Institute for Advanced Computing And Software Development (IACSD) Akurdi, Pune

Big Data Technologies

Dr. D.Y. Patil Educational Complex, Sector 29, Behind Akurdi Railway Station,

Nigdi Pradhikaran, Akurdi, Pune - 411044.

BIG DATA TECHNOLOGIES

What is Big data?

Big data is a field that treats ways to analyze, systematically extract information from, or otherwise deal with data sets that are too large or complex to be dealt with by traditional data-processing application software.

popular definition of big data is data characterized by three Vs: volume, velocity, and variety. Where volume means richness of data,*Velocity* means that data is generated at a fast rate. *Variety* refers to the fact that data can be unstructured, semi-structured, or multi-structured.Big data also called as data that captures more nuances about the entity that it represents.

Difference in traditional relational database (RDBMS) table and BIg Data:

If this data were stored in a relational database table, it would have thousands of columns and rows. But big data is in the form of volume, velocity, and variety.And also Standard relational databases could not easily handle big data.

MPP (massively parallel processing) :

Technology for processing and analyzing large datasets has been extensively researched and available in the form of proprietary commercial products for a long time. MPP databases use a "shared nothing" architecture, where data is stored and processed across a cluster of nodes. Each node comes with its own set of CPUs, memory, and disks. They communicate via a network interconnect. Data is partitioned across a cluster of nodes. There is no contention among the nodes, so they can all process data in parallel. But proprietary MPP products are expensive. Not everybody can afford them.

WHat is Hadoop?

- It is open source system to process large data sets across cluster of commoditySevres hardware.
- It provides simple frame work for large scale data processing
- It is suitable for the batch processing and Extract transform load for large scaledata.

Advantages of Hadoop?

- cheaper to use cluster of commodity servers for storing data and processing.
- Cheaper fault tolerance
- Moving code from one computer to another computer is efficient and faster.
- Writing a distributed application can be made easy by separating core data processing logic from distributed computing logic.

Key Conceptual Components of Hadoop are

- A cluster Manager(YARN)
- A distributed compute engine(MAPREDUCE)
- A distributed file system(HDFS)

Hadoop distributed file system(HDFS):

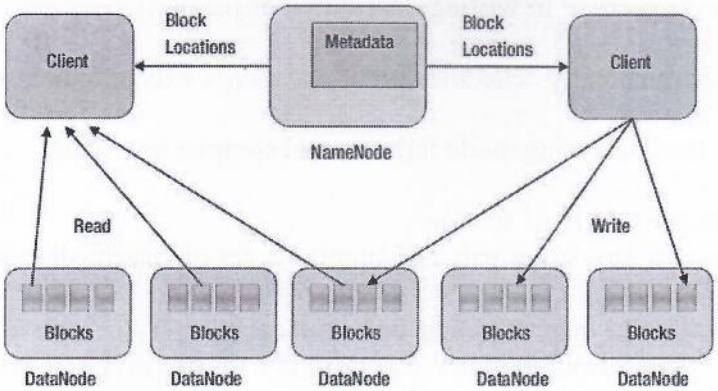


Fig1: HDFS Architecture

- This file system stores data file across a cluster of commodity server and providesfast access to big files and large data sets.
- It has blocked structured file system
- It splits a file into fixed-size blocks, also known as *partitions* or *splits*.(default Block size 128MB
- HDFS application can parallelize file-level read and write operations, making it much faster to read or write a large HDFS file distributed across a bunch of disks on different computers than reading or writing a large file stored on a single disk.
- HDFS replicates each file block on multiple machines. The default replication factor is 3.
- A HDFS cluster consists of two types of nodes: 1NameNode 2. DataNode

1. Name Node:

   It manages the file system namespace and stores all the metadata for a file. To provide fast access to the metadata, a NameNode stores all the metadata inmemory. NameNode periodically receives two types of messages from the DataNodes in an HDFS cluster. One is called Heartbeat and the other is called Blockreport.

2. Data Node:

   A DataNode stores the actual file content in the form of file blocks.A DataNode sends a heartbeat message to inform the NameNode that it is functioning properly. A Blockreport contains a list of all the data blocks on a DataNode.

- Working of HDFS:

Read files: When a client application wants to read a file, it first contacts a NameNode. The NameNode responds with the locations of all the blocks that comprise that file. A block location identifies the DataNode that holds data for that file block. A client then directly sends a read request to the DataNodes for each file block. A NameNode is not involved in the actual data transfer from a DataNode to aclient.

Write File:when a client application wants to write data to an HDFS file, itfirst contacts the NameNode and asks it to create a new entry in the HDFS namespace. The NameNode checks whether a file with the same name alreadyexists and whether the client has permissions to create a new file.

MapReduce:

- MapReduce is a distributed compute engine
- MapReduce provides a computing framework for processing large datasets inparallel across a cluster of computers.
- It abstracts cluster computing and provides higher-level constructs for writing

distributed data processing applications

- It enables programmers with no expertise in writing distributed or parallel applications to write applications
- The MapReduce framework automatically schedules an application's executionacross a set of machines in a cluster
- Mapreduce is responssible for load balancing, node failures, and complexinternode communication.
- Two building blocks of the MapReduce:1. Map 2. Reduce
- The map function takes as input a key-value pair and outputs a set of intermediate key-value pairs.
- The MapReduce framework calls the map function once for each key-value pair in the input dataset. Next, it sorts the output from the map functions and groups all intermediate values associated with the same intermediate key. It then passes them as input to the reduce function.
- The reduce function aggregates those values and outputs the aggregated value along with the intermediate key that it received as its input.
- **Hive is used to write mapReduce application because it is simple as SQL.**Hive provides a SQL-like query language called Hive Query Language (HiveQL) for processing and analyzing data stored in any Hadoop-compatible storage system. Hive is a data warehouse software.
- MapReduce is one of many programming models available for processing large data sets in Hadoop.
- Hadoop framework efficiently maintains task parallelization, job scheduling, resource allocation and data distribution in the backend, the MapReduce framework simply has two major components, **a mapper and a reducer**, for data analysis.
- A mapper maps every key/value record in the dataset by arbitrary intermediate keys and a reducer generates final key/value pairs by applying computations on the aggregated pairs.
- The strength of MapReduce framework lies in running such simple but powerful functions with Hadoop's automatic parallelization, distribution of large-scale computations and fault tolerance features using commodity hardware.
- The top-level unit of each MapReduce task is a job. A job has several mappers and reducers allocated by the underlying scheduler depending on various factors including the size of input and available physical resources. The developer, with a minimum knowledge of a distributed system, simply needs to write Map and Reduce functions which are available as Hadoop APIs in various programming languages, to take advantage of the framework. The MapReduce model can be applied to various applications including distributed grep, graph problems, inverted index and distributed sort. Figure 1 describes a workflow of a common MapReduce job.International Journal of Computer Science & Information Tec.

