence, AND, OR, WHERE and HAVING expression will have side effects.

VN2 Solutions Pvt. Ltd.

So when you are designing and using UDF, you have to be very careful especially with null handling as these results runtime exceptions.

```
"""
No guarantee Name is not null will execute first
If convertUDF(Name) like '%John%' execute first then
you will get runtime error
"""

spark.sql("select Seqno, convertUDF(Name) as Name from NAME_TABLE " + \
    "where Name is not null and convertUDF(Name) like '%John%'") \
.show(truncate=False)
```

5.2 Handling null check

UDF's are error-prone when not designed carefully. for example, when you have a column that contains the value `null` on some records

```
""" null check """

columns = ["Seqno","Name"]
data = [("1", "john jones"),
        ("2", "tracey smith"),
        ("3", "amy sanders"),
        ('4',None)]

df2 = spark.createDataFrame(data=schema=columns)
df2.show(truncate=False)
df2.createOrReplaceTempView("NAME_TABLE2")

spark.sql("select convertUDF(Name) from NAME_TABLE2") \
.show(truncate=False)
```

Note that from the above snippet, record with "Seqno 4" has value "None" for "name" column. Since we are not handling null with UDF function, using this on DataFrame returns below error. Note that in Python None is considered null.

```
AttributeError: 'NoneType' object has no attribute 'split'

at
org.apache.spark.api.python.BasePythonRunner$ReaderIterator.handlePythonException(PythonRunner.scala:456)
at
org.apache.spark.sql.execution.python.PythonUDFRunner$$anon$1.read(PythonUDFRunner.scala:81)
```

VN2 Solutions Pvt. Ltd.

```
at
org.apache.spark.sql.execution.python.PythonUDFRunner$$anon$1.read(PythonUDF
unner.scala:64)
    at
org.apache.spark.api.python.BasePythonRunner$ReaderIterator.hasNext(PythonRunne
r.scala:410)
    at
org.apache.spark.InterruptibleIterator.hasNext(InterruptibleIterator.scala:37)
    at scala.collection.Iterator$$anon$12.hasNext(Iterator.scala:440)
```

Below points to remember

- Its always best practice to check for null inside a UDF function rather than checking for null outside.
- In any case, if you can't do a null check in UDF at lease use IF or CASE WHEN to check for null and call UDF conditionally.

```
spark.udf.register("_nullsafeUDF", lambda str: convertCase(str) if not str is None else
"" , StringType())

spark.sql("select _nullsafeUDF(Name) from NAME_TABLE2") \
.show(truncate=False)

spark.sql("select Seqno, _nullsafeUDF(Name) as Name from NAME_TABLE2 " + \
    " where Name is not null and _nullsafeUDF(Name) like '%John%'") \
.show(truncate=False)
```

This executes successfully without errors as we are checking for null/none while registering UDF.

5.3 Performance concern using UDF

UDF's are a black box to PySpark hence it can't apply optimization and you will lose all the optimization PySpark does on Dataframe/Dataset. When possible you should use Spark SQL built-in functions as these functions provide optimization. Consider creating UDF only when existing built-in SQL function doesn't have it.

6. Complete PySpark UDF Example

Below is complete UDF function example in Scala

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, udf
from pyspark.sql.types import StringType

spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()
```

VN2 Solutions Pvt. Ltd.

```
columns = ["Seqno","Name"]
data = [("1", "john jones"),
       ("2", "tracey smith"),
       ("3", "amy sanders")]

df = spark.createDataFrame(data=data,schema=columns)

df.show(truncate=False)

def convertCase(str):
    resStr=""
    arr = str.split(" ")
    for x in arr:
        resStr= resStr + x[0:1].upper() + x[1:len(x)] + " "
    return resStr

""" Converting function to UDF """
convertUDF = udf(lambda z: convertCase(z))

df.select(col("Seqno"), \
          convertUDF(col("Name")).alias("Name") ) \
.show(truncate=False)

def upperCase(str):
    return str.upper()

upperCaseUDF = udf(lambda z:upperCase(z),StringType())

df.withColumn("Cureated Name", upperCaseUDF(col("Name"))) \
.show(truncate=False)

""" Using UDF on SQL """
spark.udf.register("convertUDF", convertCase,StringType())
df.createOrReplaceTempView("NAME_TABLE")
spark.sql("select Seqno, convertUDF(Name) as Name from NAME_TABLE") \
.show(truncate=False)

spark.sql("select Seqno, convertUDF(Name) as Name from NAME_TABLE " + \
          "where Name is not null and convertUDF(Name) like '%John%'") \
.show(truncate=False)

""" null check """

columns = ["Seqno","Name"]
data = [("1", "john jones"),
```

```
("2", "tracey smith"),
("3", "amy sanders"),
('4',None)]  
  
df2 = spark.createDataFrame(data=data,schema=columns)
df2.show(truncate=False)
df2.createOrReplaceTempView("NAME_TABLE2")  
  
spark.udf.register("_nullsafeUDF", lambda str: convertCase(str) if not str is None else
"" , StringType())
  
spark.sql("select _nullsafeUDF(Name) from NAME_TABLE2") \
.show(truncate=False)  
  
spark.sql("select Seqno, _nullsafeUDF(Name) as Name from NAME_TABLE2 " + \
" where Name is not null and _nullsafeUDF(Name) like '%John%'") \
.show(truncate=False)
```

PySpark map() Transformation

PySpark map (`map()`) is an RDD transformation that is used to apply the transformation function (`lambda`) on every element of RDD/DataFrame and returns a new RDD. In this article, you will learn the syntax and usage of the RDD `map()` transformation with an example and how to use it with DataFrame.

RDD `map()` transformation is used to apply any complex operations like adding a column, updating a column, transforming the data e.t.c, the output of map transformations would always have the same number of records as input.

- **Note1:** DataFrame doesn't have `map()` transformation to use with DataFrame hence you need to DataFrame to RDD first.
- **Note2:** If you have a heavy initialization use PySpark `mapPartitions()` transformation instead of `map()`, as with `mapPartitions()` heavy initialization executes only once for each partition instead of every record.

Related: [Spark map\(\) vs mapPartitions\(\) Explained with Examples](#)

First, let's create an RDD from the list.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.master("local[1]") \
.appName("SparkByExamples.com").getOrCreate()  
  
data = ["Project","Gutenberg's","Alice's","Adventures",
"in","Wonderland","Project","Gutenberg's","Adventures",
"in","Wonderland","Project","Gutenberg's"]
```

```
rdd=spark.sparkContext.parallelize(data)
```

map() Syntax

```
map(f, preservesPartitioning=False)
```

PySpark map() Example with RDD

In this PySpark `map()` example, we are adding a new element with value 1 for each element, the result of the RDD is `PairRDDFunctions` which contains key-value pairs, word of type String as Key and 1 of type Int as value.

```
rdd2=rdd.map(lambda x: (x,1))
for element in rdd2.collect():
    print(element)
```

This yields below output.

```
('Project', 1)
('Gutenberg's', 1)
('Alice's', 1)
('Adventures', 1)
('in', 1)
('Wonderland', 1)
('Project', 1)
('Gutenberg's', 1)
('Adventures', 1)
('in', 1)
('Wonderland', 1)
('Project', 1)
('Gutenberg's', 1)
```

PySpark map() Example with DataFrame

PySpark DataFrame doesn't have `map()` transformation to apply the lambda function, when you wanted to apply the custom transformation, you need to convert the DataFrame to RDD and apply the `map()` transformation. Let's use another dataset to explain this.

```
data = [('James','Smith','M',30),
        ('Anna','Rose','F',41),
        ('Robert','Williams','M',62),
```

VN2 Solutions Pvt. Ltd.

```
]

columns = ["firstname","lastname","gender","salary"]
df = spark.createDataFrame(data=data, schema = columns)
df.show()
+-----+-----+-----+
|firstname|lastname|gender|salary|
+-----+-----+-----+
|  James|  Smith|    M|   30|
|  Anna|  Rose|    F|   41|
| Robert|Williams|    M|   62|
+-----+-----+-----+


# Refering columns by index.
rdd2=df.rdd.map(lambda x:
 (x[0]+","+x[1],x[2],x[3]*2)
 )
df2=rdd2.toDF(["name","gender","new_salary"]  )
df2.show()
+-----+-----+-----+
|      name|gender|new_salary|
+-----+-----+-----+
| James,Smith|    M|      60|
| Anna,Rose|    F|      82|
|Robert,Williams|    M|     124|
+-----+-----+
```

Note that aboveI have used index to get the column values, alternatively, you can also refer to the DataFrame column names while iterating.

```
# Referring Column Names
rdd2=df.rdd.map(lambda x:
 (x["firstname"]+","+x["lastname"],x["gender"],x["salary"]*2)
 )
```

Another alternative

```
# Referring Column Names
rdd2=df.rdd.map(lambda x:
 (x.firstname+","+x.lastname,x.gender,x.salary*2)
 )
```

You can also create a custom function to perform an operation.

Below `func1()` function executes for every DataFrame row from the lambda function.

```
# By Calling function
```

```
def func1(x):
    firstName=x.firstname
    lastName=x.lastname
    name=firstName+","+lastName
    gender=x.gender.lower()
    salary=x.salary*2
    return (name,gender,salary)
```

Complete PySpark map() example

Below is complete example of PySpark map() transformation.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

data = ["Project",
"Gutenberg's",
"Alice's",
"Adventures",
"in",
"Wonderland",
"Project",
"Gutenberg's",
"Adventures",
"in",
"Wonderland",
"Project",
"Gutenberg's"]

rdd=spark.sparkContext.parallelize(data)

rdd2=rdd.map(lambda x: (x,1))
for element in rdd2.collect():
    print(element)

data = [('James','Smith','M',30),
        ('Anna','Rose','F',41),
        ('Robert','Williams','M',62),
        ]

columns = ["firstname","lastname","gender","salary"]
df = spark.createDataFrame(data=data, schema = columns)
df.show()
```

VN2 Solutions Pvt. Ltd.

```
rdd2=df.rdd.map(lambda x:  
    (x[0]+","+x[1],x[2],x[3]*2)  
 )  
df2=rdd2.toDF(["name","gender","new_salary"] )  
df2.show()  
  
#Referring Column Names  
rdd2=df.rdd.map(lambda x:  
    (x["firstname"]+","+x["lastname"],x["gender"],x["salary"]*2)  
 )  
  
#Referring Column Names  
rdd2=df.rdd.map(lambda x:  
    (x.firstname+","+x.lastname,x.gender,x.salary*2)  
 )  
  
def func1(x):  
    firstName=x.firstname  
    lastName=x.lastname  
    name=firstName+","+lastName  
    gender=x.gender.lower()  
    salary=x.salary*2  
    return (name,gender,salary)  
  
rdd2=df.rdd.map(lambda x: func1(x))
```

PySpark flatMap() Transformation

PySpark `flatMap()` is a transformation operation that flattens the RDD/DataFrame (array/map DataFrame columns) after applying the function on every element and returns a new PySpark RDD/DataFrame. In this article, you will learn the syntax and usage of the PySpark `flatMap()` with an example.

First, let's create an RDD from the list.

```
data = ["Project Gutenberg's",  
       "Alice's Adventures in Wonderland",  
       "Project Gutenberg's",  
       "Adventures in Wonderland",  
       "Project Gutenberg's"]  
rdd=spark.sparkContext.parallelize(data)  
for element in rdd.collect():  
    print(element)
```

This yields the below output

```
Project Gutenberg's  
Alice's Adventures in Wonderland  
Project Gutenberg's  
Adventures in Wonderland  
Project Gutenberg's
```

flatMap() Syntax

```
flatMap(f, preservesPartitioning=False)
```

flatMap() Example

Now, let's see with an example of how to apply a flatMap() transformation on RDD. In the below example, first, it splits each record by space in an RDD and finally flattens it. Resulting RDD consists of a single word on each record.

```
rdd2=rdd.flatMap(lambda x: x.split(" "))  
for element in rdd2.collect():  
    print(element)
```

This yields below output.

```
Project  
Gutenberg's  
Alice's  
Adventures  
in  
Wonderland  
Project  
Gutenberg's  
Adventures  
in  
Wonderland  
Project  
Gutenberg's
```

Complete PySpark flatMap() example

Below is the complete example of flatMap() function that works with RDD.

VN2 Solutions Pvt. Ltd.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

data = ["Project Gutenberg's",
        "Alice's Adventures in Wonderland",
        "Project Gutenberg's",
        "Adventures in Wonderland",
        "Project Gutenberg's"]
rdd=spark.sparkContext.parallelize(data)
for element in rdd.collect():
    print(element)

#Flatmap
rdd2=rdd.flatMap(lambda x: x.split(" "))
for element in rdd2.collect():
    print(element)
```

Using flatMap() transformation on DataFrame

Unfortunately, PySpark DataFrame doesn't have flatMap() transformation however, DataFrame has explode() SQL function that is used to flatten the column. Below is a complete example.

```
import pyspark
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('pyspark-by-examples').getOrCreate()

arrayData = [
    ('James',['Java','Scala'], {'hair':'black','eye':'brown'}),
    ('Michael',['Spark','Java',None], {'hair':'brown','eye':None}),
    ('Robert',['CSharp',''], {'hair':'red','eye':''}),
    ('Washington',None,None),
    ('Jefferson',['1','2'],{})]
df = spark.createDataFrame(data=arrayData, schema =
['name','knownLanguages','properties'])

from pyspark.sql.functions import explode
df2 = df.select(df.name,explode(df.knownLanguages))
df2.printSchema()
df2.show()
```

This example flattens the array column “knownLanguages” and yields below output

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

```
+-----+-----+
|   name|  col|
+-----+-----+
| James| Java|
| James| Scala|
| Michael| Spark|
| Michael| Java|
| Michael| null|
| Robert|CSharp|
| Robert|    |
|Jefferson|   1|
|Jefferson|   2|
+-----+-----+
```

PySpark – Loop/Iterate Through Rows in DataFrame

PySpark provides `map()`, `mapPartitions()` to loop/iterate through rows in RDD/DataFrame to perform the complex transformations, and these two returns the same number of records as in the original DataFrame but the number of columns could be different (after add/update).

PySpark also provides `foreach()` & `foreachPartitions()` actions to loop/iterate through each Row in a DataFrame but these two returns nothing, In this article, I will explain how to use these methods to get DataFrame column values and process.

- [Using map\(\) to loop through DataFrame](#)
- [Using foreach\(\) to loop through DataFrame](#)
- [Using pandas\(\) to Iterate](#)
- [Collect Data As List and Loop Through in Python](#)

PySpark Loop Through Rows in DataFrame Examples

In order to explain with examples, let's create a DataFrame

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

data = [('James','Smith','M',30),('Anna','Rose','F',41),
        ('Robert','Williams','M',62),
        ]
columns = ["firstname","lastname","gender","salary"]
df = spark.createDataFrame(data=data, schema = columns)
df.show()
+-----+-----+-----+-----+
```

VN2 Solutions Pvt. Ltd.

```
|firstname|lastname|gender|salary|
+-----+-----+-----+
| James| Smith| M| 30|
| Anna| Rose| F| 41|
| Robert|Williams| M| 62|
+-----+-----+-----+
```

Mostly for simple computations, instead of iterating through using `map()` and `foreach()`, you should use either [DataFrame select\(\)](#) or [DataFrame withColumn\(\)](#) in conjunction with PySpark SQL functions.

```
from pyspark.sql.functions import concat_ws,col,lit
df.select(concat_ws(",","df.firstname,df.lastname).alias("name"), \
    df.gender,lit(df.salary*2).alias("new_salary")).show()
+-----+-----+-----+
|      name|gender|new_salary|
+-----+-----+-----+
| James,Smith|   M|     60|
| Anna,Rose|   F|     82|
| Robert,Williams|   M|    124|
+-----+-----+-----+
```

Below I have `map()` example to achieve same output as above.

Using `map()` to Loop Through Rows in DataFrame

[PySpark map\(\) Transformation](#) is used to loop/iterate through the PySpark DataFrame/RDD by applying the transformation function (`lambda`) on every element (Rows and Columns) of RDD/DataFrame. PySpark doesn't have a `map()` in DataFrame instead it's in RDD hence we need to convert DataFrame to RDD first and then use the `map()`. It returns an RDD and you should [Convert RDD to PySpark DataFrame](#) if needed. If you have a heavy initialization use PySpark `mapPartitions()` transformation instead of `map()`, as with `mapPartitions()` heavy initialization executes only once for each partition instead of every record.

```
# Refering columns by index.
rdd=df.rdd.map(lambda x:
    (x[0]+","+x[1],x[2],x[3]*2)
)
df2=rdd.toDF(["name","gender","new_salary"])
df2.show()
```

The above example iterates through every row in a DataFrame by applying transformations to the data, since I need a DataFrame back, I have converted the result of RDD to DataFrame with new column names. Note that here I have used index to get the column values, alternatively, you can also refer to the DataFrame column names while iterating.

VN2 Solutions Pvt. Ltd.

```
# Referring Column Names
rdd2=df.rdd.map(lambda x:
    (x["firstname"]+","+x["lastname"],x["gender"],x["salary"]*2)
)
```

Another alternative

```
# Referring Column Names
rdd2=df.rdd.map(lambda x:
    (x.firstname+","+x.lastname,x.gender,x.salary*2)
)
```

You can also create a custom function to perform an operation.

Below `func1()` function executes for every DataFrame row from the lambda function.

```
# By Calling function
def func1(x):
    firstName=x.firstname
    lastName=x.lastName
    name=firstName+","+lastName
    gender=x.gender.lower()
    salary=x.salary*2
    return (name,gender,salary)

rdd2=df.rdd.map(lambda x: func1(x))
```

Using `foreach()` to Loop Through Rows in DataFrame

Similar to `map()`, `foreach()` also applied to every row of DataFrame, the difference being `foreach()` is an action and it returns nothing. Below are some examples to iterate through DataFrame using `foreach`.

```
# Foreach example
def f(x): print(x)
df.foreach(f)

# Another example
df.foreach(lambda x:
    print("Data
==>"+x["firstname"]+","+x["lastname"]+","+x["gender"]+","+str(x["salary"]*2))
)
```

Using pandas() to Iterate

If you have a small dataset, you can also [Convert PySpark DataFrame to Pandas](#) and use pandas to iterate through. Use `spark.sql.execution.arrow.enabled` config to enable Apache Arrow with Spark. Apache Spark uses Apache Arrow which is an in-memory columnar format to transfer the data between Python and JVM.

```
# Using pandas
import pandas as pd
spark.conf.set("spark.sql.execution.arrow.enabled", "true")
pandasDF = df.toPandas()
for index, row in pandasDF.iterrows():
    print(row['firstname'], row['gender'])
```

Collect Data As List and Loop Through

You can also [Collect the PySpark DataFrame to Driver](#) and iterate through Python, you can also use `toLocalIterator()`.

```
# Collect the data to Python List
dataCollect = df.collect()
for row in dataCollect:
    print(row['firstname'] + "," + row['lastname'])

#Using toLocalIterator()
dataCollect=df.rdd.toLocalIterator()
for row in dataCollect:
    print(row['firstname'] + "," + row['lastname'])
```

PySpark Random Sample with Example

PySpark provides

a `pyspark.sql.DataFrame.sample()`, `pyspark.sql.DataFrame.sampleBy()`, `RDD.sample()`, and `RDD.takeSample()` methods to get the random sampling subset from the large dataset, In this article I will explain with Python examples.

If you are working as a Data Scientist or Data analyst you are often required to analyze a large dataset/file with billions or trillions of records, processing these large datasets takes some time hence during the analysis phase it is recommended to use a random subset sample from the large files.

1. PySpark SQL sample() Usage & Examples

PySpark sampling (`pyspark.sql.DataFrame.sample()`) is a mechanism to get random sample records from the dataset, this is helpful when you have a larger dataset and wanted to analyze/test a subset of the data for example 10% of the original file. Below is the syntax of the `sample()` function.

```
sample(withReplacement, fraction, seed=None)
```

`fraction` – Fraction of rows to generate, range [0.0, 1.0]. Note that it doesn't guarantee to provide the exact number of the fraction of records.

`seed` – Seed for sampling (default a random seed). Used to reproduce the same random sampling.

`withReplacement` – Sample with replacement or not (default False).

Let's see some examples.

1.1 Using `fraction` to get a random sample in PySpark

By using fraction between 0 to 1, it returns the approximate number of the fraction of the dataset. For example, `0.1` returns 10% of the rows. However, this does not guarantee it returns the exact 10% of the records.

Note: If you run these examples on your system, you may see different results.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder \
    .master("local[1]") \
    .appName("SparkByExamples.com") \
    .getOrCreate()

df=spark.range(100)
print(df.sample(0.06).collect())
//Output: [Row(id=0), Row(id=2), Row(id=17), Row(id=25), Row(id=26), Row(id=44),
Row(id=80)]
```

My DataFrame has 100 records and I wanted to get 6% sample records which are 6 but the `sample()` function returned 7 records. This proves the `sample` function doesn't return the exact fraction specified.

