UNIT-4

SPARK

Introduction to Data Analysis with Spark: What is a Apache Spark, A unified Spark, Who uses Spark and for what? A Brief History of Spark, Spark version and releases, Storage layers for Spark.
Programming with RDDs: RDD Basics, Creating RDDs, RDD Operations, Passing functions to Spark, Common Transformations and Actions, Persistence.
Spark SQL: Linking with Spark SQL, Using Spark SQL in Applications, Loading and Saving Data, JDBC/ODBC Server, User-defined functions, Spark SQL Performance.

Q) What is Spark? Explain features of Spark.

Apache Spark is an open-source, distributed processing system used for big data analytics. It utilizes in-memory caching, and optimized query execution for fast analytic queries against data of any size.

It provides development APIs in Java, Scala, Python and R, and supports code reuse across multiple nodes—batch processing, interactive queries, real-time analytics, machine learning, and graph processing.

Spark was initially started by Matei Zaharia at UC Berkeley's AMPLab in 2009, and open sourced in 2010. In 2013, the project was donated to the Apache Software Foundation and switched its license to Apache 2.0 from BSD. In February 2014, Spark became a Top-Level Apache Project.

Spark is used by organizations from any industry, including at FINRA, Yelp, Zillow, DataXu, Urban Institute, and CrowdStrike.

Spark uses Hadoop in two ways – one is storage and second is processing. Since Spark has its own cluster management computation, it uses Hadoop for storage purpose only.

Features of Spark:

1. **Swift Processing:** Apache Spark offers high data processing speed. Thatis about 100x faster in memory and 10x faster on the disk. However, it is only possible by reducing the number of read-write to disk.

2. **Dynamic in Nature:** Basically, it is possible to develop a parallelapplication in Spark. Since there are 80 high-level operators available in Apache Spark.

3. **In-Memory Computation in Spark:** The increase in processing speed ispossible due to in-memory processing. It enhances the processing speed.

4. **Reusability:** We can easily reuse spark code for batch-processing or joinstream against historical data. Also to run ad-hoc queries on stream state.

5. **Spark Fault Tolerance:** Spark offers fault tolerance. It is possiblethrough Spark's core abstraction-RDD.

6. **Real-Time Stream Processing:** We can do real-time stream processing in Spark. Basically, Hadoop does not support real-time processing.

7. **Lazy Evaluation in Spark:** All the transformations we make in SparkRDD are Lazy in nature that is it does not give the result right away rather a new RDD is formed from the existing one. Thus, this increases the efficiency of the system.

8. **Polyglot:** Spark provides high-level APIs in Java, Scala, Python, and R.Spark code can be written in any of these four languages. It also provides a shell in Scala and Python.

9. **Support for Sophisticated Analysis:** There are dedicated tools in ApacheSpark. Such as for streaming data interactive/declarative queries, machine learning which add-on to map and reduce.

10. **Integrated with Hadoop:** As we know Spark is flexible. It can runindependently and also on Hadoop YARN Cluster Manager. Even it can read existing Hadoop data.

11. **Spark GraphX:** In Spark, a component for graph and graph-parallelcomputation, we have GraphX.

12. **Cost Efficient:** For Big data problem as in Hadoop, a large amount ofstorage and the large data centre is required during replication. Hence, Spark programming turns out to be a cost-effective solution.

Q) Explain Spark Architecture in-detail.

Apache Spark Architecture is based on two main abstractions-
- Resilient Distributed Datasets (RDD)
- Directed Acyclic Graph (DAG)

RDDs are the building blocks of any Spark application. RDDs Stands for:
- **Resilient:** Fault tolerant and is capable of rebuilding data on failure
- **Distributed:** Distributed data among the multiple nodes in a cluster
- **Dataset:** Collection of partitioned data with values

Directed Acyclic Graph (DAG)

Direct - Transformation is an action which transitions data partition state from A to B.

Acyclic -Transformation cannot return to the older partition

DAG is a sequence of computations performed on data where each node is an RDD partition and edge is a transformation on top of data. The DAG abstraction helps eliminate the HadoopMapReduce multistage execution model and provides performance enhancements over Hadoop.

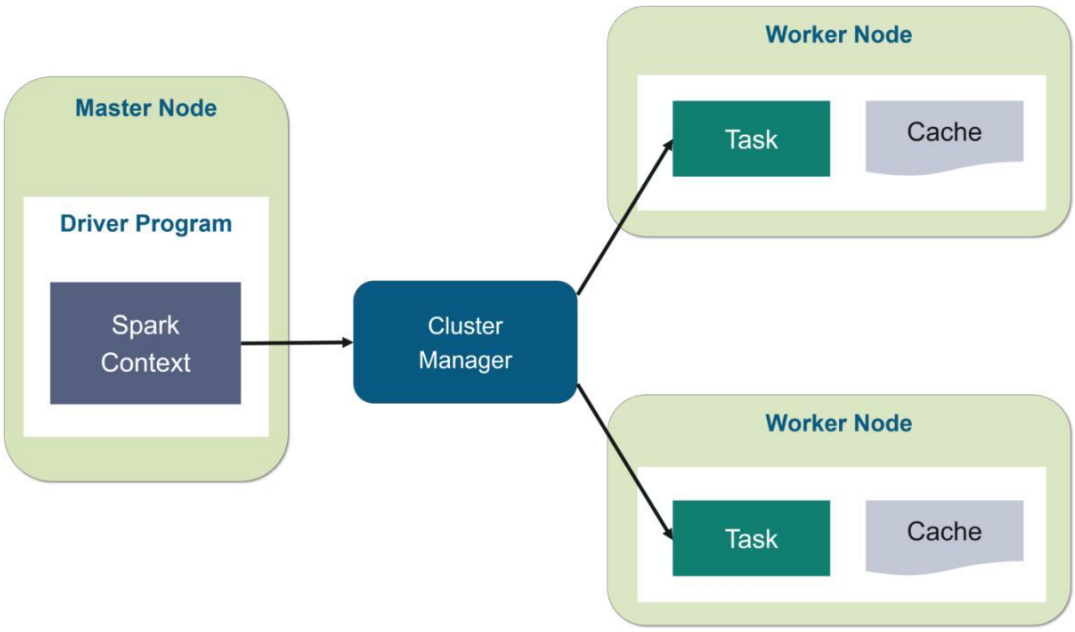


Fig: Spark Architecture

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

i. Master Daemon – (Master/Driver Process)
ii. Worker Daemon –(Slave Process/Executor)

A spark cluster has a single Master and any number of Slaves/Workers.

Spark Driver – Master Node of a Spark Application(Master):

- 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. It is similar to your database connection.
- 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.

Executor/Worker Node(Slave):

- 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.

Cluster Manager:

An external service responsible for acquiring resources on the spark cluster and allocating them to a spark job.

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.

Application:

- 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.
- **spark-submit** is the single script used to submit a spark program and launches the application on the cluster.

Q) Write a brief note on: Spark Unified Stack.

The Spark project contains multiple closely integrated components. At its core, Spark is a ―computational engine‖ that is responsible for scheduling,

distributing, and monitoring applications consisting of many computational tasks across many worker machines, or a *computing cluster*.

