D.N.R.COLLEGE, PG COURSES (AUTONOMOUS), BHIMAVARAM

(Affiliated to Adikavi Nannaya University)

MCA DEPARTMENT

E-CONTENT

Presented by

A.DURGA DEVI

II MCA – III SEM - Big Data Analytics

Introduction to Big Data: Big Data-definition, Characteristics of Big Data (Volume, Variety, Velocity, Veracity, Validity), Importance of Big Data , Patterns for Big Data Development, Data in the Warehouse and Data in Hadoop.

Introduction to Hadoop: Hadoop- definition, Understanding distributed systems and Hadoop, Comparing SQL databases and Hadoop, History of Hadoop, Starting Hadoop - The building blocks of Hadoop, NameNode, DataNode, Secondary NameNode, JobTracker and Task Tracker.

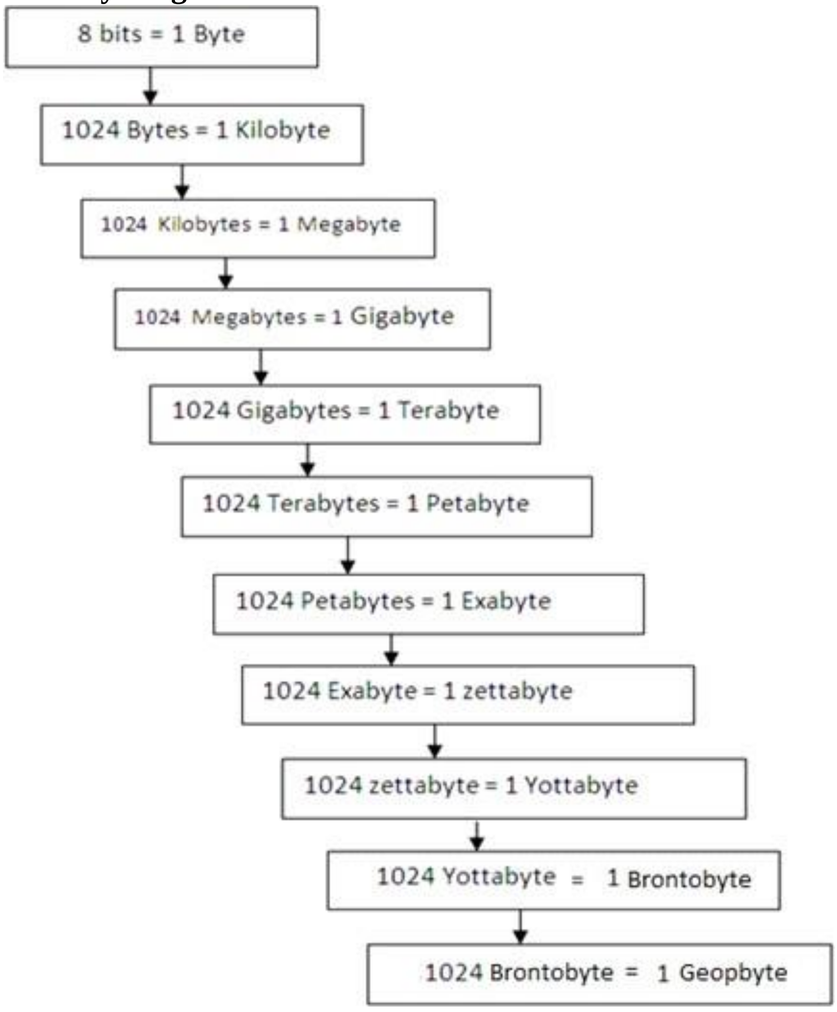
2. Introduction to Big Data

à What is Big Data and Big Data Analytics (BDA)?

"***Big Data*** is an evolving term that describes any large amount of structured, semi-structured and unstructured data that has the potential to be mined for information."

"Big Data Analytics (BDA) is the process of examining **large data** sets containing a variety of **data** types -- i.e., **big data** -- to uncover hidden patterns, unknown correlations, market trends, customer preferences and other useful business information."

Hierarchy of memory usage units:


8 bits = 1 Byte
1024 Bytes = 1 Kilobyte
1024 Kilobytes = 1 Megabyte
1024 Megabytes = 1 Gigabyte
1024 Gigabytes = 1 Terabyte
1024 Terabytes = 1 Petabyte
1024 Petabytes = 1 Exabyte
1024 Exabyte = 1 zettabyte
1024 zettabyte = 1 Yottabyte
1024 Yottabyte = 1 Brontobyte
1024 Brontobyte = 1 Geopbyte

àCharacteristics of Big Data (or) Why is Big Data different from any other data?

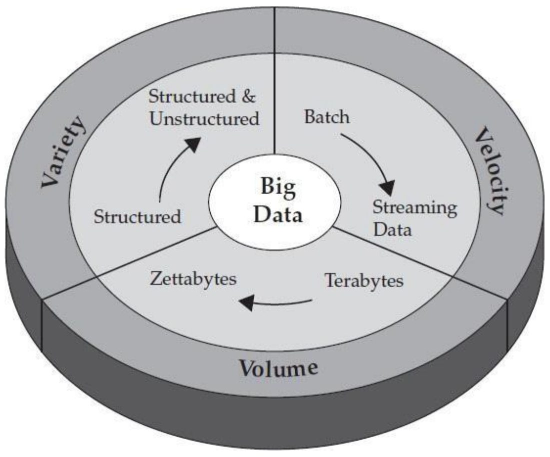
There are "Five V‟s" that characterize this data: Volume, Velocity, Variety, Veracity and Validity.

- **Volume (Data Quantity):**

Most organizations were already struggling with the increasing size of their databases as the Big Data tsunami hit the data stores.

- **Velocity (Data Speed):**
  There are two aspects to velocity. They are throughput of data and the other representing latency.
- Throughput which represents the data moving in the pipes.
- Latency is the other measure of velocity. Analytics used to be a “store and report” environment where reporting typically contained data as of yesterday—popularly represented as “D-1.”
- **Variety (Data Types):**
  The source data includes unstructured text, sound, and video in addition to structured data. A number of applications are gathering data from emails, documents, or blogs.
- **Veracity (Data Quality):**
  Veracity represents both the credibility of the data source as well as the suitability of the data for the target audience.
- **Validity (Data Correctness):**
  Validity meaning is the data correct and accurate for the future use. Clearly valid data is key to making the right decisions.

As per IBM, the number of characteristics of Big Data is V^3 and described in the following Figure:



Importance of Big Data:

- Access to social data from search engines and sites like Facebook, twitter are enabling organizations to fine tune their business strategies. Marketing agencies are learning about the response for their campaigns, promotions, and other advertising mediums.

- Traditional customer feedback systems are getting replaced by new systems designed with „Big Data‟ technologies. In these new systems, Big Data and natural language processing technologies are being used to read and evaluate consumer responses.

- Based on information in the social media like preferences and product perception of their consumers, product companies and retail organizations are planning their production.
- Determining root causes of failures, issues and defects in near-real time
- Detecting fraudulent behavior before it affects the organization.

When to you use Big Data technologies?

- Big Data solutions are ideal for analyzing not only raw structured data, but semi-structured and unstructured data from a wide variety of sources.
- Big Data solutions are ideal when all, or most, of the data needs to be analyzed versus a sample of the data; or a sampling of data isn‟t nearly as effective as a larger set of data from which to derive analysis.
- Big Data solutions are ideal for iterative and exploratory analysis when business measures on data are not predetermined.

Patterns for Big Data Development:

The following six most common usage patterns represent great Big Data opportunities—business problems that weren‟t easy to solve before—and help us gain an understanding of how Big Data can help us (or how it‟s helping our competitors make us less competitive if we are not paying attention).

- IT for IT Log Analytics
- The Fraud Detection Pattern
- The Social Media Pattern
- The Call Center Mantra: “This Call May Be Recorded for Quality Assurance Purposes”
- Risk: Patterns for Modeling and Management
- Big Data and the Energy Sector

• **IT for IT Log Analytics:** IT departments need logs at their disposal, and today they just can‟t store enough logs and analyze them in a cost-efficient manner, so logs are typically kept for emergencies and discarded as soon as possible. Another reason why IT departments keep large amounts of data in logs is to look for rare problems. It is often the case that the most common problems are known and easy to deal with, but the problem that happens “once in a while” is typically more difficult to diagnose and prevent from occurring again.

But there are more reasons why log analysis is a Big Data problem apart from its large nature. The nature of these logs is semi-structured and raw, so they aren‟t always suitedfor traditional database processing. In addition, log formats are constantly changing due to hardware and software upgrades, so they can‟t be tied to strict inflexible analysis paradigms.

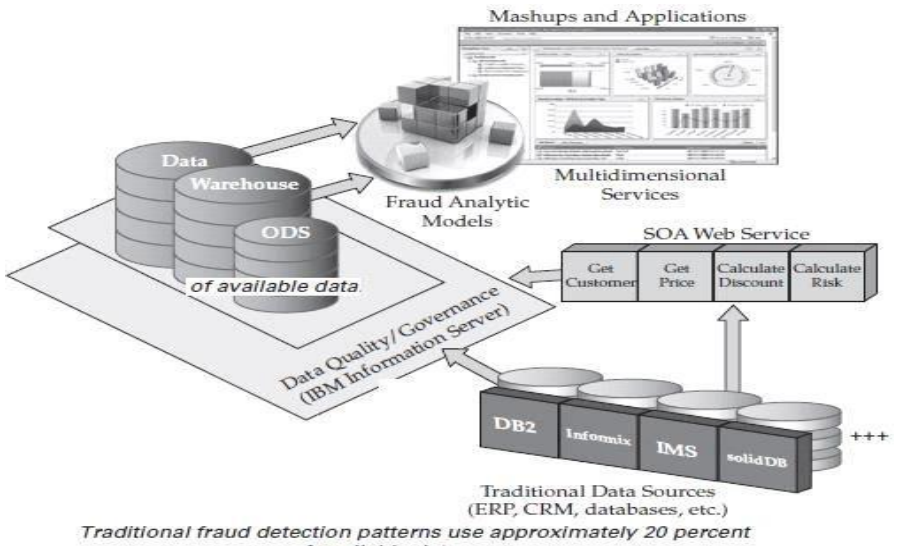
Finally, not only do we need to perform analysis on the longevity of the logs to determine trends and patterns and to find failures, but also we need to ensure the analysis is done on all the data.

Log analytics is actually a pattern that IBM established after working with a number of companies, including some large financial services sector (FSS) companies. This use case comes up with quite a few customers since; for that reason, this pattern is called *IT for IT*. If we are new to this usage pattern and wondering just who is interested in IT for IT Big Data solutions, we should know that this is an internal use case within an organization itself. An internal IT for IT implementation is well suited for any organization with a large data center footprint, especially if it is relatively complex. For example, service-oriented architecture (SOA) applications with lots of moving parts, federated data centers, and so on, all suffer from the same issues outlined in this section.

Some of large insurance and retail clients need to know the answers to such questions as, “What are the precursors to failures?”, “How are these systems all related?”, and more. These are the types of questions that conventional monitoring doesn‟t answer; a Big Data platform finally offers the opportunity to get some new and better insights into the problems at hand.

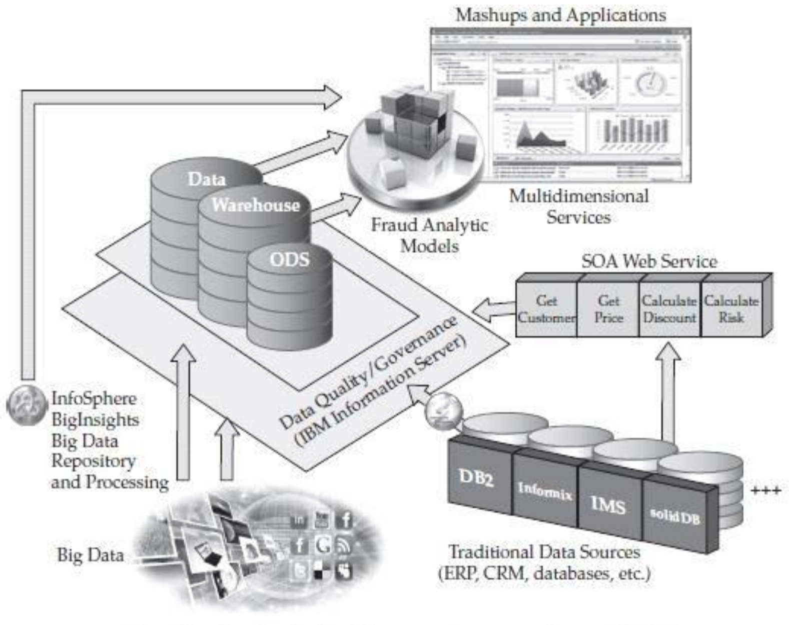
• **The Fraud Detection Pattern:** Fraud detection comes up a lot in the financial services vertical, we will find it in any sort of claims- or transaction-based environment (online auctions, insurance claims, underwriting entities, and so on). Pretty much anywhere some sort of financial transaction is involved presents a potential for misuse and the universal threat of fraud. If we influence a Big Data platform, we have the opportunity to do more than we have ever done before to identify it or, better yet, stop it.