1.2 Using `seed` to reproduce the same Samples in PySpark

Every time you run a `sample()` function it returns a different set of sampling records, however sometimes during the development and testing phase you may need to regenerate the same sample every time as you need to compare the results from your previous run. To get consistent same random sampling uses the same slice value for every run. Change slice value to get different results.

```
print(df.sample(0.1,123).collect())
```

VN2 Solutions Pvt. Ltd.

```
//Output: 36,37,41,43,56,66,69,75,83  
  
print(df.sample(0.1,123).collect())  
//Output: 36,37,41,43,56,66,69,75,83  
  
print(df.sample(0.1,456).collect())  
//Output: 19,21,42,48,49,50,75,80
```

Here, first 2 examples I have used seed value 123 hence the sampling results are the same and for the last example, I have used 456 as a seed value generate different sampling records.

1.3 Sample withReplacement (May contain duplicates)

some times you may need to get a random sample with repeated values. By using the value true, results in repeated values.

```
print(df.sample(True,0.3,123).collect()) //with Duplicates  
//Output: 0,5,9,11,14,14,16,17,21,29,33,41,42,52,52,54,58,65,65,71,76,79,85,96  
print(df.sample(0.3,123).collect()) // No duplicates  
//Output:  
0,4,17,19,24,25,26,36,37,41,43,44,53,56,66,68,69,70,71,75,76,78,83,84,88,94,96,97,98
```

On first example, values 14, 52 and 65 are repeated values.

1.4 Stratified sampling in PySpark

You can get Stratified sampling in PySpark without replacement by using `sampleBy()` method. It returns a sampling fraction for each stratum. If a stratum is not specified, it takes zero as the default.

sampleBy() Syntax

```
sampleBy(col, fractions, seed=None)  
col – column name from DataFrame
```

fractions – It's Dictionary type takes key and value.

sampleBy() Example

```
df2=df.select((df.id % 3).alias("key"))  
print(df2.sampleBy("key", {0: 0.1, 1: 0.2},0).collect())  
//Output: [Row(key=0), Row(key=1), Row(key=1), Row(key=1), Row(key=0),  
Row(key=1), Row(key=1), Row(key=0), Row(key=1), Row(key=1), Row(key=1)]
```

2. PySpark RDD Sample

PySpark RDD also provides `sample()` function to get a random sampling, it also has another signature `takeSample()` that returns an `Array[T]`.

RDD sample() Syntax & Example

VN2 Solutions Pvt. Ltd.

PySpark RDD sample() function returns the random sampling similar to DataFrame and takes a similar types of parameters but in a different order. Since I've already covered the explanation of these parameters on DataFrame, I will not be repeating the explanation on RDD, If not already read I recommend reading the DataFrame section above.

sample() of RDD returns a new RDD by selecting random sampling. Below is a syntax.

```
sample(self, withReplacement, fraction, seed=None)
```

Below is an example of RDD sample() function

```
rdd = spark.sparkContext.range(0,100)
print(rdd.sample(False,0.1,0).collect())
//Output: [24, 29, 41, 64, 86]
print(rdd.sample(True,0.3,123).collect())
//Output: [0, 11, 13, 14, 16, 18, 21, 23, 27, 31, 32, 32, 48, 49, 49, 53, 54, 72, 74, 77,
77, 83, 88, 91, 93, 98, 99]
```

RDD takeSample() Syntax & Example

RDD takeSample() is an action hence you need to careful when you use this function as it returns the selected sample records to driver memory. Returning too much data results in an out-of-memory error similar to [collect\(\)](#).

Syntax of RDD takeSample() .

```
takeSample(self, withReplacement, num, seed=None)
```

Example of RDD takeSample()

```
print(rdd.takeSample(False,10,0))
//Output: [58, 1, 96, 74, 29, 24, 32, 37, 94, 91]
print(rdd.takeSample(True,30,123))
//Output: [43, 65, 39, 18, 84, 86, 25, 13, 40, 21, 79, 63, 7, 32, 26,
```

PySpark fillna() & fill() – Replace NULL/None Values

In PySpark, `DataFrame.fillna()` or `DataFrameNaFunctions.fill()` is used to replace NULL/None values on all or selected multiple DataFrame columns with either **zero(0)**, **empty string, space, or any constant literal** values.

VN2 Solutions Pvt. Ltd.

While working on PySpark DataFrame we often need to replace null values since certain operations on null value return error hence, we need to graciously handle nulls as the first step before processing. Also, while writing to a file, it's always best practice to replace null values, not doing this result nulls on the output file.

As part of the cleanup, sometimes you may need to [Drop Rows with NULL/None Values in PySpark DataFrame](#) and [Filter Rows by checking IS NULL/NOT NULL conditions](#).

In this article, I will use both `fill()` and `fillna()` to replace null/none values with an empty string, constant value, and zero(0) on Dataframe columns integer, string with Python examples.

- [PySpark fillna\(\) and fill\(\) Syntax](#)
- [Replace NULL/None Values with Zero \(0\)](#)
- [Replace NULL/None Values with Empty String](#)

Before we start, Let's [read a CSV into PySpark DataFrame](#) file, where we have no values on certain rows of String and Integer columns, PySpark assigns null values to these no value columns.

The file we are using here is available at [GitHub small zipcode.csv](#)

```
from pyspark.sql import SparkSession
spark = SparkSession.builder \
    .master("local[1]") \
    .appName("SparkByExamples.com") \
    .getOrCreate()

filePath="resources/small_zipcode.csv"
df = spark.read.options(header='true', inferSchema='true') \
    .csv(filePath)

df.printSchema()
df.show(truncate=False)
```

This yields the below output. As you see columns type, city and population columns have null values.

```
+---+-----+-----+-----+-----+
|id |zipcode|type   |city           |state|population|
+---+-----+-----+-----+-----+
|1  |704    |STANDARD|null          |PR   |30100    |
|2  |704    |null    |PASEO COSTA DEL SUR|PR   |null      |
|3  |709    |null    |BDA SAN LUIS     |PR   |3700     |
|4  |76166  |UNIQUE  |CINGULAR WIRELESS|TX   |84000    |
|5  |76177  |STANDARD|null          |TX   |null      |
+---+-----+-----+-----+-----+
```

Now, let's see how to replace these null values.

VN2 Solutions Pvt. Ltd.

PySpark fillna() & fill() Syntax

PySpark provides `DataFrame.fillna()` and `DataFrameNaFunctions.fill()` to replace NULL/None values. These two are aliases of each other and returns the same results.

```
fillna(value, subset=None)
fill(value, subset=None)
```

- **value** – Value should be the data type of int, long, float, string, or dict. Value specified here will be replaced for NULL/None values.
- **subset** – This is optional, when used it should be the subset of the column names where you wanted to replace NULL/None values.

PySpark Replace NULL/None Values with Zero (0)

PySpark `fill(value:Long)` signatures that are available in `DataFrameNaFunctions` is used to replace NULL/None values with numeric values either zero(0) or any constant value for all integer and long datatype columns of PySpark DataFrame or Dataset.

```
#Replace 0 for null for all integer columns
df.na.fill(value=0).show()
```

```
#Replace 0 for null on only population column
df.na.fill(value=0,subset=["population"]).show()
```

Above both statements yields the same output, since we have just an integer column `population` with null values Note that it replaces only Integer columns since our value is 0.

```
+---+-----+-----+-----+-----+
|id |zipcode|type   |city      |state|population|
+---+-----+-----+-----+-----+
|1  |704    |STANDARD|null      |PR   |30100    |
|2  |704    |null    |PASEO COSTA DEL SUR|PR   |0        |
|3  |709    |null    |BDA SAN LUIS     |PR   |3700     |
|4  |76166  |UNIQUE  |CINGULAR WIRELESS |TX   |84000    |
|5  |76177  |STANDARD|null      |TX   |0        |
+---+-----+-----+-----+-----+
```

PySpark Replace Null/None Value with Empty String

Now let's see how to replace NULL/None values with an empty string or any constant values String on all DataFrame String columns.

```
df.na.fill("").show(false)
```

Yields below output. This replaces all String type columns with empty/blank string for all NULL values.

```
+---+-----+-----+-----+
|id |zipcode|type   |city      |state|population|
+---+-----+-----+-----+
```

VN2 Solutions Pvt. Ltd.

1	704	STANDARD	PR	30100		
2	704	PASEO COSTA DEL SUR	PR	null		
3	709	BDA SAN LUIS	PR	3700		
4	76166	UNIQUE	CINGULAR WIRELESS	TX	84000	
5	76177	STANDARD	TX	null		

Now, let's replace NULL's on specific columns, below example replace column `type` with empty string and column `city` with value "unknown".

```
df.na.fill("unknown",["city"]) \
    .na.fill("",["type"]).show()
```

Yields below output. This replaces null values with an empty string for `type` column and replaces with a constant value "unknown" for `city` column.

1	704	STANDARD	unknown	PR	30100	
2	704	PASEO COSTA DEL SUR	PR	null		
3	709	BDA SAN LUIS	PR	3700		
4	76166	UNIQUE	CINGULAR WIRELESS	TX	84000	
5	76177	STANDARD	unknown	TX	null	

Alternatively you can also write the above statement as

```
df.na.fill({"city": "unknown", "type": ""}) \
    .show()
```

Complete Code

Below is complete code with Scala example. You can use it by copying it from here or use the GitHub to download the source code.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder \
    .master("local[1]") \
    .appName("SparkByExample.com") \
    .getOrCreate()

filePath="resources/small_zipcode.csv"
df = spark.read.options(header='true', inferSchema='true') \
    .csv(filePath)

df.printSchema()
```

```
df.show(truncate=False)

df.fillna(value=0).show()
df.fillna(value=0,subset=["population"]).show()
df.na.fill(value=0).show()
df.na.fill(value=0,subset=["population"]).show()

df.fillna(value "").show()
df.na.fill(value "").show()

df.fillna("unknown",["city"]) \
    .fillna("",["type"]).show()

df.fillna({"city": "unknown", "type": ""}) \
    .show()

df.na.fill("unknown",["city"]) \
    .na.fill("",["type"]).show()

df.na.fill({"city": "unknown", "type": ""}) \
    .show()
```

PySpark Pivot and Unpivot DataFrame

PySpark pivot() function is used to rotate/transpose the data from one column into multiple Dataframe columns and back using unpivot(). Pivot() It is an aggregation where one of the grouping columns values is transposed into individual columns with distinct data.

This tutorial describes and provides a PySpark example on how to create a Pivot table on DataFrame and Unpivot back.

- [Pivot PySpark DataFrame](#)
- [Pivot Performance improvement in PySpark 2.0](#)
- [Unpivot PySpark DataFrame](#)
- [Pivot or Transpose without aggregation](#)

Let's create a [PySpark DataFrame](#) to work with.

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import expr
#Create spark session
```

VN2 Solutions Pvt. Ltd.

```
data = [("Banana",1000,"USA"), ("Carrots",1500,"USA"), ("Beans",1600,"USA"), \
        ("Orange",2000,"USA"),("Orange",2000,"USA"),("Banana",400,"China"), \
        ("Carrots",1200,"China"),("Beans",1500,"China"),("Orange",4000,"China"), \
        ("Banana",2000,"Canada"),("Carrots",2000,"Canada"),("Beans",2000,"Mexico")]

columns= ["Product","Amount","Country"]
df = spark.createDataFrame(data = data, schema = columns)
df.printSchema()
df.show(truncate=False)
```

DataFrame 'df' consists of 3 columns Product, Amount, and Country as shown below.

```
root
|-- Product: string (nullable = true)
|-- Amount: long (nullable = true)
|-- Country: string (nullable = true)

+-----+-----+-----+
|Product|Amount|Country|
+-----+-----+-----+
|Banana |1000  |USA    |
|Carrots|1500  |USA    |
|Beans  |1600  |USA    |
|Orange |2000  |USA    |
|Orange |2000  |USA    |
|Banana |400   |China  |
|Carrots|1200  |China  |
|Beans  |1500  |China  |
|Orange |4000  |China  |
|Banana |2000  |Canada |
|Carrots|2000  |Canada |
|Beans  |2000  |Mexico |
+-----+-----+-----+
```

Pivot PySpark DataFrame

PySpark SQL provides `pivot()` function to rotate the data from one column into multiple columns. It is an aggregation where one of the grouping columns values is transposed into individual columns with distinct data. To get the total amount exported to each country of each product, will do group by `Product`, pivot by `Country`, and the sum of `Amount`.

```
pivotDF = df.groupBy("Product").pivot("Country").sum("Amount")
pivotDF.printSchema()
pivotDF.show(truncate=False)
```

This will transpose the countries from DataFrame rows into columns and produces the below output. where ever data is not present, it represents as null by default.

```
root
|-- Product: string (nullable = true)
|-- Canada: long (nullable = true)
|-- China: long (nullable = true)
|-- Mexico: long (nullable = true)
|-- USA: long (nullable = true)

+-----+-----+-----+-----+
|Product|Canada|China|Mexico|USA |
+-----+-----+-----+-----+
|Orange |null  |4000 |null  |4000|
|Beans  |null  |1500 |2000 |1600|
|Banana |2000 |400  |null  |1000|
|Carrots|2000 |1200 |null  |1500|
+-----+-----+-----+-----+
```

Pivot Performance improvement in PySpark 2.0

version 2.0 on-wards performance has been improved on Pivot, however, if you are using the lower version; note that pivot is a very expensive operation hence, it is recommended to provide column data (if known) as an argument to function as shown below.

```
countries = ["USA","China","Canada","Mexico"]
pivotDF = df.groupBy("Product").pivot("Country", countries).sum("Amount")
pivotDF.show(truncate=False)
```

Another approach is to do two-phase aggregation. PySpark 2.0 uses this implementation in order to improve the performance [Spark-13749](#)

```
pivotDF = df.groupBy("Product","Country") \
    .sum("Amount") \
    .groupBy("Product") \
    .pivot("Country") \
    .sum("sum(Amount)") \
pivotDF.show(truncate=False)
```

The above two examples return the same output but with better performance.

Unpivot PySpark DataFrame

Unpivot is a reverse operation, we can achieve by rotating column values into rows values. PySpark SQL doesn't have unpivot function hence will use the `stack()` function. Below code converts column countries to row.

```
from pyspark.sql.functions import expr
```

VN2 Solutions Pvt. Ltd.

```
unpivotExpr = "stack(3, 'Canada', Canada, 'China', China, 'Mexico', Mexico) as  
(Country,Total)"  
unPivotDF = pivotDF.select("Product", expr(unpivotExpr)) \  
.where("Total is not null")  
unPivotDF.show(truncate=False)  
unPivotDF.show()
```

It converts pivoted column “country” to rows.

```
+-----+-----+-----+  
| Product|Country|Total|  
+-----+-----+-----+  
| Orange| China| 4000|  
| Beans | China| 1500|  
| Beans | Mexico| 2000|  
| Banana | Canada| 2000|  
| Banana | China| 400|  
| Carrots | Canada| 2000|  
| Carrots | China| 1200|  
+-----+-----+-----+
```

Transpose or Pivot without aggregation

Can we do PySpark DataFrame transpose or pivot without aggregation?

Of course you can, but unfortunately, you can't achieve using the Pivot function. However, pivoting or transposing the DataFrame structure without aggregation from rows to columns and columns to rows can be easily done using PySpark and Scala hack. please refer to [this](#) example.

Complete Example

```
import pyspark  
from pyspark.sql import SparkSession  
from pyspark.sql.functions import expr  
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()  
  
data = [("Banana",1000,"USA"), ("Carrots",1500,"USA"), ("Beans",1600,"USA"), \  
("Orange",2000,"USA"),("Orange",2000,"USA"),("Banana",400,"China"), \  
("Carrots",1200,"China"),("Beans",1500,"China"),("Orange",4000,"China"), \  
("Banana",2000,"Canada"),("Carrots",2000,"Canada"),("Beans",2000,"Mexico")]  
  
columns= ["Product","Amount","Country"]
```

```
df = spark.createDataFrame(data = data, schema = columns)
df.printSchema()
df.show(truncate=False)

pivotDF = df.groupBy("Product").pivot("Country").sum("Amount")
pivotDF.printSchema()
pivotDF.show(truncate=False)

pivotDF = df.groupBy("Product","Country") \
    .sum("Amount") \
    .groupBy("Product") \
    .pivot("Country") \
    .sum("sum(Amount)")
pivotDF.printSchema()
pivotDF.show(truncate=False)

""" unpivot """
unpivotExpr = "stack(3, 'Canada', Canada, 'China', China, 'Mexico', Mexico) as
(Country,Total)"
unPivotDF = pivotDF.select("Product", expr(unpivotExpr)) \
    .where("Total is not null")
unPivotDF.show(truncate=False)
```

PySpark partitionBy() – Write to Disk Example

- Post author:[NNK](#)
- Post category:[PySpark](#)

PySpark `partitionBy()` is a function of `pyspark.sql.DataFrameWriter` class which is used to partition the large dataset (DataFrame) into smaller files based on one or multiple columns while writing to disk, let's see how to use this with Python examples.

Partitioning the data on the file system is a way to improve the performance of the query when dealing with a large dataset in the Data lake. A Data Lake is a centralized repository of structured, semi-structured, unstructured, and binary data that allows you to store a large amount of data as-is in its original raw format.

By following the concepts in this article, it will help you to create an efficient [Data Lake](#) for production size data.

1. What is PySpark Partition?

PySpark partition is a way to split a large dataset into smaller datasets based on one or more partition keys. When you create a DataFrame from a file/table, based on certain parameters PySpark creates the DataFrame with a certain number of partitions in memory. This is one of the main advantages of PySpark DataFrame over Pandas

VN2 Solutions Pvt. Ltd.

DataFrame. Transformations on partitioned data run faster as they execute transformations parallelly for each partition.

PySpark supports partition in two ways; partition in memory (DataFrame) and partition on the disk (File system).

Partition in memory: You can partition or repartition the DataFrame by calling `repartition()` or `coalesce()` transformations.

Partition on disk: While writing the PySpark DataFrame back to disk, you can choose how to partition the data based on columns

using `partitionBy()` of `pyspark.sql.DataFrameWriter`. This is similar to [Hives partitions scheme](#).

2. Partition Advantages

As you are aware PySpark is designed to process large datasets with 100x faster than the tradition processing, this wouldn't have been possible with out partition. Below are some of the advantages using PySpark partitions on memory or on disk.

- Fast accessed to the data
- Provides the ability to perform an operation on a smaller dataset

Partition at rest (disk) is a feature of many databases and data processing frameworks and it is key to make jobs work at scale.

3. Create DataFrame

Let's [Create a DataFrame by reading a CSV file](#). You can find the dataset explained in this article at [Github zipcodes.csv file](#)

```
df=spark.read.option("header",True) \
    .csv("/tmp/resources/simple-zipcodes.csv")
df.printSchema()

#Display below schema
root
|-- RecordNumber: string (nullable = true)
|-- Country: string (nullable = true)
|-- City: string (nullable = true)
|-- Zipcode: string (nullable = true)
|-- state: string (nullable = true)
```

From above DataFrame, I will be using `state` as a partition key for our examples below.

4. PySpark partitionBy()

PySpark `partitionBy()` is a function of `pyspark.sql.DataFrameWriter` class which is used to partition based on column values while writing DataFrame to Disk/File system.

```
Syntax: partitionBy(self, *cols)
```

VN2 Solutions Pvt. Ltd.

When you write PySpark DataFrame to disk by calling `partitionBy()`, PySpark splits the records based on the partition column and stores each partition data into a sub-directory.

```
#partitionBy()
df.write.option("header",True) \
    .partitionBy("state") \
    .mode("overwrite") \
    .csv("/tmp/zipcodes-state")
```

On our DataFrame, we have a total of 6 different states hence, it creates 6 directories as shown below. The name of the sub-directory would be the partition column and its value (partition column=value).

Note: While writing the data as partitions, PySpark eliminates the partition column on the data file and adds partition column & value to the folder name, hence it saves some space on storage. To validate this, open any partition file in a text editor and check.

```
$ ls -lrt zipcodes-state
total 24
drwxr-xr-x 1 prabha 197121 0 Mar  4 21:18 's
drwxr-xr-x 1 prabha 197121 0 Mar  4 21:18 's
drwxr-xr-x 1 prabha 197121 0 Mar  4 21:18 's
drwxr-xr-x 1 prabha 197121 0 Mar  4 21:18 's
drwxr-xr-x 1 prabha 197121 0 Mar  4 21:18 's
drwxr-xr-x 1 prabha 197121 0 Mar  4 21:18 's
-rw-r--r-- 1 prabha 197121 0 Mar  4 21:18 -
```

`partitionBy("state")` example output

On each directory, you may see one or more part files (since our dataset is small, all records for each state are kept in a single part file). You can change this behavior by `repartition()` the data in memory first. Specify the number of partitions (part files) you would want for each state as an argument to the `repartition()` method.