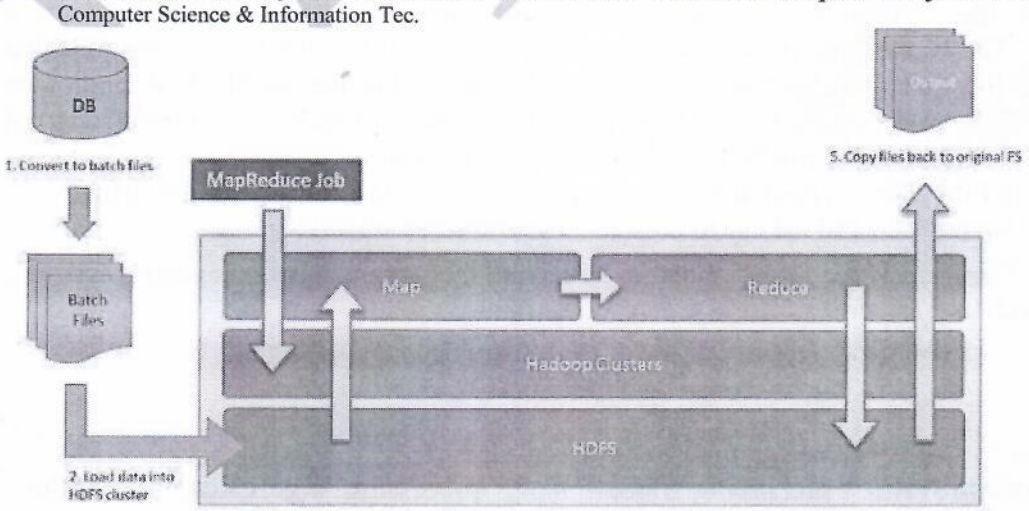


Figure 1. A workflow of typical MapReduce job.

. The Input data files stored in HDFS are split into M pieces of typically 64MB per piece and distributed across the cluster. Once a MapReduce job is submitted to the Hadoop system, several map and reduce tasks are generated and each idle container is assigned either a map task or a reduce task. A container who is assigned a map task loads the contents of the corresponding input split and invokes MAP method once for each record. Optionally on the user's request, SETUP and CLOSE methods may run prior to the first or after the last MAP method call respectively. Upon each MAP method call, it passes key and value variables to EMIT method, which then pairs are temporarily stored in a circular in-memory output buffer along with corresponding metadata.

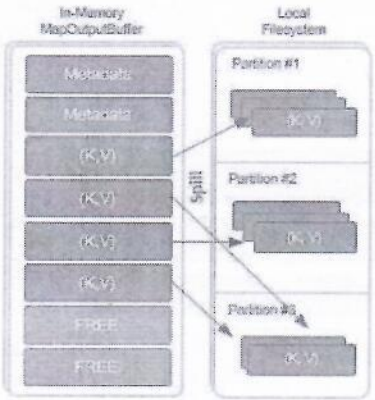


Figure 2. Circular map output buffer.

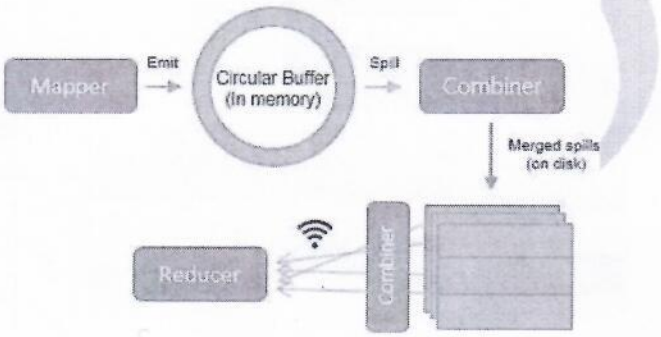


Figure 3. MapReduce shuffle phase.

Figure 2 describes a structure of a circular map output buffer. Once the contents of a buffer reaches certain threshold size (80% by default), all key/value pairs are partitioned based on their keys and finally spilled to local disk as a single spill file perbuffer. The number of partitions is equal to the total number of reduce tasks allocatedfor the job. Combiners, which are mini reduce tasks that combine intermediate results.may occasionally run on each partition prior to disk spills. Once all records have beenprocessed, spill files of a task are merged as a single partitioned output file. Then each

partition is transferred to the corresponding reducer across the network. This stage ofthe task is referred to as the shuffle phase.

A container in Hadoop is a collection of physical resources allocated by the ResourceManager upon job submission. The number of allocated containers varies by the required resources for the submitted job. RMContainerAllocator class is responsible for allocating either map or reduce tasks to containers. In our system, upon assigning a map task to a container, each container establishes a connection to the local Redis cache and updates the total number of allocated map tasks within the same node under a configured key. Each map task also updates its status on task completion in the local cache allowing other map tasks in the same node to be aware of overall job status. Each map task compares the total number of map tasks to the completed map tasks stored in the cache to verify if it is the last mapper running in thenode.

Instead of storing intermediate results in an isolated associative array, they are stored in the local Redis cache. For memory efficiency, each intermediate key-value pair is stored under one of many hash buckets.

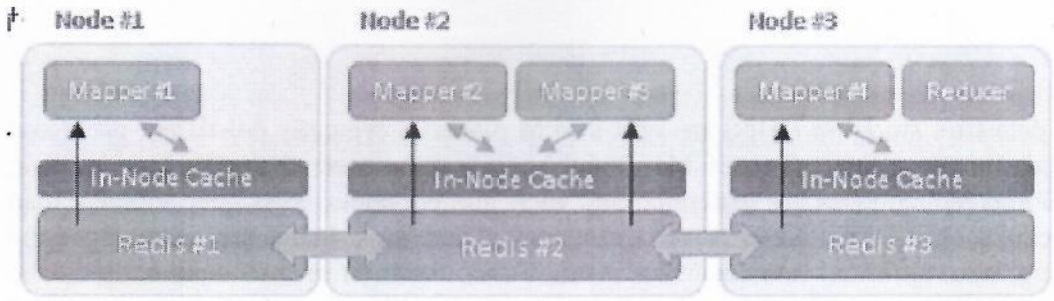


Figure 4. System overview.

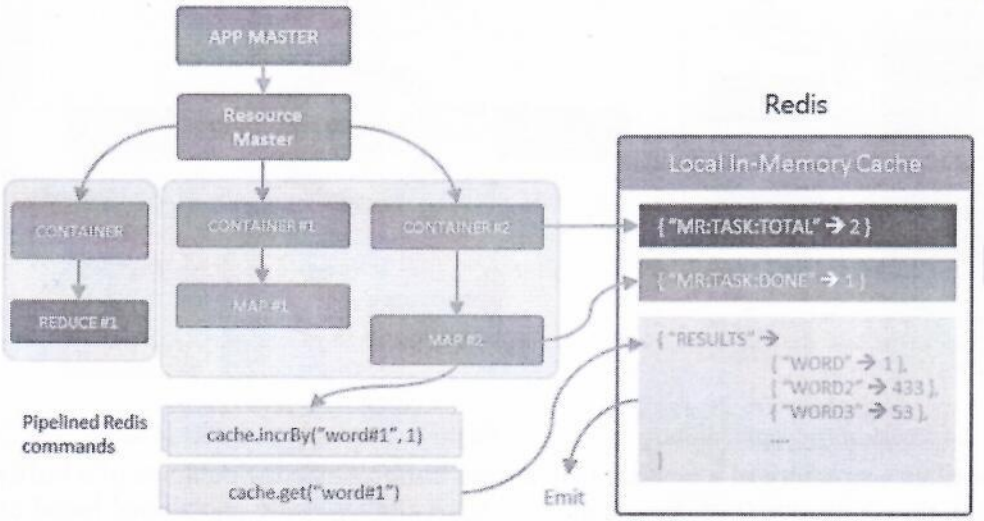


Figure 5. System workflow.

Figure 4 and 5 shows an overview and a workflow of our systemrespectively

Big Data and storage model?

Data and storage models are the basis for big data ecosystem stacks. While storagemodel captures the physical aspects and features for data storage, data model captures the logical representation and structures for data processing and management.