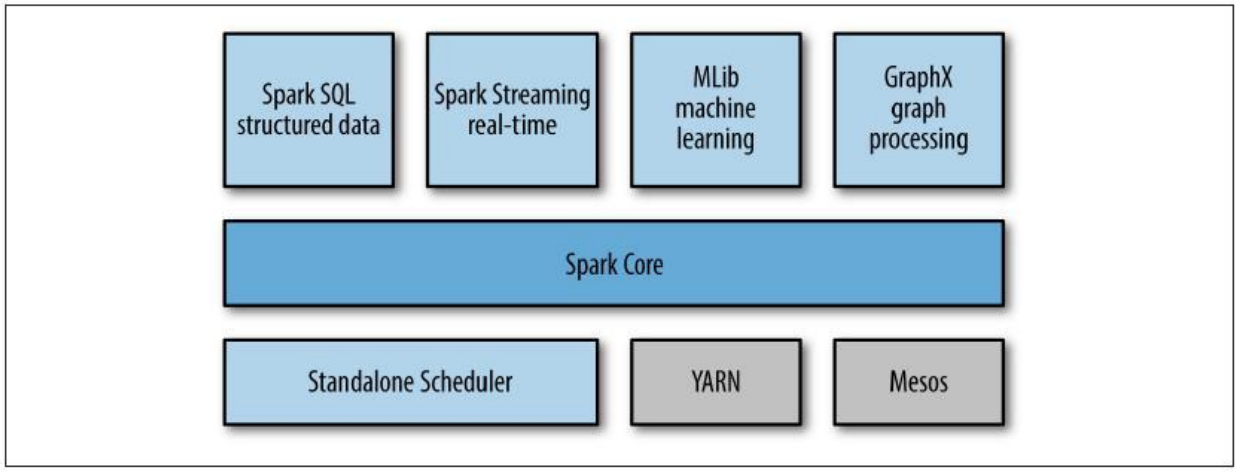


Fig.*The Spark stack*

Spark Core

- Spark Core contains the basic functionality of Spark, including components for task scheduling, memory management, fault recovery, interacting with storage systems, and more.
- Spark Core is also home to the API that defines *resilient distributed datasets* (RDDs), which are Spark's main programming abstraction.
- Spark Core provides many APIs for building and manipulating these collections.

Spark SQL

- Spark SQL was added to Spark in version 1.0.
- Spark SQL is Spark's package for working with structured data. It allows querying data via SQL as well as the Apache Hive variant of SQL—called the Hive Query Language (HQL)—and it supports many sources of data, including Hive tables and JSON.
- Spark SQL allows developers to intermix SQL queries with the programmatic data manipulations supported by RDDs in Python, Java, and Scala, all within a single application, thus combining SQL with complex analytics.

Spark Streaming

- Spark Streaming is a Spark component that enables processing of live streams of data.
  Eg.logfiles generated by production web servers.
- Spark Streaming was designed to provide the same degree of fault tolerance, throughput, and scalability as Spark Core.

MLlib

- Spark comes with a library containing common machine learning (ML) functionality, called MLlib.
- MLlib provides multiple types of machine learning algorithms, including classification, regression, clustering, and collaborative filtering, as well as supporting functionality such as model evaluation and data import.

GraphX

- GraphX is a library for manipulating graphs (e.g., a social network's friend graph) and performing graph-parallel computations.
- Like Spark Streaming and Spark SQL, GraphX extends the Spark RDD API, allowing us to create a directed graph with arbitrary properties attached to each vertex and edge.
- GraphX also provides various operators for manipulating graphs (e.g., subgraph and mapVertices) and a library of common graph algorithms (e.g., PageRank and triangle counting).

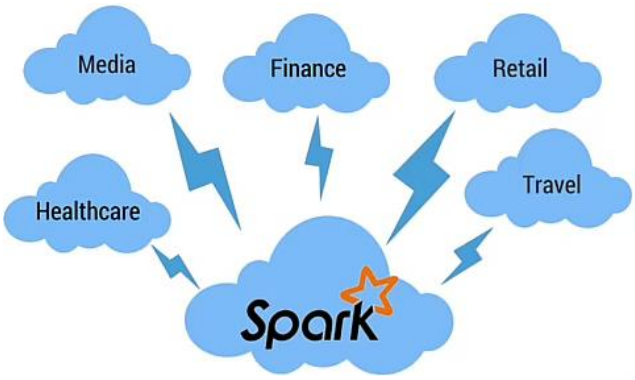
Cluster Managers

- Spark is designed to efficiently scale up from one to many thousands of compute nodes.
- To achieve this while maximizing flexibility, Spark can run over a variety of *cluster managers*, including Hadoop YARN, Apache Mesos, and a simplecluster manager included in Spark itself called the Standalone Scheduler.

Q) Who uses Spark? Explain applications of spark.

Data Scientists and Data Engineers uses Spark. Their main task is to analyze and model data. They may have experience with SQL, statistics, predictive modeling (machine learning), and programming, usually in Python, Matlab, or R. Data scientists also have experience with techniques necessary to transform data into formats that can be analyzed for insights.

Spark is a general-purpose framework for cluster computing, it is used for a diverse range of applications.



Applications of Spark

1. **Finance:** Banks are using Spark to access and analyse the social media profiles, call recordings, complaint logs, emails, forum discussions, etc. to gain insights which can help them make right business decisions for credit risk assessment, targeted advertising and customer segmentation.

2. **E-Commerce(Alibaba & ebay):** Information about real time transaction can be passed to streaming clustering algorithms like alternating least squares (collaborative filtering algorithm) or K-means clustering algorithm.

   The results can be combined with data from other sources like social media profiles, product reviews on forums, customer comments, etc. to enhance the recommendations to customers based on new trends.

3. **Health Care(MyFitnessPal):** MyFitnessPal helps people achieve a healthy lifestyle through better diet and exercise. MyFitnessPal uses apache spark to clean the data entered by users with the end goal of identifying high quality food items.
   Media & Entertainment

   4. Gaming(Tencent, Riot):

   Apache Spark is used in the gaming industry to identify patterns from the real-time in-game events and respond to them to harvest lucrative business opportunities like targeted advertising, auto adjustment of gaming levels based on complexity, player retention and many more.

   Spark improves the gaming experience of the users, it also helps in processing different game skins, different game characters, in-game points, and much more. It helps with performance improvement, offers, and efficiency. Riot can now detect the cause which made the game slow and laggy, so they can solve problems on time without impacting users.

5. Media:

Yahoo uses Apache Spark for personalizing its news webpages and for targeted advertising. It uses machine learning algorithms that run on Apache Spark to find out what kind of news - users are interested to read and categorizing the news stories to find out what kind of users would be interested in reading each category of news.

Netflix uses Apache Spark for real-time stream processing to provide online recommendations to its customers. Streaming devices at Netflix send events which capture all member activities and play a vital role in personalization. It processes 450 billion events per day which flow to server side applications and are directed to Apache Kafka.

6. Travel:

TripAdvisor, a leading travel website that helps users plan a perfect trip is using Apache Spark to speed up its personalized customer recommendations. TripAdvisor uses apache spark to provide advice to millions of travellers by comparing hundreds of websites to find the best hotel prices for its customers. The time taken to read and process the reviews of the hotels in a readable format is done with the help of Apache Spark.

OpenTable, an online real time reservation service, with about 31000 restaurants and 15 million diners a month, uses Spark for training its recommendation algorithms and for NLP of the restaurant reviews to generate new topic models.

Q) Briefly explain about Spark Storage Layers.(OR) Explain how to load and save data in spark.

Spark can create distributed datasets from any file stored in the Hadoop distributed filesystem (HDFS) or other storage systems supported by the Hadoop APIs (including your local filesystem, Amazon S3, Cassandra, Hive, HBase, etc).

It's important to remember that Spark does not require Hadoop; it simply has support for storage systems implementing the Hadoop APIs. Spark supports text files, SequenceFiles, Avro, Parquet, and any other Hadoop InputFormat.