5. PySpark `partitionBy()` Multiple Columns

You can also create partitions on multiple columns using PySpark `partitionBy()`. Just pass columns you want to partition as arguments to this method.

```
#partitionBy() multiple columns
df.write.option("header",True) \
    .partitionBy("state","city") \
    .mode("overwrite") \
    .csv("/tmp/zipcodes-state")
```

It creates a folder hierarchy for each partition; we have mentioned the first partition as state followed by city hence, it creates a city folder inside the state folder (one folder for each city in a state).

```
$ ls -lrt zipcodes-state/state=AL
total 12
drwxr-xr-x 1 prabha 197121 0 Mar  4 22:16 'city=SPRING%20GARDEN'/
drwxr-xr-x 1 prabha 197121 0 Mar  4 22:16 'city=SPRINGVILLE'/
drwxr-xr-x 1 prabha 197121 0 Mar  4 22:16 'city=SPRUCE%20PINE'/

```

partitionBy("state","city") multiple columns

6. Using repartition() and partitionBy() together

For each partition column, if you wanted to further divide into several partitions, use repartition() and partitionBy() together as explained in the below example. repartition() creates specified number of partitions in memory. The partitionBy() will write files to disk for each memory partition and partition column.

```
#Use repartition() and partitionBy() together
dfRepart.repartition(2)
    .write.option("header",True) \
    .partitionBy("state") \
    .mode("overwrite") \
    .csv("c:/tmp/zipcodes-state-more")
```

If you look at the folder, you should see only 2 part files for each state. Dataset has 6 unique states and 2 memory partitions for each state, hence the above code creates a maximum total of $6 \times 2 = 12$ part files.

```
$ ls -lrt zipcodes-state-more/state=AL
total 2
-rw-r--r-- 1 prabha 197121 65 Mar  5 18:12 part-00001-40b295a0-ea4a-4c4c-b740-aab6231612d3.c000.csv
-rw-r--r-- 1 prabha 197121 91 Mar  5 18:12 part-00000-40b295a0-ea4a-4c4c-b740-aab6231612d3.c000.csv
```

Note: Since total zipcodes for each US state differ in large, California and Texas have many whereas Delaware has very few, hence it creates a Data Skew (Total rows per each part file differs in large).

7. Data Skew – Control Number of Records per Partition File

Use option `maxRecordsPerFile` if you want to control the number of records for each partition. This is particularly helpful when your data is skewed (Having some partitions with very low records and other partitions with high number of records).

```
#partitionBy() control number of partitions
df.write.option("header",True) \
    .option("maxRecordsPerFile", 2) \
    .partitionBy("state") \
    .mode("overwrite") \
    .csv("/tmp/zipcodes-state")
```

The above example creates multiple part files for each `state` and each part file contains just 2 records.

8. Read a Specific Partition

Reads are much faster on partitioned data. This code snippet retrieves the data from a specific partition `"state=AL and city=SPRINGVILLE"`. Here, It just reads the data from that specific folder instead of scanning a whole file (when not partitioned).

```
dfSinglePart=spark.read.option("header",True) \
    .csv("c:/tmp/zipcodes-state/state=AL/city=SPRINGVILLE")
dfSinglePart.printSchema()
dfSinglePart.show()

#Displays
root
 |-- RecordNumber: string (nullable = true)
 |-- Country: string (nullable = true)
 |-- Zipcode: string (nullable = true)

+-----+-----+-----+
|RecordNumber|Country|Zipcode|
+-----+-----+-----+
|      54355|    US|  35146|
+-----+-----+-----+
```

While reading specific Partition data into DataFrame, it does not keep the partitions columns on DataFrame hence, you `printSchema()` and DataFrame is missing `state` and `city` columns.

9. PySpark SQL – Read Partition Data

This is an example of how to write a Spark DataFrame by preserving the partition columns on DataFrame.

VN2 Solutions Pvt. Ltd.

```
parqDF = spark.read.option("header",True) \
    .csv("/tmp/zipcodes-state")
parqDF.createOrReplaceTempView("ZIPCODE")
spark.sql("select * from ZIPCODE where state='AL' and city = 'SPRINGVILLE'" ) \
    .show()

#Display
+-----+-----+-----+-----+
|RecordNumber|Country|Zipcode|state|      city|
+-----+-----+-----+-----+
|      54355|    US|  35146|   AL|SPRINGVILLE|
+-----+-----+-----+-----+
```

The execution of this query is also significantly faster than the query without partition. It filters the data first on `state` and then applies filters on the `city` column without scanning the entire dataset.

10. How to Choose a Partition Column When Writing to File system?

When creating partitions you have to be very cautious with the number of partitions you would create, as having too many partitions creates too many sub-directories on HDFS which brings unnecessarily and overhead to NameNode (if you are using Hadoop) since it must keep all metadata for the file system in memory.

Let's assume you have a US census table that contains zip code, city, state, and other columns. Creating a partition on the state, splits the table into around 50 partitions, when searching for a zipcode within a state (`state='CA'` and `zipCode ='92704'`) results in faster as it needs to scan only in a `state=CA` partition directory.

Partition on zipcode may not be a good option as you might end up with too many partitions.

Another good example of partition is on the Date column. Ideally, you should partition on Year/Month but not on a date.

Conclusion

While you are create Data Lake out of Azure, HDFS or AWS you need to understand how to partition your data at rest (File system/disk), PySpark `partitionBy()` and `repartition()` help you partition the data and eliminating the Data Skew on your large datasets.

PySpark ArrayType Column With

Examples: PySpark `pyspark.sql.types.ArrayType` (`ArrayType` extends `DataType` class) is used to define an array data type column on DataFrame that holds the same type of elements, In this article, I will explain how to create a DataFrame `ArrayType` column using `org.apache.spark.sql.types.ArrayType` class and applying some SQL functions on the array columns with examples.

While working with structured files ([Avro](#), [Parquet](#) e.t.c) or semi-structured ([JSON](#)) files, we often get data with complex structures like `MapType`, `ArrayType`, `StructType` e.t.c. I will try my best to cover some mostly used functions on `ArrayType` columns.

What is PySpark `ArrayType`

PySpark `ArrayType` is a collection data type that extends the `DataType` class which is a superclass of all types in PySpark. All elements of `ArrayType` should have the same type of elements.

Create PySpark `ArrayType`

You can create an instance of an `ArrayType` using `ArrayType()` class, This takes arguments `valueType` and one optional argument `valueContainsNull` to specify if a value can accept null, by default it takes True. `valueType` should be a PySpark type that extends `DataType` class.

```
from pyspark.sql.types import StringType, ArrayType
arrayCol = ArrayType(StringType(), False)
```

Create PySpark `ArrayType` Column Using `StructType`

Let's create a DataFrame with few array columns by using [PySpark StructType & StructField classes](#).

```
data = [
("James,,Smith",["Java","Scala","C++"],["Spark","Java"],"OH","CA"),
("Michael,Rose,",["Spark","Java","C++"],["Spark","Java"],"NY","NJ"),
("Robert,,Williams",["CSharp","VB"],["Spark","Python"],"UT","NV")
]
```

```
from pyspark.sql.types import StringType, ArrayType, StructType, StructField
schema = StructType([
    StructField("name", StringType(), True),
    StructField("languagesAtSchool", ArrayType(StringType()), True),
    StructField("languagesAtWork", ArrayType(StringType()), True),
    StructField("currentState", StringType(), True),
```

VN2 Solutions Pvt. Ltd.

```
StructField("previousState", StringType(), True)
])

df = spark.createDataFrame(data=data,schema=schema)
df.printSchema()
df.show()
```

This snippet creates two Array columns `languagesAtSchool` and `languagesAtWork` which defines languages learned at School and languages using at work. For the rest of the article, I will use these array columns of DataFrame and provide examples of PySpark SQL array functions. `printSchema()` and `show()` from above snippet display below output.

```
root
|-- name: string (nullable = true)
|-- languagesAtSchool: array (nullable = true)
|   |-- element: string (containsNull = true)
|-- languagesAtWork: array (nullable = true)
|   |-- element: string (containsNull = true)
|-- currentState: string (nullable = true)
|-- previousState: string (nullable = true)
+-----+-----+-----+-----+
|      name | languagesAtSchool | languagesAtWork | currentState | previousState |
+-----+-----+-----+-----+
| James,,Smith | [Java, Scala, C++] | [Spark, Java] | OH | CA |
| Michael,Rose, | [Spark, Java, C++] | [Spark, Java] | NY | NJ |
| Robert,,Williams | [CSharp, VB] | [Spark, Python] | UT | NV |
+-----+-----+-----+-----+
```

PySpark ArrayType (Array) Functions

PySpark SQL provides several Array functions to work with the ArrayType column, In this section, we will see some of the most commonly used SQL functions.

`explode()`

Use `explode()` function to create a new row for each element in the given array column. There are various [PySpark SQL explode functions](#) available to work with Array columns.

```
from pyspark.sql.functions import explode
df.select(df.name,explode(df.languagesAtSchool)).show()

+-----+---+
|      name |  col |
+-----+---+
| James,,Smith |  Java |
```

VN2 Solutions Pvt. Ltd.

```
| James,,Smith| Scala|
| James,,Smith| C++|
| Michael,Rose,| Spark|
| Michael,Rose,| Java|
| Michael,Rose,| C++|
| Robert,,Williams| CSharp|
| Robert,,Williams| VB|
+-----+-----+
```

Split()

`split()` sql function returns an array type after splitting the string column by delimiter. Below example split the name column by comma delimiter.

```
from pyspark.sql.functions import split
df.select(split(df.name,",").alias("nameAsArray")).show()

+-----+
|      nameAsArray|
+-----+
| [James, , Smith]|
| [Michael, Rose, ]|
|[Robert, , Williams]|
+-----+
```

array()

Use `array()` function to create a new array column by merging the data from multiple columns. All input columns must have the same data type. The below example combines the data from `currentState` and `previousState` and creates a new column `states`.

```
from pyspark.sql.functions import array
df.select(df.name,array(df.currentState,df.previousState).alias("States")).show()

+-----+-----+
|      name| States|
+-----+-----+
| James,,Smith|[OH, CA]|
| Michael,Rose,|[NY, NJ]|
| Robert,,Williams|[UT, NV]|
+-----+-----+
```

array_contains()

`array_contains()` sql function is used to check if array column contains a value. Returns `null` if the array is `null`, `true` if the array contains the `value`, and `false` otherwise.

```
from pyspark.sql.functions import array_contains
df.select(df.name,array_contains(df.languagesAtSchool,"Java")
    .alias("array_contains")).show()

+-----+-----+
|      name|array_contains|
+-----+-----+
| James,,Smith|      true|
| Michael,Rose,|      true|
| Robert,,Williams|  false|
+-----+-----+
```

PySpark MapType (Dict) Usage with Examples

PySpark `MapType` (also called map type) is a data type to represent Python Dictionary (dict) to store key-value pair, a `MapType` object comprises three fields, `keyType` (a `DataType`), `valueType` (a `DataType`) and `valueContainsNull` (a `BooleanType`).

What is PySpark MapType

PySpark `MapType` is used to represent map key-value pair similar to python Dictionary (Dict), it extends `DataType` class which is a superclass of all types in PySpark and takes two mandatory arguments `keyType` and `valueType` of type `DataType` and one optional boolean argument `valueContainsNull`. `keyType` and `valueType` can be any type that extends the `DataType` class. for e.g `StringType`, `IntegerType`, `ArrayType`, `MapType`, `StructType` (struct) e.t.c.

1. Create PySpark MapType

In order to use `MapType` data type first, you need to import it from `pyspark.sql.types.MapType` and use `MapType()` constructor to create a map object.

```
from pyspark.sql.types import StringType, MapType
mapCol = MapType(StringType(),StringType(),False)
```

MapType Key Points:

- The First param `keyType` is used to specify the type of the key in the map.
- The Second param `valueType` is used to specify the type of the value in the map.
- Third param `valueContainsNull` is an optional boolean type that is used to specify if the value of the second param can accept `Null/None` values.
- The key of the map won't accept `None/Null` values.
- PySpark provides several SQL functions to work with `MapType`.

2. Create MapType From StructType

Let's see how to create a `MapType` by using `PySpark StructType & StructField`, `StructType()` constructor takes list of `StructField`, `StructField` takes a fieldname and type of the value.

```
from pyspark.sql.types import StructField, StructType, StringType, MapType
schema = StructType([
    StructField('name', StringType(), True),
    StructField('properties', MapType(StringType(),StringType()),True)
])
```

Now let's create a `DataFrame` by using above `StructType` schema.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()
dataDictionary = [
    ('James',{'hair':'black','eye':'brown'}),
    ('Michael',{'hair':'brown','eye':None}),
    ('Robert',{'hair':'red','eye':'black'}),
    ('Washington',{'hair':'grey','eye':'grey'}),
    ('Jefferson',{'hair':'brown','eye':None})
]
df = spark.createDataFrame(data=dataDictionary, schema = schema)
df.printSchema()
df.show(truncate=False)
df.printSchema() yields the Schema and df.show() yields the DataFrame output.
```

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

+-----+-----+
|Name    |properties          |
+-----+-----+
|James   |[eye -> brown, hair -> black]|
|Michael|[eye ->, hair -> brown]   |
|Robert  |[eye -> black, hair -> red]  |
|Washington|[eye -> grey, hair -> grey] |
|Jefferson|[eye -> , hair -> brown]  |
+-----+-----+
```

3. Access PySpark MapType Elements

Let's see how to extract the key and values from the PySpark DataFrame Dictionary column. Here I have used PySpark map transformation to read the values of properties (MapType column)

```
df3=df.rdd.map(lambda x: \
(x.name,x.properties["hair"],x.properties["eye"])) \
.toDF(["name","hair","eye"])
df3.printSchema()
df3.show()
```

```
root
|-- name: string (nullable = true)
|-- hair: string (nullable = true)
|-- eye: string (nullable = true)
```

```
+-----+----+----+
|    name| hair| eye|
+-----+----+----+
|  James|black|brown|
| Michael|brown| null|
| Robert| red|black|
|Washington| grey| grey|
| Jefferson|brown|   |
+-----+----+----+
```

Let's use another way to get the value of a key from Map using `getItem()` of `Column` type, this method takes a key as an argument and returns a value.

```
df.withColumn("hair",df.properties.getItem("hair")) \
.withColumn("eye",df.properties.getItem("eye")) \
.drop("properties") \
.show()
```

```
df.withColumn("hair",df.properties["hair"]) \
.withColumn("eye",df.properties["eye"]) \
.drop("properties") \
.show()
```

4. Functions

Below are some of the MapType Functions with examples.

4.1 – explode

```
from pyspark.sql.functions import explode
df.select(df.name,explode(df.properties)).show()

+-----+---+---+
|    name| key|value|
+-----+---+---+
|    James| eye|brown|
|    James|hair|black|
| Michael| eye| null|
| Michael|hair|brown|
| Robert| eye|black|
| Robert|hair| red|
|Washington| eye| grey|
|Washington|hair| grey|
| Jefferson| eye|   |
| Jefferson|hair|brown|
+-----+---+---+
```

4.2 map_keys() – Get All Map Keys

```
from pyspark.sql.functions import map_keys
df.select(df.name,map_keys(df.properties)).show()

+-----+-----+
|    name|map_keys(properties)|
+-----+-----+
|    James|      [eye, hair]|
| Michael|      [eye, hair]|
| Robert|      [eye, hair]|
|Washington|      [eye, hair]|
| Jefferson|      [eye, hair]|
+-----+-----+
```

In case if you wanted to get all map keys as Python List. **WARNING: This runs very slow.**

```
from pyspark.sql.functions import explode, map_keys
keysDF = df.select(explode(map_keys(df.properties))).distinct()
keysList = keysDF.rdd.map(lambda x:x[0]).collect()
print(keysList)
#['eye', 'hair']
```

4.3 map_values() – Get All map Values

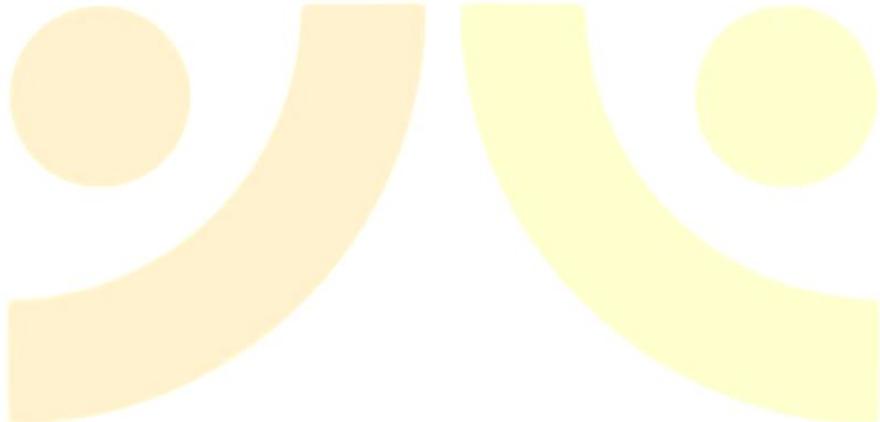
VN2 Solutions Pvt. Ltd.

```
from pyspark.sql.functions import map_values
df.select(df.name,map_values(df.properties)).show()

+-----+-----+
|    name|map_values(properties)|
+-----+-----+
|  James|[brown, black]|
| Michael|[, brown]|
| Robert|[black, red]|
|Washington|[grey, grey]|
| Jefferson|[, brown]|
+-----+-----+
```

Conclusion

MapType is a map data structure that is used to store key key-value pairs similar to Python Dictionary (Dic), keys and values type of map should be of a type that extends DataType. Key won't accept null/None values whereas map of the key can have None/Null value.



SQL

[Structured Query Language]

PLACEMENT PREPARATION [EXCLUSIVE NOTES]

MUST SAVE AND SHARE

Curated By- HIMANSHU KUMAR(LINKEDIN)

<https://www.linkedin.com/in/himanshukumarmahuri>

QUESTIONS COVERED:-

1. What is SQL?
2. What is Database ?
3. What are the differences between SQL and PL/SQL?
4. What is the difference between BETWEEN and IN operators in SQL?
5. What is the use of LIKE operator?
6. What is the use of 'WHERE' Clause?
7. What is the use of 'Having' Clause?
8. What are SQL Commands?
Explain types of SQL Commands
9. What are normalization and denormalization and why do we need them?
10. What are Nested Queries in SQL?
11. What are different types of Normalization?
12. What is Stored Procedures in SQL ?
13. What are different types of case manipulation functions available in SQL.
14. What is the difference between CHAR and VARCHAR2 datatype in SQL?
15. What is the use of CREATE, INSERT INTO, UPDATE and DELETE Clauses?
16. What is the use of ADD, DROP and MODIFY Commands?
17. What are VIEWS in SQL?
18. What are JOINS in SQL?
19. What is the use of GROUP BY Clause?
20. What are Aggregate Functions?
21. What is Cursor in SQL ?
22. What is the difference between Implicit and Explicit Cursor?
23. What is the difference between VIEW and CURSOR in SQL?
24. What are the advantages of PL/SQL functions?
25. Explain BETWEEN and IN Clause.
26. What is the difference between DROP and TRUNCATE?
27. What are Constraints in SQL?
28. What is a TRIGGER?
29. What is the use of LIMIT and OFFSET in SQL?
30. What are different types of operators present in SQL?

What is SQL?

Structured Query Language is a computer language that we use to interact with a relational database. SQL is a tool for organizing, managing, and retrieving archived data from a computer database. The original name was given by IBM as Structured English Query Language, abbreviated by the acronym SEQUEL. When data needs to be retrieved from a database, SQL is used to make the request. The DBMS processes the SQL query retrieves the requested data and returns it to us. Rather, SQL statements describe how a collection of data should be organized or what data should be extracted or added to the database.

In common usage, SQL encompasses DDL and DML commands for create, updates, modified or other operations on database structure.

SQL uses:

- **Data definition:** It is used to define the structure and organization of the stored data and relationships among the stored data items.
- **Data retrieval:** SQL can also be used for data retrieval.
- **Data manipulation:** If the user wants to add new data, remove data, or modifying in existing data then SQL provides this facility also.
- **Access control:** SQL can be used to restrict a user's ability to retrieve, add, and modify data, protecting stored data against unauthorized access.
- **Data sharing:** SQL is used to coordinate data sharing by concurrent users, ensuring that changes made by one user do not inadvertently wipe out changes made at nearly the same time by another user.

SQL also differs from other computer languages because it describes what the user wants the computer to do rather than how the computer should do it. (In more technical terms, SQL is a declarative or descriptive language rather than a procedural one.) SQL contains no IF statement for testing conditions, and no GOTO, DO, or FOR statements for program flow control. Rather, SQL statements describe how a collection of data is to be organized, or what data is to be retrieved or added to the database. The sequence of steps to do those tasks is left for the DBMS to determine.

Features of SQL:

- SQL may be utilized by quite a number of users, which include people with very little programming experience.
- SQL is a Non-procedural language.

- We can without difficulty create and replace databases in SQL. It isn't a time-consuming process.
- SQL is primarily based totally on ANSI standards.
- SQL does now no longer have a continuation individual.
- SQL is entered into SQL buffer on one or extra lines.
- SQL makes use of a termination individual to execute instructions immediately. It makes use of features to carry out a few formatting.
- It uses functions to perform some formatting.