Traditionally, in fraud cases, samples and models are used to identify customers that characterize a certain kind of profile. The problem with this is that although it works, we are profiling a segment and not the granularity at an individual transaction or person level. As per customer experiences, it is estimated that only 20 percent (or maybe less) of the available information that could be useful for fraud modeling is actually being used. The traditional approach is shown in the following Figure.



Traditional fraud detection patterns use approximately 20 percent of available data.

We can use BigInsights to provide an flexible and cost-effective repository to establish what of the remaining 80 percent of the information is useful for fraud modeling, and then feed newly discovered high-value information back into the fraud model as shownin the following Figure.


Mashups and Applications
Multidimensional Services
Data Warehouse
ODS
Fraud Analytic Models
SOA Web Service
Get Customer
Get Price
Calculate Discount
Calculate Risk
Data Quality / Governance (IBM Information Server)
InfoSphere BigInsights Big Data Repository and Processing
Big Data
DB2
Informix
IMS
solidDB
+++
Traditional Data Sources (ERP, CRM, databases, etc.)

A modern-day fraud detection ecosystem synergizes a Big Data platform with traditional processes

A modern-day fraud detection ecosystem provides a low-cost Big Data platform for exploratory modeling and discovery. Typically, fraud detection works after a transaction gets stored only to get pulled out of storage and analyzed; storing something to instantly pull it back out again feels like latency to us. With Streams, we can apply the fraud detection models as the transaction is happening.

• **The Social Media Pattern:** Perhaps the most talked about Big Data usage pattern is social media and customer sentiment. More specifically, we can determine how sentiment is impacting sales, the effectiveness or receptiveness of marketing campaigns, the accuracy of marketing mix (product, price, promotion, and placement), and so on.

Social media analytics is a pretty hot topic, so hot in fact that IBM has built a solution specifically to accelerate our use of it: Cognos Consumer Insights (CCI). CCI can tell what people are saying, how topics are trending in social media, and all sorts of things that affect the business, all packed into a rich visualization engine.

• **The Call Center Mantra: "This Call May Be Recorded for Quality Assurance Purposes":** It seems that when we want our call with a customer service representative (CSR) to be recorded for quality assurance purposes, it seems the *may* part never works in our favor. The challenge of call center efficiencies is somewhat similar to the fraud detection pattern.

Call centers of all kinds want to find better ways to process information to address what‟s going on in the business with lower latency. This is a really interesting Big Data use case, because it uses analytics-in-motion and analytics-at-rest. Using in-motion analytics (Streams) means that we basically build our models and find out what‟s interesting based upon the conversations that have been converted from voice to text or with voice analysis as the call is happening. Using at-rest analytics (BigInsights), we build up these models and thenpromote them back into Streams to examine and analyze the calls that are actually happening in real time: it‟s truly a closed-loop feedback mechanism.

• **Risk: Patterns for Modeling and Management:** Risk modeling and management is another big opportunity and common Big Data usage pattern. Risk modeling brings into focus a frequent question when it comes to the Big Data usage patterns, "How much of our data do we use in our modeling?" The financial crisis of 2008, the associated subprime loan crisis, and its outcome has made risk modeling and management a key area of focus for financial institutions.

Two problems are associated with this usage pattern: "How much of the data will we use for our model?" and "How can we keep up with the data‟s velocity?" The answer to the second question, unfortunately, is often, "We can‟t." Finally, consider that financial services trend to move their risk model and dashboards to inter-day positions rather than just close-of-day positions, and we can see yet another challenge that can‟t be solved with traditional systems alone. Another characteristic of today‟s financial markets is that there are massive trading volumes requires better model and manage risk.

• **Big Data and the Energy Sector:** The energy sector provides many Big Data use case challenges in how to deal with the massive volumes of sensor data from remote installations. Many companies are using only a fraction of the data being collected, because they lack the infrastructure to store or analyze the available scale of data.

Vestas is primarily engaged in the development, manufacturing, sale, and maintenance of power systems that use wind energy to generate electricity through its wind turbines. Its product range includes land and offshore wind turbines. At the time of wrote thisbook, it had more than 43,000 wind turbines in 65 countries on 5 continents. Vestas used IBM BigInsights platform to achieve their vision is about the generation of clean energy.

Data in the Warehouse and Data in Hadoop:

Traditional warehouses are mostly ideal for analyzing structured data from various systems and producing insights with known and relatively stable measurements. On the other hand, Hadoop-based platform is well suited to deal with semi-structured and unstructured data, as well as when a data discovery process is needed.

The authors could say that data warehouse data is trusted enough to be "public," while Hadoop data isn"t as trusted (*public* can mean vastly distributed within the company and not for external consumption), and although this will likely change in the future, today this is something that experience suggests characterizes these repositories.

Hadoop-based repository scheme stores the entire business entity and the reliability of the Tweet, transaction, Facebook post, and more is kept intact. Data in Hadoop might seemof low value today. IT departments pick and choose high-valued data and put it through difficult cleansing and transformation processes because they know that data has a *high known value per byte*.

Unstructured data can"t be easily stored in a warehouse. A Big Data platform can store all of the data in its native business object format and get value out of it through massive parallelism on readily available components.

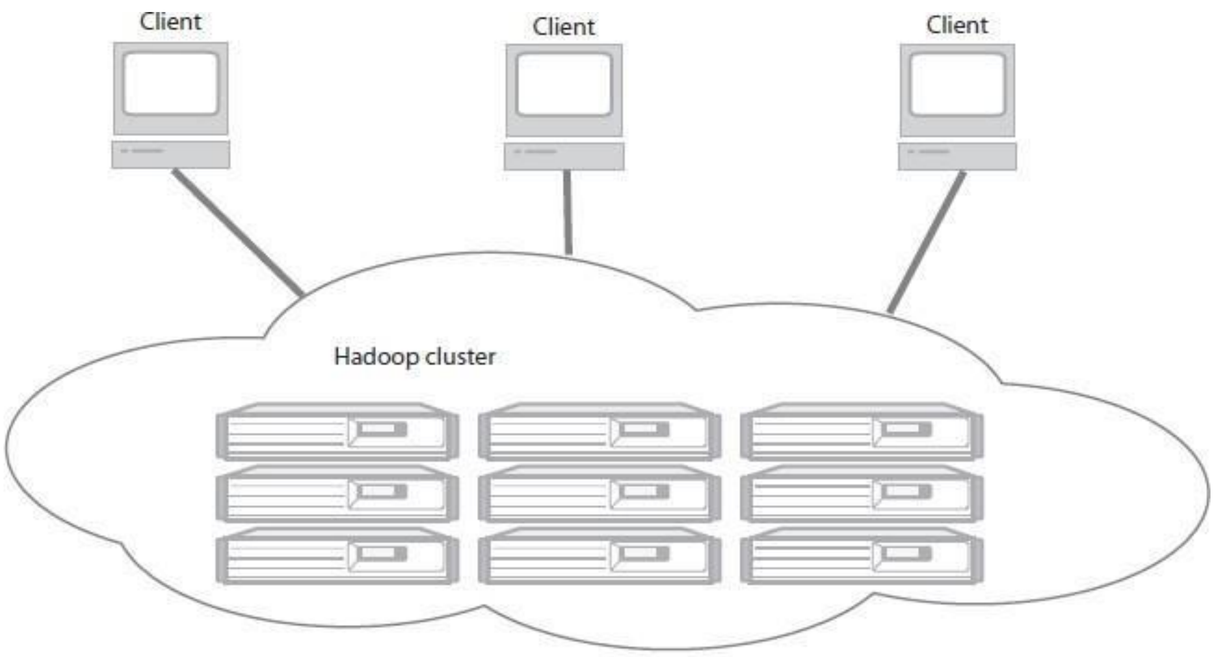
&&&&&&&&&

1. Introduction to Hadoop

Hadoop: Hadoop is an open source framework for writing and running distributed applications that process large amounts of data. Distributed computing is a wide and varied field, but the key distinctions of Hadoop are that it is:

- *Accessible*—Hadoop runs on large clusters of commodity machines or on cloud computing services such as Amazon's Elastic Compute Cloud (EC2).
- *Robust*—Because it is intended to run on commodity hardware, Hadoop is architected with the assumption of frequent hardware malfunctions (errors). It can gracefully handle most such failures.
- *Scalable*—Hadoop scales linearly to handle larger data by adding more nodes to the cluster.
- *Simple*—Hadoop allows users to quickly write efficient parallel code.

The following Figure illustrates how one interacts with a Hadoop cluster. As we can see, a Hadoop cluster is a set of commodity machines networked together in one location. Data storage and processing all occur within this "cloud" of machines. Different users can submit computing "jobs" to Hadoop from individual clients, which can be their own desktop machines in remote locations from the Hadoop cluster.



A Hadoop cluster has many parallel machines that store and process large data sets. Client computers send jobs into this computer cloud and obtain results.

Understanding distributed systems and Hadoop:

A lot of low-end/commodity machines tied together as a single functional is known as *distributed system*. A high-end machine with four I/O channels each having a throughput of 100 MB/sec will require three hours to *read* a 4 TB data set. With Hadoop, this same data set will be divided into smaller (typically 64 MB) blocks that are spread among many machines in the cluster via the Hadoop Distributed File System (HDFS). With a modest degree of replication, the cluster machines can read the data set in parallel and provide a much higher throughput. And such a cluster of commodity machines turns out to be cheaper than one high-end server.

Comparing SQL databases and Hadoop:

SQL (*structured* query language) is designed for structured data. Many of Hadoop's initial applications deal with unstructured data such as text. From this perspective Hadoop provides a more general paradigm than SQL. SQL is a query language which can be implemented on top of Hadoop as the execution engine. But in practice, SQL databases tend to refer to a whole set of legacy technologies, with several dominant vendors, optimized for a historical set of applications. The following concepts explain a more detailed comparison of Hadoop with typical SQL databases on specific dimensions.

1. Scale-Out Instead of Scale-Up: Scaling commercial relational databases is expensive. Their design is friendlier to scaling up. To run a bigger database we need to buy a bigger machine which is expensive. Unfortunately, at some point there won't be a big enough machine available for the larger data sets. Hadoop is designed to be a scale-out architecture operating on a cluster of commodity PC machines. Adding more resources means adding more machines to the Hadoop cluster. A Hadoop cluster with ten to hundreds of commodity

machines is standard. In fact, other than for development purposes, there's no reason to run Hadoop on a single server.

2. Key/Value Pairs Instead of Relational Tables: A fundamental principle of relational databases is that data resides in tables having relational structure defined by a schema. Hadoop uses key/value pairs as its basic data unit, which is flexible enough to work with the less-structured data types. Hadoop, data can originate in any form (Structured/unstructured/ semi-structured), but it eventually transforms into (key/value) pairs for the processing functions to work on.

3. Functional Programming (Mapreduce) instead of Declarative Queries (Sql): SQL is fundamentally a high-level declarative language. By executing queries, the required data will be retrieved from database. Under MapReduce we specify the actual steps in processing the data, which is more similar to an execution plan for a SQL engine. Under SQL we have query statements; under MapReduce we have scripts and codes. MapReduce allows to process data in a more general fashion than SQL queries. For example, we can build complex statistical models from our data or reformat our image data. SQL is not well designed for such tasks.

4. Offline Batch Processing Instead of Online Transactions: Hadoop is designed for offline processing and analysis of large-scale data. It doesn't work for random reading and writing of a few records, which is the type of load for online transaction processing. In fact, Hadoop is best used as a write-once , read-many-times type of data store. In this aspect it's similar to data warehouses in the SQL world. Hadoop relates to distributed systems and SQL databases at a high level.

Understanding MapReduce:

MapReduce is a data processing model. The main advantage is easy scaling of data processing over multiple computing nodes. Under the MapReduce model, the data processing primitives are called *mappers* and *reducers*. Decomposing a data processing application into mappers and reducers is sometimes nontrivial. But, once we write an application in the MapReduce form, scaling the application to run over hundreds, thousands, or even tens of thousands of machines in a cluster is merely a configuration change. This is the reason what has attracted many programmers to the MapReduce model.

Ex: To count the number of times each word occurs in a set of documents. We have a set of documents having only one document with only one sentence:

Do as I say, not as I do.

We derive the word counts shown as the following.

Word	Count
as	2
do	2
i	2
not	1
say	1

When the set of documents is small, a straightforward program will do the job and pseudo-code is:

```
define wordCount as Multiset;
for each document in documentSet {
    T = tokenize(document);
    for each token in T {
        wordCount[token]++;
    }
}
display(wordCount);
```

The program loops through all the documents. For each document, the words are extracted one by one using a tokenization process. For each word, its corresponding entry in a multiset called wordCount is incremented by one. At the end, a display() function prints out all the entries in wordCount.