Data Serialization:

The process of converting data in memory to a format in which it can be stored on disk or sent over a network is called *serialization*. Data serialized text formats are CSV, JSON, XML. Data serialized is also in binary format which compact and not human readable .Binary Data formats are **Avro,Thrift, Protocol buffer.**

The reverse process of reading data from disk or network into memory is called *deserialization.*

Avro: Avro uses a self-describing binary format. When data is serialized using Avro, schema is stored along with data. Therefore, an Avro file can be later read by any application.Avro automatically handles field addition and removal, and forward and backward compatibility—all without any awareness by an application.

Thrift: Thrift is a language-independent data serialization framework.Thrift providesa code-generation tool and a set of libraries for serializing data and transmitting it across a network.

Protocol Buffers: Just like Thrift and Avro, it is language neutral developed by Google.It provides a compiler and a set of libraries that a developer can use to serialize data.

SequenceFile: SequenceFile is a binary flat file format for storing key-value pairs.It is commonly used in Hadoop as an input and output file format. MapReduce also uses SequenceFiles to store the temporary output from map functions.

Columnar Storage:

Columnar storage is more efficient than row-oriented storage for analytic applications. It enables faster analytics and requires less disk space.row-oriented storage is that data cannot be efficiently compressed.

columnar file formats are RCfile,ORC and Parquet.

RCFile: Record Columnar File format is used in HDFS for storing Hive tables. RCFile first splits a table into row groups, and then stores each row group in columnarformat. The row groups are distributed across a cluster.

ORC:RCFile first splits a table into row groups, and then stores each row group incolumnar format. The row groups are distributed across a cluster.

Advantages of ORC:1.stores row indexes 2. provides better compression 3.Also allows generic compression.4.The ORC file format not only stores data efficiently,but also allows efficient queries

Parquet:. It can be used with any data processing framework, including Hadoop MapReduce and Spark.It was designed to support complex nested data structures.itnot only supports a variety of data encodings and compression techniques, but alsoallows compression schemes to be specified on a per-column basis.

Messaging Systems

Data usually flows from one application to another. It is produced by one applicationand used by one or more other applications. A simple way to send data from one application to another is to connect them to each other directly. coupling between

producers and consumers requires them to be run at the same time or to implement acomplex buffering mechanism. Therefore, direct connections between producers and consumers does not scale.

A flexible and scalable solution is to use a message broker or messaging system. Messaging systems in the big data applications are kfka and zeroMQ.

KAFKA

Kafka is a distributed messaging system or message broker. it is a distributed, partitioned, replicated commit log service, which can be used as a publish-subscribe messaging system. A single broker can handle several hundredmegabytes of reads and writes per second from thousands of applications.

- Kafka runs as a cluster of nodes, each of which is called a *broker*.
- Messages sent through Kafka are categorized into *topics*.
- An application that publishes messages to a Kafka topic is called a *producer*.
- A *consumer* is an application that subscribes to a Kafka topic and processes messages.

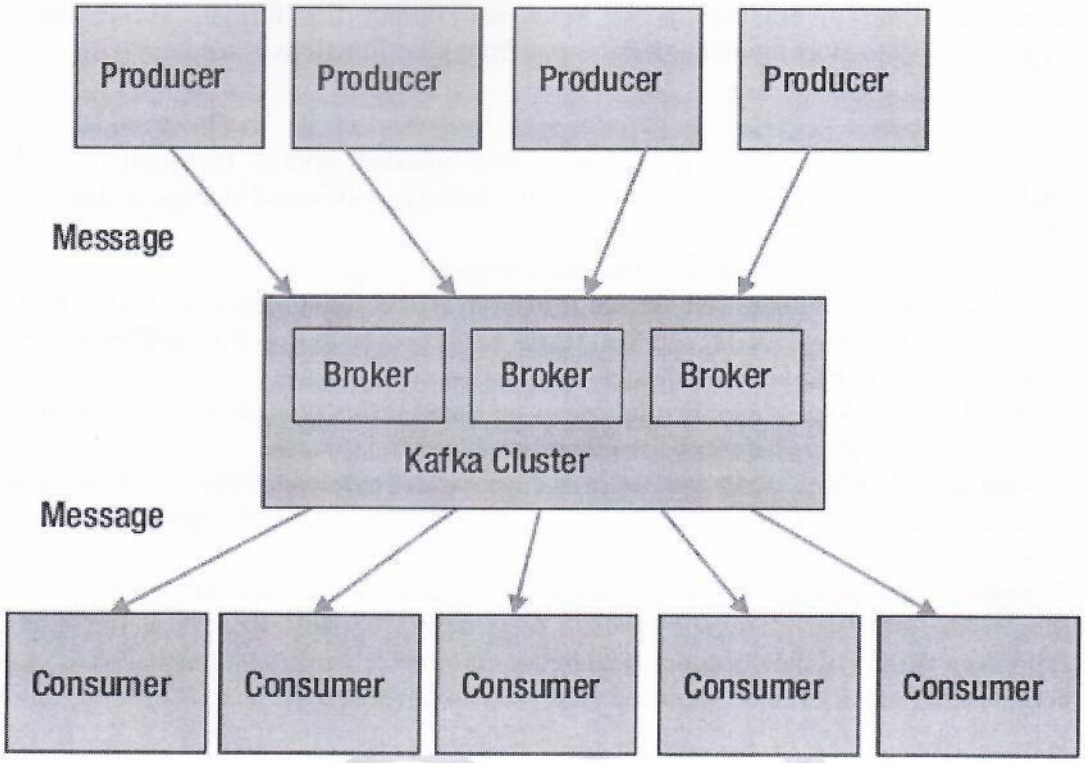


Fig 2. Kafka Architecture

i. Kafka splits a topic into partitions. Each partition is an ordered immutable sequence of messages.
ii. Newmessages are appended to a partition.
iii. Each message in a partition is assigned a unique sequential identifier called *offset*.Partitions are distributed across the nodes in a Kafka cluster.
iv. In addition, they are replicated for fault tolerance.
v. Partitioning of topics helps with scalability and parallelism.
vi. A topic need not fit on a single machine. It can grow to any size. Growth in topicsize can be handled by adding more nodes to a Kafka cluster.

Kafka supports both queuing and publish-subscribe messaging models using an abstraction called *consumer group*. Each message published to a topic is delivered to a single consumer within each subscribing consumer group. Thus, if all the consumers

subscribing to a topic belong to the same consumer group, then Kafka acts as a queuing messaging system, where each message is delivered to only one consumer. On the other hand, if each consumer subscribing a topic belongs to a different consumer group, then Kafka acts as a publish-scribe messaging system, where each message is broadcast to all consumers subscribing a topic.If each consumer subscribing a topic belongs to a different consumer group, then Kafka acts as a publish-scribe messaging system, where each message is broadcast to all consumers subscribing a topic.

HBase

1. HBase is also a distributed, scalable, and fault-tolerant NoSQL data store designed for storing large datasets.
2. It runs on top of HDFS. It has similar characteristics as Cassandra, since both are inspired by
   Bigtable, a data store invented by Google.
3. HBase stores data in tables. A table consists of rows. A row consists of column families. A column
   family consists of versioned columns.