Table 5-1. Common supported file formats

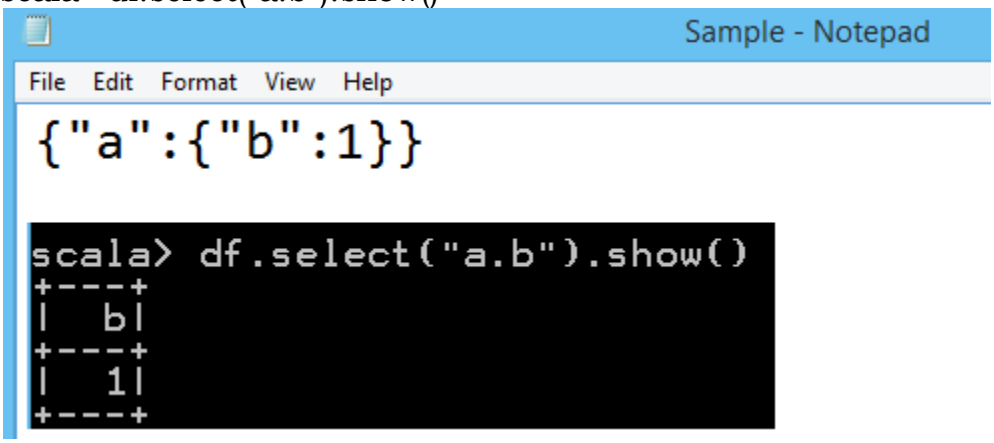
Format name	Structured	Comments
Text files	No	Plain old text files. Records are assumed to be one per line.
JSON	Semi	Common text-based format, semistructured; most libraries require one record per line.
CSV	Yes	Very common text-based format, often used with spreadsheet applications.
SequenceFiles	Yes	A common Hadoop file format used for key/value data.
Protocol buffers	Yes	A fast, space-efficient multilanguage format.
Object files	Yes	Useful for saving data from a Spark job to be consumed by shared code. Breaks if you change your classes, as it relies on Java Serialization.

Text file:

```
scala> val rd1 =
spark.sparkContext.wholeTextFiles("C:/Users/chpraneeth/Desktop/
links.txt")
scala> rd1.collect()
rd1. saveAsTextFile(“Tout”)
```

JSON File:

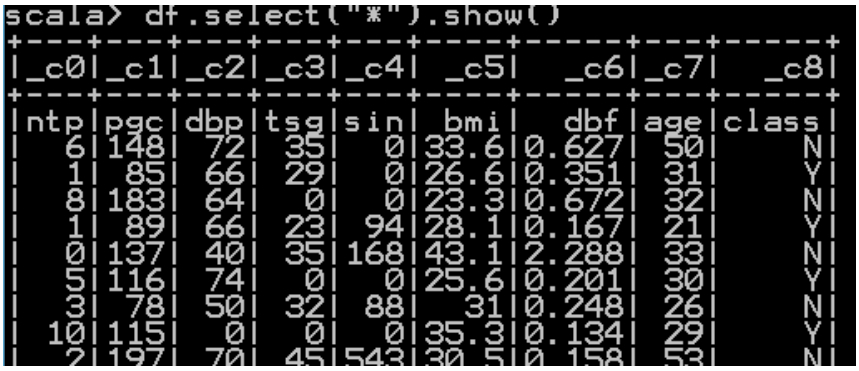
```
scala> val df = spark.read.json("Sample.json")
scala> df.select("a.b").show()
```




```
scala> df.write.json("Jout")
```

CSV file:

```
scala> val df = spark.read.csv("pima_diabetes.csv")
scala> df.select("*").show()
```



```
scala> df.select("*").show()
+---+---+---+---+---+-----+-----+---+-----+
|_c0|_c1|_c2|_c3|_c4|  _c5|  _c6|_c7|  _c8|
+---+---+---+---+---+-----+-----+---+-----+
|ntp|pgc|dbp|tsg|sin|  bmi|  dbf|age|class|
|  6|148|  72|  35|   0|33.6|0.627|  50|    N|
|  1|  85|  66|  29|   0|26.6|0.351|  31|    Y|
|  8|183|  64|   0|   0|23.3|0.672|  32|    N|
|  1|  89|  66|  23|  94|28.1|0.167|  21|    Y|
|  0|137|  40|  35|168|43.1|2.288|  33|    N|
|  5|116|  74|   0|   0|25.6|0.201|  30|    Y|
|  3|  78|  50|  32|  88|   31|0.248|  26|    N|
| 10|115|   0|   0|   0|35.3|0.134|  29|    Y|
|  2|197|  70|  45|543|30.5|0.158|  53|    N|
```

scala> df.write.csv("Cout")

Sequence File:
val data = sc.sequenceFile(inFile, classOf[Text], classOf[IntWritable]).
map{case (x, y) => (x.toString, y.get())}

val data = sc.parallelize(List(("Panda", 3), ("Kay", 6), ("Snail", 2)))
data.saveAsSequenceFile(outputFile)

Q) What is RDD? Explain the features of RDD.

RDDs are the building blocks of any Spark application. RDDs Stands for:

- **Resilient:** Fault tolerant and is capable of rebuilding data on failure
- **Distributed:** Distributed data among the multiple nodes in a cluster
- **Dataset:** Collection of partitioned data with values

There are various advantages/features of using RDD. Some of them are

1. **In-memory computation:** Basically, while storing data in RDD, data isstored in memory for as long as you want to store. It improves the performance by an order of magnitudes by keeping the data in memory.

2. **Lazy Evaluation:** Spark Lazy Evaluation means the data inside RDDs arenot evaluated on the go. Basically, only after an action triggers all the changes or the computation is performed. Therefore, it limits how much work it has to do.

3. **Fault Tolerance:** If any worker node fails, by using lineage of operations,we can re-compute the lost partition of RDD from the original one. Hence, it is possible to recover lost data easily.

4. **Immutability:** Immutability means once we create an RDD, we can notmanipulate it. Moreover, we can create a new RDD by performing any transformation. Also, we achieve consistency through immutability.

5. **Persistence:** In in-memory, we can store the frequently used RDD. Also,we can retrieve them directly from memory without going to disk. It results in the speed of the execution. Moreover, we can perform multiple operations on the same data. It is only possible by storing the data explicitly in memory by calling persist() or cache() function.

6. **Partitioning:** Basically, RDD partition the records logically. Also,distributes the data across various nodes in the cluster. Moreover, the logical divisions are only for processing and internally it has no division. Hence, it provides parallelism.

7. **Parallel:** While we talk about parallel processing, RDD processes the data parallelly over the cluster.

8. **Location-Stickiness:** To compute partitions, RDDs are capable ofdefining placement preference. Moreover, placement preference refers to information about the location of RDD. Although, the DAGScheduler places the partitions in such a way that task is close to data as much as possible. Moreover, it speeds up computation.

9. **Coarse-grained Operation:** Generally, we apply coarse-grainedtransformations to Spark RDD. It means the operation applies to the whole dataset not on the single element in the data set of RDD in Spark.

10. **No limitation:** There are no limitations to use the number of SparkRDD. We can use any no. of RDDs. Basically, the limit depends on the size of disk and memory.

Q) Define RDD. Explain the workflow of RDD. Explain Transformations and Actions on RDD.

RDDs are the building blocks of any Spark application. RDDs Stands for:

- **Resilient:** Fault tolerant and is capable of rebuilding data on failure
- **Distributed:** Distributed data among the multiple nodes in a cluster
- **Dataset:** Collection of partitioned data with valuesBasically, there are 2 ways to create

Spark RDDs:

1. Parallelized collections

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

In python:

```
nums = sc.parallelize([1, 2, 3, 4])
>>>nums.collect()
[1, 2, 3, 4]
```

In scala:

```
scala>val rd1 = spark.sparkContext.parallelize(Seq(1,3,3))
scala> rd1.collect()
res: Array[Int] = Array(1, 3, 3)
```

2. External datasets

One can create Spark RDDs, by calling a textFile method. Hence, this method takes URL of the file and reads it as a collection of lines.
Eg. lines = sc.textFile("README.txt")

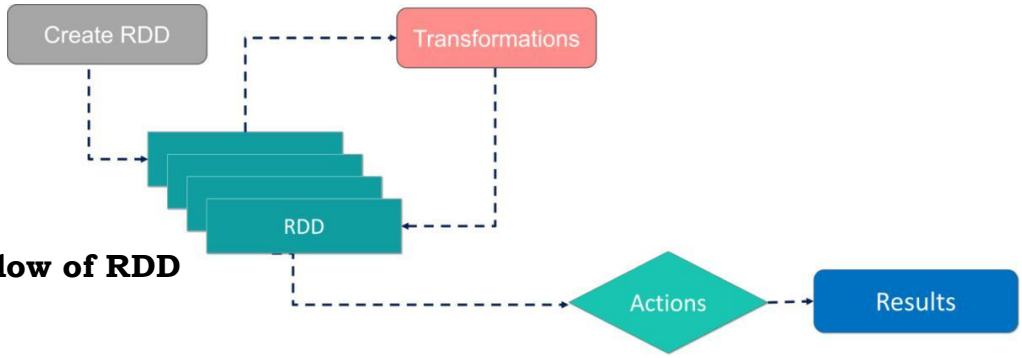


Fig. Workflow of RDD

With RDDs, you can perform two types of operations:

1. **Transformations:** They are the operations that are applied to create a newRDD.