The above code works fine until the set of documents we want to process becomes large. If it is large, to speed it up by rewriting the program so that it distributes the work over several machines. Each machine will process a distinct fraction of the documents. When all the machines have completed this, a second phase of processing will combine the result of all the machines. The pseudocode for the first phase, to be distributed over many machines, is

```
define wordCount as Multiset;
for each document in documentSubset {
    T = tokenize(document);
    for each token in T {
        wordCount[token]++;
    }
}
sendToSecondPhase(wordCount);
```

The pseudo-code for the second phase is:

```
define totalWordCount as Multiset;
for each wordCount received from firstPhase {
    multisetAdd (totalWordCount, wordCount);
}
```

This word counting program is getting complicated. To make it work across a cluster of distributed machines, we need to add a number of functionalities:

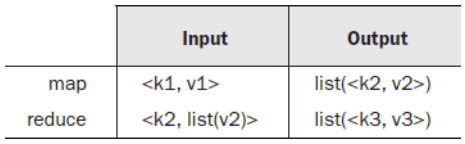
a. Store files over many processing machines (of phase one).

b. Write a disk-based hash table permitting processing without being limited by RAM capacity.
c. Partition the intermediate data (that is, wordCount) from phase one.
d. Shuffle the partitions to the appropriate machines in phase two.

Scaling the same program in MapReduce:

MapReduce programs are executed in two main phases, called *mapping* and *reducing*. Each phase is defined by a data processing function, and these functions are called *mapper* and *reducer*, respectively. In the mapping phase, MapReduce takes the input data and feeds each data element to the mapper. In the reducing phase, the reducer processes all the outputs from the mapper and arrives at a final result. In simple terms, the mapper is meant to *filter and transform* the input into something that the reducer can *aggregate* over.

The MapReduce framework was designed in writing scalable, distributed programs. This two-phase design pattern is using in scaling many programs, and became the basis of the framework. Partitioning and shuffling are common design patterns along with mapping and reducing. The MapReduce framework provides a default implementation that works in most situations. MapReduce uses *lists* and *(key/value) pairs* as its main data primitives. The keys and values are often integers or strings but can also be dummy values to be ignored or complex object types. The map and reduce functions must obey the following constraint on the types of keys and values.



	Input	Output
map	<k1, v1>	list(<k2, v2>)
reduce	<k2, list(v2)>	list(<k3, v3>)

In the MapReduce framework we write applications by specifying the mapper and reducer. The following steps explain the complete data flow:

1. The input to the application must be structured as a list of (key/value) pairs, list (<k_1, v_1>). The input format for processing multiple files is usually list (<String filename, String file_content>).
2. The list of (key/value) pairs is broken up and each individual (key/value) pair, <k_1, v_1>, is processed by calling the map function of the mapper. For word counting, mapper takes <String filename, String file_content> and promptly ignores filename. It can output a list of <String word, Integer count>, we can output a list of <String word, Integer 1> with repeated entries and let the complete aggregation be done later. That

is, in the output list we can have the (key/value) pair <"foo", 3> once or we can have the pair <"foo", 1> three times.

3. The output of all the mappers are (conceptually) aggregated into one giant list of <k_2, v_2> pairs. All pairs sharing the same k2 are grouped together into a new (key/value) pair, <k_2, list(v_2)>. The framework asks the reducer to process each one of these aggregated (key/value) pairs individually. For example, the map output for one document may be a list with pair <"foo", 1> three times, and the map output for another document may be a list with pair <"foo", 1> twice. The aggregated pair the reducer will see is <"foo", list(1,1,1,1,1)>. In word counting, the output of our reduceris <"foo", 5>, which is the total number of times "foo" has occurred in our document

```
map(String filename, String document) {
    List<String> T = tokenize(document); for
    each token in T {

    emit ((String)token, (Integer) 1);

    }

}

reduce(String token, List<Integer> values) {
    Integer sum = 0;

    for each value in values {
    sum = sum + value;
```

}set. Each reducer works on a different word. The MapReduce framework automatically collects all the <k_3, v_3> pairs and writes them to file(s).

Pseudo-code for map and reduce functions for word counting:

In the above pseudo-code, a special function is used in the framework called emit() which is used to generate the elements in the list one at a time. The emit() function further relieves the programmer from managing a large list. But Hadoop makes building scalable distributed programs easy.

History of Hadoop:

Hadoop is a versatile (flexible) tool that allows new users to access the power of distributed computing. By using distributed storage and transferring code instead of data, Hadoop avoids the costly transmission step when working with large data sets. Moreover, the redundancy of data allows Hadoop to recover should a single node fail. It is easy of creating programs with Hadoop using the MapReduce framework. On a fully configured cluster, "running Hadoop" means running a set of daemons, or resident programs, on the different servers in the network. These daemons have specific roles; some exist only on one server, some exist across multiple servers. The daemons include:

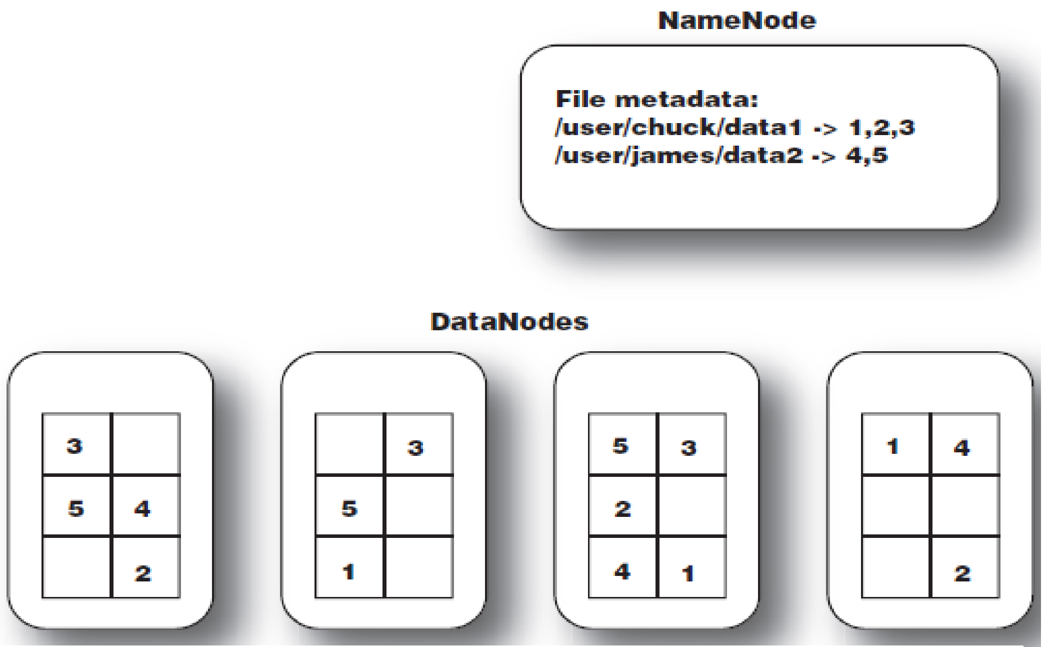
1. NameNode
2. DataNode
3. Secondary NameNode
4. JobTracker
5. TaskTracker

1. NameNode: The distributed storage system is called the *Hadoop File System*, or HDFS. The NameNode is the master of HDFS that directs the slave DataNode daemons to perform the low-level I/O tasks. The NameNode is the bookkeeper of HDFS; it keeps track of how thefiles are broken down into file blocks, which nodes store those blocks, and the overall health of the distributed filesystem.

The function of the NameNode is memory and I/O intensive. As such, the server hosting the NameNode typically doesn't store any user data or perform any computations for a MapReduce program to lower the workload on the machine. This means that the NameNode server doesn't double as a DataNode or a TaskTracker.

There is unfortunately a negative aspect to the importance of the NameNode—it's a single point of failure of the Hadoop cluster. For any of the other daemons, if their host nodes fail for software or hardware reasons, the Hadoop cluster will likely continue to function smoothly or we can quickly restart it and Not so for the NameNode.

2. DataNode: Each slave machine in the cluster will host a DataNode daemon to perform the grunt work of the distributed filesystem—reading and writing HDFS blocks to actual files on the local filesystem. When we want to read or write a HDFS file, the file is broken into blocks and the NameNode will tell the client which DataNode each block resides in. The client communicates directly with the DataNode daemons to process the local files corresponding to the blocks. Furthermore, a DataNode may communicate with other DataNodes to replicate its data blocks for redundancy. The following figure illustrates the



NameNode/DataNode interaction in HDFS. The NameNode keeps track of the file metadata—which files are in the system and how each file is broken down into blocks. The DataNodes provide backup store of the blocks and constantly report to the NameNode to keep the metadata current.
roles of NameNode and DataNodes.

The data1 file takes up three blocks, which we denote 1, 2, and 3, and the data2 file consists of blocks 4 and 5. The content of the files are distributed among the DataNodes. In this illustration, each block has three replicas. For example, block 1 (used for data1) is replicated over the three rightmost DataNodes. This ensures that if any one DataNode crashes or becomes inaccessible over the network, we’ll still be able to read the files.

DataNodes are constantly reporting to the NameNode. Each of the DataNodes informs the NameNode of the blocks it’s currently storing. After this mapping is complete, the DataNodes continually poll the NameNode to provide information regarding local changes as well as receive instructions to create, move, or delete blocks from the local disk.

3. Secondary NameNode: The Secondary NameNode (SNN) is an assistant daemon for monitoring the state of the cluster HDFS. Like the NameNode, each cluster has one SNN, and it typically resides on its own machine as well. No other DataNode or TaskTracker daemons run on the same server. The SNN differs from the NameNode in that this process doesn't receive or record any real-time changes to HDFS. Instead, it communicates with the NameNode to take snapshots of the HDFS metadata at intervals defined by the cluster configuration.

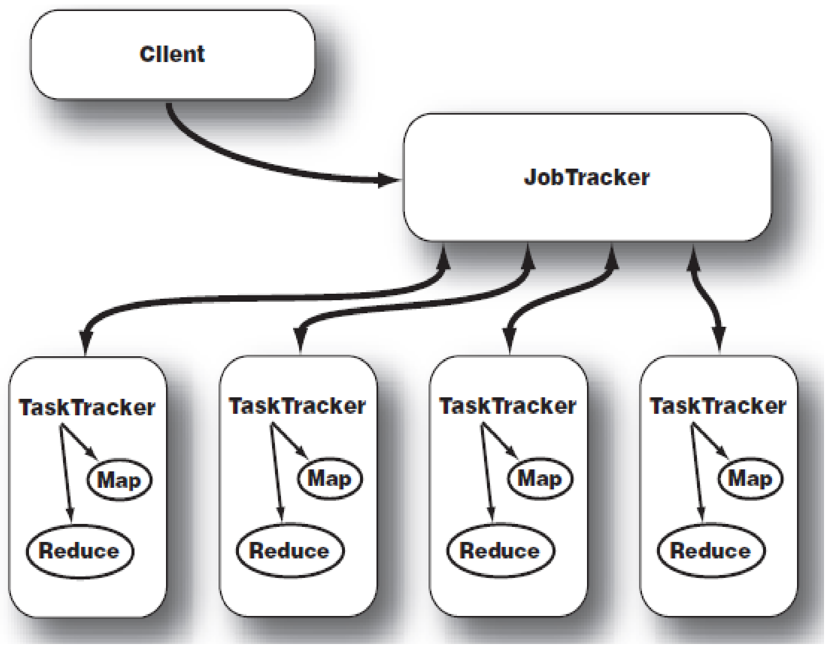
The NameNode is a single point of failure for a Hadoop cluster, and the SNNsnapshots help minimize the downtime and loss of data. However, a NameNode failure requires human intervention to reconfigure the cluster to use the SNN as the primary NameNode.

4. JobTracker: There is only one JobTracker daemon per Hadoop cluster. It's typically run on a server as a master node of the cluster. The JobTracker daemon is the link between our application and Hadoop. Once we submit our code to the cluster, the JobTracker determines the execution plan by determining which files to process, assigns nodes to different tasks, and monitors all tasks as they're running. If a task fails, the JobTracker will automatically relaunch the task, possibly on a different node, up to a predefined limit of retries.

5. TaskTracker: The JobTracker is the master control for overall execution of a MapReduce job and the TaskTrackers manage the execution of individual tasks on each slave node. The interaction between JobTracker and TaskTracker is shown in the following diagram.