4. A table and column in HBase are very different from a table and column in a relational database.
5. An HBase table is essentially a sparse, distributed, persistent, multidimensional, sorted . Map.
6. Map is a data structure supported by most programming languages. It is a container for storing key-value pairs.
7. It is a very efficient data structure for looking up values by keys. Generally, the order of keys is not defined and an application does not care about the order since it gives a key to the Map and gets back a value for that key.
8. Note that the Map data structure should not be confused with the map function in Hadoop MapReduce.
9. The **map function** is a functional language concept for transforming data.
10. The Map data structure is called by different names in different programming languages. For example, in PHP, it is called an *associative array*. In Python, it is known as a *dictionary*. Ruby calls it Hash. Java and Scala call it Map.An HBase table is **a sorted multi-dimensional or multi-level Map**. The first level key is the **row key**, which allows an application to quickly read a row from billions of rows. The second level key is the column family. The third-level key is the column name, also known as a ***column qualifier.***

DATA MIGRATION FROM HIVE TO HBASE

Why you want Load HBase Table from Apache Hive?

Apache Hive and HDFS are generally write-once and read-many systems. Data is inserted or appended to a file which has table on top of it. Generally, you **cannot update or overwrite Hive table without deleting the whole file and writing it again with the updated data set**. To overcome this issue, we will send Hive tabledata to HBase that require update values.

It includes an optional library for interacting with HBase. This is where the bridge layer between the two systems is implemented. The primary interface you use when accessing HBase from Hive queries is called the BaseStorageHandler. You can also

interact with HBase tables directly via Input and Output formats, but the handler issimpler and works for most uses.

Table creation in HIVE:

```
CREATE TABLE hive_employee_bank2(
id INT, name STRING, type STRING, balance INT
)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'WITH
SERDEPROPERTIES ("hbase.columns.mapping" =
":key,cf1:name,cf2:type,cf2:balance")
TBLPROPERTIES ("hbase.table.name" = "hbase_employee_bank2");

show tables;
```

Use the HBaseStorageHandler to register HBase tables with the Hive metastore

The hbase.columns.mapping property is mandatory. The hbase.table.name property isoptional.

For integrating HBase with Hive, Storage Handlers in Hive is used.

Storage Handlers are a combination of InputFormat, OutputFormat, SerDe, andspecific code that Hive uses to identify an external entity as a Hive table

Lets check if the table got created in HBase:

list

describe 'hbase_employee_bank2'

Loading the data:

We can not directly load data into hbase table "hive_employee_bank" with load datainpath hive command. We have to copy data into it from another Hive table. Lets create another hive table with the same schema as hive_employee_bank

```
create table bank(
id INT, name STRING, type STRING, balance INT
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
INSERT INTO TABLE hive_employee_bank SELECT * FROM bank;
```

Lets check it in HBase:

scan 'hbase_employee_bank2'

Apache Airflow

1. How will you describe Airflow?

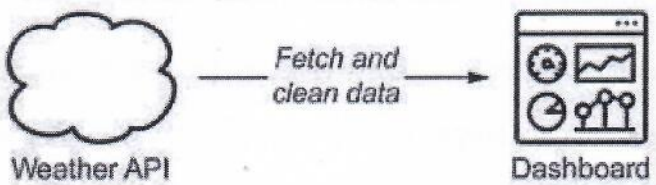
Apache Airflow is referred to as an open-source platform that is used for workflow management. This one is a data transformation pipeline Extract, Transform, Load (ETL) workflow orchestration tool. It initiated its operations back in October 2014 at Airbnb. At that time, it offered a solution to manage the increasingly complicated workflows of a company. This Airflow tool allowed them to programmatically write, schedule and regulate the workflows through an inbuilt Airflow user interface.

Introducing data pipelines

Data pipelines generally consist of several tasks or actions that need to be executedto achieve the desired result. For example, say we want to build a small weather dashboard that tells us what the weather will be like in the coming week (figure 1.1). To implement this live weather dashboard, we need to perform something likethe following steps:

- Fetch weather forecast data from a weather API.
- Clean or otherwise transform the fetched data (e.g., converting temperatures from Fahrenheit toCelsius or vice versa), so that the data suits our purpose.
- Push the transformed data to the weather dashboard.

Figure 1.1 Overview of the weather dashboard use case, in which weather data is fetched from an external API and fed into a dynamic dashboard



2. What are the problems resolved by Airflow?

Some of the issues and problems resolved by Airflow include:

- Maintaining an audit trail of every completed task
- Scalable in nature
- Creating and maintaining a relationship between tasks with ease
- Comes with a UI that can track and monitor the execution of the workflow and more.

3. What are some of the features of Apache Airflow?

Some of the features of Apache Airflow include:

- It helps schedule all the jobs and their historical status
- Helps in supporting executions through web UI and CRUD operations on DAG
- Helps view Directed Acyclic Graphs and the relation dependencies

4. How does Apache Airflow act as a Solution?

Airflow solves a variety of problems, such as:

- **Failures:** This tool assists in retrying in case there is a failure.
- **Monitoring:** It helps in checking if the status has been succeeded or failed.
- **Dependency:** There are two different types of dependencies, such as:
    - Data Dependencies that assist in upstreaming the data
    - Execution Dependencies that assist in deploying all the new changes
- **Scalability:** It helps centralize the scheduler
- **Deployment:** It is useful in deploying changes with ease
- **Processing Historical Data:** It is effective in backfilling historical data

5. Define the basic concepts in Airflow.

Airflow has four basic concepts, such as:

- **DAG:** It acts as the order's description that is used for work
- **Task Instance:** It is a task that is assigned to a DAG
- **Operator:** This one is a Template that carries out the work
- **Task:** It is a parameterized instance

6. Define integrations of the Airflow.

Some of the integrations that you'll find in Airflow include:

- Apache Plg
- Amazon EMR
- Kubernetes
- Amazon S3
- AWS Glue
- Hadoop
- Azure Data Lake

7. What do you know about the command line?

The command line is used to run Apache Airflow. There are some significant commands that everybody should know, such as:

- Airflow run is used for running a task
- Airflow show DAG is used for showcasing tasks and their dependencies
- Airflow task is used for debugging tasks
- Airflow Webserver is used for beginning the GUI
- Airflow backfill is used for running a specific part of DAG

8. How would you create a new DAG?

There are two different methods to create a new DAG, such as:

- By writing a Python code
- By testing the code

9. What do you mean by Xcoms?

Cross Communication (XComs) is a mechanism that allows tasks to talk to one another. The default tasks get isolated and can run on varying machines. They can be comprehended by a Key and by dag_id and task_id

10. Define Jinja Templates.

Jinja templates assist by offering pipeline authors that contain a specific set of inbuilt Macros and Parameters. Normally, it's a template that contains Expressions and Variables.

1. Explain the design of workflow in Airflow.

To design workflow in this tool, a Directed Acyclic Graph (DAG) is used. When creating a workflow, you must contemplate how it could be divided into varying tasks that can be independent. And then, the tasks are combined into a graph to create a logical whole.

The overall, comprehensive logic of the workflow is dependent on the graph's shape. An Airflow DAG can come with multiple branches, and you can select the ones to follow and the ones to skip during the execution of the workflow

Also:

- Airflow can be halted, completed, and can run workflows by resuming from the last unfinished task.
- It is crucial to keep in mind that Airflow operators can run multiple times when designing. Every task should be independent and capable of being performed several times without leading to any unintentional consequences.