2. **Actions:** They are applied on an RDD to instruct Apache Spark to applycomputation and pass the result back to the driver.

To summarize, every Spark program and shell session will work as follows:

1. Create some input RDDs from external data.
2. Transform them to define new RDDs using transformations like filter().
3. Ask Spark to persist() any intermediate RDDs that will need to be reused.
4. Launch actions such as count() and first() to kick off a parallel computation, which is then optimized and executed by Spark.

Table Basic RDD transformations on an RDD containing {1, 2, 3, 3}

Function name	Purpose	Example	Result
`map()`	Apply a function to each element in the RDD and return an RDD of the result.	`rdd.map(x => x + 1)`	`{2, 3, 4, 4}`
`flatMap()`	Apply a function to each element in the RDD and return an RDD of the contents of the iterators returned. Often used to extract words.	`rdd.flatMap(x => x.to(3))`	`{1, 2, 3, 2, 3, 3, 3}`
`filter()`	Return an RDD consisting of only elements that pass the condition passed to `filter()`.	`rdd.filter(x => x != 1)`	`{2, 3, 3}`
`distinct()`	Remove duplicates.	`rdd.distinct()`	`{1, 2, 3}`

Table Two-RDD transformations on RDDs containing {1, 2, 3} and {3, 4, 5}

Function name	Purpose	Example	Result
`union()`	Produce an RDD containing elements from both RDDs.	`rdd.union(other)`	`{1, 2, 3, 3, 4, 5}`
`intersection()`	RDD containing only elements found in both RDDs.	`rdd.intersection(other)`	`{3}`
`subtract()`	Remove the contents of one RDD (e.g., remove training data).	`rdd.subtract(other)`	`{1, 2}`
`cartesian()`	Cartesian product with the other RDD.	`rdd.cartesian(other)`	`{(1, 3), (1, 4), … (3,5)}`

Actions:

The most common action on basic RDDs you will likely use is reduce(), which takes a function that operates on two elements of the type in your RDD and returns a new element of the same type.

A simple example of such a function is +, which we can use to sum our RDD.

Eg.

reduce() in Python

```
>>>  sum = nums.reduce(lambda x, y: x + y)
>>>  print sum
```

Table Basic actions on an RDD containing {1, 2, 3, 3}

Function name	Purpose	Example	Result
`collect()`	Return all elements from the RDD.	`rdd.collect()`	`{1, 2, 3, 3}`
`count()`	Number of elements in the RDD.	`rdd.count()`	`4`
`countByValue()`	Number of times each element occurs in the RDD.	`rdd.countByValue()`	`{(1, 1), (2, 1), (3, 2)}`
`reduce(func)`	Combine the elements of the RDD together in parallel (e.g., `sum`).	`rdd.reduce((x, y) => x + y)`	`9`
`fold(zero)(func)`	Same as `reduce()` but with the provided zero value.	`rdd.fold(0)((x, y) => x + y)`	`9`
`aggregate(zeroValue) (seqOp, combOp)`	Similar to `reduce()` but used to return a different type.	`rdd.aggregate((0, 0)) ((x, y) => (x._1 + y, x._2 + 1), (x, y) => (x._1 + y._1, x._2 + y._2))`	`(9, 4)`
`foreach(func)`	Apply the provided function to each element of the RDD.	`rdd.foreach(func)`	Nothing
`take(num)`	Return `num` elements from the RDD.	`rdd.take(2)`	`{1, 2}`
`top(num)`	Return the top `num` elements the RDD.	`rdd.top(2)`	`{3, 3}`
`takeOrdered(num)(ordering)`	Return `num` elements based on provided ordering.	`rdd.takeOrdered(2)(myOrdering)`	`{3, 3}`

Q) Explain the difference between map() and flatmap()

The map() transformation takes in a function and applies it to each element in the RDD with the result of the function being the new value of each element in the resulting RDD.

Eg.

Python squaring the values in an RDD

```
>>>  nums = sc.parallelize([1, 2, 3, 4])
>>>  squared = nums.map(lambda x: x * x).collect()
>>>  print squared
```

[1, 4, 9, 16]

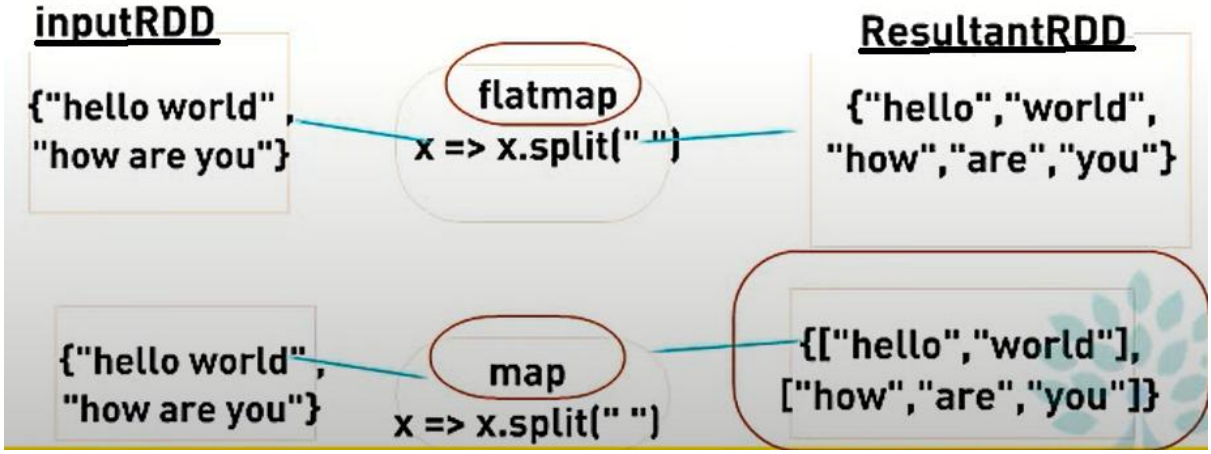


Fig.*Difference between flatMap() and map() on an RDD*

The flatMap() transformation takes in a function and applies it to each element in the RDD and return an RDD of the contents of the iterators returned i.e; flatmap returns multiple values for each element in the source RDD, Often used to extract words.

Eg.

flatMap() in Python, splitting lines into words

```
>>>  lines = sc.parallelize(["hello world", "hi"])
>>>  words = lines.flatMap(lambda line: line.split(""))
>>>  words.first()
```

'hello'

Q) Explain about Paired RDD operations.

Paired RDD is a distributed collection of data with the key-value pair.

Transformations on Pair RDDs:
Since pair RDDs contain tuples, we need to pass functions that operate on tuples rather than on individual elements.