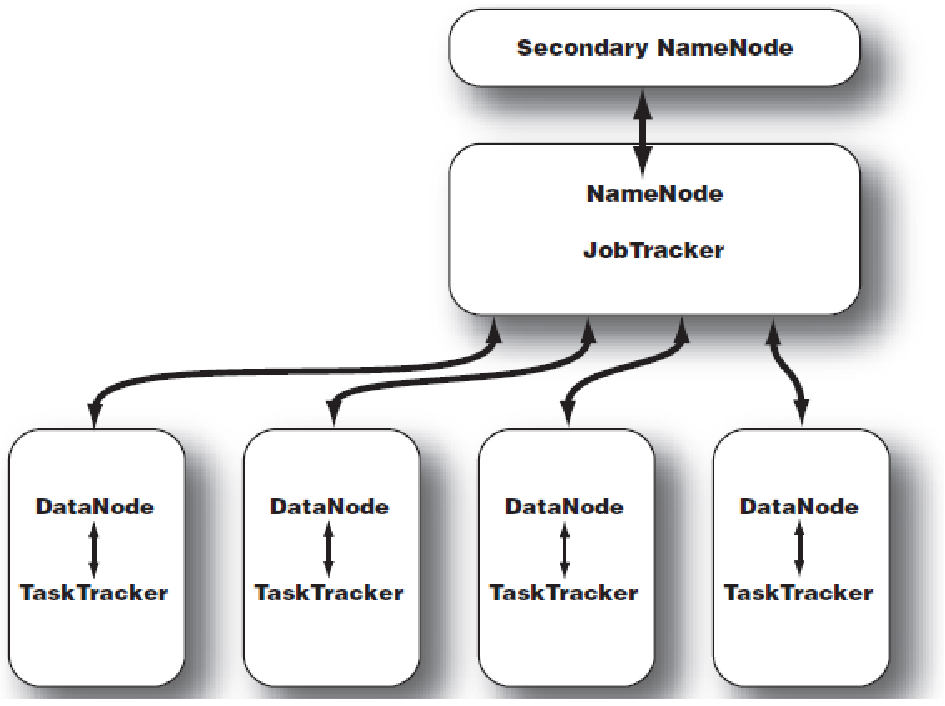
Each TaskTracker is responsible for executing the individual tasks that the JobTracker assigns. Although there is a single TaskTracker per slave node, each TaskTracker can spawn multiple JVMs to handle many maps or reduce tasks in parallel.

One responsibility of the TaskTracker is to constantly communicate with the JobTracker. If the JobTracker fails to receive a heartbeat from a TaskTracker within a specified amount of time, it will assume the TaskTracker has crashed and will resubmit the corresponding tasks to other nodes in the cluster.



JobTracker and TaskTracker interaction. After a client calls the JobTracker to begin a data processing job, the JobTracker partitions the work and assigns different map and reduce tasks to each TaskTracker in the cluster

The topology of one typical Hadoop cluster in described in the following figure:



Topology of a typical Hadoop cluster. It's a master/slave architecture in which the NameNode and JobTracker are masters and the DataNodes and TaskTrackers are slaves.

UNIT - II

1. Real Time Analytics: Examples, What is Apache Spark, Why Spark when Hadoop is there, Spark Features, Getting started with Spark, Spark Eco System, Architecture and its working, Data Structures of Spark, Spark components, Using Spark with Hadoop, Usecase.

2. MapReduce Programming: Writing basic Map Reduce programs - Getting the patent data set, constructing the basic template of a Map Reduce program, Counting things, Programming with RDDs-Basics.

Real Time Analytics

Real-time analytics refers to the practice of collecting and analyzing streaming data as it is generated, with minimal latency between the generation of data and the analysis of that data. Real-time analytics is often used in applications where the timeliness of the data is critical, such as personalized advertisements or offers, smart pricing, or predictive maintenance. Real-time analytics is built on the foundational capability of data streaming.

Who Uses Real-Time Analytics in an Organization?

Real-time analytics can be used by various stakeholders across an organization, depending on their roles and responsibilities. Here are some examples of who typically uses real-time analytics:

1. **Data Analysts:** Responsible for creating reports and analyzing data to provide insights and actionable information to decision-makers in real time.
2. **Business Managers:** Use real-time analytics to monitor key performance indicators (KPIs) and make data-driven decisions to improve business operations.
3. **Operations Managers:** Use real-time analytics to monitor and optimize production processes, supply chain logistics and customer service.
4. **IT Managers:** Use real-time analytics to monitor system performance, identify and mitigate cybersecurity risks, and ensure business continuity.
5. **Marketing Managers:** Use real-time analytics to monitor social media activity, track customer engagement and adjust marketing strategies.
6. **Customer Service Managers:** Use real-time analytics to monitor customer feedback, identify trends, and respond to customer inquiries and complaints.

What is Apache Spark?

–Apache Spark is a cluster computing platform designed to be fast and general purpose.‖

Apache Spark is an open-source distributed cluster-computing framework. Spark is a data processing engine developed to provide faster and easy-to-use analytics than Hadoop MapReduce.

Why Spark when Hadoop is there?

1. **In-memory Processing:** In-memory processing is faster when compared to Hadoop, as there is no time spent in moving data/processes in and out of the disk. Spark is 100 times faster than MapReduce as everything is done here in memory. We should look at Hadoop as a general purpose Framework that supports multiple models and we should look at Spark as an alternative to Hadoop MapReduce rather than a replacement to Hadoop.
2. **Stream Processing:** Apache Spark supports stream processing, which involves continuous input and output of data. Stream processing is also called real-time processing.
3. Spark stores data in-memory whereas Hadoop stores data on disk. Hadoop uses replication to achieve fault tolerance whereas Spark uses different data storage model, resilient distributed datasets (RDD), uses a clever way of guaranteeing fault tolerance that minimizes network I/O.
4. **Lazy Evaluation:** Apache Spark starts evaluating only when it is absolutely needed. This plays an important role in contributing to its speed.
5. **Less Lines of Code:** Although Spark is written in both Scala and Java, the implementation is in Scala, so the number of lines is relatively lesser in Spark when compared to Hadoop.

Hadoop Vs. Spark:

1. **Low Processing Speed:** In Hadoop, the MapReduce algorithm, which is a parallel and distributed algorithm, processes really large datasets. These are the tasks need to be performed here.
2. **Map:** Map takes some amount of data as input and converts it into another set of data, which again is divided into key/value pairs.
3. **Reduce:** The output of the Map task is fed into Reduce as input. In the Reduce task, as the name suggests, those key/value pairs are combined into a smaller set of tuples. The Reduce task is always done after Mapping.

4. **Batch Processing:** Hadoop deploys batch processing, which is collecting data and then processing it in bulk later. Although batch processing is efficient for processing high volumes of data, it does not process streamed data. Because of this, the performance is lower.
5. **No Data Pipelining:** Hadoop does not support data pipelining (i.e., a sequence of stages where the previous stage‘s output ID is the next stage‘s input).
6. **Not Easy to Use:** MapReduce developers need to write their own code for each and every operation, which makes it really difficult to work with. And also, MapReduce has no interactive mode.
7. **Latency:** In Hadoop, the MapReduce framework is slower, since it supports different formats, structures, and huge volumes of data.
8. **Lengthy Line of Code:** Since Hadoop is written in Java, the code is lengthy. And, this takes more time to execute the program.

Having outlined all these drawbacks of Hadoop, it is clear that there was a scope for improvement, which is why Spark was introduced.

Spark Features:

The features that make Spark one of the most extensively used Big Data platforms are:

1. Lighting-fast processing speed: Big Data processing is all about processing large volumes of complex data. Hence, when it comes to Big Data processing, organizations and enterprises want such frameworks that can process massive amounts of data at high speed. As we mentioned earlier, Spark apps can run up to 100x faster in memory and 10x faster on disk in Hadoop clusters.

It relies on Resilient Distributed Dataset (RDD) that allows Spark to transparently store data on memory and read/write it to disc only if needed. This helps to reduce most of the disc read and write time during data processing.

2. Ease of use: Spark allows you to write scalable applications in Java, Scala, Python, and R, so developers get the scope to create and run Spark applications in their preferred programming languages. Moreover, Spark is equipped with a built-in set of over 80 high- level operators. You can use Spark interactively to query data from Scala, Python, R, and SQL shells.

3. It offers support for sophisticated analytics: Not only does Spark support simple ―map‖ and ―reduce‖ operations, but it also supports SQL queries, streaming data, and advanced analytics, including ML and graph algorithms. It comes with a powerful stack of libraries such as SQL & DataFrames and MLlib (for ML), GraphX, and Spark Streaming. What‗s fascinating is that Spark lets you combine the capabilities of all these libraries within a single workflow/application.

4. Real-time stream processing: Spark is designed to handle real-time data streaming. While MapReduce is built to handle and process the data that is already stored in Hadoop clusters, Spark can do both and also manipulate data in real-time via Spark Streaming.

Unlike other streaming solutions, Spark Streaming can recover the lost work and deliver the exact semantics out-of-the-box without requiring extra code or configuration. Plus, it also lets you reuse the same code for batch and stream processing and even for joining streaming data to historical data.

5. It is flexible: Spark can run independently in cluster mode, and it can also run on Hadoop YARN (**Y**et **A**nother **R**esource **N**egotiator is a resource manager of Hadoop is created by splitting the processing engine and the management function of MapReduce), Apache Mesos, Kubernetes, and even in the cloud. Furthermore, it can access diverse data sources. For instance, Spark can run on the YARN cluster manager and read any existing Hadoop data. It can read from any Hadoop data sources like HBase, HDFS, Hive, and Cassandra. This aspect of Spark makes it an ideal tool for migrating pure Hadoop applications, provided the apps‗ use-case is Spark-friendly.

6. Active and expanding community: Developers from over 300 companies have contributed to design and build Apache Spark. Ever since 2009, more than 1200 developers have actively contributed to making Spark what it is today! Naturally, Spark is backed by an active community of developers who work to improve its features and performance continually. To reach out to the Spark community, you can make use of mailing lists for any queries, and you can also attend Spark meet up groups and conferences.

Getting started with Spark: Installing Spark on Windows 10:

Step I: Install Scala

1. Install Scala: Download Scala from the following link:

http://downloads.lightbend.com/scala/2.11.8/scala- 2.11.8.msi

a. Set environmental variables:

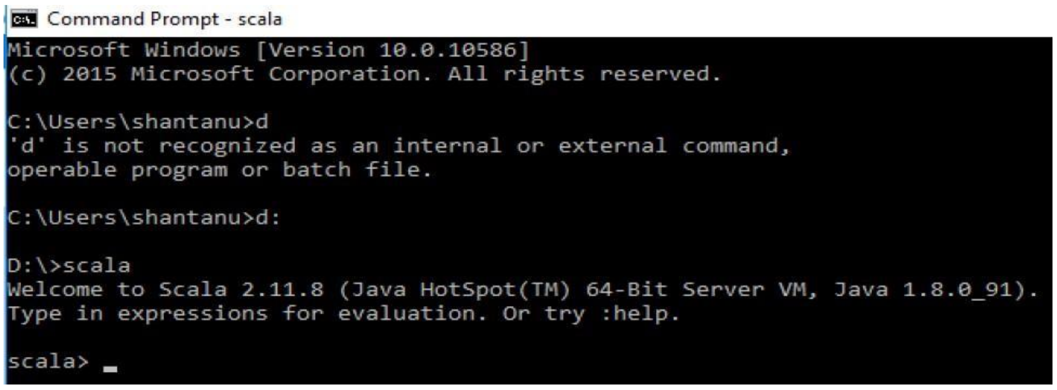
i. User variable:

- ➢ Variable: SCALA_HOME
- ➢ Value: C:\Program Files (x86)\scala

ii. System variable:

- ➢ Variable: PATH
- ➢ Value: C:\Program Files (x86)\scala\bin

b. Check it on cmd, see below.



```
Microsoft Windows [Version 10.0.10586]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\Users\shantanu>d
'd' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\shantanu>d:

D:\>scala
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_91).
Type in expressions for evaluation. Or try :help.

scala> _
```

Step II: Install Java

1. Install Java 8: Download Java 8 from the link and install it:

http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

a. Set environmental variables:

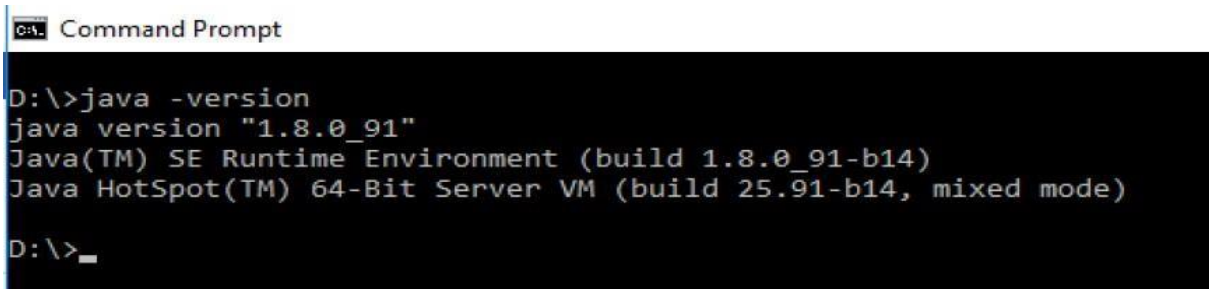
i. User variable:

- ➢ Variable: JAVA_HOME
- ➢ Value: C:\Program Files\Java\jdk1.8.0_91

ii. System variable:

- ➢ Variable: PATH
- ➢ Value: C:\Program Files\Java\jdk1.8.0_91\bin

b. Check on cmd, see below:



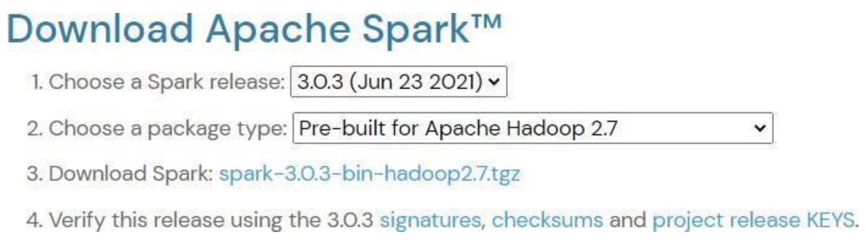
Command Prompt

```
D:\>java -version
java version "1.8.0_91"
Java(TM) SE Runtime Environment (build 1.8.0_91-b14)
Java HotSpot(TM) 64-Bit Server VM (build 25.91-b14, mixed mode)

D:\>_
```

Step III: Install Apache Spark

1. Download the Apache Spark form the following link:

https://spark.apache.org/downloads.html



Download Apache Spark™

1. Choose a Spark release: 3.0.3 (Jun 23 2021)
2. Choose a package type: Pre-built for Apache Hadoop 2.7
3. Download Spark: spark-3.0.3-bin-hadoop2.7.tgz
4. Verify this release using the 3.0.3 signatures, checksums and project release KEYS.