2. What do you know about Airflow Architecture and its components?

There are four primary Airflow components, such as:

- **Web Server**
  This one is an Airflow UI that is developed on the Flask and offers an overview of the complete health of a variety of DAGs. Furthermore, it also helps visualize an array of components and states of each DAG. For the Airflow setup, the web server also lets you manage different configurations, roles, and users.
- **Scheduler**
  Every 'n' second, the scheduler navigates through the DAGs and helps schedule the tasks that have to be executed. The scheduler has an internal component as well, which is known as the Executor. Just as the name

suggests, it helps execute the tasks, and the scheduler orchestrates all of them. In Airflow, you'll find a variety of Executors, such as KubernetesExecutor, CeleryExecutor, LocalExecutor, and SequentialExecutor.

- **Worker**
  Basically, workers are liable for running those tasks that the Executor has provided to them.
- **Metadata Database**
  Airflow supports an extensive range of metadata storage databases. These comprise information regarding DAGs and their runs along with other Airflow configurations, such as connections, roles and users. Also, the database is used by the webserver to showcase the states and runs of the DAGs.

3. Define the types of Executors in Airflow.

The Executors, as mentioned above, are such components that execute tasks. Thus, Airflow has a variety of them, such as:

- **SequentialExecutor**
  SequentialExecutor only executes one task at a time. Herein, the workers and the scheduler both use a similar machine.
- **KubernetesExecutor**
  This one runs every task in its own Kubernetes pod. On-demand, it spins up the worker pods, thus, enabling the efficient use of resources
- **LocalExecutor**
  In most ways, this one is the same as the SequentialExecutor. However, the only difference is that it can run several tasks at a time.
- **CeleryExecutor**
  Celery is typically a Python framework that is used for running distributed asynchronous tasks. Thus, it has been a page of Airflow for a long time now. CeleryExecutors come with a fixed number of workers that are always on the standby to take tasks whenever available.

4. Can you define the pros and cons of all Executors in Airflow?

Here are the pros and cons of Executors in Airflow.

Executors	Pros	Cons
SequentialExecutor	• Simple and straightforward setup • A good way to test DAGs is when they're in the development stage	• Not scalable • Can't perform several tasks at a time • Not suitable for use in production
LocalExecutor	• Can perform multiple tasks at a time • Can be used to run DAGs when they're in the development stage	• Not scalable • Only one failure point • Unsuitable for use in production
CeleryExecutor	• Scalable • Responsible for handling workers • Can create a new one if there's a failure	• Needs RabbitMQ or • Redis to queue tasks Complicated setup
KubernetesExecutor	• Offers the advantages of LocalExecutor and CeleryExecutor in one as far as simplicity and scalability go	• Complex documentation • Complicated setup

	• Fine control of task-allocation resources	

5. How can you define a workflow in Airflow?

To define workflows in Airflow, Python files are used. The DAG Python class lets you create a Directed Acyclic Graph, which represents the workflow.

```
from Airflow.models import DAG
from airflow.utils.dates import days_ago

args = {
   'start_date': days_ago(0),
}

dag = DAG(
   dag_id='bash_operator_example',
   default_args=args,
   schedule_interval='* * * * *',
)
```

You can use the beginning date to launch any task on a certain date. The schedule interval also specifies how often every workflow is scheduled to run. Also, '* * * * *' represents that the tasks should run each minute.

Scala Fundamentals

Scala is a hybrid programming language that supports both object-oriented and functional programming. It supports functional programming concepts such as immutable data structures and functions as first-class citizens. For object-oriented programming, it supports concepts such as class, object, and trait. It also supports encapsulation, inheritance, polymorphism, and other important object-oriented concepts. Scala is a statically typed language. A Scala application is compiled by the Scala compiler. It is a type-safe language and the Scala compiler enforces type safety at compile time. This helps reduce the number of bugs in an application.
Scala is a hybrid object-oriented and functional programming language, it emphasizes functional programming. That is what makes it a powerful language.

You will reap greater benefit from using Scala as a functional programming languagethan if you used it just as another object-oriented programming language.
You launch it by typing scala in a terminal.

```
$ cd /path/to/scala-binaries
$ bin/scala
scala> println("hello world")
```

Basic Types

Variable Type	Description
Byte	8-bit signed integer
Short	16-bit signed integer
Int	32-bit signed integer
Long	64-bit signed integer
Float	32-bit single precision float
Double	64-bit double precision float
Char	16-bit unsigned Unicode character
String	A sequence of Chars
Boolean	true or false

Variables

Scala has two types of variables: mutable and immutable.A mutable variable is declared using the keyword var; whereas an immutable variable is declared usingthe keyword val. The following two statements are equivalent. val y:

Int = 10;
val y = 10

Function

A function is a block of executable code that returns a value. It is conceptuallysimilar to a function in mathematics; it takes inputs and returns an output.
Scala treats functions as first-class citizens. A function can be used like a variable. It can be passed as an input to another function. It can be defined as an unnamed function literal, like a string literal. It can be assigned to a variable. It can be defined inside another function. It can be returned as an output from another function. A function in Scala is defined with the keyword def.

Example:

```
def add(firstInput: Int, secondInput: Int): Int = {val sum
= firstInput + secondInput
return sum
}
```

Methods

A method is a function that is a member of an object.

Higher-Order Methods

A method that takes a function as an input parameter is called a *higher-order method.*
The following example shows a simple higher-order function.

```
def encode(n: Int, f: (Int) => Long): Long = {val x =
n * 10
f(x)
}
```

Classes

A class is an object-oriented programming concept. It provides a higher-level programming abstraction. At a very basic level, it is a code organization technique that allows you to bundle

data and all of its operations together. Conceptually, it represents an entity with properties and behavior. A class in Scala is similar to that in other object-oriented languages. It consists of fields and methods.

An example :

```
class Car(mk: String, ml: String, cr: String) {val
make = mk
val model = mlvar
color = cr
def repaint(newColor: String) = {color =
newColor
}
}
```

An instance of a class is created using the keyword new.

```
val mustang = new Car("Ford", "Mustang", "Red")val
corvette = new Car("GM", "Corvette", "Black")
```

Tuples

A *tuple* is a container for storing two or more elements of different types. It is immutable; it cannot be modified after it has been created. It has a lightweight syntax:

```
val twoElements = ("10", true)
val threeElements = ("10", "harry", true)
```

An element in a tuple has a one-based index. The following code sample shows the syntax for accessing elements in a tuple.

```
val first = threeElements._1 val second
= threeElements._2val third =
threeElements._3
```

List

A *List* is a linear sequence of elements of the same type. It is a recursive data structure

The following code shows a few ways to create a list.

```
val xs = List(10,20,30,40)val ys =
(1 to 100).toList val zs =
someArray.toList
```

Basic operations on a list include

- Fetching the first element. For this operation, the List class provides a methodnamed head.
- Fetching all the elements except the first element. For this operation, the Listclass provides a method named tail.
- Checking whether a list is empty. For this operation, the List class provides amethod named isEmpty, which returns true if a list is empty.

Vector

The Vector class is a hybrid of the List and Array classes. It combines the performance characteristics of both Array and List. It provides constant-time indexedaccess and constant-time linear access. It allows both fast random access and fast functional updates.

An example is shown next.

```
val v1 = Vector(0, 10, 20, 30, 40)val v2 =
v1 :+ 50
val v3 = v2 :+ 60val v4 =
v3(4) val v5 = v3(5)
```

Sets

Set is an unordered collection of distinct elements. It does not contain duplicates. Inaddition, you cannot access an element by its index, since it does not have one.

An example of a set is shown next.

```
val fruits = Set("apple", "orange", "pear", "banana")
```

Sets support two basic operations.