Transformations on one pair RDD (example: {(1, 2), (3, 4), (3, 6)})

Function name	Purpose	Example	Result
`reduceByKey(func)`	Combine values with the same key.	`rdd.reduceByKey( (x, y) => x + y)`	`{(1, 2), (3, 10)}`
`groupByKey()`	Group values with the same key.	`rdd.groupByKey()`	`{(1, [2]), (3, [4, 6])}`
`combineBy Key(createCombiner, mergeValue, mergeCombiners, partitioner)`	Combine values with the same key using a different result type.	See Examples 4-12 through 4-14.	
`mapValues(func)`	Apply a function to each value of a pair RDD without changing the key.	`rdd.mapValues(x => x+1)`	`{(1, 3), (3, 5), (3, 7)}`
`flatMapValues(func)`	Apply a function that returns an iterator to each value of a pair RDD, and for each element returned, produce a key/value entry with the old key. Often used for tokenization.	`rdd.flatMapValues(x => (x to 5)`	`{(1, 2), (1, 3), (1, 4), (1, 5), (3, 4), (3, 5)}`
`keys()`	Return an RDD of just the keys.	`rdd.keys()`	`{1, 3, 3}`
`values()`	Return an RDD of just the values.	`rdd.values()`	`{2, 4, 6}`
`sortByKey()`	Return an RDD sorted by the key.	`rdd.sortByKey()`	`{(1, 2), (3, 4), (3, 6)}`

Transformations on two pair RDDs (rdd = {(1, 2), (3, 4), (3, 6)} other = {(3, 9)})
Eg.
In Python:

```
>>>rdd = sc.parallelize({(1, 2), (3, 4), (3, 6})
>>>other = sc.parallelize({(3,9)})
>>>print rdd.collect()

[(1,2),(3,4),(3,6)]
>>>print other.collect()
[(3,9)]
```

In Scala:

```
scala> val rd1 = spark.sparkContext.parallelize(Seq((1,2),(3,4),(3,6)))
scala> rd1.collect()
res: Array[(Int, Int)] = Array((1,2), (3,4), (3,6))
```

Function name	Purpose	Example	Result
`subtractByKey`	Remove elements with a key present in the other RDD.	`rdd.subtractByKey(other)`	`{(1, 2)}`
`join`	Perform an inner join between two RDDs.	`rdd.join(other)`	`{(3, (4, 9)), (3, (6, 9))}`
`rightOuterJoin`	Perform a join between two RDDs where the key must be present in the first RDD.	`rdd.rightOuterJoin(other)`	`{(3,(Some(4),9)), (3,(Some(6),9))}`
`leftOuterJoin`	Perform a join between two RDDs where the key must be present in the other RDD.	`rdd.leftOuterJoin(other)`	`{(1,(2,None)), (3, (4,Some(9))), (3, (6,Some(9)))}`
`cogroup`	Group data from both RDDs sharing the same key.	`rdd.cogroup(other)`	`{(1,([2],[])), (3, ([4, 6],[9]))}`

Actions Available on Pair RDDs
Actions on pair RDDs (example {(1, 2), (3, 4), (3, 6)})

Function	Description	Example	Result
`countByKey()`	Count the number of elements for each key.	`rdd.countByKey()`	`{(1, 1), (3, 2)}`
`collectAsMap()`	Collect the result as a map to provide easy lookup.	`rdd.collectAsMap()`	`Map{(1, 2), (3, 4), (3, 6)}`
`lookup(key)`	Return all values associated with the provided key.	`rdd.lookup(3)`	`[4, 6]`

Q) What is spark? State the advantages of using Apache spark over Hadoop MapReduce for Big data processing.

Key Features	**Apache Spark**	**HadoopMapReduce**
Ease of Programming	Easy to code	Difficult to code
Absraction	Uses RDD abstraction	No Abstraction
Speed	10–100 times faster than MapReduce	Slower
Analytics	Supports streaming, Machine Learning, complex analytics, etc.	Comprises simple Map and Reduce tasks
Suitable for	Real-time streaming	Batch processing
Complexity	Easy to write and debug	Difficult to write and debug
Processing Location	In-memory	Local disk
Developed using the laguage	Scala	Java
Supported Languages	Python,Scala,Java,R,SQL	Java,Python,Ruby,Perl,C,C++
Cost	High because of huge amount of RAM	Less cost
Security	Evolving	High compared to Spark
Coding	Less no.of lines	More no.of lines
SQL	Through Spark SQL	Through HiveQL

Q) Explain various statistical operation on RDDs.

Spark provides several descriptive statistics operations on RDDs containing numeric data.

count() Number of elements in the RDD
mean() Average of the elementsin RDD

sum() Total sum of all elements in RDD
max() Maximum valuein RDD
min() Minimum valuein RDD
variance() Variance of the elements
sampleVariance() Variance of the elements, computed for a sample
stdev() Standard deviation
sampleStdev() sample standard deviation

Eg.1.

```
>>>  nums = sc.parallelize([1, 2, 3, 4])
>>>  nums.count()
4
>>>  nums.mean()
2.5
```

Eg. 2. Removing outliers in Python

```
>>>  distanceNumerics = sc.parallelize([1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
     1000])
>>>  stats = distanceNumerics.stats()
>>>  stddev = stats.stdev()
>>>  mean = stats.mean()
>>>  reasonableDistances = distanceNumerics.filter( lambda x:
     math.fabs(x - mean) < 3 * stddev)
>>>  print reasonableDistances.collect()
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Q) Write a program for wordcount in spark.

Wordcount.py

```
importpyspark

import random

if not 'sc' in globals():
    sc = pyspark.SparkContext()
text_file = sc.textFile("/home/hadoop/Desktop/dept.txt")
counts = text_file.flatMap(lambda line: line.split(","))
\ .map(lambda word: (word, 1)) .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("/home/hadoop/Desktop/word")
```

Q) What is the need of Spark SQL? Explain how to connect to Spark SQL.

Spark SQL is easier and efficient to load and query structured data. It provides 3 capabilities:

1. It can load data from a variety of structured sources (e.g., JSON, Hive)
2. It lets you query the data using SQL, both inside a Spark program and from external tools that connect to Spark SQL through standard database connectors (JDBC/ODBC), such as business intelligence tools like Tableau.
3. When used within a Spark program, Spark SQL provides rich integration between SQL and regular Python/Java/Scala code, including the ability to join RDDs and SQL tables, expose custom functions in SQL

Spark SQL provides a special type of RDD called SchemaRDD. A SchemaRDD is an RDD of Row objects, each representing a record. A SchemaRDD also knows the schema (i.e., data fields) of its rows.

SchemaRDDs can be created from external data sources, from the results of queries, or from regular RDDs.

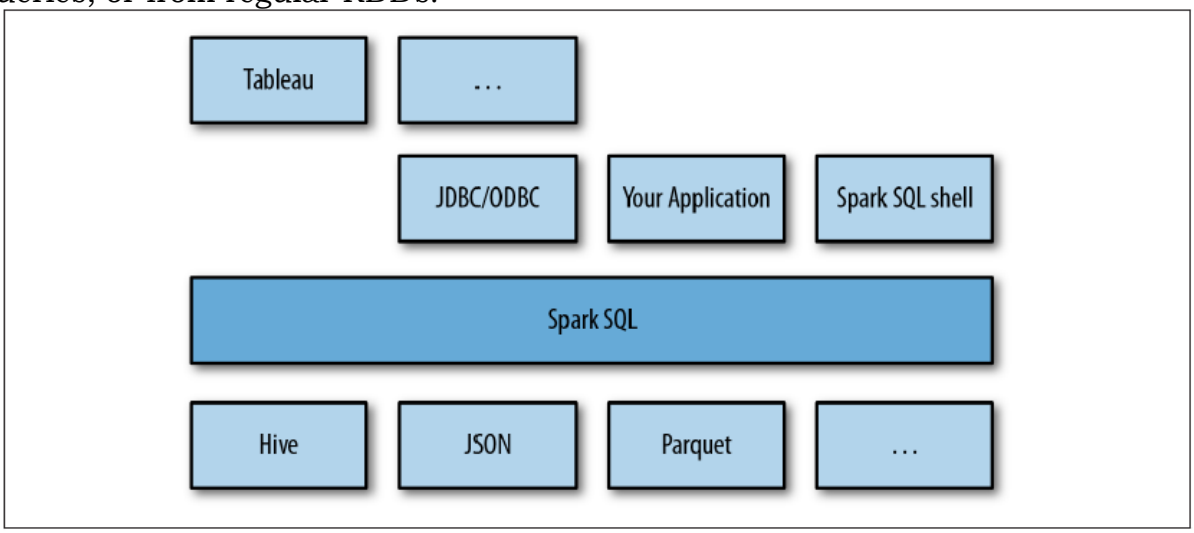


Figure 9-1. Spark SQL usage

Spark SQL can be built with or without Apache Hive, the Hadoop SQL engine. SparkSQL with Hive support allows us to access Hive tables, UDFs (user-defined functions).

When programming against Spark SQL we have two entry points depending on whether we need Hive support. The recommended entry point is the HiveContext to provide access to HiveQL and other Hive-dependent functionality. The more basic SQLContext provides a subset of the Spark SQL support that does not depend on Hive.

HiveQL is the recommended query language for working with Spark SQL. Many resources have been written on HiveQL.

To connect Spark SQL to an existing Hive installation, you must copy your hive-site.xml file to Spark's configuration directory ($SPARK_HOME/conf).

Q) Explain how to use Spark SQL in applications with an example.

We can use Spark SQL in our programs using Scala or Python or Java or R.

1. First we need to import HiveContext and SQLContext in our program. Scala users no need to import HiveContext
2. Create SQLContext
3. Read the input file
4. Register the input *schema RDD as any temporary table*
5. Write necessary SQL Queries and display the results

Code in Scala:

```
import org.apache.spark.sql.hive.HiveContext
import org.apache.spark.sql.SQLContext
//sc is not required to import or create in scala
val hiveCtx = new HiveContext(sc) // Constructing a SQL context
in Scala
val input = hiveCtx.jsonFile("iris.json")
// Register the input schema RDD as temporary table iris
input.registerTempTable("iris")
// Select records based on petalLength
val topRows = hiveCtx.sql("select petalLength,petalWidth from iris order by
petalLength limit 5")
topRows.show()
```

```
scala> topRows.show()
+-----------+----------+
|petalLength|petalWidth|
+-----------+----------+
|        1.0|       0.2|
|        1.1|       0.1|
|        1.2|       0.2|
|        1.2|       0.2|
|        1.3|       0.2|
+-----------+----------+
```

Q) Explain about SchemaRDD and their data types.

SchemaRDDs are similar to tables in a traditional database. Under the hood, a SchemaRDD is an RDD composed of Row objects with additional schema information of the types in each column.
Both loading data and executing queries return SchemaRDDs.
SchemaRDDs are also regular RDDs, so you can operate on them using existing RDD transformations like map() and filter().

Types stored by SchemaRDD:

TINYINT	'1 byte signed integer',	-128 to 127
SMALLINT	'2 byte signed integer',	-32, 768 to 32, 767
INT	'4 byte signed integer',	–2,147,483,648 to 2,147,483,647
BIGINT	'8 byte signed integer',	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
FLOAT	'Single precision floating point',	
DOUBLE	'Double precision floating point',	
DECIMAL	'Precise decimal type based on Java BigDecimal Object',	
TIMESTAMP	'YYYY-MM-DD HH:MM:SS.fffffffff" (9 decimal place precision)',	
BOOLEAN	'TRUE or FALSE boolean data type',	
STRING	'Character String data type',	
BINARY	'Data Type for Storing arbitrary	

b. Complex Data Types in Hive
In this category of Hive data types following data types are come-

- Array
- MAP
- STRUCT
- UNION

ARRAY<TINYINT> 'A collection of fields all of the same data type indexed BY an integer'

MAP<STRING,INT> 'A Collection of Key,Value Pairs where the Key is a Primitive Type and the Value can be anything. The chosen data types for the keys and values must remain the same per map'

STRUCT<first : SMALLINT, second : FLOAT, third : STRING>
'A nested complex data structure'

UNIONTYPE<INT,FLOAT,STRING>
'A Complex Data Type that can hold One of its Possible Data Types at Once'

Q) What is caching in Spark SQL.

Caching in Spark SQL works a bit differently. Since we know the types of each column, Spark is able to store the data more efficiently.

To make sure that we cache using the memory efficient representation, rather than the full objects, we should use the special hiveCtx.cacheTable("tableName") method.

You can also cache tables using HiveQL/SQL statements. To cache or uncache a table simply run CACHE TABLE tableName or UNCACHE TABLE tableName. This is most commonly used with command-line clients to the JDBC server.

Q) How to connect with JDBC/ODBC server in Spark SQL? Explain how to work with it.

Spark SQL also provides JDBC connectivity, which is useful for connecting business intelligence (BI) tools to a Spark cluster and for sharing a cluster across multiple users.

The JDBC server runs as a standalone Spark driver program that can be shared by multiple clients. Any client can cache tables in memory, query them, and so on, and the cluster resources and cached data will be shared among all of them.

Spark SQL's JDBC server corresponds to the HiveServer2 in Hive. It is also known as the "Thrift server" since it uses the Thrift communication protocol. JDBC server requires Spark be built with Hive support.

The server can be launched with **sbin/start-thriftserver.sh** in Spark directory. This script takes many of the same options as spark-submit. By default it listens on localhost:10000, but we can change these with either environment variables.

Launching the JDBC server:
./sbin/start-thriftserver.sh --master sparkMaster

Connecting to the JDBC server with Beeline:
./bin/beeline -u jdbc:hive2://localhost:10000

Many external tools can also connect to Spark SQL via its ODBC driver. The Spark SQL ODBC driver is produced by **Simba** and can be downloaded from various Spark vendors (e.g., Databricks Cloud, Datastax, and MapR). It is commonly used by business intelligence (BI) tools such as Microstrategy or Tableau.

Working with Beeline

Within the Beeline client, you can use standard HiveQL commands to create, list, and query tables.

Eg.
spark.sql("create table t1(id int, name string)")
spark.sql("insert into t1 values(1,'chp')")
spark.sql("insert into t1 values(2,'vr')")
val df = spark.sql("select * from t1")
df.show()

Q) Illustrate User-Defined Functions in Spark SQL.

User-defined functions, or UDFs, allow you to register custom functions in Python, Java, and Scala to call within SQL.

In Scala and Python, we can use the native function and lambda syntax of the language, and in Java we need only extend the appropriate UDF class.

Eg.

In Python:
hiveCtx. udf.register ("strLenPython", **lambda** x: len(x), IntegerType())
lengthSchemaRDD = hiveCtx.sql**("SELECT strLenScala('species') FROM iris where species = 'setosa'")**

In Scala:
scala> hiveCtx.udf.register ("strLenScala", (_**: String**).length)
scala> val speciesLength = hiveCtx.sql**("SELECT strLenScala(iris.species) FROM iris where species = 'setosa' limit 1")**

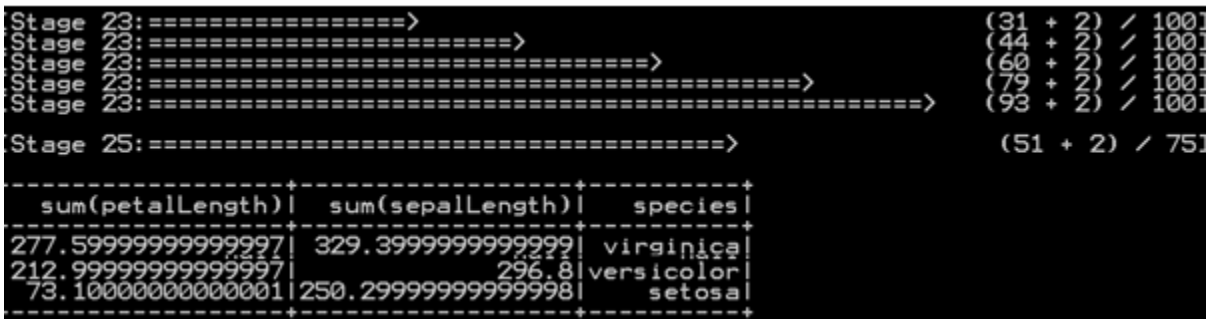
```
scala> speciesLength.show()
[Stage 1:>