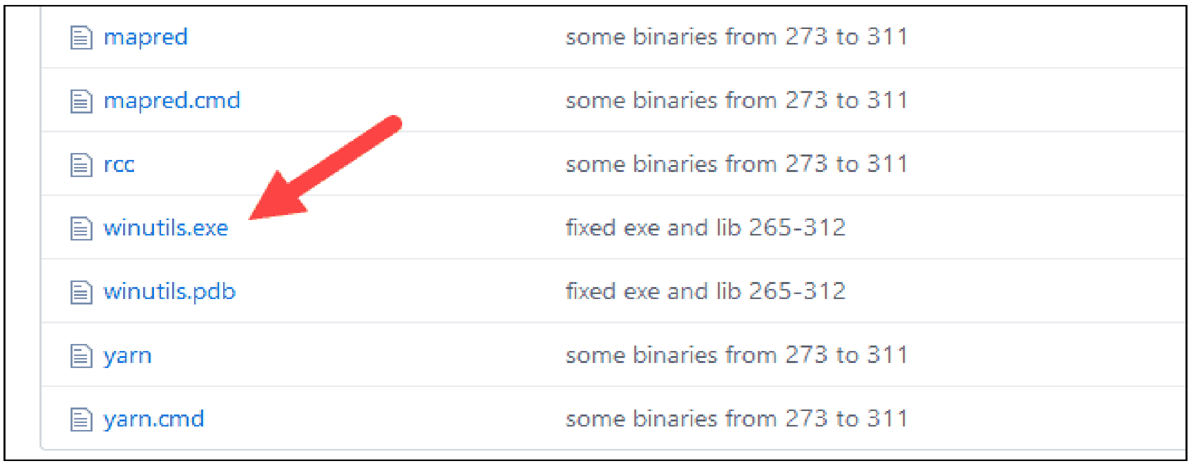
2. Right-click the file and extract it to *C:\* drive or any drive using the tool you have on your system (e.g., 7-Zip/Winrar).

3. You can find the folder name as ―C:\spark-3.0.3-bin-hadoop2.7‖. I stored in my system in ―E drive‖.

Add winutils.exe File

Download the **winutils.exe** file for the underlying Hadoop version for the Spark installation you downloaded.

1. Navigate to this URL https://github.com/cdarlint/winutils and inside the **bin** folder, locate **winutils.exe**, and click it.



mapred	some binaries from 273 to 311
mapred.cmd	some binaries from 273 to 311
rcc	some binaries from 273 to 311
winutils.exe	fixed exe and lib 265-312
winutils.pdb	fixed exe and lib 265-312
yarn	some binaries from 273 to 311
yarn.cmd	some binaries from 273 to 311

2. Find the **Download** button on the right side to download the file.

3. Copy the winutils.exe file from the Downloads folder to **C:\spark-3.0.3-bin-hadoop2.7\bin**.

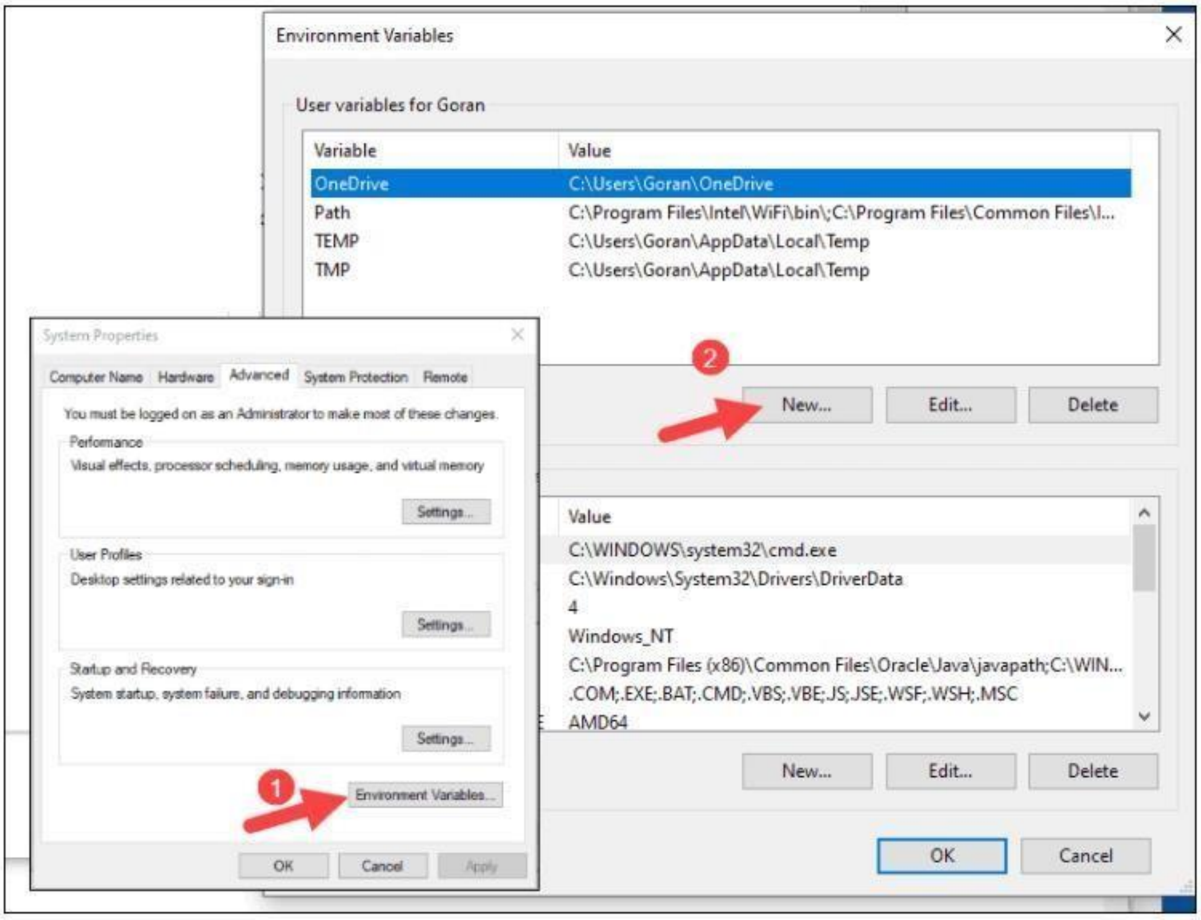
Step IV: Configure Environment Variables

Configuring environment variables in Windows adds the Spark and Hadoop locations to your system PATH. It allows you to run the Spark shell directly from a command prompt window.

1. Click **Start** and type *environment*.

2. Select the result labeled ***Edit the system environment variables***.

3. A System Properties dialog box appears. In the lower-right corner, click **Environment Variables** and then click **New** in the next window.



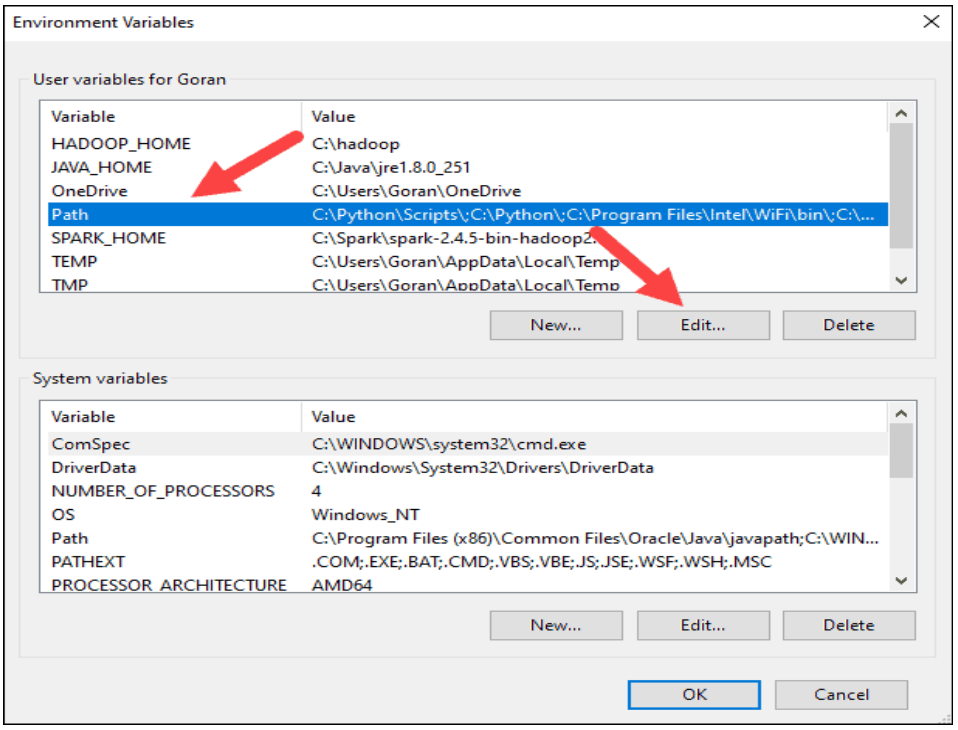
Variable	Value
OneDrive	C:\Users\Goran\OneDrive
Path	C:\Program Files\Intel\WiFi\bin\;C:\Program Files\Common Files\I...
TEMP	C:\Users\Goran\AppData\Local\Temp
TMP	C:\Users\Goran\AppData\Local\Temp

Value
C:\WINDOWS\system32\cmd.exe
C:\Windows\System32\Drivers\DriverData
4
Windows_NT
C:\Program Files (x86)\Common Files\Oracle\Java\javapath;C:\WIN...
.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
AMD64

4. For *Variable Name* type ***SPARK_HOME***.

5. For *Variable Value* type **E:\spark-3.0.3-bin-hadoop2.7\bin** and click OK. If you changed the folder path, use that one instead.

6. In the top box, click the **Path** entry, then click **Edit**. Be careful with editing the system path. Avoid deleting any entries already on the list.