Rules for SQL:

- A ';' is used to end SQL statements.
- Statements may be split across lines but keywords may not.
- Identifiers, operator names, literals are separated by one or more spaces or other delimiters.
- A comma(,) separates parameters without a clause.
- A space separates a clause.
- Reserved words can not be used as identifiers unless enclosed with double quotes.
- Identifiers can contain up to 30 characters.
- Identifiers must start with an alphabetic character.
- Characters and date literals must be enclosed within single quotes.
- Numeric literals can be represented by simple values.
- Comments may be enclosed between /* and */ symbols and maybe multi-line.

Role of SQL :

SQL plays many different roles:

- SQL is an interactive question language. Users type SQL instructions into an interactive SQL software to retrieve facts and show them on the screen, presenting a convenient, easy-to-use device for ad hoc database queries.
- SQL is a database programming language. Programmers embed SQL instructions into their utility packages to access the facts in a database. Both user-written packages and database software packages (consisting of document writers and facts access tools) use this approach for database access.
- SQL is a client/server language. Personal computer programs use SQL to communicate over a network with database servers that save shared facts. This client/server architecture is utilized by many famous enterprise-class applications.
- SQL is an Internet facts access language. Internet net servers that have interaction with company facts and Internet utility servers all use SQL as a widespread language for getting access to company databases, frequently through embedding SQL database get entry to inside famous scripting languages like PHP or Perl.

- SQL is a distributed database language. Distributed database control structures use SQL to assist distribute facts throughout many linked pc structures. The DBMS software program on every gadget makes use of SQL to speak with the opposite structures, sending requests for facts to get entry to.
- SQL is a database gateway language. In a pc community with a mixture of various DBMS products, SQL is frequently utilized in a gateway that lets in one logo of DBMS to speak with every other logo. SQL has for this reason emerged as a useful, effective device for linking people, pc packages, and pc structures to the facts saved in a relational database.

Finally, SQL is not a particularly structured language, especially when compared with highly structured languages such as C, Pascal, or Java. Instead, SQL statements resemble English sentences, complete with “noise words” that don’t add to the meaning of the statement but make it read more naturally. The SQL has quite a few inconsistencies and also some special rules to prevent you from constructing SQL statements that look perfectly legal but that don’t make sense.

Despite the inaccuracy of its name, SQL has emerged as the standard language for using relational databases. SQL is both a powerful language and one that is relatively easy to learn. So SQL is a database management language. The database administrator answerable for handling a minicomputer or mainframe database makes use of SQL to outline the database shape and manipulate get entry to to the saved data.

What is Database ?

The **Database** is an essential part of our life. As we encounter several activities that involve our interaction with databases, for example in the bank, in the railway station, in school, in a grocery store, etc. These are the instances where we need to store a large amount of data in one place and fetch these data easily.

A database is a collection of data that is organized, which is also called structured data. It can be accessed or stored in a computer system. It can be managed through a Database Management System (DBMS), a software used to manage data. Database refers to related data in a structured form.

In a database, data is organized into tables consisting of rows and columns and it is indexed so data can be updated, expanded, and deleted easily. Computer databases typically contain file records data like transactions money in one bank

account to another bank account, sales and customer details, fee details of students, and product details. There are different kinds of databases, ranging from the most prevalent approach, the relational database, to a distributed database, cloud database, and NoSQL databases.

- **Relational Database:**

A relational database is made up of a set of tables with data that fits into a predefined category.

- **Distributed Database:**

A distributed database is a database in which portions of the database are stored in multiple physical locations, and in which processing is dispersed or replicated among different points in a network.

- **Cloud Database:**

A cloud database is a database that typically runs on a cloud computing platform. Database service provides access to the database. Database services make the underlying software-stack transparent to the user.

These interactions are the example of a traditional database where data is of one type—that is textual. In advancement of technology has led to new applications of database systems. New media technology has made it possible to store images, video clips. These essential features are making multimedia databases.

Nowadays, people are becoming smart - before taking any decisions they analyze facts and figures related to it, which come from these databases. As the databases have made it easier to manage information, we are able to catch criminals and do deep research.

What are the differences between SQL and PL/SQL?

SQL	PL/SQL
SQL is a query execution or commanding language	PL/SQL is a complete programming language
SQL is a data-oriented language.	PL/SQL is a procedural language
SQL is very declarative in nature.	PL/SQL has a procedural nature.
It is used for manipulating data.	It is used for creating applications.
We can execute one statement at a time in SQL	We can execute blocks of statements in PL/SQL
SQL tells databases, what to do?	PL/SQL tells databases how to do.
We can embed SQL in PL/SQL	We can not embed PL/SQL in SQL

What is the difference between BETWEEN and IN operators in SQL?

BETWEEN

The BETWEEN operator is used to fetch rows based on a range of values.

For example,

```
SELECT * FROM Students
```

```
WHERE ROLL_NO BETWEEN 20 AND 30;
```

This query will select all those rows from the table Students where the value of the field ROLL_NO lies between 20 and 30.

IN

The IN operator is used to check for values contained in specific sets.

For example,

```
SELECT * FROM Students
```

```
WHERE ROLL_NO IN (20,21,23);
```

This query will select all those rows from the table Students where the value of the field ROLL_NO is either 20 or 21 or 23.

What is the use of LIKE operator?

The **LIKE** operator of SQL is used for this purpose. It is used to fetch filtered data by searching for a particular pattern in the where clause.

The Syntax for using LIKE is,

```
SELECT column1,column2 FROM table_name WHERE column_name LIKE pattern;
```

LIKE: operator name

pattern: exact value extracted from the pattern to get related data in result set.

For Example: Find all employees from the table 'Employees' whose name start with an 'A'.

The required query is:

```
SELECT * FROM Employees WHERE EmpName like 'A%' ;
```

What is the use of 'WHERE' Clause?

WHERE keyword is used for fetching filtered data in a result set.

- It is used to fetch data according to a particular criteria.
- WHERE keyword can also be used to filter data by matching patterns.

Basic Syntax:

```
SELECT column1,column2 FROM table_name WHERE column_name operator value;
```

column1 , column2: fields int the table

table_name: name of table

column_name: name of field used for filtering the data

operator: operation to be considered for filtering

value: exact value or pattern to get related data in result

List of operators that can be used with where clause:

operator	description
>	Greater Than
>=	Greater than or Equal to
<	Less Than
<=	Less than or Equal to
=	Equal to
≠	Not Equal to
BETWEEN	In an inclusive Range
LIKE	Search for a pattern
IN	To specify multiple possible values for a column

EXAMPLE -

Student				
ROLL_NO	NAME	ADDRESS	PHONE	Age
1	Ram	Delhi	XXXXXXXXXX	18
2	RAMESH	GURGAON	XXXXXXXXXX	18
3	SUJIT	ROHTAK	XXXXXXXXXX	20
4	SURESH	Delhi	XXXXXXXXXX	18
3	SUJIT	ROHTAK	XXXXXXXXXX	20
2	RAMESH	GURGAON	XXXXXXXXXX	18

Queries

To fetch record of students with age equal to 20

```
SELECT * FROM Student WHERE Age=20;
```

Output:

ROLL_NO	NAME	ADDRESS	PHONE	Age
3	SUJIT	ROHTAK	XXXXXXXXXX	20
3	SUJIT	ROHTAK	XXXXXXXXXX	20

What is the use of 'Having' Clause?

Having clause extracts the rows based on the conditions given by the user in the query.

Having clause has to be paired with the group by clause in order to extract the data.

Otherwise, an error is produced.

Syntax:

```
select select_list from table_name  
group by group_list  
having conditions
```

Example:

```
select roll_number  
from student  
having name like 'R%'
```

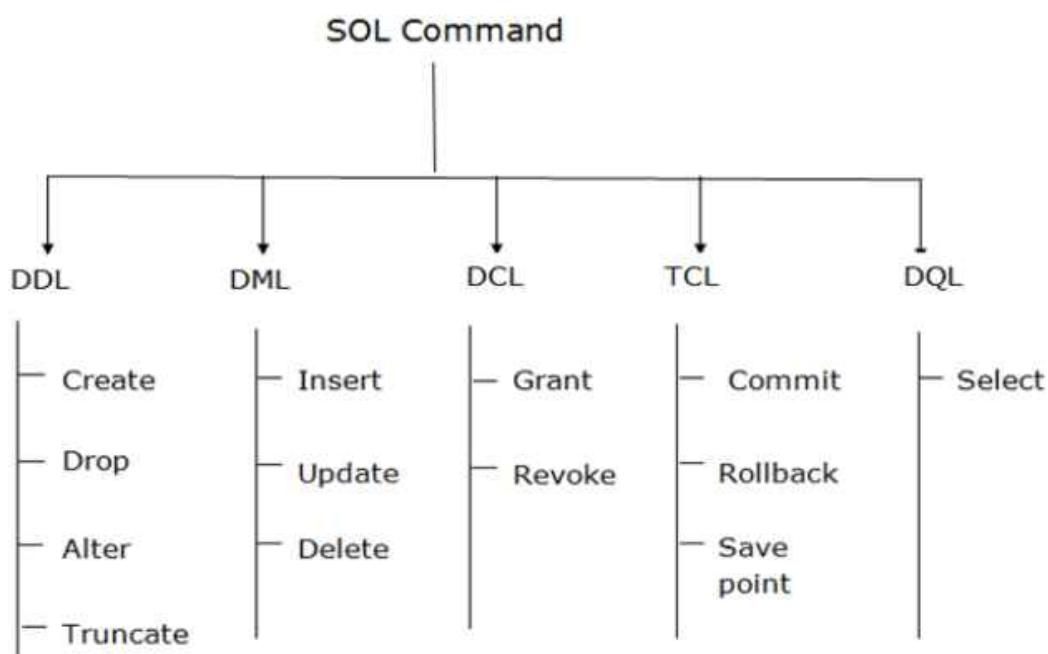
What are SQL Commands? Explain types of SQL Commands

Structured Query Language(SQL) as we all know is the database language by the use of which we can perform certain operations on the existing database, and also we can use this language to create a database. SQL uses certain commands like Create, Drop, Insert, etc. to carry out the required tasks.

These SQL commands are mainly categorized into four categories as:

1. DDL - Data Definition Language
2. DQL - Data Query Language
3. DML - Data Manipulation Language
4. DCL - Data Control Language

Though many resources claim there to be another category of SQL clauses **TCL - Transaction Control Language**. So we will see in detail about TCL as well.



1. **DDL(Data Definition Language):** DDL or Data Definition Language actually consists of the SQL commands that can be used to define the database schema. It simply deals with descriptions of the database schema and is used to create and modify the structure of database objects in the database.

Examples of DDL commands:

- o [CREATE](#) - is used to create the database or its objects (like table, index, function, views, store procedure, and triggers).
- o [DROP](#) - is used to delete objects from the database.
- o [ALTER](#) -is used to alter the structure of the database.
- o [TRUNCATE](#)-is used to remove all records from a table, including all spaces allocated for the records are removed.
- o [COMMENT](#) -is used to add comments to the data dictionary.

- o **RENAME** –is used to rename an object existing in the database.

2. DQL (Data Query Language) :

DML statements are used for performing queries on the data within schema objects. The purpose of the DQL Command is to get some schema relation based on the query passed to it.

Example of DQL:

- o **SELECT** – is used to retrieve data from the database.
3. **DML(Data Manipulation Language):** The SQL commands that deal with the manipulation of data present in the database belong to DML or Data Manipulation Language and this includes most of the SQL statements.

Examples of DML:

- o **INSERT** – is used to insert data into a table.
 - o **UPDATE** - is used to update existing data within a table.
 - o **DELETE** – is used to delete records from a database table.
4. **DCL(Data Control Language) :** DCL includes commands such as GRANT and REVOKE which mainly deal with the rights, permissions, and other controls of the database system.

Examples of DCL commands:

- o **GRANT** -gives user's access privileges to the database.
 - o **REVOKE**-withdraw user's access privileges given by using the GRANT command.
5. **TCL(transaction Control Language):** TCL commands deal with the transaction within the database.
- ### Examples of TCL commands:
- o **COMMIT**– commits a Transaction.
 - o **ROLLBACK**– rollbacks a transaction in case of any error occurs.
 - o **SAVEPOINT** –sets a savepoint within a transaction.
 - o **SET TRANSACTION** –specify characteristics for the transaction.

What are normalization and denormalization and why do we need them?

Normalization: Normalization is the method used in a database to reduce the data redundancy and data inconsistency from the table. It is the technique in which Non-redundancy and consistency data are stored in the set schema. By using normalization the number of tables is increased instead of decreased.

Denormalization: Denormalization is also the method that is used in a database. It is used to add redundancy to execute the query quickly. It is a technique in which data are combined to execute the query quickly. By using denormalization the number of tables is decreased which oppose to the normalization.

Difference between Normalization and Denormalization:

	NORMALIZATION	DENORMALIZATION
IMPLEMENTATION	Decomposes data into different tables to reduce redundancy.	Combines data to improve the access time
QUERY EXECUTION SPEED	Speed of update, delete and write operations is higher.	Speed of read operations is higher, but that of update and write operations is slower.
MEMORY CONSUMPTION	Memory consumption is less as data redundancy is less.	Memory consumption is more as redundancy is introduced.
NUMBER OF TABLES	Number of tables is more on account of decomposition of data.	Combines tables and hence number of tables are less.
DATA INTEGRITY	Data integrity is maintained.	Data integrity might not be maintained.

What are Nested Queries in SQL?

In nested queries, a query is written inside a query. The result of the inner query is used in the execution of the outer query. We will use **STUDENT**, **COURSE**,

STUDENT_COURSE tables for understanding nested queries.

STUDENT

S_ID	S_NAME	S_ADDRESS	S_PHONE	S_AGE
S1	RAM	DELHI	9455123451	18
S2	RAMESH	GURGAON	9652431543	18
S3	SUJIT	ROHTAK	9156253131	20
S4	SURESH	DELHI	9156768971	18

COURSE

C_ID	C_NAME
C1	DSA
C2	Programming
C3	DBMS

STUDENT_COURSE

S_ID	C_ID
S1	C1
S1	C3
S2	C1
S3	C2
S4	C2
S4	C3

There are mainly two types of nested queries:

- **Independent Nested Queries:** In independent nested queries, query execution starts from innermost query to outermost queries. The execution of the inner query is independent of the outer query, but the result of the inner query is used in the execution of the outer query. Various operators like IN, NOT IN, ANY, ALL etc are used in writing independent nested queries.

IN: If we want to find out **S_ID** who are enrolled in **C_NAME** 'DSA' or 'DBMS', we can write it with the help of independent nested query and IN operator. From **COURSE** table, we can find out **C_ID** for **C_NAME** 'DSA' or DBMS' and we can use these **C_IDs** for finding **S_IDs** from **STUDENT_COURSE** TABLE.

STEP 1: Finding **C_ID** for **C_NAME** ='DSA' or 'DBMS'

Select **C_ID** from **COURSE** where **C_NAME** = 'DSA' or **C_NAME** = 'DBMS'

STEP 2: Using **C_ID** of step 1 for finding **S_ID**

Select **S_ID** from **STUDENT_COURSE** where **C_ID** IN

(SELECT **C_ID** from **COURSE** where **C_NAME** = 'DSA' or **C_NAME**='DBMS');

The inner query will return a set with members C1 and C3 and the outer query will return those **S_IDs** for which **C_ID** is equal to any member of the set (C1 and C3 in this case). So, it will return S1, S2 and S4.

Note: If we want to find out names of **STUDENTs** who have either enrolled in 'DSA' or 'DBMS', it can be done as:

Select **S_NAME** from **STUDENT** where **S_ID** IN

(Select **S_ID** from **STUDENT_COURSE** where **C_ID** IN

(SELECT **C_ID** from **COURSE** where **C_NAME**='DSA' or **C_NAME**='DBMS'));

NOT IN: If we want to find out **S_IDs** of **STUDENTs** who have neither enrolled in 'DSA' nor in 'DBMS', it can be done as:

Select **S_ID** from **STUDENT** where **S_ID** NOT IN

(Select **S_ID** from **STUDENT_COURSE** where **C_ID** IN

(SELECT **C_ID** from **COURSE** where **C_NAME**='DSA' or **C_NAME**='DBMS'));

The innermost query will return a set with members C1 and C3. Second inner query will return those **S_IDs** for which **C_ID** is equal to any member of set (C1 and C3 in this case) which are S1, S2 and S4. The outermost query will return those **S_IDs** where **S_ID** is not a member of set (S1, S2 and S4). So it will return S3.

- **Co-related Nested Queries:** In co-related nested queries, the output of inner query depends on the row which is being currently executed in outer query. e.g.; If we want to find out **S_NAME** of **STUDENTs** who are enrolled in **C_ID 'C1'**, it can be done with the help of co-related nested query as:

```
Select S_NAME from STUDENT S where EXISTS
( select * from STUDENT_COURSE SC where S.S_ID=SC.S_ID and
SC.C_ID='C1');
```

For each row of **STUDENT S**, it will find the rows from **STUDENT_COURSE** where **S.S_ID = SC.S_ID** and **SC.C_ID='C1'**. If for a **S_ID** from **STUDENT S**, atleast a row exists in **STUDENT_COURSE SC** with **C_ID='C1'**, then inner query will return true and corresponding **S_ID** will be returned as output.

What are different types of Normalization?

There are four types of normalization:

- 1. 1NF**: It is known as the first normal form and is the simplest type of normalization that you can implement in a database. A table to be in its first normal form should satisfy the following conditions:
 - Every column must have a single value and should be atomic.
 - Duplicate columns from the same table should be removed.
 - Separate tables should be created for each group of related data and each row should be identified with a unique column.

2. 2NF: It is known as the second normal form. A table to be in its second normal form should satisfy the following conditions:

- The table should be in its 1NF i.e. satisfy all the conditions of 1NF.
- Every non-prime attribute of the table should be fully functionally dependent on the primary key i.e. every non-key attribute should be dependent on the primary key in such a way that if any key element is deleted then even the non_key element will be saved in the database.

3. 3NF: It is known as the third normal form. A table to be in its second normal form should satisfy the following conditions:

- The table should be in its 2NF i.e. satisfy all the conditions of 2NF.
- There is no transitive functional dependency of one attribute on any attribute in the same table.

4. BCNF: BCNF stands for Boyce-Codd Normal Form and is an advanced form of

3NF. It is also referred to as 3.5NF for the same reason. A table to be in its BCNF normal form should satisfy the following conditions:

- o The table should be in its 3NF i.e. satisfy all the conditions of 3NF.
- o For every functional dependency of any attribute A on B ($A \rightarrow B$), A should be the super key of the table. It simply implies that A can't be a non-prime attribute if B is a prime attribute

What is Stored Procedures in SQL ?

Stored Procedures are created to perform one or more DML operations on Database. It is nothing but the group of SQL statements that accepts some input in the form of parameters and performs some task and may or may not returns a value.

Syntax : Creating a Procedure

```
CREATE or REPLACE PROCEDURE name(parameters)

IS

variables;

BEGIN

//statements;

END;
```

The most important part is parameters. Parameters are used to pass values to the Procedure. There are 3 different types of parameters, they are as follows:

1. **IN:**

This is the Default Parameter for the procedure. It always receives the values from calling program.

2. **OUT:**

This parameter always sends the values to the calling program.

3. **IN OUT:**

This parameter performs both the operations. It Receives value from as well as sends the values to the calling program.

Example:

Imagine a table named with emp_table stored in Database. We are Writing a Procedure to update a Salary of Employee with 1000.

```
CREATE or REPLACE PROCEDURE INC_SAL(eno IN NUMBER, up_sal OUT NUMBER)
IS
BEGIN
UPDATE emp_table SET salary = salary+1000 WHERE emp_no = eno;
COMMIT;
SELECT sal INTO up_sal FROM emp_table WHERE emp_no = eno;
END;
```

- Declare a Variable to Store the value coming out from Procedure :

```
VARIABLE v NUMBER;
```

- Execution of the Procedure:

```
EXECUTE INC_SAL(1002, :v);
```

- To check the updated salary use SELECT statement:

```
SELECT * FROM emp_table WHERE emp_no = 1002;
```

- or Use print statement:

PRINT :v

What are different types of case manipulation functions available in SQL.

There are three types of case manipulation functions available in SQL. They are-

LOWER: The purpose of this function is to return the string in lowercase. It takes a string as an argument and returns the string by converting it into lower case.

Syntax: `LOWER('string')`

UPPER: The purpose of this function is to return the string in uppercase. It takes a string as an argument and returns the string by converting it into uppercase.

Syntax:

`UPPER('string')`

INITCAP: The purpose of this function is to return the string with the first letter in uppercase and the rest of the letters in lowercase.

Syntax: `INITCAP('string')`

What is the difference between CHAR and VARCHAR2 datatype in SQL?

Both of these data types are used for characters, but varchar2 is used for character strings of variable length, whereas char is used for character strings of fixed length. For example, if we specify the type as `char(5)` then we will not be allowed to store a string of any other length in this variable, but if we specify the type of this variable as `varchar2(5)` then we will be allowed to store strings of variable length. We can store a string of length 3 or 4 or 2 in this variable.

What is the use of CREATE, INSERT INTO, UPDATE and DELETE Clauses?