+------------------------+
|UDF:strLenScala(species)|
+------------------------+
|                       6|
+------------------------+
```

Q) Give Spark SQL Performance tuning parameters.

Spark SQL is for more than just users who are familiar with SQL. Spark SQL makes it very easy to perform conditional aggregate operations, like counting the sum of multiple columns.

Eg.
scala> **val res = hiveCtx.sql("SELECT SUM(petalLength), SUM**(sepalLength), species **FROM** iris **GROUP BY species")**
scala> res.show()



When **caching data**, Spark SQL uses an in-memory columnar storage. This not only takes up less space when cached, but if our subsequent queries depend only on subsets of the data, Spark SQL minimizes the data read.

Predicate push-down allows Spark SQL to move some parts of our query "down" to the engine we are querying. If we wanted to read only certain records in Spark, the standard way to handle this would be to read in the entire dataset and then execute a filter on it.

In Spark SQL, if the underlying data store supports retrieving only subsets of the key range, or another restriction, Spark SQL is able to push the restrictions in our query down to the data store, resulting in potentially much less data being read.

Table 9-2. Performance options in Spark SQL

Option	Default	Usage
`spark.sql.codegen`	`false`	When `true`, Spark SQL will compile each query to Java bytecode on the fly. This can improve performance for large queries, but `codegen` can slow down very short queries.
`spark.sql.inMemoryColumnarStorage.compressed`	`false`	Compress the in-memory columnar storage automatically.
`spark.sql.inMemoryColumnarStorage.batchSize`	`1000`	The batch size for columnar caching. Larger values may cause out-of-memory problems
`spark.sql.parquet.compression.codec`	`snappy`	Which compression codec to use. Possible options include `uncompressed`, `snappy`, `gzip`, and `lzo`.

Enabling codegen in scala
conf.set("spark.sql.codegen", "true")

Beeline command for enabling codegen
beeline> **set** spark.**sql**.codegen=**true**;
SET spark.**sql**.codegen=**true**
spark.**sql**.codegen=**true**
Time taken: 1.196 seconds

HIVE

Q) Explain different data types in Hive.

Hive Data types are used for specifying the column/field type in Hive tables.
Mainly Hive Data Types are classified into 5 major categories, let's discuss them one by one:
a. Primitive Data Types in Hive
Primitive Data Types also divide into 3 types which are as follows:

- Numeric Data Type
- Date/Time Data Type
- String Data Type

TINYINT '1 byte signed integer', -128 to 127
SMALLINT '2 byte signed integer', -32, 768 to 32, 767
INT '4 byte signed integer', –2,147,483,648 to 2,147,483,647
BIGINT '8 byte signed integer', –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
FLOAT 'Single precision floating point',

DOUBLE 'Double precision floating point',
DECIMAL 'Precise decimal type based on Java BigDecimal Object',
TIMESTAMP 'YYYY-MM-DD HH:MM:SS.fffffffff" (9 decimal place precision)',
BOOLEAN 'TRUE or FALSE boolean data type',
STRING 'Character String data type',
BINARY 'Data Type for Storing arbitrary

b. Complex Data Types in Hive

In this category of Hive data types following data types are come-

- Array
- MAP
- STRUCT
- UNION

ARRAY<TINYINT> 'A collection of fields all of the same data type indexed BY an integer',
MAP<STRING,INT> 'A Collection of Key,Value Pairs where the Key is a Primitive Type and the Value can be anything. The chosen data types for the keys and values must remain the same per map',

STRUCT<first : SMALLINT, second : FLOAT, third : STRING>
'A nested complex data structure',
UNIONTYPE<INT,FLOAT,STRING>
'A Complex Data Type that can hold One of its Possible Data Types at Once'
eg.
t1.txt
1001,cse^1,1|2|3,A|50000,3,true
1002,cse^2,1|2,B|40000,good,true

Creating a table t1:

```
create table t1(id int,class map<string,int>,sections array<int>,hostel
struct<grade:string,fee:double>,rating uniontype<int,string>,exist boolean)
row format delimited
fields terminated by ','
collection items terminated by '|'
map keys terminated by '^'
lines terminated by '\n'
stored as textfile;
```

Q) What is Hive Query Language(HQL)? Explain various DDL and DML statements in Hive

Hive query language provides basic SQL like operations.
Basic tasks of HQL are:

1. Create and Manage tables and partitions
2. Support various relational, arithmentic and logic operations
3. Evaluate functions
4. Down load the contents of a table to a local directory or result of queries to HDFS directory.

HIVE DDL Statements:

These statements are used to build and modify the tables and other objects in the database.

1. Create/Drop/Alter Database
2. Create/Drop/truncate Table
3. Alter Table/partition/column
4. Create/Drop/Alter view
5. Create/Drop/Alter index
6. Show
7. Describe

HIVE DML Statements:

These statements are used to retrive, store, modify, delete and update data in database. The DML commands are as follows:

1. Loading files into table.
2. Inserting data into Hive tables from queries.

1. Creating and Managing Databases and Tables

faculty.txt	**dept.txt**
1,chp,10000	2@cse
2,pnr,20000	3@mca
3,kry,30000	4@cse

hive> create database chp;

OK

Time taken: 0.116 seconds

NOTE: TO CREATE DATABASE WITH COMMENTS AND DATABASE PROPERTIES

create database if not exists chp **comment** 'employee details' with **dbproperties**('creator'='praneeth');

hive> show databases;

OK

chp

default

hive> use chp;

OK

Time taken: 0.018 seconds

hive> describe database chp;

NOTE: SHOWS ONLY DB NAME, COMMENT AND DB DIRECTORY

hive> describe database extended chp;

NOTE: SHOWS DB PROPERTIES ALSO.

2. Create tables emp and dept and load data from text files on hdfs.

hdfs dfs -mkdir /chp/data
hdfs dfs -put /home/pvp/Desktop/hive_data/*.txt /chp/data

hive> create table emp(id int,name string,sal double) row format delimited fields terminated by ',';
OK
Time taken: 8.331 seconds
hive> show tables;
OK
emp

hive> create table dept(eid int,dept string) row format delimited fields terminated by '@';
OK
Time taken: 0.088 seconds

3. Loading data into the tables

hive> load data inpath '/chp/data/faculty.txt' into table emp;
hive> load data inpath '/chp/data/dept.txt' into table dept;

4. Retreiving data from the tables.

hive> select * from emp;
OK
1 chp 10000.0
2 pnr 20000.0
3 kry 30000.0
Time taken: 0.379 seconds, Fetched: 3 row(s)
hive> select * from dept;
OK
2 cse
3 mca
4 cse
Time taken: 0.133 seconds, Fetched: 4 row(s)

Q) Briefly explain joins in Hive.

JOIN is a clause that is used for combining specific fields from two tables by using values common to each one. It is used to combine records from two or more tables in the database. It is more or less similar to SQL JOIN.

Inner join: The HiveQL INNER JOIN returns all the rows which are common in both the tables.

hive> select * from emp join dept on (emp.id=dept.eid);
2 pnr 20000.0 2 cse
3 kry 30000.0 3 mca

Left outer join: A LEFT JOIN returns all the values from the left table, plus the matched values from the right table, or NULL in case of no matching JOIN predicate.

hive> select * from emp left outer join dept on (emp.id=dept.eid);
1 chp 10000.0 NULL NULL
2 pnr 20000.0 2 cse
3 kry 30000.0 3 mca

Right Outer Join: The HiveQL RIGHT OUTER JOIN returns all the rows from the right table, even if there are no matches in the left table. If the ON clause matches 0 (zero) records in the left table, the JOIN still returns a row in the result, but with NULL in each column from the left table.

hive> select * from emp right outer join dept on (emp.id=dept.eid);
2 pnr 20000.0 2 cse
3 kry 30000.0 3 mca
NULL NULL NULL 4 cse

Full outer join: The HiveQL FULL OUTER JOIN combines the records of both the left and the right outer tables that fulfil the JOIN condition. The joined table contains either all the records from both the tables, or fills in NULL values for missing matches on either side.

hive> select * from emp full outer join dept on (emp.id=dept.eid);
1 chp 10000.0 NULL NULL
2 pnr 20000.0 2 cse
3 kry 30000.0 3 mca
NULL NULL NULL 4 cse

NOTE: Inner Join In Spark SQL using scala:

val i = spark.sql("select * from emp join dept on (emp.id=dept.eid)")
val.show() //display the result of the join.