Environment Variables

User variables for Goran

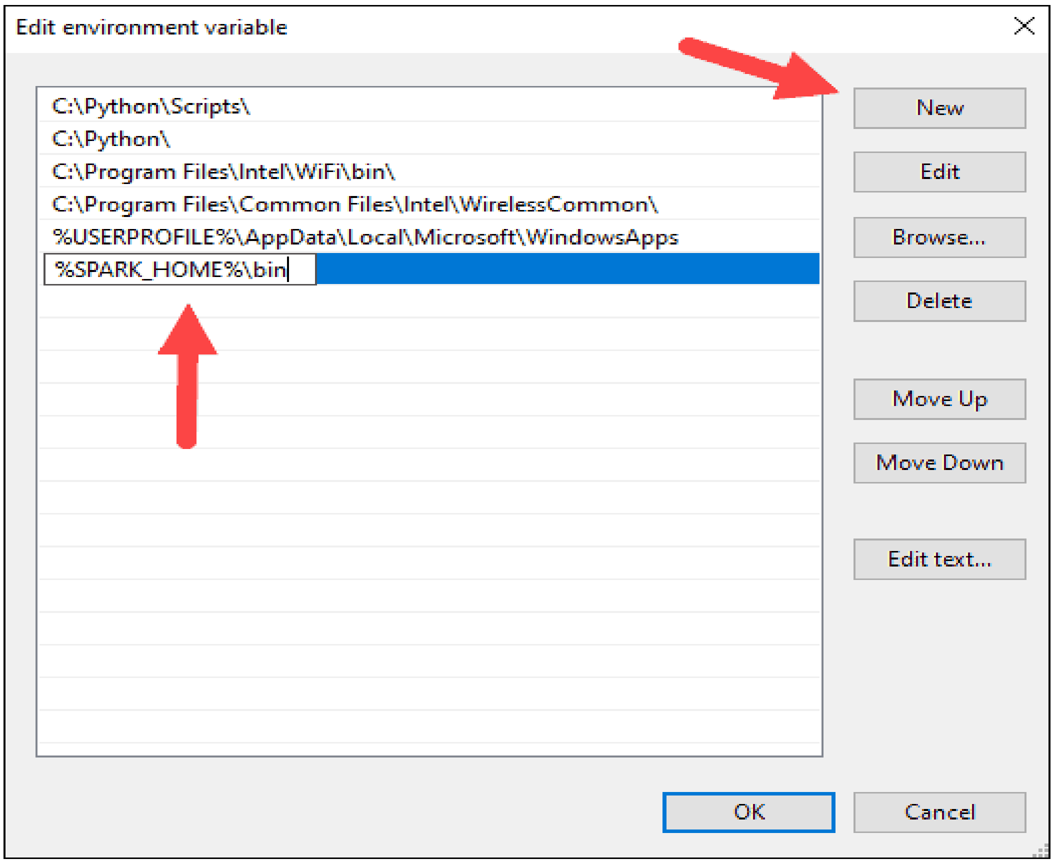
Variable	Value
HADOOP_HOME	C:\hadoop
JAVA_HOME	C:\Java\jre1.8.0_251
OneDrive	C:\Users\Goran\OneDrive
Path	C:\Python\Scripts\;C:\Python\;C:\Program Files\Intel\WiFi\bin\;C:\...
SPARK_HOME	C:\Spark\spark-2.4.5-bin-hadoop2.7
TEMP	C:\Users\Goran\AppData\Local\Temp
TMP	C:\Users\Goran\AppData\Local\Temp

System variables

Variable	Value
ComSpec	C:\WINDOWS\system32\cmd.exe
DriverData	C:\Windows\System32\Drivers\DriverData
NUMBER_OF_PROCESSORS	4
OS	Windows_NT
Path	C:\Program Files (x86)\Common Files\Oracle\Java\javapath;C:\WIN...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
PROCESSOR_ARCHITECTURE	AMD64

7. You should see a box with entries on the left. On the right, click **New**.

8. The system highlights a new line. Enter the path to the Spark folder **E:\spark-3.0.3-bin-hadoop2.7\bin**. We recommend using **%SPARK_HOME%\bin** to avoid possible issues with the path.


Edit environment variable

```
C:\Python\Scripts\
C:\Python\
C:\Program Files\Intel\WiFi\bin\
C:\Program Files\Common Files\Intel\WirelessCommon\
%USERPROFILE%\AppData\Local\Microsoft\WindowsApps
%SPARK_HOME%\bin
```

9. Repeat this process for Hadoop and Java.

- For Hadoop, the variable name is **HADOOP_HOME** and for the value use the path of the folder you created earlier: **C:\hadoop.** Add **C:\hadoop\bin** to the **Path variable** field, but we recommend using **%HADOOP_HOME%\bin**.
- For Java, the variable name is **JAVA_HOME** and for the value use the path to your Java JDK directory (in our case it's **C:\Program Files\Java\jdk1.8.0_251**).

10. Click **OK** to close all open windows.

Step V: Launch Spark

1. Open a new command-prompt window using the right-click and **Run as administrator**:

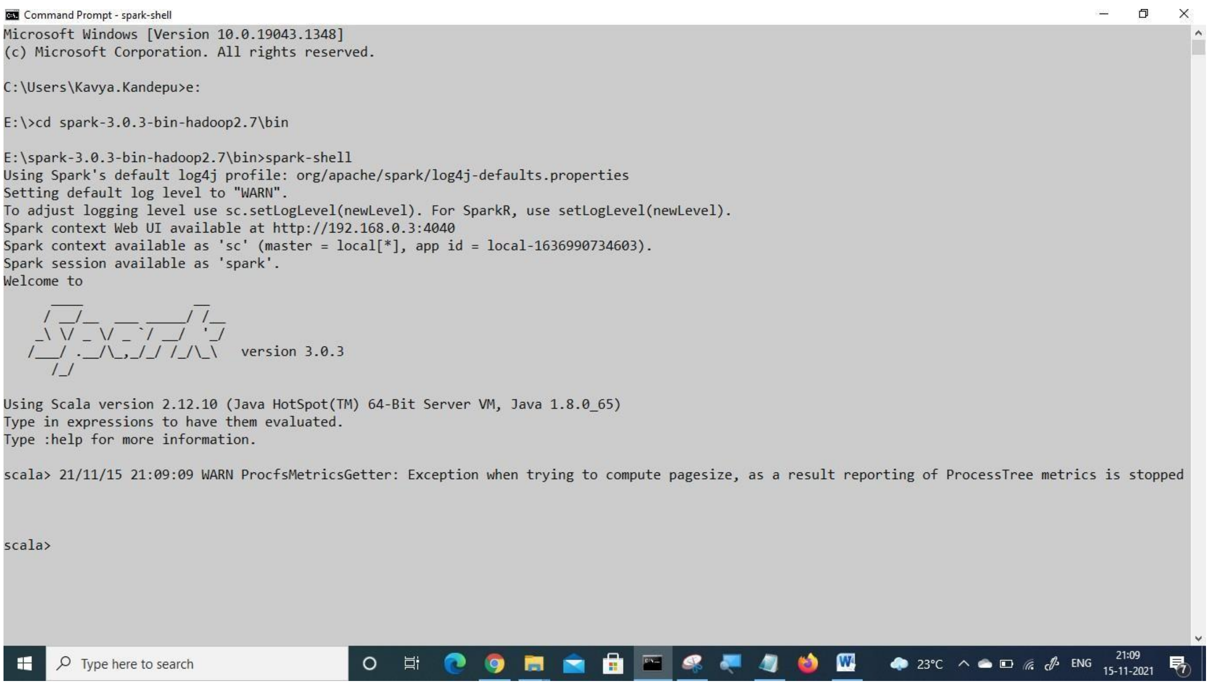
2. To start Spark, enter:

E:\spark-3.0.3-bin-hadoop2.7\bin\spark-shell

If you set the **environment path** correctly, you can type **spark-shell** to launch Spark.

3. The system should display several lines indicating the status of the application. You may get a Java pop-up. Select **Allow access** to continue.

Finally, the Spark logo appears, and the prompt displays the **Scala shell**.

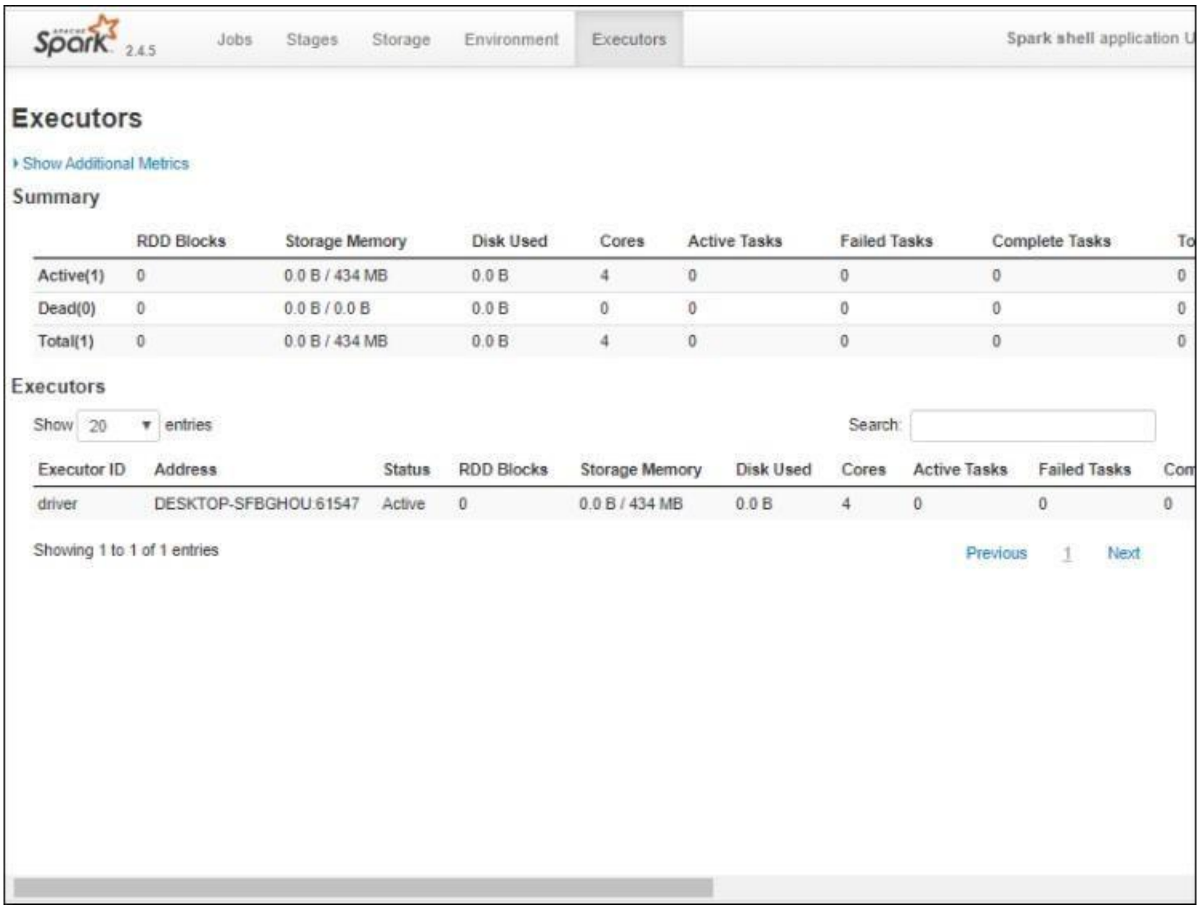


```
Command Prompt - spark-shell
Microsoft Windows [Version 10.0.19043.1348]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Kavya.Kandepu>e:

E:\>cd spark-3.0.3-bin-hadoop2.7\bin

E:\spark-3.0.3-bin-hadoop2.7\bin>spark-shell
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Spark context Web UI available at http://192.168.0.3:4040
Spark context available as 'sc' (master = local[*], app id = local-1636990734603).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 3.0.3
      /_/

Using Scala version 2.12.10 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_65)
Type in expressions to have them evaluated.
Type :help for more information.

scala> 21/11/15 21:09:09 WARN ProcfsMetricsGetter: Exception when trying to compute pagesize, as a result reporting of ProcessTree metrics is stopped


scala>
```

4. Open a web browser and navigate to **http://localhost:4040/**.

5. You can replace **localhost** with the name of your system.

6. You should see an Apache Spark shell Web UI. The example below shows the *Executors* page.



Executors

Show Additional Metrics

Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	To
Active(1)	0	0.0 B / 434 MB	0.0 B	4	0	0	0	0
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0
Total(1)	0	0.0 B / 434 MB	0.0 B	4	0	0	0	0

Executors

Show 20 entries Search:

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Com
driver	DESKTOP-SFBGHOU:61547	Active	0	0.0 B / 434 MB	0.0 B	4	0	0	0

Showing 1 to 1 of 1 entries

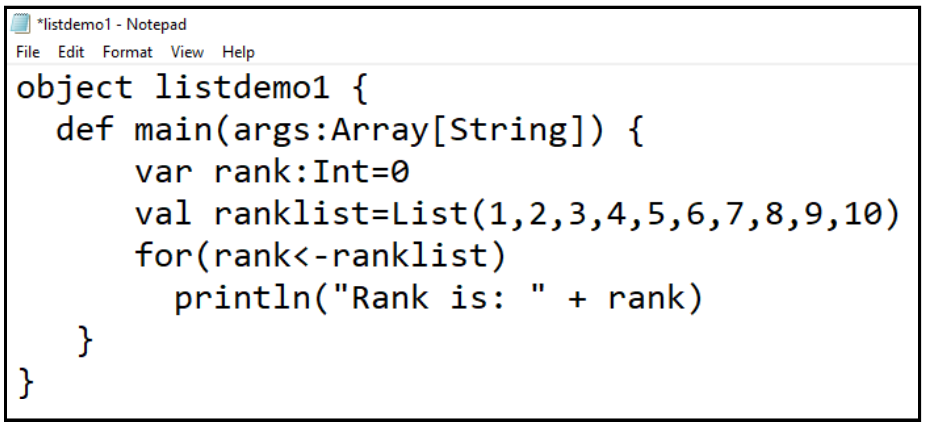
Previous 1 Next

7. To exit Spark and close the Scala shell, press **ctrl-d** in the command-prompt window.

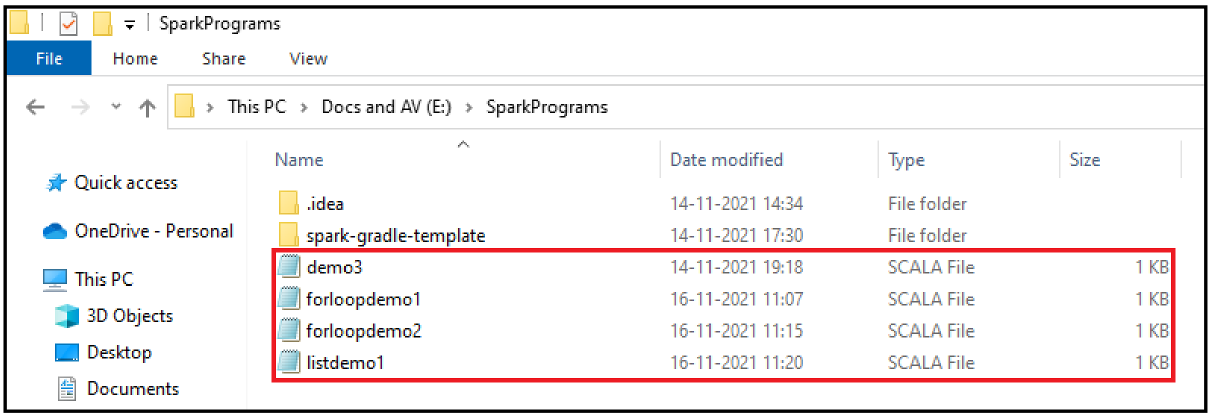
Step VI: Execution of Scala Program

1. Scala program can execute in two methods. a) Text Editor (eg. Notepad) & Command Prompt b) Intellij IDEA. Here explained the first method only.