1. CREATE Clause

There are two CREATE statements available in SQL:

0. CREATE DATABASE
1. CREATE TABLE

CREATE DATABASE

A **Database** is defined as a structured set of data. So, in SQL the very first step to store the data in a well structured manner is to create a database. The **CREATE DATABASE** statement is used to create a new database in SQL.

Syntax:

```
CREATE DATABASE database_name;  
database_name: name of the database.
```

Example Query: This query will create a new database in SQL and name the database as *university*.

```
CREATE DATABASE university;
```

CREATE TABLE

We have learned above about creating databases. Now to store the data we need a table to do that. The **CREATE TABLE** statement is used to create a table in SQL. We know that a table comprises rows and columns. So while creating tables we have to provide all the information to SQL about the names of the columns, type of data to be stored in columns, size of the data, etc. Let us now dive into details on how to use the **CREATE TABLE** statement to create tables in SQL.

Syntax:

```
CREATE TABLE table_name  
(  
    column1 data_type(size),  
    column2 data_type(size),  
    column3 data_type(size),  
    ....  
);  
  
table_name: name of the table.  
column1 name of the first column.  
data_type: Type of data we want to store in the particular column.  
          For example, int for integer data.  
size: Size of the data we can store in a particular column. For example, if for a column we specify the data_type as int and size as 10 then this column can store an integer a number of maximum 10 digits.
```

Example Query: This query will create a table named Students with four columns, ROLL_NO, NAME, ADDRESS, and AGE.

```
CREATE TABLE Student
(
    ROLL_NO int,
    AGE int,
    NAME varchar(20),
    ADDRESS varchar(20)
);
```

This query will create a table named Student. The ROLL_NO and AGE field is of type int. The next two columns NAME and ADDRESS are of type varchar and can store characters and the size 20 specifies that these two fields can hold a maximum of 20 characters.

2. INSERT INTO Clause

The INSERT INTO statement of SQL is used to insert a new row in a table. There are two ways of using INSERT INTO statement for inserting rows:

0. **Only values:** First method is to specify only the value of data to be inserted without the column names.

Syntax

```
INSERT INTO table_name VALUES (value1, value2, value3,...);
table_name: name of the table.
value1, value2,... : value of first column, second
column,... for the new record
```

1. **Values with Column Name:** In the second method we will specify both the columns which we want to fill and their corresponding values as shown below:

Syntax:

```
INSERT INTO table_name (column1, column2, column3,...)
VALUES ( value1, value2, value3,...);
table_name: name of the table.
column1: name of first column, second column ...
value1, value2, value3 : value of first column, second
column,... for the new record
```

Empty Student table After Creation

```
roll_no | name | address | age
-----+-----+-----+-----
(0 rows)
```

Example - Method 1 (Inserting only values) :

```
INSERT INTO Students VALUES ('1','ALEX','NOIDA','19');
```

Student Table will look like this.

```
roll_no | name | address | age
-----+-----+-----+-----
    1 | ALEX | NOIDA   | 19
(1 row)
```

Example - Method 2 (Inserting Values with Column Name) :

```
INSERT INTO Students (ROLL_NO, NAME, Address,Age) VALUES
('2','ALLEN','DELHI','19');
```

Student Table will look like this.

```
roll_no | name | address | age
-----+-----+-----+-----
    1 | ALEX | NOIDA   | 19
    2 | ALLEN | DELHI   | 19
(2 rows)
```

3. UPDATE Clause

The UPDATE statement in SQL is used to update the data of an existing table in database. We can update single columns as well as multiple columns using UPDATE statement as per our requirement.

Basic Syntax

```
UPDATE table_name SET column1 = value1, column2 = value2, ...
WHERE condition;

table_name: name of the table
column1: name of first , second, third column....
value1: new value for first, second, third column....
condition: condition to select the rows for which the
values of columns needs to be updated.
```

NOTE: In the above query the **SET** statement is used to set new values to the particular column and the **WHERE** clause is used to select the rows for which the columns are needed to be updated. If we have not used the

WHERE clause then the columns in **all** the rows will be updated. So the WHERE clause is used to choose the particular rows.

Example Query for Updating Multiple Columns

```
UPDATE Students SET NAME = 'BROOK', ADDRESS = 'GURUGRAM' WHERE  
ROLL_NO = 1;
```

The Student Table will look like.

roll_no	name	address	age
1	BROOK	GURUGRAM	19
2	ALLEN	DELHI	19
(2 rows)			

4. DELETE Clause

The DELETE Statement in SQL is used to delete existing records from a table. We can delete a single record or multiple records depending on the condition we specify in the WHERE clause.

Basic Syntax

```
DELETE FROM table_name WHERE some_condition;  
table_name: name of the table  
some_condition: condition to choose particular record.
```

Note: We can delete single as well as multiple records depending on the condition we provide in WHERE clause. If we omit the WHERE clause then all of the records will be deleted and the table will be empty.

Example Query for Deleting Record

```
DELETE FROM Students WHERE NAME = 'ALLEN';
```

The Student table will look like this after deleting ALLEN's record.

roll_no	name	address	age
1	BROOK	GURUGRAM	19
(1 row)			

What is the use of ADD, DROP and MODIFY Commands?

ALTER TABLE is used to add, delete/drop or modify columns in the existing table. It is also used to add and drop various constraints on the existing table.

ALTER TABLE - ADD

ADD is used to add columns into the existing table. Sometimes we may require to add additional information, in that case we do not require to create the whole database again, **ADD** comes to our rescue.

Syntax:

```
ALTER TABLE table_name  
    ADD (Columnname_1 datatype,  
         Columnname_2 datatype,  
         ...  
         Columnname_n datatype);
```

ALTER TABLE - DROP

DROP COLUMN is used to drop column in a table. Deleting the unwanted columns from the table.

Syntax:

```
ALTER TABLE table_name  
    DROP COLUMN column_name;
```

ALTER TABLE-MODIFY

It is used to modify the existing columns in a table. Multiple columns can also be modified at once.

*Syntax may vary slightly in different databases. **Syntax(Oracle,MySQL,MariaDB):**

```
ALTER TABLE table_name  
    MODIFY column_name column_type;
```

Syntax(SQL Server):

```
ALTER TABLE table_name  
  
ALTER COLUMN column_name column_type;
```

Queries

Sample Table:

Student

ROLL_NONAME

1	Ram
2	Abhi
3	Rahul
4	Tanu

QUERY:

- To ADD 2 columns AGE and COURSE to table Student.

```
ALTER TABLE Student ADD (AGE number(3),COURSE varchar(40));
```

OUTPUT:

ROLL_NODENAMEAGECOURSE

1	Ram
2	Abhi
3	Rahul
4	Tanu

- MODIFY column COURSE in table Student

```
ALTER TABLE Student MODIFY COURSE varchar(20);
```

After running the above query maximum size of Course Column is reduced to 20 from 40.

- DROP column COURSE in table Student.

```
ALTER TABLE Student DROP COLUMN COURSE;
```

OUTPUT:

ROLL_NONAMEAGE

1	Ram
2	Abhi
3	Rahul
4	Tanu

What are VIEWS in SQL?

Views in SQL are kind of virtual tables. A view also has rows and columns as they are in a real table in the database. We can create a view by selecting fields from one or more tables present in the database. A View can either have all the rows of a table or specific rows based on certain condition.

Sample Tables:

StudentDetails

S_ID	NAME	ADDRESS
1	Harsh	Kolkata
2	Ashish	Durgapur
3	Pratik	Delhi
4	Dhanraj	Bihar
5	Ram	Rajasthan

StudentMarks

ID	NAME	MARKS	AGE
1	Marsh	90	19
2	Ashish	50	20
3	Pratik	80	19
4	Dhanraj	95	21
5	Ram	85	18

Creating Views

We can create View using CREATE VIEW statement. A View can be created from a single table or multiple tables.

SYNTAX:

```
CREATE VIEW view_name AS
SELECT column1, column2.....
FROM table_name
WHERE condition;
```

view_name: Name for the View

table_name: Name of the table

condition: Condition to select rows

Example

Creating View from a single table:

In this example we will create a View named DetailsView from the table StudentDetails.

Query:

```
CREATE VIEW DetailsView AS  
SELECT NAME, ADDRESS  
FROM StudentDetails  
WHERE S_ID < 5;
```

To see the data in the View, we can query the view in the same manner as we query a table.

```
SELECT * FROM DetailsView;
```

Output:

OUTPUT

NAME	ADDRESS
Harsh	Kolkata
Ashish	Durgapur
Pratik	Delhi
Dhanraj	Bihar

Creating View from multiple tables:

In this example we will create a View named MarksView from two tables StudentDetails and StudentMarks. To create a View from multiple tables we can simply include multiple tables in the SELECT statement.

Query:

```
CREATE VIEW MarksView AS  
SELECT StudentDetails.NAME, StudentDetails.ADDRESS, StudentMarks.MARKS  
FROM StudentDetails, StudentMarks  
WHERE StudentDetails.NAME = StudentMarks.NAME;
```

To display data of View MarksView:

```
SELECT * FROM MarksView;
```

Output:

NAME	ADDRESS	MARKS	AGE
Harsh	Kolkata	90	19
Ashish	Durgapur	50	20
Pratik	Delhi	80	19
Dhanraj	Bihar	95	21
Ram	Rajasthan	85	18

What are JOINS in SQL?

A SQL Join statement is used to combine data or rows from two or more tables based on a common field between them. Different types of Joins are:

- INNER JOIN
- LEFT JOIN
- RIGHT JOIN
- FULL JOIN

Consider the two tables below:

Student

ROLL_NO	NAME	ADDRESS	PHONE	Age
1	HARSH	DELHI	XXXXXXXXXX	18
2	PRATIK	BIHAR	XXXXXXXXXX	19
3	RIYANKA	SILIGURI	XXXXXXXXXX	20
4	DEEP	RAMNAGAR	XXXXXXXXXX	18
5	SAPTARSHI	KOLKATA	XXXXXXXXXX	19
6	DHANRAJ	BARABAJAR	XXXXXXXXXX	20
7	ROHIT	BALURGHAT	XXXXXXXXXX	18
8	NIRAJ	ALIPUR	XXXXXXXXXX	19

StudentCourse

COURSE_ID	ROLL_NO
1	1
2	2
2	3
3	4
1	5
4	9
5	10
4	11

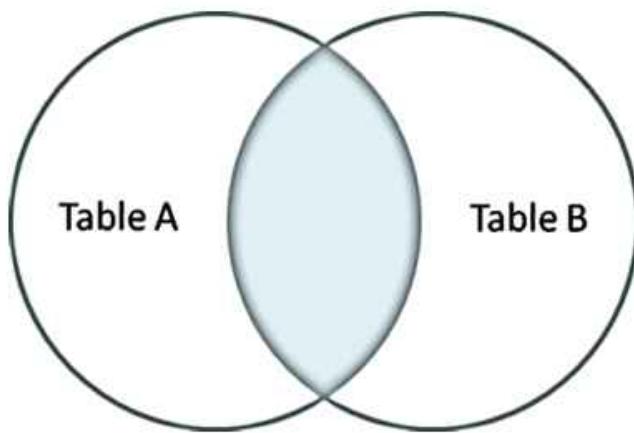
The simplest Join is INNER JOIN.

1. **INNER JOIN:** The INNER JOIN keyword selects all rows from both the tables as long as the condition satisfies. This keyword will create the result-set by combining all rows from both the tables where the condition satisfies i.e., value of the common field will be the same.

Syntax:

2. `SELECT table1.column1,table1.column2,table2.column1,....`
3. `FROM table1`
4. `INNER JOIN table2`
5. `ON table1.matching_column = table2.matching_column;`
6. `table1:` First table.
7. `table2:` Second table
8. `matching_column:` Column common to both the tables.

Note: We can also write JOIN instead of INNER JOIN. JOIN is the same as INNER JOIN.



Example Queries(INNER JOIN)

- o This query will show the names and age of students enrolled in different courses.
- o

```
SELECT StudentCourse.COURSE_ID, Student.NAME, Student.AGE
FROM Student
```
- o

```
INNER JOIN StudentCourse
```
- o

```
ON Student.ROLL_NO = StudentCourse.ROLL_NO;
```

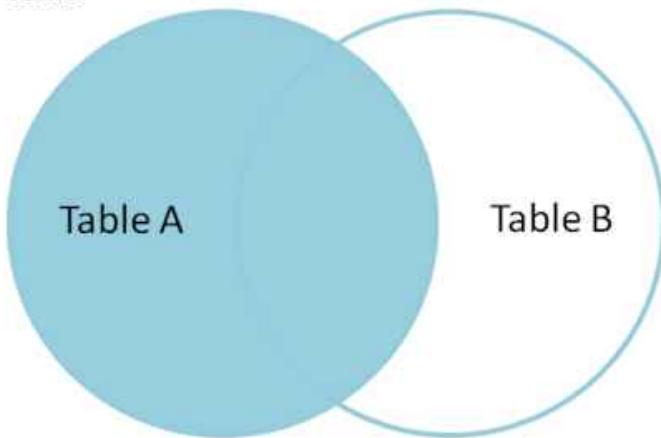
Output:

COURSE_ID	NAME	Age
1	HARSH	18
2	PRATIK	19
2	RIYANKA	20
3	DEEP	18
1	SAPTARHI	19

9. **LEFT JOIN**: This join returns all the rows of the table on the left side of the join and matching rows for the table on the right side of the join. The rows for which there is no matching row on the right side, the result-set will contain *null*. LEFT JOIN is also known as LEFT OUTER JOIN. **Syntax:**

```
10. SELECT table1.column1,table1.column2,table2.column1,....  
11. FROM table1  
12. LEFT JOIN table2  
13. ON table1.matching_column = table2.matching_column;  
14. table1: First table.  
15. table2: Second table  
16. matching_column: Column common to both the tables.
```

Note: We can also use LEFT OUTER JOIN instead of LEFT JOIN, both are same.



Example Queries(LEFT JOIN):

```
SELECT Student.NAME,StudentCourse.COURSE_ID  
FROM Student  
LEFT JOIN StudentCourse  
ON StudentCourse.ROLL_NO = Student.ROLL_NO;
```

Output:

NAME	COURSE_ID
HARSH	1
PRATIK	2
RIYANKA	2
DEEP	3
SAPTARHI	1
DHANRAJ	NULL
ROHIT	NULL
NIRAJ	NULL

17. **RIGHT JOIN:** RIGHT JOIN is similar to LEFT JOIN. This join returns all the rows of the table on the right side of the join and matching rows for the table on the left side of join. The rows for which there is no matching row on the left side, the result-set will contain *null*. RIGHT JOIN is also known as RIGHT OUTER JOIN.

Syntax:

```
18. SELECT table1.column1,table1.column2,table2.column1,....  
19. FROM table1  
20. RIGHT JOIN table2  
21. ON table1.matching_column = table2.matching_column;  
22. table1: First table.  
23. table2: Second table  
24. matching_column: Column common to both the tables.
```

Note: We can also use RIGHT OUTER JOIN instead of RIGHT JOIN, both are the same.

Example Queries(RIGHT JOIN):

```

SELECT Student.NAME,StudentCourse.COURSE_ID
FROM Student
RIGHT JOIN StudentCourse
ON StudentCourse.ROLL_NO = Student.ROLL_NO;

```

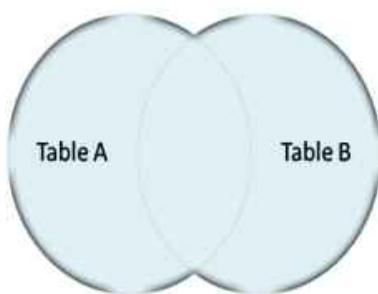
Output:

NAME	COURSE_ID
HARSH	1
PRATIK	2
RIYANKA	2
DEEP	3
SAPTARHI	1
NULL	4
NULL	5
NULL	4

25. **FULL JOIN:** FULL JOIN creates the result-set by combining results of both LEFT JOIN and RIGHT JOIN. The result-set will contain all the rows from both the tables. The rows for which there is no matching, the result-set will contain *NULL* values.

Syntax:

26. SELECT table1.column1,table1.column2,table2.column1,....
27. FROM table1
28. FULL JOIN table2
29. ON table1.matching_column = table2.matching_column;
30. table1: First table.
31. table2: Second table
32. matching_column: Column common to both the tables.



Example Queries(FULL JOIN):

```
SELECT Student.NAME,StudentCourse.COURSE_ID  
  
FROM Student  
  
FULL JOIN StudentCourse  
  
ON StudentCourse.ROLL_NO = Student.ROLL_NO;
```

NAME	COURSE_ID
HARSH	1
PRATIK	2
RIYANKA	2
DEEP	3
SAPTARHI	1
DHANRAJ	NULL
ROHIT	NULL
NIRAJ	NULL
NULL	9
NULL	10
NULL	11

Output:

What is the use of GROUP BY Clause?

The GROUP BY Statement in SQL is used to arrange identical data into groups with the help of some functions. i.e if a particular column has same values in different rows then it will arrange these rows in a group.

Important Points:

- GROUP BY clause is used with the SELECT statement.
- In the query, GROUP BY clause is placed after the WHERE clause.
- In the query, GROUP BY clause is placed before ORDER BY clause if used any.

Syntax:

```
SELECT column1, function_name(column2)
FROM table_name
WHERE condition
GROUP BY column1, column2
ORDER BY column1, column2;

function_name: Name of the function used for example, SUM() , AVG().
table_name: Name of the table.
condition: Condition used.
```

Example query to demonstrate GROUP BY Clause.

```
SELECT address, AVG(age) FROM Students GROUP BY address;
```

Output of the Above query will look like this.

address	avg
DELHI	19.000000000000000
NOIDA	20.000000000000000
GURUGRAM	19.000000000000000

(3 rows)

This returns the Average Age for each address present in Students Table.

What are Aggregate Functions?

In database management an **aggregate function** is a function where the values of multiple rows are grouped together as input on certain criteria to form a single value of more significant meaning.

We will be using below Students table to explain the **Example Queries**.

Students Table

roll_no	name	address	age
1	BROOK	GURUGRAM	19
2	ALEX	NOIDA	19
3	ALLEN	NOIDA	21
4	ROBIN	DELHI	20
5	CALVIN	DELHI	18

Various Aggregate Functions

- 1) Count ()
- 2) Sum()
- 3) Avg()
- 4) Min()
- 5) Max()

1. **Count()** Example queries to demonstrate above function.

```
SELECT COUNT(*) FROM Students;
```

Output : 5

Returns total number of records .i.e 5.

SELECT COUNT(age) FROM Students;

Output : 5

Return number of Non Null values over the column age. i.e 5.

SELECT COUNT(DISTINCT age) FROM Students;

Output : 4

Return number of distinct Non Null values over the column age .i.e 4

2. **Sum() Example** queries to demonstrate above function.

SELECT SUM(age) FROM Students;

Output : 97

Sum all Non Null values of Column salary i.e.,97

SELECT SUM(DISTINCT age) FROM Students;

Output : 78

Sum of all distinct Non-Null values i.e., 78.

3. **Avg() Example** queries to demonstrate above function.

SELECT AVG(age) FROM Students;

Output : 19.40

$\text{Avg}(\text{age}) = \text{sum}(\text{age}) / \text{Count}(\text{age}) = 97/5 = 19.40$

SELECT AVG(DISTINCT age) FROM Students;

Output : 19.50

$\text{Avg}(\text{Distinct age}) = \text{sum}(\text{Distinct age}) / \text{Count}(\text{Distinct age}) = 78/4 = 19.50$

4. **Min() Example** query to demonstrate above function.

SELECT MIN(age) FROM Students;

Output : 18

Return minimum non null value over the column age. i.e 18.

5. **Max() Example** query to demonstrate above function.

```
SELECT MAX(age) FROM Students;
```

Output : 21

Return maximum non null value over the column age. i.e 21.

What is Cursor in SQL?

Cursor is a Temporary Memory or Temporary Work Station. It is Allocated by Database Server at the Time of Performing DML(Data Manipulation Language) operations on Table by User. Cursors are used to store Database Tables. There are 2 types of Cursors: Implicit Cursors, and Explicit Cursors. These are explained as following below.

1. **Implicit Cursors:** Implicit Cursors are also known as Default Cursors of SQL SERVER. These Cursors are allocated by SQL SERVER when the user performs DML operations.
2. **Explicit Cursors :** Explicit Cursors are Created by Users whenever the user requires them. Explicit Cursors are used for Fetching data from Table in Row-By-Row Manner.

How to create Explicit Cursor:

1. **Declare Cursor Object. Syntax :** `DECLARE cursor_name CURSOR FOR SELECT * FROM table_name`

```
DECLARE s1 CURSOR FOR SELECT * FROM studDetails
```

2. **Open Cursor Connection. Syntax :** `OPEN cursor_connection`

```
OPEN s1
```

3. **Fetch Data from cursor.** There are total 6 methods to access data from cursor. They are as follows :

FIRST is used to fetch only the first row from cursor table.

LAST is used to fetch only last row from cursor table.

NEXT is used to fetch data in forward direction from cursor table.

PRIOR is used to fetch data in backward direction from cursor table.

ABSOLUTE n is used to fetch the exact n^{th} row from cursor table.