- contains: Returns true if a set contains the element passed as input to this method.
- isEmpty: Returns true if a set is empty

Map

Map is a collection of key-value pairs. In other languages, it known as a dictionary, associative array, or hash map. It is an efficient data structure for looking up a value by its key. It should not be confused with the map in Hadoop MapReduce. That mapprefers to an operation on a collection.

The following code snippet shows how to create and use a Map.

```
val capitals = Map("USA" -> "Washington D.C.", "UK" -> "London", "India" -> "NewDelhi")
val indiaCapital = capitals("India")
```

Higher-Order Methods on Collection Classes

The real power of Scala collections comes from their higher-order methods.

A higher-order method takes a function as its input parameter. It is important to notethat a higher-order method does not mutate a collection.

map

The map method of a Scala collection applies its input function to all the elements inthe collection and returns another collection.

```
val xs = List(1, 2, 3, 4)
val ys = xs.map((x: Int) => x * 10.0)
```

flatMap

The flatMap method of a Scala collection is similar to map. It takes a function as input, applies it to each element in a collection, and returns another collection as aresult.

Example:

```
val line = "Scala is fun"
val SingleSpace = " "
val words = line.split(SingleSpace)
val arrayOfChars = words flatMap {_.toList}
```

filter

The filter method applies a predicate to each element in a collection and returns another collection consisting of only those elements for which the predicate returned true. A predicate is function that returns a Boolean value. It returns either true or false.

```
val xs = (1 to 100).toList
val even = xs filter {_ %2 == 0}
```

foreach

The foreach method of a Scala collection calls its input function on each element of the collection, but does not return anything. It is similar to the map method. The onlydifference between the two methods is that map returns a collection and foreach doesnot return anything. It is a rare method that is used for its side effects.

```
val words = "Scala is fun".split(" ")
```

```
words.foreach(println)
```

reduce

The reduce method returns a single value. As the name implies, it reduces a collectionto a single value. The input function to the reduce method takes two inputs at a time and returns one value. Essentially, the input function is a binary operator that must beboth associative and commutative.

EXAmple.

```
val xs = List(2, 4, 6, 8, 10)
val sum = xs reduce {(x,y) => x + y}
val product = xs reduce {(x,y) => x * y}
val max = xs reduce {(x,y) => if (x > y) x else y}val min
= xs reduce {(x,y) => if (x < y) x else y}
```

SPARK

Spark is the most active open source project in the big data world. It has become hotter than Hadoop. It is considered the successor to Hadoop MapReduce. Spark adoption is growing rapidly. Many organizations are replacing MapReduce with Spark.Spark is an in-memory cluster computing framework for processing and analyzing large amounts of data.

Key features of Spark

1. Easy to Use

Spark provides a simpler programming model than that provided by MapReduce. Developing a distributed data processing application with Spark is a lot easier than developing the same application with MapReduce. Spark offers a rich application programming interface (API) for developing big data applications; it comes with 80- plus data processing operators.

2. Fast

Spark is orders of magnitude faster than Hadoop MapReduce. It can be hundreds of times faster than Hadoop MapReduce if data fits in memory. Even if data does not fit in memory, Spark can be up to ten times faster than Hadoop MapReduce.

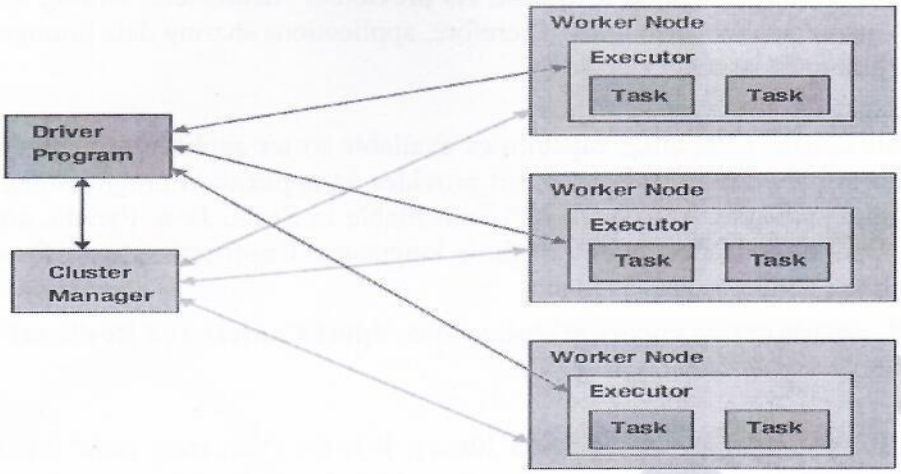
3. General Purpose

Spark provides a unified integrated platform for different types of data processing jobs. It can be used for batch processing, interactive analysis, stream processing, machine learning, and graph computing. In contrast, Hadoop MapReduce is designed just for batch processing. Therefore, a developer using MapReduce has to use different frameworks for stream processing and graph computing.

Spark comes pre-packaged with an integrated set of libraries for batch processing, interactive analysis, stream processing, machine learning, and graph computing. With Spark, you can use a single framework to build a data processing pipeline that involves different types of data processing tasks. There is no need to learn multiple frameworks or deploy separate clusters for different types of data processing jobs. Thus, Spark helps reduce operational complexity and avoids code as well as data duplication.

Spark Architecture

A Spark application involves five key entities: a driver program, a cluster manager,workers, executors, and tasks



Workers

A worker provides CPU, memory, and storage resources to a Spark application. Theworkers run a Spark application as distributed processes on a cluster of nodes.

Cluster Managers

Spark uses a cluster manager to acquire cluster resources for executing a job. A cluster manager, as the name implies, manages computing resources across a cluster of worker nodes. It provides low-level scheduling of cluster resources across applications. It enables multiple applications to share cluster resources and run on the same worker nodes. Spark currently supports three cluster managers: standalone, Mesos, and YARN. Mesos and YARN allow you to run Spark and Hadoop applications simultaneously on the same worker nodes.

Executors

An executor is a JVM (Java virtual machine) process that Spark creates on each worker for an application. It executes application code concurrently in multiple threads. It can also cache data in memory or disk. An executor has the same lifespan as the application for which it is created. When a Spark application terminates, all the executors created for it also terminate.

Tasks

A task is the smallest unit of work that Spark sends to an executor. It is executed by a thread in an executor on a worker node. Each task performs some computations to either return a result to a driver program or partition its output for shuffle.

Spark creates a task per data partition. An executor runs one or more tasks concurrently. The amount of parallelism is determined by the number of partitions. More partitions mean more tasks processing data in parallel.

How an Application Works

A Spark application processes data in parallel across a cluster of nodes. When a Spark application is run, Spark connects to a cluster manager and acquires executors on the worker nodes. As mentioned earlier, a Spark application submits a data processing algorithm as a job. Spark splits a job into a directed acyclic graph (DAG) of stages. It then schedules the execution of these stages on the executors using a low-level scheduler provided by a cluster manager. The executors run the tasks submitted by Spark in parallel.

Every Spark application gets its own set of executors on the worker nodes. This design provides a few benefits. First, tasks from different applications are isolated from each other since they run in different JVM processes. A misbehaving task from one application cannot crash another Spark application. Second, scheduling of tasks becomes easier. Spark has to schedule the tasks belonging to only one application at a time. It does not have to handle the complexities of scheduling tasks from multiple concurrently running applications. However, this design also has one disadvantage. Since applications run in separate JVM processes, they cannot easily share data. Even though they may be running on the same worker nodes, they

cannot share data without writing it to disk. As previously mentioned, writing and reading data from disk are expensive operations. Therefore, applications sharing data through disk will experience performance issues.