2. Open the Notepad and type the code and save it with ***„.scala‟*** extension. For example,

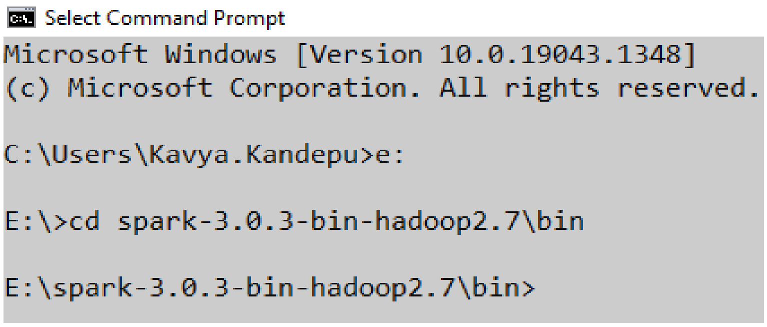


```
object listdemo1 {
  def main(args:Array[String]) {
      var rank:Int=0
      val ranklist=List(1,2,3,4,5,6,7,8,9,10)
      for(rank<-ranklist)
        println("Rank is: " + rank)
   }
}
```



Name	Date modified	Type	Size
.idea	14-11-2021 14:34	File folder	
spark-gradle-template	14-11-2021 17:30	File folder	
demo3	14-11-2021 19:18	SCALA File	1 KB
forloopdemo1	16-11-2021 11:07	SCALA File	1 KB
forloopdemo2	16-11-2021 11:15	SCALA File	1 KB
listdemo1	16-11-2021 11:20	SCALA File	1 KB

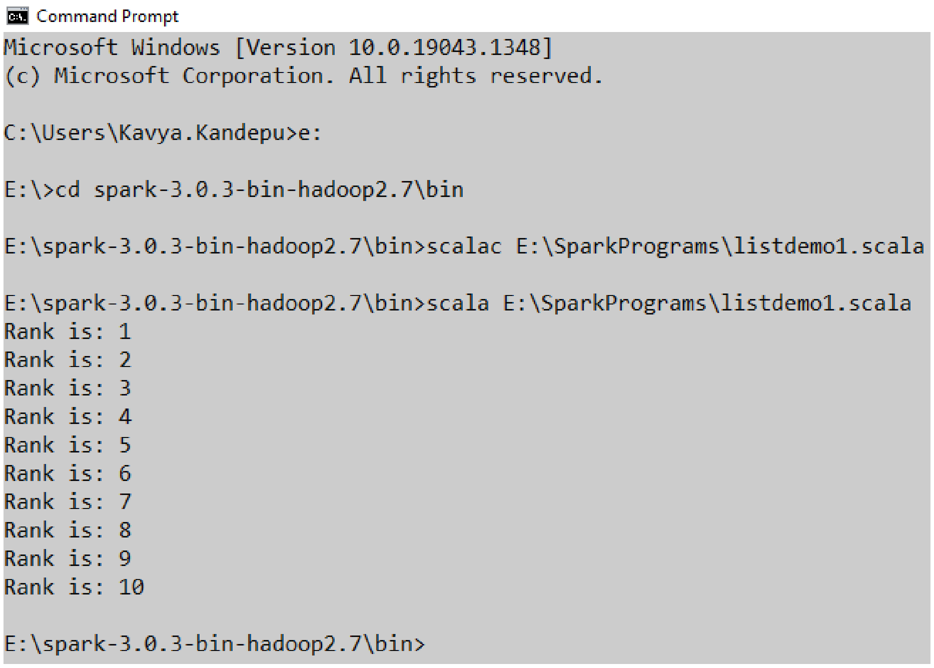
3. Open the command prompt and change the directory to where –Spark software‖ is stored.



```
Microsoft Windows [Version 10.0.19043.1348]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Kavya.Kandepu>e:

E:\>cd spark-3.0.3-bin-hadoop2.7\bin

E:\spark-3.0.3-bin-hadoop2.7\bin>
```

4. Compile using „***scalac***" command and if no errors, execute it. Otherwise, correct the errors, save & compile once again. Does this until all errors should be cleared.

5. Execute using „***scala***"command.



```
Microsoft Windows [Version 10.0.19043.1348]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Kavya.Kandepu>e:

E:\>cd spark-3.0.3-bin-hadoop2.7\bin

E:\spark-3.0.3-bin-hadoop2.7\bin>scalac E:\SparkPrograms\listdemo1.scala

E:\spark-3.0.3-bin-hadoop2.7\bin>scala E:\SparkPrograms\listdemo1.scala
Rank is: 1
Rank is: 2
Rank is: 3
Rank is: 4
Rank is: 5
Rank is: 6
Rank is: 7
Rank is: 8
Rank is: 9
Rank is: 10

E:\spark-3.0.3-bin-hadoop2.7\bin>
```

Note 1: Once the ***Scala*** program compiled successfully, ***".class"*** file will be created in the ***"bin folder of Spark"***. To execute Scala program in the ***"bin"*** folder, the file name should not extended by ***".scala"***.

Note 2:

- ❖ To open the Python version of the Spark shell, which we also refer to as the PySpark Shell, go into your Spark directory and type:

  bin\pyspark

- ❖ To open the Scala version of the shell, type:

  bin\spark-shell

- ❖ The shell prompt should appear within a few seconds. When the shell starts, you will notice a lot of log messages. You may need to press Enter once to clear the log output and get to a shell prompt.
- ❖ To exit either shell, press Ctrl-D.

<u>Steps to Install "IntelliJ IDEA" for Spark-Scala</u>

1. Download –**Git**‖ for Windows 10 and install it.

https://git-scm.com/downloads

2. Download **IntelliJ IDEA "Community"** version and install it.

https://www.jetbrains.com/idea/download/#section=windows

3. Create a folder to store the Spark Programs and open it.

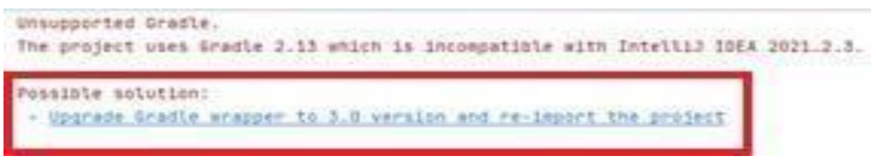
4. Open the properties by keeping the mouse pointer within the folder and select –git Bash here‖.

5. Type or copy and paste the following command and press enter. This will create a folder name as —spark-gradle-template‖.

```
$ git clone https://github.com/faizanahemad/spark-gradle-template.git
```

6. Open IntelliJ IDE and Select File→New→Project from Existing Sources→Browse & Select the folder created in Step: 4 and click on –OK‖.

7. If it displays –Upgrade…‖ in the bottom of the window (Build window), click on it and wait until update is over.



```
Unsupported Gradle.
The project uses Gradle 2.13 which is incompatible with IntelliJ IDEA 2021.2.3.
Possible solution:
 - Upgrade Gradle wrapper to 3.0 version and re-import the project
```

8. Open the project→SRC→main→scala→open properties→New→Package→Type the Package Name and click on OK.

9. The User-defined Package will be created under the Scala directory.

10. Select the Package→Open Properties→New→Scala Class→select –Scala Object‖→Type the name→Press Enter Key.

11. The file will be created with ―.scala‖ extension and open it.

12. Open the newly created file and type the code.

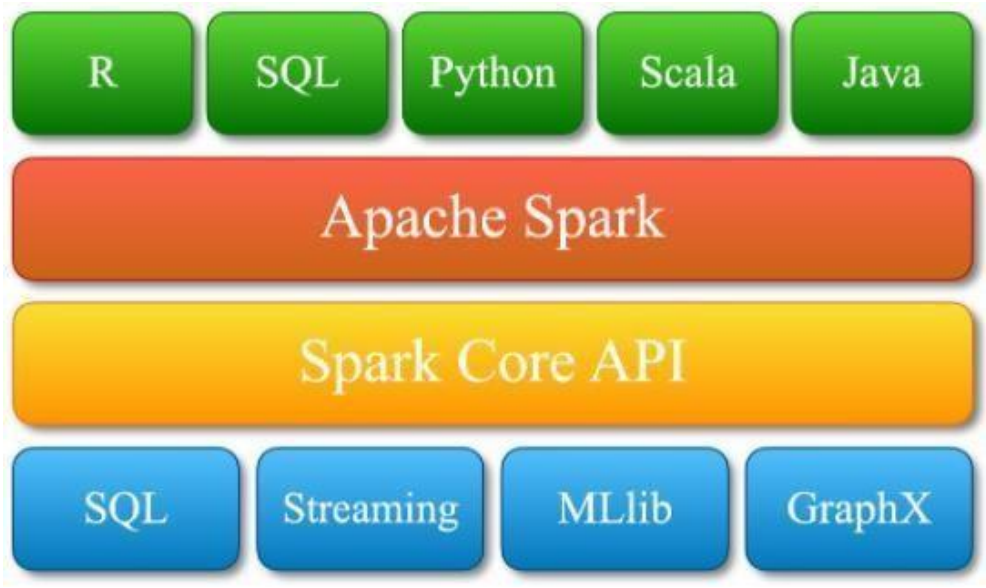
13. Import the required classes to the source code and save the file.

14. Build (Run/Execute) – Keep the mouse pointer in the program editor, open properties and select ―Run file (defined by you)‖.

Spark Eco System (Spark Components):

The Apache Spark ecosystem is an open-source distributed cluster-computing framework. Spark is a data processing engine developed to provide faster and easier analytics than Hadoop MapReduce.

Apache Spark provides a robust set of components to perform different types of operations such as GraphX, Spark Streaming, Spark Core, SparkR, Spark SQL, and MLlib. This is shown in the following figure.



1. Spark Core:

Apache Spark Core is an execution model that is responsible for job scheduling, memory management, interaction with a storage system, and so on. It supports multiple application and its core functionality are access using Java, Python, or Scala APIs.

2. Spark SQL:

Apache Spark SQL is a component of Spark that is used to process structured and semistructured dataset. Spark SQL supports DSL(Domain Specific Language) language to perform different kinds of operations. Spark SQL allows querying structured data inside Spark programs, using either SQL or a familiar DataFrame API. It is usable in Java, Scala, Python, and R.

3. Spark Streaming:

Apache Spark Streaming is the core component of Spark that is used to perform streaming processing for live data. Spark streaming process input data in mini-batches and apply RDD transformations on mini-batches. The source of input data would be Kinesis, Flume, Kafka, and so on.

4. MLlib: Machine Learning Library:

Apache Spark MLlib is a distributed machine-learning framework of Spark. It is designed to make practical machine learning scalable and easy. There are some common machine learning and statistical algorithms implemented with Spark MLlib such as Summary Statistics, Stratified Sampling, Correlations, Hypothesis Testing, Random Data Generation, and Kernel Density Estimation.

5. GraphX:

Apache Spark GraphX is a component of Spark architecture that is used for graph processing. GraphX can extend the Spark RDD that is immutable. Some of the Apache Spark GraphX algorithms are PageRank, Connected components, Label propagation, SVD++, Strongly connected components, Triangle count.

6. SparkR (R on Spark):

Apache SparkR is a Spark component to use R programming language with Spark. It supports different types of operations on data frames such as filtering, aggregation, selection, and so on.