RELATIVE n is used to fetch the data in incremental way as well as decremental way.

Syntax : `FETCH NEXT/FIRST/LAST/PRIOR/ABSOLUTE n/RELATIVE n FROM cursor_name`

```
FETCH FIRST FROM s1
FETCH LAST FROM s1
FETCH NEXT FROM s1

FETCH PRIOR FROM s1
FETCH ABSOLUTE 7 FROM s1

FETCH RELATIVE -2 FROM s1
```

4. **Close cursor connection.** **Syntax :** `CLOSE cursor_name`

```
CLOSE s1
```

5. **Deallocate cursor memory.** **Syntax :** `DEALLOCATE cursor_name`

```
DEALLOCATE s1
```

What is the difference between Implicit and Explicit Cursor?

Databases such as ORACLE have a memory area, where processing of instructions and fetched data takes place. A cursor is a pointer which is pointing to this area. The data contained in this memory area is also known as **Active Set**. Cursors can be broadly classified into **Implicit Cursors** and **Explicit Cursors**.

Difference between Implicit and Explicit Cursors :

Implicit cursor in Oracle	Explicit Cursor in Oracle
<ul style="list-style-type: none"> Implicit Cursor in Oracle are declared by PL/SQL implicitly for all DML Statements and for single row queries Example: Select Statement issued directly within the BEGIN..END part of a block opens up an implicit cursor. 	<ul style="list-style-type: none"> Declared and named by the programmer Explicit Cursors allow multiple rows to be processed from the query. Use explicit cursor to individually process all the rows returned by multiple row select statement
Example	
<pre>DECLARE v_salary number; BEGIN SELECT emp_salary INTO v_salary FROM emp where empno=1111; DBMS_OUTPUT.PUT_LINE(v_salary); END; /</pre>	
Four attributes	
SQL%NOTFOUND SQL%FOUND SQL%ISOPEN SQL%ROWCOUNT	

What is the difference between VIEW and CURSOR in SQL?

1. View:

A view is a virtual table that not actually exist in the database but it can be produced upon request by a particular user. A view is an object that gives the user a logical view of data from a base table we can restrict to what user can view by allowing them to see an only necessary column from the table and hide the other database details. View also permits users to access data according to their requirements, so the same data can be access by a different user in a different way according to their needs.

2Cursor : A cursor is a temporary work area created in memory for processing and storing the information related to an SQL statement when it is executed. The temporary work area is used to store the data retrieved from the database and manipulate data according to need. It contains all the necessary information on data access by the select statement. It can hold a set of rows called active set but can access only a single row at a time. There are two different types of cursors -

1. Implicit Cursor
2. Explicit Cursor

Difference between View and Cursor in SQL :

S.No.	View	Cursor
1.	A view is a virtual table that gives logical view of data from base table.	A cursor is a temporary workstation created in database server when SQL statement is executed.
2.	Views are dynamic in nature which means any changes made in base table are immediately reflected in view.	Cursor can be static as well as dynamic in nature.

There are some steps for creating a Explicit cursor -

3.	We can perform CRUD operations on view like create, insert, delete and update.	<ul style="list-style-type: none"> • Declare the cursor in declaration section. • Open the cursor in execution section. • Fetch the cursor to retrieve data into PL/SQL variable. • Close the cursor to release allocated memory.
4.	There are two types of view i.e. Simple View (created from single table) and Complex Cursor (created from multiple tables).	Cursor has two types i.e. Implicit Cursor (pre-defined) and Explicit Cursor(user defined).
5.	View is a database object similar to table so it can be used with both SQL and PL/SQL.	Cursor is defined and used within the block of stored procedure which means it can be only used with PL/SQL.
6.	General Syntax of Creating View : <pre>CREATE VIEW "VIEW_NAME" AS "SQL Statement";</pre>	General Syntax of Creating View : <pre>CURSOR cursor_name IS select_statement;</pre>

What are the advantages of PL/SQL functions?

Advantages of PL / SQL functions as follows:

- We can make a single call to the database to run a block of statements. Thus, it improves the performance against running SQL multiple times. This will reduce the number of calls between the database and the application.

- We can divide the overall work into small modules which becomes quite manageable, also enhancing the readability of the code.
- It promotes reusability.
- It is secure since the code stays inside the database, thus hiding internal database details from the application(user). The user only makes a call to the PL/SQL functions. Hence, security and data hiding is ensured.

Explain BETWEEN and IN Clause.

We will be using below Students table to explain the **Example Queries**.

Students Table

roll_no	name	address	age
1	BROOK	GURUGRAM	19
2	ALEX	NOIDA	19
3	ALLEN	NOIDA	21
4	ROBIN	DELHI	20
5	CALVIN	DELHI	18

• BETWEEN Clause

The SQL BETWEEN condition allows you to easily test if an expression is within a range of values (inclusive). The values can be text, date, or numbers. It can be used in a SELECT, INSERT, UPDATE, or DELETE statement. The SQL BETWEEN Condition will return the records where expression is within the range of value1 and value2.

Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE column_name BETWEEN value1 AND value2;
```

Example : Query to get Student Records, Where Student's age is between 19 and 21.

```
SELECT * FROM Students WHERE age BETWEEN 19 AND 21;
```

Output of the query will look like this.

roll_no	name	address	age
1	BROOK	GURUGRAM	19
2	ALEX	NOIDA	19
3	ALLEN	NOIDA	21
4	ROBIN	DELHI	20

(4 rows)

Example : Query to get Student Records, Where Student's age is **not** between 19 and 21.

```
SELECT * FROM Students WHERE age NOT BETWEEN 19 AND 21;
```

Output of the query will look like this.

roll_no	name	address	age
5	CALVIN	DELHI	18

(1 row)

- **IN Clause**

IN operator allows you to easily test if the expression matches any value in the list of values. It is used to remove the need of multiple OR condition in SELECT, INSERT, UPDATE or DELETE. You can also use NOT IN to exclude the rows in your list.

Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN (list_of_values);
```

Example : Query to get Student Records, Where Student's age is in the set {18,20,21}.

```
SELECT * FROM Students WHERE age IN (18,20,21);
```

Output of the query will look like this.

roll_no	name	address	age
3	ALLEN	NOIDA	21
4	ROBIN	DELHI	20
5	CALVIN	DELHI	18

(3 rows)

Example : Query to get Student Records, Where Student's age is **not** in the

set {18,20,21}.

```
SELECT * FROM Students WHERE age NOT IN (18,20,21);
```

Output of the query will look like this.

roll_no	name	address	age
1	BROOK	GURUGRAM	19
2	ALEX	NOIDA	19

(2 rows)

What is the difference between DROP and TRUNCATE?

S.NO.	DROP	TRUNCATE
1.	The DROP command is used to remove table definition and its contents.	Whereas the TRUNCATE command is used to delete all the rows from the table.
2.	In the DROP command, table space is freed from memory.	While the TRUNCATE command does not free the table space from memory.
3.	DROP is a DDL(Data Definition Language) command.	Whereas the TRUNCATE is also a DDL(Data Definition Language) command.
4.	In the DROP command, a view of the table does not exist.	While in this command, a view of the table exists.
5.	In the DROP command, integrity constraints will be removed.	While in this command, integrity constraints will not be removed.
6.	In the DROP command, undo space is not used.	While in this command, undo space is used but less than DELETE.
7.	The DROP command is quick to perform but gives rise to complications.	While this command is faster than DROP.

What are Constraints in SQL?

Constraints are the rules that we can apply to the type of data in a table. That is, we can specify the limit on the type of data that can be stored in a particular column in a table using constraints.

The available constraints in SQL are:

NOT NULL: This constraint tells that we cannot store a null value in a column. That

is, if a column is specified as NOT NULL then we will not be able to store null in this particular column anymore.

UNIQUE: This constraint when specified with a column, tells that all the values in the column must be unique. That is, the values in any row of a column must not be repeated.

PRIMARY KEY: A primary key is a field that can uniquely identify each row in a table. And this constraint is used to specify a field in a table as the primary key.

FOREIGN KEY: A Foreign key is a field that can uniquely identify each row in another table. And this constraint is used to specify a field as a Foreign key.

CHECK: This constraint helps to validate the values of a column to meet a particular condition. That is, it helps to ensure that the value stored in a column meets a specific condition.

DEFAULT: This constraint specifies a default value for the column when no value is specified by the user.

How to specify constraints? We can specify constraints at the time of creating the table using CREATE TABLE statement. We can also specify the constraints after creating a table using ALTER TABLE statement.

Syntax: Below is the syntax to create constraints using CREATE TABLE statement at the time of creating the table.

```
CREATE TABLE sample_table
(
column1 data_type(size) constraint_name,
column2 data_type(size) constraint_name,
column3 data_type(size) constraint_name,
....;
);
sample_table: Name of the table to be created.
data_type: Type of data that can be stored in the field.
constraint_name: Name of the constraint. for example- NOT NULL,
UNIQUE, PRIMARY KEY etc.
```

Let us see each of the constraints in detail.

1. NOT NULL –

If we specify a field in a table to be NOT NULL. Then the field will never accept the null value. That is, you will be not allowed to insert a new row in the table without specifying any value to this field.

For example, the below query creates a table Student with the fields ID and NAME as NOT NULL. That is, we are bound to specify values for these two fields every time we wish to insert a new row.

```
CREATE TABLE Student
(
ID int(6) NOT NULL,
NAME varchar(10) NOT NULL,
ADDRESS varchar(20)
);
```

2. UNIQUE – This constraint helps to uniquely identify each row in the table. i.e. for a particular column, all the rows should have unique values. We can have more than one UNIQUE column in a table.

For example, the below query creates a table Student where the field ID is specified as UNIQUE. i.e, no two students can have the same ID. Unique constraint in detail.

```
CREATE TABLE Student
(
ID int(6) NOT NULL UNIQUE,
NAME varchar(10),
ADDRESS varchar(20)
);
```

3. PRIMARY KEY –

Primary Key is a field that uniquely identifies each row in the table. If a field in a table is the primary key, then the field will not be able to contain NULL values as well as all the rows should have unique values for this field. So, in other words, we can say that this is a combination of NOT NULL and UNIQUE constraints.

A table can have only one field as a primary key. The below query will create a table named Student and specifies the field ID as a primary key.

```
CREATE TABLE Student
(
ID int(6) NOT NULL UNIQUE,
NAME varchar(10),
ADDRESS varchar(20),
PRIMARY KEY(ID)
);
```

4. FOREIGN KEY –

Foreign Key is a field in a table that uniquely identifies each row of another table. That is this field points to the primary key of another table. This usually creates a kind of link between the tables.

Consider the two tables as shown below:

Orders

O_ID	ORDER_NO	C_ID
1	2253	3
2	3325	3
3	4521	2
4	8523	1

Customers

C_ID	NAME	ADDRESS
1	RAMESH	DELHI
2	SURESH	NOIDA
3		DHARMESHGURGAON

As we can see clearly that the field C_ID in the Orders table is the primary key in the Customers table, i.e. it uniquely identifies each row in the Customers table. Therefore, it is a Foreign Key in the Orders table.

Syntax:

```
CREATE TABLE Orders
(
O_ID int NOT NULL,
ORDER_NO int NOT NULL,
C_ID int,
PRIMARY KEY (O_ID),
FOREIGN KEY (C_ID) REFERENCES Customers(C_ID)
```

```
)
```

(i) CHECK -

Using the CHECK constraint we can specify a condition for a field, which should be satisfied at the time of entering values for this field.

For example, the below query creates a table Student and specifies the condition for the field AGE as (AGE >= 18). That is, the user will not be allowed to enter any record in the table with AGE < 18. Check constraint in detail

```
CREATE TABLE Student
(
ID int(6) NOT NULL,
NAME varchar(10) NOT NULL,
AGE int NOT NULL CHECK (AGE >= 18)
);
```

(ii) DEFAULT -

This constraint is used to provide a default value for the fields. That is, if at the time of entering new records in the table if the user does not specify any value for these fields then the default value will be assigned to them.

For example, the below query will create a table named Student and specify the default value for the field AGE as 18.

```
CREATE TABLE Student
(
ID int(6) NOT NULL,
NAME varchar(10) NOT NULL,
AGE int DEFAULT 18
);
```

What is a TRIGGER?

A **trigger** is a stored procedure in database which automatically invokes whenever a special event in the database occurs. For example, a trigger can be invoked when a row is inserted into a specified table or when certain table columns are being updated.

Syntax:

```
create trigger [trigger_name]
[before | after]
{insert | update | delete}
on [table_name]
[for each row]
[trigger_body]
```

Explanation of syntax:

1. `create trigger [trigger_name]`: Creates or replaces an existing trigger with the `trigger_name`.
2. `[before | after]`: This specifies when the trigger will be executed.
3. `{insert | update | delete}`: This specifies the DML operation.
4. `on [table_name]`: This specifies the name of the table associated with the trigger.
5. `[for each row]`: This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected.
6. `[trigger_body]`: This provides the operation to be performed as trigger is fired

Example:

Given Student Report Database, in which student marks assessment is recorded. In such schema, create a trigger so that the total and average of specified marks is automatically inserted whenever a record is insert.

Here, as trigger will invoke before record is inserted so, BEFORE Tag can be used.

Suppose the database Schema –

```
mysql> desc Student;
+-----+-----+-----+-----+-----+-----+
| Field | Type   | Null | Key | Default | Extra       |
+-----+-----+-----+-----+-----+-----+
| tid  | int(4) | NO  | PRI | NULL    | auto_increment |
| name | varchar(30) | YES |     | NULL    |             |
| subj1 | int(2) | YES |     | NULL    |             |
| subj2 | int(2) | YES |     | NULL    |             |
| subj3 | int(2) | YES |     | NULL    |             |
| total | int(3) | YES |     | NULL    |             |
| per   | int(3) | YES |     | NULL    |             |
+-----+-----+-----+-----+-----+-----+
7 rows in set (0.00 sec)
```

SOLUTION

```
create trigger stud_marks
before INSERT
on
```

```
Student
for each row
set Student.total = Student.subj1 + Student.subj2 + Student.subj3,
Student.per = Student.total * 60 / 100;
```

Above SQL statement will create a trigger in the student database in which whenever subjects marks are entered, before inserting this data into the database, trigger will compute those two values and insert with the entered values. i.e.,

```
mysql> insert into Student values(0, "ABCDE", 20, 20, 20, 0, 0);
Query OK, 1 row affected (0.09 sec)

mysql> select * from Student;
+----+-----+-----+-----+-----+-----+
| tid | name | subj1 | subj2 | subj3 | total | per |
+----+-----+-----+-----+-----+-----+
| 100 | ABCDE |    20 |    20 |    20 |    60 |   36 |
+----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

What is the use of LIMIT and OFFSET in SQL?

We will be using below Students table to explain the **Example Queries**.

Students Table

roll	name	address	age
1	BROOK	GURUGRAM	19
2	ALEX	NOIDA	19
3	ALLEN	NOIDA	21
4	ROBIN	DELHI	20
5	CALVIN	DELHI	18

If there are a large number of tuples satisfying the query conditions, it might be resourceful to view only a handful of them at a time.

- The **LIMIT** clause is used to set an upper limit on the number of tuples returned by SQL.
- It is important to note that this clause is not supported by all SQL versions.
- The LIMIT clause can also be specified using the SQL 2008 OFFSET/FETCH FIRST clauses.
- The limit/offset expressions must be a non-negative integer.

Example Queries to demonstrate LIMIT Clause.

```
SELECT *
FROM Students
LIMIT 3;
```

```

roll_no | name   | address | age
-----+-----+-----+
  1 | BROOK | GURUGRAM | 19
  2 | ALEX   | NOIDA   | 19
  3 | ALLEN  | NOIDA   | 21
(3 rows)

```

Output of the query gives First three Student Records.

```

SELECT *
FROM Students
ORDER BY age
LIMIT 3;

```

```

roll_no | name   | address | age
-----+-----+-----+
  5 | CALVIN | DELHI   | 18
  1 | BROOK  | GURUGRAM | 19
  2 | ALEX    | NOIDA   | 19
(3 rows)

```

Output of the query gives three Student Records in order of Ascending Ages.

The LIMIT operator can be used in situations such as the above, where we need to find the top N students in a class and based on any condition statements.

Using LIMIT along with OFFSET

LIMIT x OFFSET y simply means skip the first y entries and then return the next x entries.

OFFSET can only be used with the ORDER BY clause. It cannot be used on its own. OFFSET value must be greater than or equal to zero. It cannot be negative, else returns error.

Example query to demonstrate LIMIT and OFFSET.

```

SELECT *
FROM Students
LIMIT 3 OFFSET 2
ORDER BY roll_no;

```

```

roll_no | name   | address | age
-----+-----+-----+
  3 | ALLEN  | NOIDA   | 21
  4 | ROBIN  | DELHI   | 20
  5 | CALVIN | DELHI   | 18
(3 rows)

```

Returns 3 Student Records skipping first two records in Table.

What are different types of operators present in SQL?

Operators are the foundation of any programming language. We can define operators as symbols that help us to perform specific mathematical and logical computations on operands. In other words, we can say that an operator operates the operands. SQL operators have three different categories.

- Arithmetic operator
- Comparison operator
- Logical operator

Arithmetic operators:

We can use various arithmetic operators on the data stored in the tables. Arithmetic Operators are:

S.No.	Operator	Description
1	+	The addition is used to perform an addition operation on the data values.
2	-	This operator is used for the subtraction of the data values.
3	/	This operator works with the 'ALL' keyword and it calculates division operations.
4	*	This operator is used for multiply data values.
5	%	Modulus is used to get the remainder when data is divided by another.

Comparison operators:

Another important operator in SQL is a comparison operator, which is used to compare one expression's value to other expressions. SQL supports different types of the comparison operator, which is described below:

S.No.	Operator	Description
1	=	Equal to.
2	>	Greater than.
3	<	Less than.
4	>=	Greater than equal to.
5	<=	Less than equal to.
6	<>	Not equal to.

Logical operators:

The Logical operators are those that are true or false. They return true or false values to combine one or more true or false values.

S.No	Operator	Description
1	AND	Logical AND compares between two Booleans as expressions and returns true when both expressions are true.
2	OR	Logical OR compares between two Booleans as expressions and returns true when one of the expressions is true.
3	NOT	Not takes a single Boolean as an argument and changes its value from false to true or from true to false.

Special operators:

S.No.	Operator	Description
1	ALL	ALL is used to select all records of a SELECT STATEMENT. It compares a value to every value in a list of results from a query. The ALL must be preceded by the comparison operators and evaluates to TRUE if the query returns no rows.
2	ANY	ANY compares a value to each value in a list of results from a query and evaluates to true if the result of an inner query contains at least one row.
3	BETWEEN	The SQL BETWEEN operator tests an expression against a range. The range consists of a beginning, followed by an AND keyword and an end expression.
4	IN	The IN operator checks a value within a set of values separated by commas and retrieves the rows from the table which are matching.
5	EXISTS	The EXISTS checks the existence of a result of a subquery. The EXISTS subquery tests whether a subquery fetches at least one row. When no data is returned then this operator returns 'FALSE'.
6	SOME	SOME operator evaluates the condition between the outer and inner tables and evaluates to true if the final result returns any one row. If not, then it evaluates to false.



HIMANSHU KUMAR(LINKEDIN)
<https://www.linkedin.com/in/himanshukumarmahuri>

CREDITS- INTERNET.

DISCLOSURE- ALL THE DATA AND IMAGES ARE TAKEN FROM GOOGLE AND INTERNET.

What is PostgreSQL?

PostgreSQL (pronounced as **post-gress-Q-L**) is an open source relational database management system (DBMS) developed by a worldwide team of volunteers. PostgreSQL is not controlled by any corporation or other private entity and the source code is available free of charge.

A Brief History of PostgreSQL

PostgreSQL, originally called Postgres, was created at UCB by a computer science professor named Michael Stonebraker. Stonebraker started Postgres in 1986 as a follow-up project to its predecessor, Ingres, now owned by Computer Associates.

1. 1977-1985: A project called INGRES was developed.

- Proof-of-concept for relational databases
- Established the company Ingres in 1980
- Bought by Computer Associates in 1994

2. 1986-1994: POSTGRES

- Development of the concepts in INGRES with a focus on object orientation and the query language - Quel
- The code base of INGRES was not used as a basis for POSTGRES
- Commercialized as Illustra (bought by Informix, bought by IBM)

3. 1994-1995: Postgres95

- Support for SQL was added in 1994

Key Features of PostgreSQL

PostgreSQL runs on all major operating systems, including Linux, UNIX (AIX, BSD, HP-UX, SGI IRIX, Mac OS X, Solaris, Tru64), and Windows. It supports text, images, sounds, and video, and includes programming interfaces for C / C++, Java, Perl, Python, Ruby, Tcl and Open Database Connectivity (ODBC).

VN2 Solutions Pvt. Ltd.

PostgreSQL supports a large part of the SQL standard and offers many modern features including the following:

- Complex SQL queries
- SQL Sub-selects
- Foreign keys
- Trigger
- Views
- Transactions
- Multiversion concurrency control (MVCC)
- Streaming Replication (as of 9.0)
- Hot Standby (as of 9.0)

You can check official documentation of PostgreSQL to understand the above-mentioned features. PostgreSQL can be extended by the user in many ways. For example, by adding new:

- Data types
- Functions
- Operators
- Aggregate functions
- Index methods

Installing PostgreSQL:

Ubuntu-Linux:

Open terminal and follow the below steps:

- sudo apt update
- apt install postgresql postgresql-contrib
- sudo systemctl start postgresql.service (for ensure server is running)

Accessing Postgres and Creating Data Base:

Open terminal and follow the below steps:

- psql postgres -- (to log into postgres)
- \l -- (to see the list of data-base or info of postgres)

Creating an **employee** data-base with **empuser** as user and **emppass** as password and with all permissions

- create database employee;
- create user empuser with encrypted password 'emppass';
- grant all privileges on database employee to empuser;

After above steps the changes will be similar to below image.