Application Programming Interface (API)

Spark makes its cluster computing capabilities available to an application in the form of a library. This library is written in Scala, but it provides an application programming interface (API) in multiple languages. The Spark API is available in Scala, Java, Python, and R. You can develop a Spark application in any of these languages. Unofficial support for additional languages, such as Clojure, is also available.

The Spark API consists of two important abstractions: **SparkContext and ResilientDistributed Datasets (RDDs).**

SparkContext

SparkContext is a class defined in the Spark library. It is the main entry point into the Spark library. It represents a connection to a Spark cluster. It is also required to create other important objects provided by the Spark API. A Spark application must create an instance of the SparkContext class. Currently, an application can have only one active instance of SparkContext. To create another instance, it must first stop the active instance.

The SparkContext class provides multiple constructors. The simplest one does not take any arguments. An instance of the SparkContext class can be created as shown next.

```
val sc = new SparkContext()
```

SparkContext gets configuration settings such as the address of the Spark master, application name, and other settings from system properties. You can also provide configuration parameters to SparkContext programmatically using an instance of SparkConf, which is also a class defined in the Spark library. It can be used to set various Spark configuration parameters as shown next.

```
val config = new SparkConf().setMaster("spark://host:port").setAppName("bigapp")
val sc = new SparkContext(config)
```

In addition to providing explicit methods for configuring commonly used parameters such as the address of the Spark master, SparkConf provides a generic method forsetting any parameter using a keyvalue pair.

Resilient Distributed Datasets (RDD)

RDD represents a collection of partitioned data elements that can be operated on in parallel. It is the primary data abstraction mechanism in Spark. It is defined as an abstract class in the Spark library. Conceptually, RDD is similar to a Scala collection, except that it represents a distributed dataset and it supports lazy operations. Lazy operations The key characteristics of an RDD are :.

- ✓ Immutable
- ✓ Partitioned
- ✓ Fault Tolerant
- ✓ Interface
- ✓ Strongly Typed .
- ✓ In Memory

Creating an RDD

Since RDD is an abstract class, you cannot create an instance of the RDD classdirectly. The SparkContext class provides factory methods to create instances of concrete implementation classes. An RDD can also be created from another RDD byapplying a transformation to it. As discussed earlier, RDDs are immutable. Any operation that modifies an RDD returns a new RDD with the modified data **parallelize**

This method creates an RDD from a local Scala collection. It partitions and distributesthe elements of a Scala collection and returns an RDD representing those elements.

This method is generally not used in a production application, but useful for learningSpark.

```
val xs = (1 to 10000).toList
```

textFile

```
val rdd =sc.parallelize(xs)
```

The textFile method creates an RDD from a text file. It can read a file or multiple filesin a directory stored on a local file system, HDFS, Amazon S3, or any other

Hadoop-supported storage system. It returns an RDD of Strings, where each element represents a line in the input file.

```
val rdd = sc.textFile("hdfs://namenode:9000/path/to/file-or-directory")
```

The preceding code will create an RDD from a file or directory stored on HDFS.
The textFile method can also read compressed files. In addition, it supports wildcardsas an argument for reading multiple files from a directory. An example is shown next.

```
val rdd = sc.textFile("hdfs://namenode:9000/path/to/directory/*.gz")
```

The textFile method takes an optional second argument, which can be used to specify the number of partitions. By default, Spark creates one RDD partition for each file block. You can specify a higher number of partitions for increasing parallelism; however, a fewer number of partitions than file blocks is not allowed.

wholeTextFiles

This method reads all text files in a directory and returns an RDD of key-value pairs. Each key-value pair in the returned RDD corresponds to a single file. The key part stores the path of a file and the value part stores the content of a file. This method can also read files stored on a local file system, HDFS, Amazon S3, or any other Hadoop- supported storage system.

```
val rdd = sc.wholeTextFiles("path/to/my-data/*.txt")
```

sequenceFile

The sequenceFile method reads key-value pairs from a sequence file stored on a local file system, HDFS, or any other Hadoop-supported storage system. It returns an RDD of key-value pairs. In addition to providing the name of an input file, you have to specify the data types for the keys and values as type parameters when you call this method.

```
val rdd = sc.sequenceFile[String, String]("some-file")
```

RDD Operations

Spark applications process data using the methods defined in the RDD class or classesderived from it. These methods are also referred to as operations. Since Scala allows a method to be used with operator notation, the RDD methods are also sometimes referred to as *operators*. The beauty of Spark is that the same RDD methods can be used to process data ranging in size from a few bytes to several petabytes. In addition, a Spark application can use the same methods to process datasets stored on either a distributed storage system or a local file system. This flexibility allows a developer to develop, debug and test a Spark application on a single machine and deploy it on a large cluster without making any code change.

RDD operations can be categorized into two types: transformation and action. A transformation creates a new RDD. An action returns a value to a driver program.

Transformations

A transformation method of an RDD creates a new RDD by performing a computation on the source RDD.

RDD transformations are conceptually similar to Scala collection methods. The key difference is that the Scala collection methods operate on data that can fit in the memory of a single machine, whereas RDD methods can operate on data distributed

across a cluster of nodes. Another important difference is that RDD transformationsare lazy, whereas Scala collection methods are strict.

map

The map method is a higher-order method that takes a function as input and applies itto each element in the source RDD to create a new RDD. The input function to map must take a single input parameter and return a value.

```
val lines = sc.textFile("...")
val lengths = lines map { l => l.length}
```

filter

The filter method is a higher-order method that takes a Boolean function as input andapplies it to each element in the source RDD to create a new RDD. A Boolean function takes an input and returns true or false. The filter method returns a new RDDformed by selecting only those elements for which the input Boolean function returned true. Thus, the new RDD contains a subset of the elements in the original RDD.

flatMap

```
val lines = sc.textFile("...")
val longLines = lines filter { l => l.length > 80}
```

The flatMap method is a higher-order method that takes an input function, which returns a sequence for each input element passed to it. The flatMap method returns anew RDD formed by flattening this collection of sequence.

```
val lines = sc.textFile("...")
val words = lines flatMap { l => l.split(" ")
```

mapPartitions

The higher-order mapPartitions method allows you to process data at a partition level.Instead of passing one element at a time to its input function, mapPartitions passes a partition in the form of an iterator.

The input function to the mapPartitions method takes an iterator as input and returnsanother iterator as output. The mapPartitions method returns new RDD formed by applying a user-specified function to each partition of the source RDD.

```
val lines = sc.textFile("...")
val lengths = lines mapPartitions { iter => iter.map { l => l.length}}
```

union

The union method takes an RDD as input and returns a new RDD that contains theunion of the elements in

the source RDD and the RDD passed to it as an input.

```
val linesFile1 = sc.textFile("...")val
linesFile2 = sc.textFile("...")
val linesFromBothFiles = linesFile1.union(linesFile2)
```

intersection

The intersection method takes an RDD as input and returns a new RDD that contains the intersection of theelements in the source RDD and the RDD passed to it as an input.

```
val linesFile1 = sc.textFile("...")val
linesFile2 = sc.textFile("...")
val linesPresentInBothFiles = linesFile1.intersection(linesFile2)
```

Here is another example.

```
val mammals = sc.parallelize(List("Lion", "Dolphin", "Whale"))
val aquatics =sc.parallelize(List("Shark", "Dolphin", "Whale"))val
aquaticMammals = mammals.intersection(aquatics)
```