```
scala> i.show()
+---+----+-------+---+-----+
| id|name|    sal|eid|dname|
+---+----+-------+---+-----+
|  3| kry|30000.0|  3|  mca|
|  2| pnr|20000.0|  2|  cse|
+---+----+-------+---+-----+
```

Q) Briefly explain about Views in Hive.

A view is purely a logical construct (an alias for a query) with no physical data behind it.
So altering a view only involves changes to metadata in the metastore database, not any data files in HDFS.
When a query becomes long or complicated, a view may be used to hide the complexity by dividing the query into smaller, more manageable pieces; similar to writing a function in a programming language or the concept of layered design in software.

1. **Create a view from emp table with the fields id and name.**

hive> create view emp_view as select id,name from emp;
hive> select * from emp_view;
1 chp
2 pnr
3 kry

2. **Find no.of employees using above view.**

hive> select count(*) from emp_view;
3

3. **Drop view.**

hive> drop view emp_view;

Q) Explain about various functions in Hive.
string functions:

1. **Display employee names in uppercase**

hive> select upper(name) from emp;
CHP
PNR
KRY

2. **Display employee names from 2nd character**

hive> select substr(name,2) from emp;
hp
nr
ry

3. **Concatenate emp id and name**

hive> select concat(id,name) from emp;
1chp
2pnr
3kry

Math Functions:

1. **Find the salaries of the employees by applying ceil function.**

hive> select ceil(sal) from emp;

10000
20000
30000

2. **Find the square root of the emp salaries.**

hive> select sqrt(sal) from emp;
100.0
141.4213562373095
173.20508075688772

3. Find the length of the emp names.

hive> select name,length(name) from emp;
chp 3
pnr 3
kry 3

Aggregate functions:

1. **Find no.of employees in the table emp.**

hive> select count(*) from emp;
3

2. **Find the salary of all the employees.**

hive> select sum(sal) from emp;
60000.0

3. **Find the average salary of the employees.**

hive> select avg(sal) from emp;
20000.0

4. **Find the minimum salary of all the employees.**

hive> select min(sal) from emp;
10000.0

5. **Find the maximum salary of all the employees.**

hive> select max(sal) from emp;
30000.0

Queries:

1. **Display different department in dept**

hive> select distinct(dept) from dept;
cse
mca

2. **Find the employees who earns 10000**

hive> select name from emp where sal=10000;
chp

3. **Find the employees who earns greater than 20000**

hive> select name from emp where sal>=20000;
pnr

kry

4. Find the employee id whose name is either chp or kry

hive> select id from emp where name='chp' or name='kry';
1
3

5. Find the employee name whose dept is either cse or mca.

select emp.name from emp join dept on(emp.id=dept.eid) where dept.dept='cse' or dept='mca';
pnr
kry

(or)

select emp.name from emp left outer join dept on(emp.id=dept.eid) where dept.dept='cse' or dept.dept='mca';

pnr
kry

6. Find first 2 records in dept

hive> select * from dept limit 2;
OK
2 cse
3 mca

7. Find the no.of employees in each department.

hive> select dept,count(*) from dept group by dept;
cse 2
mca 1

8. Find the no.of employees in dept cse.

hive> select dept,count(*) from dept group by dept having dept='cse';
cse 2

9. Find the name of the employee who is earning minimum salary.

hive> select name,sal from emp order by sal limit 1;
chp 10000.0

10. Find the name of the employee who is earning maximum salary.

hive> select name,sal from emp order by sal desc limit 1;
kry 30000.0

Q) Explain about Index in Hive.

An Index acts as a reference to the records. Instead of searching all the records, we can refer to the index to search for a particular record.

In a Hive table, there are many numbers of rows and columns. If we want to perform queries only on some columns without indexing, it will take large amount of time because queries will be executed on all the columns present in the table.

Indexes are maintained in a separate table in Hive so that it won't affect the data inside the table, which contains the data.

Indexes are advised to build on the columns on which you frequently perform operations.

Building more number of indexes also degrade the performance of your query.

Types of Indexes in Hive

- Compact Indexing
- Bitmap Indexing

Differences between Compact and Bitmap Indexing

Compact indexing stores the pair of indexed column's value and its blockid.
Bitmap indexing stores the combination of indexed column value and list of rows as a bitmap.

Creating compact index:

Syntax:

hive> create index index_name on table table_name(columns,...) as 'org.apache.hadoop.hive.ql.index.compact.CompactIndexHandler' with deferred rebuild;

Here, in the place of index_name we can give any name of our choice, which will be the table's INDEX NAME.

•In the ON TABLE line, we can give the table_name for which we are creating the index and the names of the columns in brackets for which the indexes are to be created. We should specify the columns which are available only in the table.

•The org.apache.hadoop.hive.ql.index.compact.CompactIndexHandler' line specifies that a built in CompactIndexHandler will act on the created index, which means we are creating a compact index for the table.

•The WITH DEFERRED REBUILD statement should be present in the created index because we need to alter the index in later stages using this statement.

Eg.
hive>create index emp_index on table emp(name,sal) as 'org.apache.hadoop.hive.ql.index.compact.CompactIndexHandler' with deferred rebuild;

creating **bitmap index**:
Syntax:
create index index_name on table table_name(columns,...) as 'bitmap' with deferred rebuild;
eg.
hive> create index dept_index on table dept(eid) as 'bitmap' with deferred rebuild;

Q) Find different indices on table emp and dept
hive> show formatted index on emp;

idx_name tab_name col_names idx_tab_name idx_type
emp_index emp name, sal default__emp_emp_index__ compact

hive>show formatted index on dept;

idx_name tab_name col_names idx_tab_name idx_type
dept_index dept eid default__dept_dept_index__ bitmap

1) Update index emp_index
hive> alter index emp_index on emp rebuild;
2) Drop index emp_index
hive>drop index if exists emp_index on emp;

Q) Explain Partitioning in Hive.

In Hive, the query reads the entire dataset even though a where clause filter is specified on a particular column. This becomes a bottlenect in most of the MapReduce jobs as it involves huge degree of I/O.

So it is necessary to reduce I/O required by the MapReduce job to imporve the performance of the query. A very common method to reduce I/O is data partioning.

Partitions split the larger dataset into more meaningful chunks. Hive provides two kinds of partions.

Static partition: Static partitions comprise columns whose values are known at compile time.

Eg.

1) Create a partition table.

hive> create table std_partition(sid int) partitioned by (branch string) row format delimited fields terminated by ',' stored as textfile;

std1.txt
1001
1002

2) Load data into std_partition from st1.txt and partitioned column branch as cse.

hive> load data local inpath 'home/pvp/Desktop/hive_data/std1.txt' into table std_partition partition(branch='cse');
hive> select * from std_partition;
1001 cse
1002 cse
std2.txt
2001
2002

Q) Loading data into std_partition from std2.txt and partitioned column branch as mca.

hive> load data local inpath '/home/chp/Desktop/hive_data/std2.txt' into table std_partition partition(branch='mca');
hive> select * from std_partition;
1001 cse
1002 cse
2001 mca
2002 mca

Dynamic partitioning: Dynamic partition have columns whose values are known only at Execution time.
By default the dynamic partitioning will be off. We can enable it by using the following commands in hive.
hive> set hive.exec.dynamic.partition=true;
hive> set hive.exec.dynamic.partition.mode=nonstrict;
hive> set hive.exec.max.dynamic.partitions.pernode=450;

1) Create a partition table.

```
hive> create table dept_partition(id int) partitioned by (branch string);
```

2) Describe dept_partition.

```
hive> describe formatted dept_partition;
# col_name data_type
id int None
# Partition Information
# col_name data_type
branch string None
```

3) Load data into dept_partition from dept table.

```
hive>insert into table dept_partition partition(branch) select * from dept;
hive> select * from dept_partition;
OK
2 cse
4 cse
3 mca
```

4) Drop partitioned table dept_partition.

```
hive> alter table dept_partition drop partition(branch='cse');
Dropping the partition branch=cse
OK
Time taken: 1.737 seconds
hive> select * from dept_partition;
OK
3 mca
```