```
Last login: Mon Jul 25 21:03:04 on ttys011
[REDACTED] + postgres % psql postgres
psql (14.4)
Type "help" for help.

postgres=# \l
              List of databases
   Name    |  Owner   | Encoding | Collate | Ctype | Access privileges
---+-----+-----+-----+-----+-----+
postgres | [REDACTED] | UTF8    | C      | C     | 
template0 | [REDACTED] | UTF8    | C      | C     | 
template1 | [REDACTED] | UTF8    | C      | C     | 
(3 rows)

postgres=# create database employee;
CREATE DATABASE
postgres=# create user empuser with encrypted password 'emppass';
CREATE ROLE
postgres=# grant all privileges on database employee to empuser;
GRANT
postgres=# \l
              List of databases
   Name    |  Owner   | Encoding | Collate | Ctype | Access privileges
---+-----+-----+-----+-----+-----+
employee | [REDACTED] | UTF8    | C      | C     | 
postgres | [REDACTED] | UTF8    | C      | C     | 
template0 | [REDACTED] | UTF8    | C      | C     | 
template1 | [REDACTED] | UTF8    | C      | C     | 
(4 rows)

postgres=# \c employee
You are now connected to database "employee" as user [REDACTED].
employee=# [REDACTED]
```

Command for accessing data-base: \c employee (\c data-base name)

Command to drop data-base: DROP DATABASE [IF EXISTS] employee (DROP DATABASE [IF EXISTS] data-base name)

For reference: [Link](#)

Accessing PostgreSQL through DBeaver:

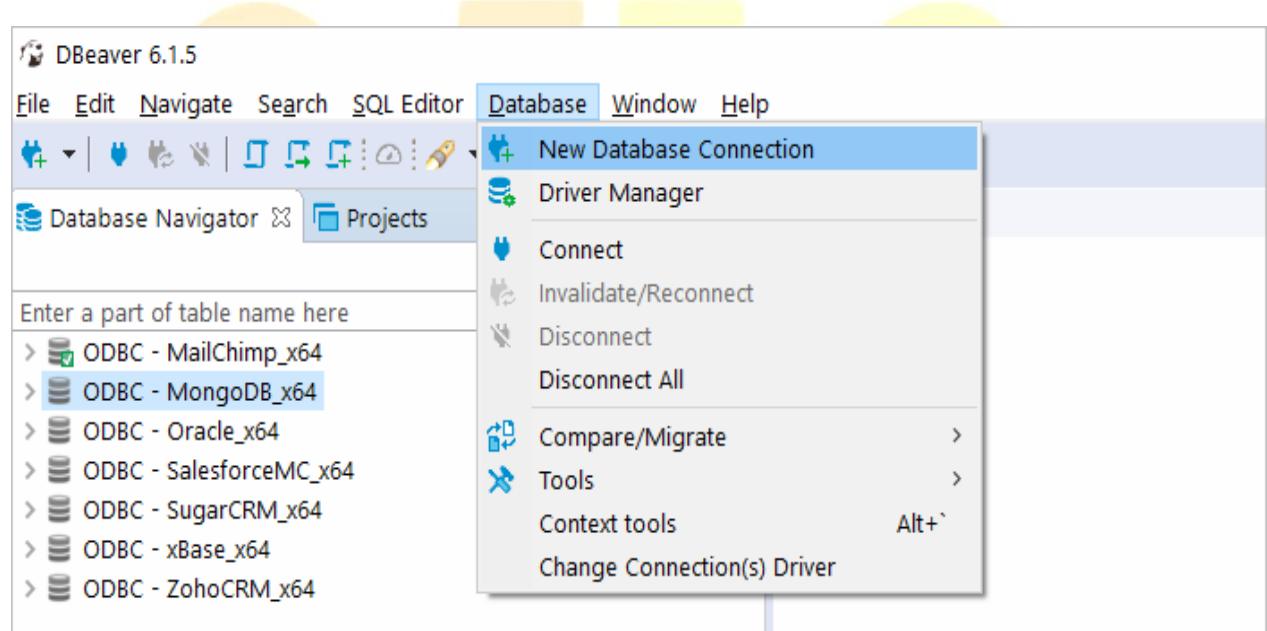
About DBeaver :

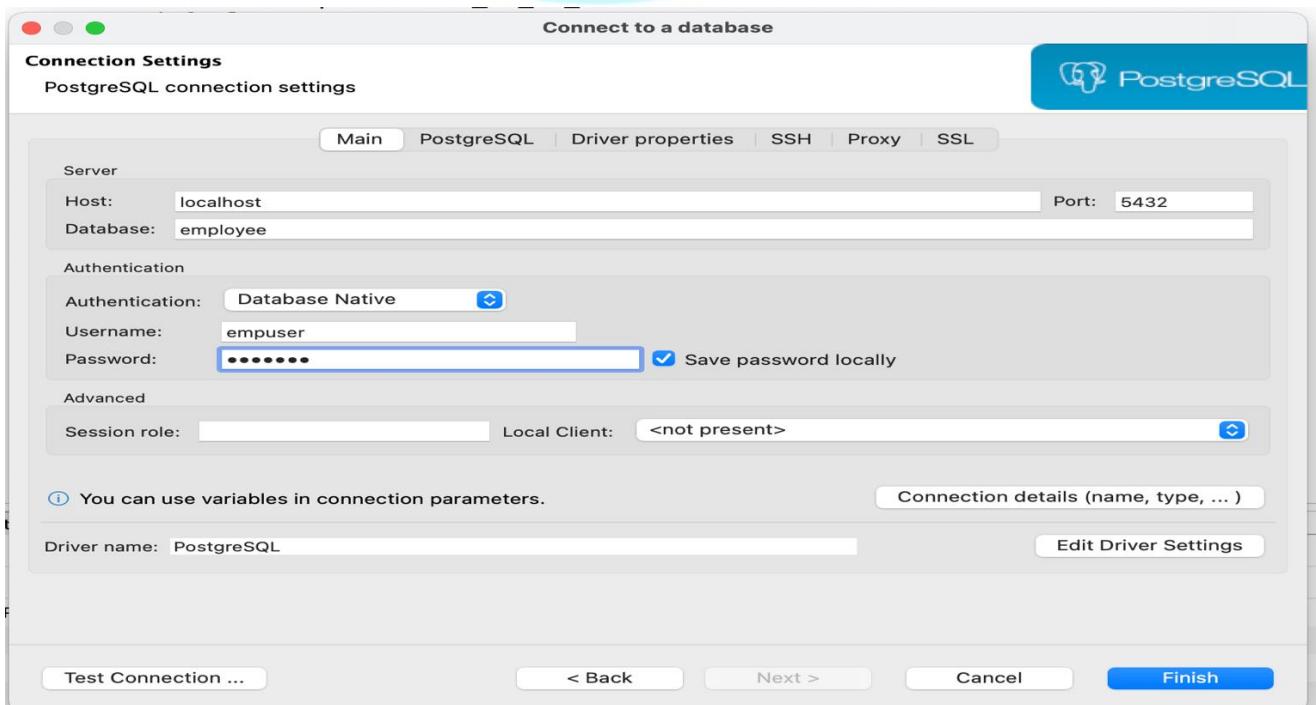
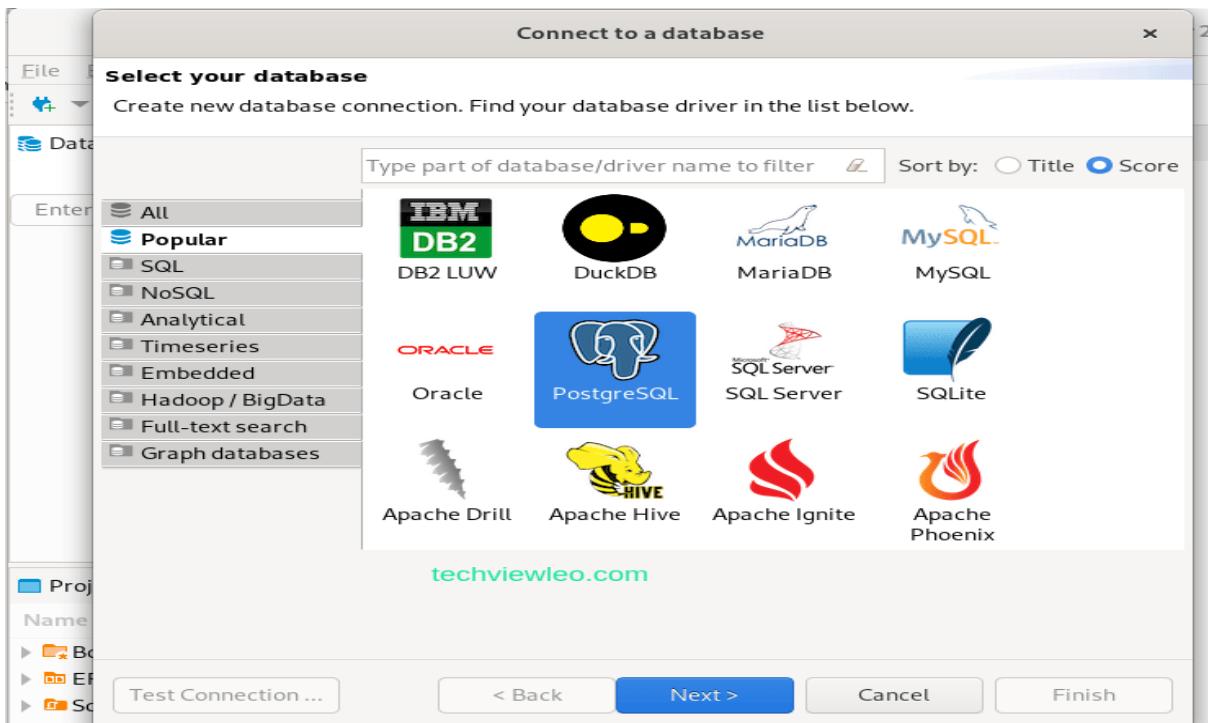
- Free multi-platform database tool for developers, database administrators, analysts and all people who need to work with databases. Supports all popular databases: MySQL, PostgreSQL, SQLite, Oracle, DB2, SQL Server, Sybase, MS Access, Teradata, Firebird, Apache Hive, Phoenix, Presto, etc.

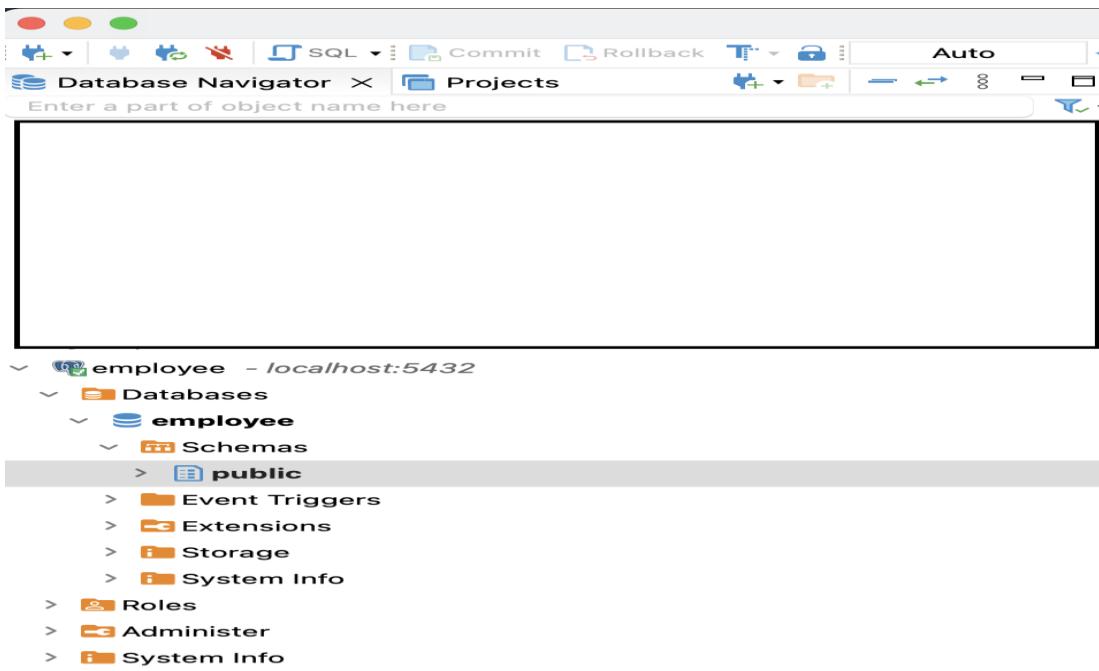
Installing DBeaver:

Open terminal and follow the below steps:

- sudo apt update
- sudo snap installdbeaver-ce (note : this will take some time to install)
- Open DBeaver once install and follow the below steps shown in images







Employee Data-Base creation is done.

Creating Tables:

The PostgreSQL CREATE TABLE statement is used to create a new table in any of the given database.

Syntax:

Basic syntax of CREATE TABLE statement is as follows –

```
CREATE TABLE table_name(
```

```
    column1 datatype,
```

```
    column2 datatype,
```

```
    column3 datatype,
```

```
....
```

```
    columnN datatype,
```

```
    PRIMARY KEY( one or more columns )
```

```
);
```

CREATE TABLE is a keyword, telling the database system to create a new table. The unique name or identifier for the table follows the CREATE TABLE statement. Initially, the empty table in the current database is owned by the user issuing the command.

Then, in brackets, comes the list, defining each column in the table and what sort of data type it is. The syntax will become clear with an example given below.

```
CREATE TABLE employee (
    id bigint NOT NULL,
    birth_date date NOT NULL,
    first_name character varying(14) NOT NULL,
    last_name character varying(16) NOT NULL,
    gender employee_gender NOT NULL,
    hire_date date NOT NULL
);
```

```
CREATE TABLE department_employee (
    employee_id bigint NOT NULL,
    department_id character(4) NOT NULL,
    from_date date NOT NULL,
    to_date date NOT NULL
);
```

Dropping tables:

- The PostgreSQL “DROP TABLE” statement is used to remove a table definition and all associated data, indexes, rules, triggers, and constraints for that table. once a table is deleted then all the information available in the table would also be lost forever.

Syntax:

- Basic syntax of DROP TABLE statement is as follows:

```
DROP TABLE table_name;
```

- We can drop the single or multiple tables at a time.
- Dropping single table:

```
DROP TABLE employee;
```

- Dropping multiple tables:

```
DROP TABLE employee, department;
```

After drop command which command, I must write.

Schema:

- A **Schema** is a named collection of tables. A schema can also contain views, indexes, sequences, data types, operators, and functions. Schemas are analogous to directories at the operating system level, except those schemas cannot be nested.

Syntax:

- The basic syntax of create schema is as follows:

```
CREATE SCHEMA name;
```

- Note: here **name** is the name of the schema.
- The basic syntax to create table in schema is as follows:

Syntax:

```
CREATE TABLE myschema.table_name(  
    column1 datatype,  
    column2 datatype,  
    column3 datatype,  
    ....  
    column datatype,  
    PRIMARY KEY( one or more columns )  
);
```

Example:

```
CREATE TABLE myschema.employee (  
    id bigint NOT NULL,  
    birth_date date NOT NULL,  
    first_name character varying(14) NOT NULL,  
    last_name character varying(16) NOT NULL,  
    gender employee_gender NOT NULL,  
    hire_date date NOT NULL  
);
```

- To drop a schema if it is empty (all objects in it have been dropped), use the below command:

Syntax:

```
DROP SCHEMA schema_name;
```

- To drop a schema including all contained objects, use the below command:

Syntax:

```
DROP SCHEMA schema_name CASCADE
```

Inserting into tables:

The PostgreSQL **INSERT INTO** statement allows one to insert new rows into a table. One can insert a single row at a time or several rows as a result of a query.

Syntax:

```
INSERT INTO TABLE_NAME (column1, column2, column3,...columnN)  
VALUES (value1, value2, value3,...valueN);
```

Example:

```
INSERT INTO employee(id, birth_date, first_name, last_name, gender,  
hire_data)  
VALUES (100001, 1953-02-09, 'Georgi', 'Facello', 'M', 1988-02-06);
```

- The following example inserts multiple rows using the multirow VALUES syntax :

Example:

```
INSERT INTO employee(id, birth_date, first_name, last_name, gender,  
hire_data)
```

```
VALUES (10001, '1953-02-09', 'Georgi', 'Facello', 'M', '1988-02-06'),  
(10002, '1964-02-06', 'Bezalel', 'Simmel', 'F', '1986-09-11'),  
(10003, '1959-03-12', 'Parto', 'Bamford', 'M', '1988-04-08');
```

Selecting from table:

- PostgreSQL **SELECT** statement is used to fetch the data from a database table, which returns data in the form of result table. These results are called result-sets.

Syntax:

```
SELECT column1, column2, columnN FROM table_name;
```

- Here, column1, column2...are the fields of a table, whose values you want to fetch. If you want to fetch all the fields available in the field then you can use the following syntax:

Syntax:

```
SELECT * FROM table_name;
```

Example:

```
SELECT id, first_name, gender, hire_date from employee;
```

	123 id	ABC first_name	ABC gender	⌚ hire_date
1	10,001	Georgi	M	1988-02-06
2	10,002	Bezalel	F	1986-09-11
3	10,003	Parto	M	1988-04-08
4	10,004	Chirstian	M	1986-01-12
5	10,005	Kyoichi	M	1989-12-09
6	10,006	Anneke	F	1989-02-06

Example:

```
SELECT * from employee;
```

	123 id	birth_date	first_name	last_name	gender	hire_date
1	10,001	1953-02-09	Georgi	Facello	M	1988-02-06
2	10,002	1964-02-06	Bezalel	Simmel	F	1986-09-11
3	10,003	1959-03-12	Parto	Bamford	M	1988-04-08
4	10,004	1954-01-05	Chirstian	Koblick	M	1986-01-12
5	10,005	1956-09-01	Kyoichi	Maliniak	M	1989-12-09
6	10,006	1954-08-04	Anneke	Preusig	F	1989-02-06

Using operators:

- An operator is a reserved word, or a character used primarily in a PostgreSQL statement's WHERE clause to perform operations, such as comparisons and arithmetic operations. Operators are used to specify conditions in a PostgreSQL statement and to serve as conjunctions for multiple conditions in a statement.

- Arithmetic operators
- Comparison operators
- Logical operators
- Bitwise operators

Arithmetic operators:

Addition:

- Addition adds the values on either side of the operator.

Example:

```
SELECT employee_id, from_date, amount+100 as "amount+100" FROM salary;
```

VN2 Solutions Pvt. Ltd.

	123 employee_id ↑↓	⌚ from_date ↑↓	123 amount+100 ↑↓
1	10,001	1988-02-06	61,117
2	10,001	1989-02-06	63,102
3	10,001	1990-01-06	67,074
4	10,001	1991-01-06	67,596
5	10,001	1992-01-06	67,961
6	10,001	1993-01-06	72,046

Subtraction:

- Subtracts right hand operand from left hand operand.

Example:

```
select employee_id, from_date, amount-100 as "amount-100" from salary;
```

	123 employee_id ↑↓	⌚ from_date ↑↓	123 amount-100 ↑↓
1	10,001	1988-02-06	59,117
2	10,001	1989-02-06	61,102
3	10,001	1990-01-06	65,074
4	10,001	1991-01-06	65,596
5	10,001	1992-01-06	65,961
6	10,001	1993-01-06	70,046

Multiplication:

- Multiplies values on either side of the operator

Example:

```
select employee_id, from_date, amount*2 as "amount*2" from salary;
```

	employee_id	from_date	amount*2
1	10,001	1988-02-06	120,234
2	10,001	1989-02-06	124,204
3	10,001	1990-01-06	132,148
4	10,001	1991-01-06	133,192
5	10,001	1992-01-06	133,922
6	10,001	1993-01-06	142,092

Division:

- Divides left hand operand by right hand operand

Example:

```
select employee_id, from_date, amount/10 as "amount/10" from salary;
```

	employee_id	from_date	amount/10
1	10,001	1988-02-06	6,011
2	10,001	1989-02-06	6,210
3	10,001	1990-01-06	6,607
4	10,001	1991-01-06	6,659
5	10,001	1992-01-06	6,696
6	10,001	1993-01-06	7,104

,mqnegtikj90o`

Comparison operators:

Equal (=):

- Checks if the values of two operands are equal or not, if yes then condition becomes true.

Example:

```
select * from salary where amount=61000;
```

	employee_id	amount	from_date	to_date
1	14,114	61,000	1997-08-11	1998-08-11
2	18,116	61,000	2001-10-03	2002-10-03
3	18,209	61,000	2001-09-04	2002-09-04
4	18,420	61,000	1999-02-04	2000-02-04
5	23,544	61,000	2001-07-01	2002-06-01
6	23,608	61,000	1986-08-12	1987-08-12

Not equal (!=):

- Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.

Example:

```
select * from salary where amount!=61000;
```

VN2 Solutions Pvt. Ltd.

Less than (<):

- Checks if the value of right operand is greater than the value of left operand, if yes then condition becomes true.

Example:

```
select * from salary where amount<70000;
```

	employee_id	amount	from_date	to_date
1	10,001	60,117	1988-02-06	1989-02-06
2	10,001	62,102	1989-02-06	1990-01-06
3	10,001	66,074	1990-01-06	1991-01-06
4	10,001	66,596	1991-01-06	1992-01-06
5	10,001	66,961	1992-01-06	1993-01-06
6	10,002	65,828	1996-03-08	1997-03-08

Greater than (>):

- Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.

Example:

```
select * from salary where amount>70000;
```

	employee_id	amount	from_date	to_date
1	10,001	71,046	1993-01-06	1993-12-06
2	10,001	74,333	1993-12-06	1994-12-06
3	10,001	75,286	1994-12-06	1995-12-06
4	10,001	75,994	1995-12-06	1996-12-06
5	10,001	76,884	1996-12-06	1997-11-06
6	10,001	80,013	1997-11-06	1998-11-06

Less than or equal to (<=):

- Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.

Example:

```
select * from salary where amount<=70000;
```

	employee_id	amount	from_date	to_date
1	10,001	60,117	1988-02-06	1989-02-06
2	10,001	62,102	1989-02-06	1990-01-06
3	10,001	66,074	1990-01-06	1991-01-06
4	10,001	66,596	1991-01-06	1992-01-06
5	10,001	66,961	1992-01-06	1993-01-06
6	10,002	65,828	1996-03-08	1997-03-08

Greater than or equal to (>=):

- Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.

Example:

```
select * from salary where amount>=70000;
```

	123 employee_id ↕	123 amount ↕	⌚ from_date ↕	⌚ to_date ↕
1	10,001	71,046	1993-01-06	1993-12-06
2	10,001	74,333	1993-12-06	1994-12-06
3	10,001	75,286	1994-12-06	1995-12-06
4	10,001	75,994	1995-12-06	1996-12-06
5	10,001	76,884	1996-12-06	1997-11-06
6	10,001	80,013	1997-11-06	1998-11-06

Not equal (<>):

- Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.

Example:

```
select * from salary where amount <> 60000;
```

	123 employee_id ↕	123 amount ↕	⌚ from_date ↕	⌚ to_date ↕
1	10,001	60,117	1988-02-06	1989-02-06
2	10,001	62,102	1989-02-06	1990-01-06
3	10,001	66,074	1990-01-06	1991-01-06
4	10,001	66,596	1991-01-06	1992-01-06
5	10,001	66,961	1992-01-06	1993-01-06
6	10,001	71,046	1993-01-06	1993-12-06

Example:

```
select * from department_employee where department_id <> 'd005';
```

VN2 Solutions Pvt. Ltd.

	employee_id	Ctrl+click to open SQL console	m_date	to_date
1	10,002	d007	1996-03-08	1999-01-01
2	10,003	d004	1995-03-12	1999-01-01
3	10,004	d004	1986-01-12	1999-01-01
4	10,005	d003	1989-12-09	1999-01-01
5	10,007	d008	1989-10-02	1999-01-01
6	10,009	d006	1986-06-02	1999-01-01

Logical operators:

- The AND, OR, and NOT keywords are PostgreSQL's Boolean operators. These keywords are mostly used to join or invert conditions in a SQL statement, specifically in the WHERE clause and the HAVING clause.

AND:

- The AND operator allows the existence of multiple conditions in a PostgreSQL statement's WHERE clause.

Example:

```
select * from department_employee where department_id = 'd005' AND  
from_date > '1988-02-06';
```

	employee_id	department_id	from_date	to_date
1	10,006	d005	1990-05-08	1999-01-01
2	10,008	d005	1998-11-03	2002-07-07
3	10,012	d005	1993-06-12	1999-01-01
4	10,014	d005	1995-05-12	1999-01-01
5	10,021	d005	1988-10-02	2003-03-07
6	10,022	d005	1999-03-09	1999-01-01

NOT:

- The NOT operator reverses the meaning of the logical operator with which it is used. Eg. NOT EXISTS, NOT BETWEEN, NOT IN etc. This is negate operator.

Example:

```
select * from department_employee where not department_id = 'd005';
```

	employee_id	department_id	from_date	to_date
1	10,002	d007	1996-03-08	1999-01-01
2	10,003	d004	1995-03-12	1999-01-01
3	10,004	d004	1986-01-12	1999-01-01
4	10,005	d003	1989-12-09	1999-01-01
5	10,007	d008	1989-10-02	1999-01-01
6	10,009	d006	1986-06-02	1999-01-01

OR:

- The OR operator is used to combine multiple conditions in a PostgreSQL statement's WHERE clause.

Example:

```
select * from department_employee where department_id = 'd005' or  
department_id = 'd004';
```

	employee_id	department_id	from_date	to_date
1	10,001	d005	1988-02-06	1999-01-01
2	10,003	d004	1995-03-12	1999-01-01
3	10,004	d004	1986-01-12	1999-01-01
4	10,006	d005	1990-05-08	1999-01-01
5	10,008	d005	1998-11-03	2002-07-07
6	10,010	d004	1997-12-11	2002-02-06

Expressions:

- An expression is a combination of one or more values, operators, and PostgreSQL functions that evaluate to a value. PostgreSQL expressions are like formulas and are written in query language. You can also use them to query the database for specific set of data.

Syntax:

```
SELECT column1, column2, columnN
```

```
FROM table_name
```

```
WHERE [CONDITION | EXPRESSION];
```

Example:

```
select * from department_employee where from_date > '1988-02-06' and  
department_id = 'd005';
```

	employee_id	department_id	from_date	to_date
1	10,006	d005	1990-05-08	1999-01-01
2	10,008	d005	1998-11-03	2002-07-07
3	10,012	d005	1993-06-12	1999-01-01
4	10,014	d005	1995-05-12	1999-01-01
5	10,021	d005	1988-10-02	2003-03-07
6	10,022	d005	1999-03-09	1999-01-01

WHERE clause:

- The PostgreSQL WHERE clause is used to specify a condition while fetching the data from single table or joining with multiple tables. If one condition is satisfied, only then it returns specific value from the table. You can filter out rows that you do not want included in the result-set by using the WHERE clause. The WHERE clause not only used in SELECT, but it is also used in UPDATE, DELETE statement etc.

Syntax:

```
SELECT column1, column2, columnN  
FROM table_name  
WHERE [search_condition]
```

Example:

```
select * from employee where gender = 'F';
```

VN2 Solutions Pvt. Ltd.

	123 id	birth_date	first_name	last_name	gender	hire_date
1	10,002	1964-02-06	Bezalel	Simmel	F	1986-09-11
2	10,006	1954-08-04	Anneke	Preusig	F	1989-02-06
3	10,007	1958-11-05	Tzvetan	Zielinski	F	1989-10-02
4	10,009	1953-07-04	Sumant	Peac	F	1986-06-02
5	10,010	1963-01-06	Duangkaew	Piveteau	F	1990-12-08
6	10,011	1953-07-11	Mary	Sluis	F	1991-10-01

Example:

```
select * from employee where gender = 'M' and birth_date > '1953-02-09';
```

	123 id	birth_date	first_name	last_name	gender	hire_date
1	10,003	1959-03-12	Parto	Bamford	M	1988-04-08
2	10,004	1954-01-05	Chirstian	Koblick	M	1986-01-12
3	10,005	1956-09-01	Kyoichi	Maliniak	M	1989-12-09
4	10,008	1959-07-02	Saniya	Kalloufi	M	1995-03-09
5	10,012	1960-04-10	Patricio	Bridgland	M	1993-06-12
6	10,013	1963-07-06	Eberhardt	Terkki	M	1986-08-10

UPDATE query:

The PostgreSQL UPDATE query is used to modify the existing records in a table. You can use WHERE clause with UPDATE query to update the selected rows. Otherwise, all the rows would be updated.

Syntax:

```
UPDATE table_name  
SET column1 = value1, column2 = value2...., columnN = valueN  
WHERE [condition];
```

Example:

```
update salary set amount = 20000 where employee_id = 10001;
```

Example:

```
update salary set employee_id = 10001, amount = 30000 where  
employee_id = 10002;
```

DELETE query:

- The PostgreSQL DELETE query is used to delete the existing records from a table. You can use WHERE clause with DELETE query to delete the selected rows. Otherwise, all the records would be deleted.

Syntax:

```
DELETE FROM table_name  
WHERE [condition];
```

Note: You can combine N number of conditions using AND or OR operators.

Example:

```
delete from employee where id = '10001';
```

Example:

```
delete from salary where employee_id = 10001 and amount > 70000;
```

- If you want to DELETE all the records from table, you do not need to use WHERE clause.

Example:

Delete from employee;

LIKE clause:

- The PostgreSQL LIKE operator is used to match text values against a pattern using wildcards. If you search expression can be matched to the pattern expression, the LIKE will return true, which is 1.
- There are 2 wildcards used in conjunction with the LIKE operators
 - Percent sign(%)
 - Underscore(_)
- The percent sign represents 0 or 1 or multiple numbers or characters. The underscore represents a single number of character. These symbols can be used in combinations.
- If either of these two signs is not used in conjunction with the LIKE clause, then the LIKE acts like the equals operator.

Syntax:

```
SELECT FROM table_name
```

```
WHERE column LIKE 'XXXX%'
```

or

```
SELECT FROM table_name
```

```
WHERE column LIKE '%XXXX%'
```

or

```
SELECT FROM table_name
```

```
WHERE column LIKE 'XXXX_'
```

or

```
SELECT FROM table_name
```

```
WHERE column LIKE '_XXXX'
```

or

```
SELECT FROM table_name
```

```
WHERE column LIKE '_XXXX_'
```

Example:

```
select * from employee where first_name like 'G%';
```

	123 id ↕	⌚ birth_date ↕	ABC first_name ↕	RBC last_name ↕	RBC gender ↕	⌚ hire_date ↕
1	10,015	1960-07-08	Guoxiang	Nooteboom	M	1987-02-07
2	10,055	1956-06-06	Georgy	Dredge	M	1994-03-04
3	10,063	1952-06-08	Gino	Leonhardt	F	1989-08-04
4	10,075	1960-09-03	Gao	Dolinsky	F	1988-07-03
5	10,121	1963-02-07	Guoxiang	Ramsay	M	1989-03-05
6	10,124	1963-11-05	Geraldo	Marwedel	M	1991-05-09
7	10,133	1963-12-12	Giri	Isaak	M	1986-03-12

Example:

```
select * from employee where first_name like 'Gen%';
```

	123 id ↕	⌚ birth_date ↕	ABC first_name ↕	RBC last_name ↕	RBC gender ↕	⌚ hire_date ↕
1	10,219	1952-02-05	Genta	Kolvik	M	1995-07-03
2	id: int8 0	1963-12-08	Gen	Tiemann	F	1995-04-09
3	11,008	1962-11-07	Gennady	Menhoudj	M	1989-06-09
4	11,060	1955-06-06	Genevieve	Pokrovskii	F	1991-03-05
5	11,196	1959-01-05	Gennadi	Breugel	M	1993-05-11
6	11,474	1954-05-04	Gennady	Akazan	M	1992-02-07

Example:

```
select * from employee where first_name like '%dy';
```

VN2 Solutions Pvt. Ltd.

	123 id	birth_date	ABC first_name	ABC last_name	ABC gender	hire_date
1	10,255	1964-08-02	Roddy	Garnick	M	1993-12-05
2	10,750	1954-02-06	Roddy	Demeyer	F	1991-04-08
3	10,756	1959-12-06	Paddy	Brizzi	M	1996-09-11
4	11,008	1962-11-07	Gennady	Menhoudj	M	1989-06-09
5	11,099	1958-12-07	Woody	Spataro	M	1987-07-12
6	11,285	1963-11-02	Randy	Koshino	M	1997-03-03

Example:

```
select * from salary where amount::text like '6000%';
```

	123 employee_id	123 amount	from_date	to_date
1	83,486	60,005	1987-02-07	1988-02-07
2	87,940	60,001	1987-12-02	1988-12-02
3	107,605	60,008	1988-01-12	1988-12-12
4	22,883	60,005	1986-10-05	1987-10-05
5	30,168	60,000	1986-05-03	1987-05-03
6	64,657	60,000	1988-02-05	1989-01-05
7	64,861	60,004	1987-12-09	1988-12-09

Example:

```
select * from employee where first_name like '%eo%';
```

VN2 Solutions Pvt. Ltd.

	123 id ↕	⌚ birth_date ↕	ABC first_name ↕	ABC last_name ↕	ABC gender ↕	⌚ hire_date ↕
1	10,032	1960-09-08	Jeong	Reistad	F	1991-08-06
2	10,055	1956-06-06	Georgy	Dredge	M	1994-03-04
3	10,165	1961-04-06	Miyeon	Macedo	M	1989-05-05
4	10,224	1955-05-05	Leon	Trogemann	M	1988-09-01
5	10,247	1966-06-06	Heon	Riefers	F	1993-02-08
6	10,337	1957-10-12	Jeong	Sadowsky	M	1995-06-08
7	10,338	1961-12-02	Heon	Ranai	M	1988-01-09

Example:

select * from employee where first_name like '_e____';

	123 id ↕	⌚ birth_date ↕	ABC first_name ↕	ABC last_name ↕	ABC gender ↕	⌚ hire_date ↕
1	10,052	1963-02-02	Heping	Nitsch	M	1989-09-05
2	10,055	1956-06-06	Georgy	Dredge	M	1994-03-04
3	10,070	1956-08-08	Reuven	Gariglano	M	1986-02-10
4	10,090	1963-06-05	Kendra	Hofting	M	1987-02-03
5	10,225	1958-01-02	Kellie	Chinen	F	1987-07-06
6	10,288	1959-02-06	Selwyn	Perri	M	1996-05-08

Example:

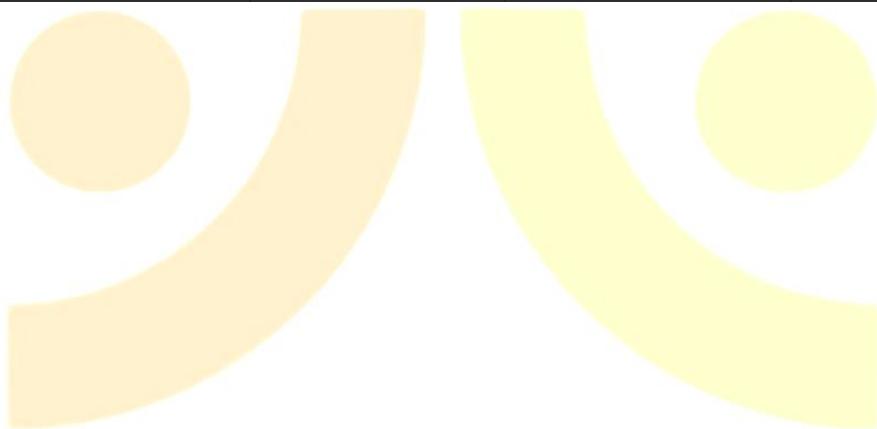
select * from employee where first_name like '_a%a';

	123 id ↕	⌚ birth_date ↕	ABC first_name ↕	ABC last_name ↕	ABC gender ↕	⌚ hire_date ↕
1	10,008	1959-07-02	Saniya	Kalloufi	M	1995-03-09
2	10,069	1960-06-09	Margareta	Bierman	F	1989-05-11
3	10,093	1964-11-06	Sailaja	Desikan	M	1996-05-11
4	10,131	1953-07-02	Magdalena	Eldridge	M	1995-05-11
5	10,144	1960-05-06	Marla	Brendel	M	1986-02-10
6	10,164	1957-07-01	Jagoda	Braunmuhl	M	1985-12-11

Example:

```
select * from salary where amount:text like '6___1';
```

	123 employee_id ↕	123 amount ↕	⌚ from_date ↕	⌚ to_date ↕
1	69,966	66,121	1987-05-11	1988-05-11
2	71,068	63,691	1987-05-10	1988-04-10
3	71,282	60,251	1987-05-02	1988-05-02
4	71,616	61,281	1986-02-03	1987-02-03
5	71,616	62,651	1987-02-03	1988-01-03
6	71,766	62,271	1987-11-08	1988-04-05
7	71,806	64,181	1986-08-06	1987-08-06



LIMIT clause:

- The PostgreSQL LIMIT clause is used to limit the data amount returned by the SELECT statement.

Syntax:

```
SELECT column1, column2, columnN  
FROM table_name  
LIMIT [no of rows]
```

Example:

```
select * from salary limit 5;
```

	employee_id	amount	from_date	to_date
1	204,388	40,000	1987-11-06	1988-11-06
2	204,399	40,000	1985-11-10	1986-11-10
3	204,399	40,444	1986-11-10	1987-11-10
4	204,399	44,491	1987-11-10	1988-10-10
5	204,401	59,468	1986-01-09	1987-01-09

- The following is the syntax of LIMIT clause when it is used along with OFFSET clause:

Syntax:

```
SELECT column1, column2, columnN  
FROM table_name  
LIMIT [no of rows] OFFSET [row num]
```

- LIMIT and OFFSET allow you to retrieve just a portion of the rows that are generated by the rest of the query.

Example:

```
select * from department_employee limit 5 offset 3;
```

	employee_id	department_id	from_date	to_date
1	10,004	d004	1986-01-12	1999-01-01
2	10,005	d003	1989-12-09	1999-01-01
3	10,006	d005	1990-05-08	1999-01-01
4	10,007	d008	1989-10-02	1999-01-01
5	10,008	d005	1998-11-03	2002-07-07

ORDER BY clause:

- The PostgreSQL ORDER BY clause is used to sort the data in ascending or descending order, based on one or more columns.

Syntax:

```
SELECT column-list  
FROM table_name  
[WHERE condition]  
[ORDER BY column1, column2, .. columnN] [ASC | DESC];
```

- You can use more than one column in the ORDER BY clause. Make sure whatever column you are using to sort, that column should be available in column list.

Example:

```
select * from employee order by birth_date;
```

VN2 Solutions Pvt. Ltd.

	123 id	⌚ birth_date	ABC first_name	ABC last_name	ABC gender	⌚ hire_date
1	87,461	1952-01-02	Moni	Decaestecker	M	1986-06-10
2	65,308	1952-01-02	Jouni	Pocchiola	M	1985-10-03
3	237,571	1952-01-02	Ronghao	Schaad	M	1988-10-07
4	91,374	1952-01-02	Eishiro	Kuzuoka	M	1992-12-02
5	207,658	1952-01-02	Kiyokazu	Whitcomb	M	1990-02-07
6	406,121	1952-01-02	Supot	Remmele	M	1991-03-01

Example:

```
select * from employee order by hire_date desc;
```

	123 id	⌚ birth_date	ABC first_name	ABC last_name	ABC gender	⌚ hire_date
25	225,657	1960-10-08	Xiaoheng	Chenoweth	F	2001-07-01
26	103,208	1954-04-10	Charmane	Cannard	M	2001-07-01
27	203,299	1953-02-07	Girolamo	Binkley	F	2001-07-01
28	94,646	1956-08-05	Mahendra	Kawashimo	F	2001-07-01
29	71,159	1953-07-07	Manton	Ghemri	F	2001-06-12
30	73,925	1959-11-08	Vasilii	Stavenow	M	2001-06-12

Example:

```
select * from salary order by amount, from_date;
```

	123 employee_id	123 amount	⌚ from_date	⌚ to_date
1	10,001	20,000	1988-02-06	1989-02-06
2	209,812	20,000	1988-02-07	1989-02-07
3	210,226	20,000	1988-02-07	1989-02-07
4	80,894	20,000	1988-02-07	1989-02-07
5	81,172	20,000	1988-02-07	1989-01-07

GROUP BY clause:

- The PostgreSQL GROUP BY clause is used in collaboration with the SELECT statement to group together those rows in a table that have identical data. This is done to eliminate redundancy in the output and/or compute aggregates that apply to these groups.
- The GROUP BY clause follows the WHERE clause in a SELECT statement and precedes the ORDER BY clause.
- The basic syntax of GROUP BY clause is given below. The GROUP BY clause must follow the conditions in the WHERE clause and must precede the ORDER BY clause if one is used.

Syntax:

```
SELECT column-list  
FROM table_name  
WHERE [ conditions ]  
GROUP BY column1, column2....columnN  
ORDER BY column1, column2....columnN
```

- You can use more than one column in the GROUP BY clause. Make sure whatever column you are using to group, that column should be available in column list.