This method is generally not used in a production application, but useful for learningSpark.

```
val xs = (1 to 10000).toList
```

textFile

```
val rdd =sc.parallelize(xs)
```

The textFile method creates an RDD from a text file. It can read a file or multiple filesin a directory stored on a local file system, HDFS, Amazon S3, or any other

Hadoop-supported storage system. It returns an RDD of Strings, where each element represents a line in the input file.

```
val rdd = sc.textFile("hdfs://namenode:9000/path/to/file-or-directory")
```

The preceding code will create an RDD from a file or directory stored on HDFS.
The textFile method can also read compressed files. In addition, it supports wildcardsas an argument for reading multiple files from a directory. An example is shown next.

```
val rdd = sc.textFile("hdfs://namenode:9000/path/to/directory/*.gz")
```

The textFile method takes an optional second argument, which can be used to specify the number of partitions. By default, Spark creates one RDD partition for each file block. You can specify a higher number of partitions for increasing parallelism; however, a fewer number of partitions than file blocks is not allowed.

wholeTextFiles

This method reads all text files in a directory and returns an RDD of key-value pairs. Each key-value pair in the returned RDD corresponds to a single file. The key part stores the path of a file and the value part stores the content of a file. This method can also read files stored on a local file system, HDFS, Amazon S3, or any other Hadoop- supported storage system.

```
val rdd = sc.wholeTextFiles("path/to/my-data/*.txt")
```

sequenceFile

The sequenceFile method reads key-value pairs from a sequence file stored on a local file system, HDFS, or any other Hadoop-supported storage system. It returns an RDD of key-value pairs. In addition to providing the name of an input file, you have to specify the data types for the keys and values as type parameters when you call this method.

```
val rdd = sc.sequenceFile[String, String]("some-file")
```

RDD Operations

Spark applications process data using the methods defined in the RDD class or classesderived from it. These methods are also referred to as operations. Since Scala allows a method to be used with operator notation, the RDD methods are also sometimes referred to as *operators*. The beauty of Spark is that the same RDD methods can be used to process data ranging in size from a few bytes to several petabytes. In addition, a Spark application can use the same methods to process datasets stored on either a distributed storage system or a local file system. This flexibility allows a developer to develop, debug and test a Spark application on a single machine and deploy it on a large cluster without making any code change.
RDD operations can be categorized into two types: transformation and action. A transformation creates a new RDD. An action returns a value to a driver program.

Transformations

A transformation method of an RDD creates a new RDD by performing a computation on the source RDD.
RDD transformations are conceptually similar to Scala collection methods. The key difference is that the Scala collection methods operate on data that can fit in the memory of a single machine, whereas RDD methods can operate on data distributed

across a cluster of nodes. Another important difference is that RDD transformationsare lazy, whereas Scala collection methods are strict.

subtract

The subtract method takes an RDD as input and returns a new RDD that containselements in the source RDD but not in the input RDD.

```
val linesFile1 = sc.textFile("...")val
linesFile2 = sc.textFile("...")
val linesInFile1Only = linesFile1.subtract(linesFile2)Here is
another example.
val mammals = sc.parallelize(List("Lion", "Dolphin", "Whale"))val
aquatics =sc.parallelize(List("Shark", "Dolphin", "Whale")) val fishes =
aquatics.subtract(mammals)
```

keyBy

The keyBy method is similar to the groupBy method. It a higher-order method that takesas input a function that returns a key for any given element in the source RDD. The keyBymethod applies this function to all the
elements in the source RDD and returns an RDD of pairs. In each returned pair, the first item is a key and the second item is an element that was mapped to that key by the inputfunction to the keyBy method. The RDD returned by keyBy will have the same number ofelements as the source RDD. The difference between groupBy and keyBy is that the seconditem in a returned pair is a collection of elements in the first case, while it is a single element in the second case.

```
case class Person(name: String, age: Int, gender: String, zip: String)val lines
= sc.textFile("...")
val people = lines map { l => {val a =
l.split(",")
Person(a(0), a(1).toInt, a(2), a(3))
}
}
val keyedByZip = people.keyBy { p => p.zip}
```

sortBy

The higher-order sortBy method returns an RDD with sorted elements from the sourceRDD. It takes two input parameters. The first input is a function that generates a key for each element in the source RDD. The second argument allows you to specify ascending or descending order for sort.

```
val numbers = sc.parallelize(List(3,2, 4, 1, 5))val
sorted = numbers.sortBy(x => x, true)
```

sample

The sample method returns a sampled subset of the source RDD. It takes three input parameters. The first parameter specifies the replacement strategy. The second parameter specifies the ratio of the sample size to source RDD size. The third parameter, which is optional, specifies a random seed for sampling.

```
val numbers = sc.parallelize((1 to 100).toList) val
sampleNumbers = numbers.sample(true, 0.2)
```

Lazy Operations

RDD creation and transformation methods are lazy operations. Spark does not immediately perform any computation when an application calls a method that returnan RDD. For example, when you read a file from HDFS using textFile method of SparkContext, Spark does not immediately read the file from disk.

Similarly, RDD transformations, which return a new RDD, are lazily computed.Spark just

keeps track of transformations applied to an RDD.
sample code:

```
val lines = sc.textFile("...")
val errorLines = lines filter { l => l.contains("ERROR")}
val warningLines = lines filter { l => l.contains("WARN")}
```

These three lines of code will seem to execute very quickly, even if you pass a file containing 100 terabytes of data to the textFile method. The reason is that the textFile method does not actually read a file right when you call it. Similarly, the filter method does not immediately iterate through all the elements in the source RDD.
Spark just makes a note of how an RDD was created and the transformations applied to it to create child RDDs. Thus, it maintains lineage information for each RDD. It uses this lineage information to construct or reconstruct an RDD when required.

Spark Jobs

RDD operations, including transformation, action and caching methods form the basis of a Spark application. Essentially, RDDs describe the Spark programming model. Now that we have covered the programming model, we will discuss how it all comes together in a Spark application. A job is a set of computations that Spark performs to return the results of an action to a driver program. An application can launch one or more jobs. It launches a job by calling an action method of an RDD. Thus, an action method triggers a job. If an action is called for an RDD that is not cached or a descendant of a cached RDD, a job starts with the reading of data from a storage system. However, if an action is called for an RDD that is cached or a descendent of a cached RDD, a job begins from the point at which the RDD or its ancestor RDD was cached.

Spark SQL:

Spark SQL brings native support for SQL to Spark and streamlines the process of querying data stored both in RDDs (Spark's distributed datasets) and in external sources. Spark SQL conveniently blurs the lines between RDDs and relational tables. Unifying these powerful abstractions makes it easy for developers to intermix SQL commands querying external data with complex analytics, all within in a single application. Concretely, Spark SQLwill allow developers to:

- Import relational data from Parquet files and Hive tables
- Run SQL queries over imported data and existing RDDs
- Easily write RDDs out to Hive tables or Parquet files

```
Example: results=spark.sql("SELECT*FROMpeople")
names = results.map(lambda p: p.name)
```

Spark MLlib:

MLlib is Spark's machine learning (ML) library. Its goal is to make practical machinelearning scalable and easy. At a high level, it provides tools such as:

- ML Algorithms: common learning algorithms such as classification, regression, clustering, and collaborative filtering
- Featurization: feature extraction, transformation, dimensionality reduction, and selection
- Pipelines: tools for constructing, evaluating, and tuning ML Pipelines
- Persistence: saving and load algorithms, models, and Pipelines
- Utilities: linear algebra, statistics, data handling, etc.

https://spark.apache.org/mllib/