Spark Architecture and its working:

Apache Spark has a well-defined layered architecture where all the spark components and layers are loosely coupled. This architecture is further integrated with various extensions and libraries. Apache Spark Architecture is based on two main abstractions:

1. Resilient Distributed Dataset (RDD)
2. Directed Acyclic Graph (DAG)

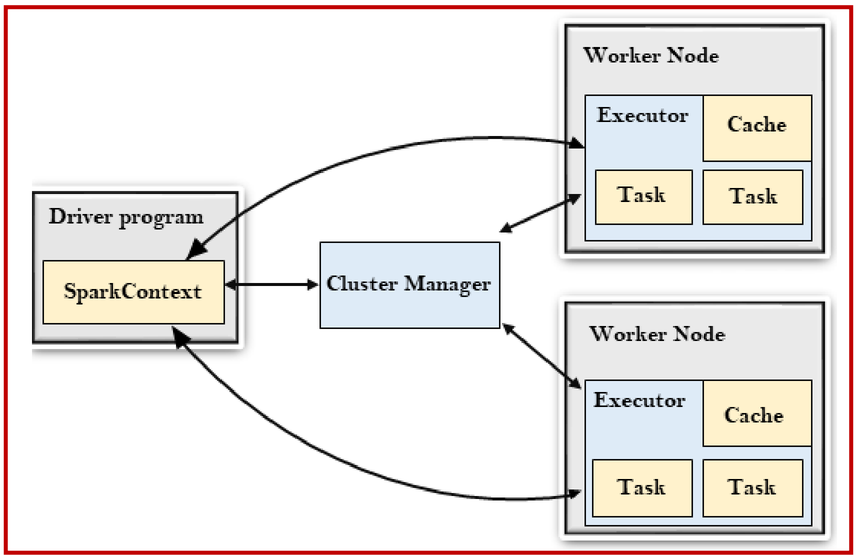
1. Resilient Distributed Datasets (RDD):

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

- Resilient: Restore the data on failure.
- Distributed: Data is distributed among different nodes.
- Dataset: Group of data.

2. Directed Acyclic Graph (DAG):

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



Spark Architecture

a) Driver Program:

The Driver Program is a process that runs the main() function of the application and creates the SparkContext object. The purpose of SparkContext is to coordinate the spark applications, running as independent sets of processes on a cluster. To run on a cluster, the SparkContext connects to a different type of cluster managers and then perform the following tasks:

- It acquires executors on nodes in the cluster.
- Then, it sends your application code to the executors. Here, the application code can be defined by JAR or Python files passed to the SparkContext.
- At last, the SparkContext sends tasks to the executors to run.

b) Cluster Manager:

- The role of the cluster manager is to allocate resources across applications. The Spark is capable enough of running on a large number of clusters.
- It consists of various types of cluster managers such as Hadoop YARN, Apache Mesos and Standalone Scheduler.
- Here, the Standalone Scheduler is a standalone spark cluster manager that facilitates to install Spark on an empty set of machines.

c) Worker Node:

- The worker node is a slave node
- Its role is to run the application code in the cluster.

d) Executor:

- An executor is a process launched for an application on a worker node.
- It runs tasks and keeps data in memory or disk storage across them.
- It read and writes data to the external sources.
- Every application contains its executor.

e) Task:

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

Data Structures of Spark:

RDD, DataFrame, and Dataset are the three most common data structures in Spark, and they make processing very large data easy and convenient. Because of the lazy evaluation algorithm of Spark, these data structures are not executed right way during creations, transformations, and functions etc.

1. Spark RDD (since Spark 1.0):

RDD stands for Resilient Distributed Dataset. It is a collection of recorded immutable partitions. RDD is the fundamental data structure of Spark whose partitions are shuffled, sent across nodes and operated in parallel. It allows programmers to perform complex in-memory analysis on large clusters in a fault-tolerant manner. RDD can handle structured and unstructured data easily and effectively as it has lots of built-in functional operators like group, map and filter etc.

However, when encountering complex logic, RDD has a very obvious disadvantage – operators cannot be re-used. This is because RDD does not know the information of the stored data, so the structure of the data is a black box which requires a user to write a very specific aggregation function to complete an execution. Therefore, RDD is preferable on unstructured data, to be used for low-level transformations and actions.

RDD provides users with a familiar object-oriented programming style, along with a distributing collection of JVM objects that indicate it is compile-time type safety. Using RDD is very flexible as it provides Java, Scala, Python and R APIs. But there is a big limitation on RDD, it cannot be used within Spark SQL as it does not have optimizations for special scenarios.

2. Spark DataFrame (since Spark 1.3):

DataFrame is a distributed dataset based on RDD which organizes the data in named columns before Spark 2.0. It is similar to a two-dimensional table in the relational database, so it introduces the database‘s schema. Because of that, it can be treated as an optimization ontop of RDD – for example, RDD knows the structure of the stored data, which allows users to perform high-level operations. With respect to that, it handles structured and semi-structured data. Users can be specific on which column to perform what operations. This allows an operator to be used into multiple columns and makes an operator reusable.

DataFrame also makes revamping operations easier and flexible. If users have extra requires on the existing operation that needs to include another column into the operation, users can just write an operator for that extra column and add it into the existing operation. Whereas, RDD needs to make a lots of changes on the existing aggregation. Compared to RDD, DataFrame does not provide compile-time type safety as it is a distributed collection ofRow objects. Like RDD, DataFrame also supports various APIs. Unlike RDD, DataFrame is able to be used with Spark SQL as the structure of data it stores, so it can provide more functional operators and allow users to perform expression-based operations and UDFs. Last but not least, it can enhance the execution efficiency, reduce the cost of loading the data, and optimize the logical plans.

3. Spark Dataset (since Spark 1.6):

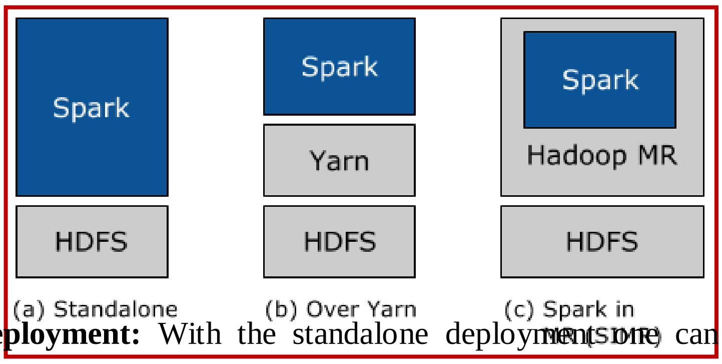
Dataset API is like an extension and enhancement of DataFrame API. Externally, Dataset is a collection of JVM objects. Internally, Dataset has an un-typed view called a DataFrame, which is a Dataset of Row since Spark 2.0. Dataset merges the advantages of RDD and DataFrame. Like RDD, it supports structured, unstructured and custom data storing, and it provides users an object-oriented programming style and compile-time type safety. Like DataFrame, it takes advantages of the Catalyst optimizer to allow users to perform structured SQL queries on data, but it is slower than DataFrame. Unlike RDD and DataFrame, it only supports Java and Scala APIs. APIs for Python and R are still under development.

Using Spark with Hadoop:

1. First, Spark is intended to enhance, not replace, the Hadoop stack. From day one, Spark was designed to read and write data from and to HDFS, as well as other storage systems, suchas HBase and Amazon‘s S3 (Amazon Simple Storage Service). As such, Hadoop users can

enrich their processing capabilities by combining Spark with Hadoop MapReduce, HBase, and other big data frameworks.

2. Second, we have constantly focused on making it as easy as possible for every Hadoop user to take advantage of Spark‘s capabilities. No matter whether you run Hadoop 1.x or Hadoop 2.0 (YARN), and no matter whether you have administrative privileges to configure the Hadoop cluster or not, there is a way for you to run Spark! In particular, there are three ways to deploy Spark in a Hadoop cluster: standalone, YARN, and SIMR (Spark In Map Reduce).



a) Standalone deployment: With the standalone deployment one can statically allocate resources on all or a subset of machines in a Hadoop cluster and run Spark side by side with Hadoop MR. The user can then run arbitrary Spark jobs on her HDFS data. Its simplicity makes this the deployment of choice for many Hadoop 1.x users.

b) Hadoop Yarn deployment: Hadoop users who have already deployed or are planning to deploy Hadoop Yarn can simply run Spark on YARN without any pre-installation or administrative access required. This allows users to easily integrate Spark in their Hadoop stack and take advantage of the full power of Spark, as well as of other components running on top of Spark.

c) Spark In MapReduce (SIMR): For the Hadoop users that are not running YARN yet, another option, in addition to the standalone deployment, is to use SIMR to launch Spark jobs inside MapReduce. With SIMR, users can start experimenting with Spark and use its shell within a couple of minutes after downloading it! This tremendously lowers the barrier of deployment, and lets virtually everyone play with Spark.

Interoperability with other Systems:

Spark interoperates not only with Hadoop, but with other popular big data technologies as well.

a) Apache Hive: Through Shark, Spark enables Apache Hive users to run their unmodified queries much faster. Hive is a popular data warehouse solution running on top of Hadoop, while Shark is a system that allows the Hive framework to run on top of Spark instead of Hadoop. As a result, Shark can accelerate Hive queries by as much as 100x when the input data fits into memory, and up 10x when the input data is stored on disk.

b) AWS EC2: Users can easily run Spark (and Shark) on top of Amazon‘s EC2 either using the scripts that come with Spark, or the hosted versions of Spark and Shark on Amazon‘s Elastic MapReduce.

c) Apache Mesos: Spark runs on top of Mesos, a cluster manager system which provides efficient resource isolation across distributed applications, including MPI and Hadoop. Mesos enables fine grained sharing which allows a Spark job to dynamically take advantage of the idle resources in the cluster during its execution. This leads to considerable performance improvements, especially for long running Spark jobs.

Usecase/What are the use cases for Spark?

Some of the top use cases for Apache Spark are explained as the following.

1. Streaming Data:

Apache Spark‘s key use case is its ability to process streaming data. With so much data being processed on a daily basis, it has become essential for companies to be able to stream and analyze it all in real-time. And Spark Streaming has the capability to handle this extra workload. Some experts even theorize that Spark could become the go-to platform for stream-computing applications, no matter the type. The reason for this claim is that Spark Streaming unifies disparate data processing capabilities, allowing developers to use a single framework to accommodate all their processing needs. Among the general ways that Spark Streaming is being used by businesses today are:

a) **Streaming ETL** – Traditional ETL (Extract, Transform, Load) tools used for batch processing in data warehouse environments must read data, convert it to a database compatible format, and then write it to the target database. With Streaming ETL, data is continually cleaned and aggregated before it is pushed into data stores.

b) **Data Enrichment** – This Spark Streaming capability enriches live data by combining it with static data, thus allowing organizations to conduct more complete real-time data analysis. Online advertisers use data enrichment to combine historical customer data with live customer behavior data and deliver more personalized and targeted ads in real-time and in context with what customers are doing.

c) **Trigger Event Detection** – Spark Streaming allows organizations to detect and respond quickly to rare or unusual behaviors (―trigger events‖) that could indicate a potentially serious problem within the system. Financial institutions use triggers to detect fraudulent transactions and stop fraud in their tracks. Hospitals also use triggers to detect potentially dangerous health changes while monitoring patient vital signs—sending automatic alerts to the right caregivers who can then take immediate and appropriate action.

d) **Complex Session Analysis –** Using Spark Streaming, events relating to live sessions—such as user activity after logging into a website or application—can be grouped together and quickly analyzed. Session information can also be used to continuously update machine learning models. Companies such as Netflix use this functionality to gain immediate insights as to how users are engaging on their site and provide more real-time movie recommendations.

2. Machine Learning:

Another of the many Apache Spark use cases is its machine learning capabilities. Spark comes with an integrated framework for performing advanced analytics that helps users run repeated queries on sets of data—which essentially amounts to processing machine learning algorithms. Among the components found in this framework is Spark‘s scalable Machine Learning Library (MLlib). The MLlib can work in areas such as clustering, classification, and dimensionality reduction, among many others. All this enables Spark to be used for some very common big data functions, like predictive intelligence, customer segmentation for marketing purposes, and sentiment analysis. Companies that use a recommendation engine will find that Spark gets the job done fast.

Network security is a good business case for Spark‘s machine learning capabilities. Utilizing various components of the Spark stack, security providers can conduct real-time inspections of data packets for traces of malicious activity. At the front end, Spark Streaming allows security analysts to check against known threats prior to passing the packets on to the storage platform. Upon arrival in storage, the packets undergo further analysis via other stack components such as MLlib. Thus security providers can learn about new threats as they evolve—staying ahead of hackers while protecting their clients in real-time.

3. Interactive Analysis:

Among Spark‘s, most notable features are its capability for interactive analytics. MapReduce was built to handle batch processing, and SQL-on-Hadoop engines such as Hive or Pig are frequently too slow for interactive analysis. However, Apache Spark is fast enough to perform exploratory queries without sampling. Spark also interfaces with a number of development languages including SQL, R, and Python. By combining Spark with visualization tools, complex data sets can be processed and visualized interactively.

Debuting in April or May of this year, the next version of Apache Spark (Spark 2.0) will have a new feature—Structured Streaming—that will give users the ability to perform interactive queries against live data. Combining live streaming with other types of data analysis, Structured Streaming is predicted to provide a boost to Web analytics by allowing users to run interactive queries against a Web visitor‘s current session. It could also be used to apply machine learning algorithms to live data. In this scenario, the algorithms would be trained on old data and then redirected to incorporate new—and potentially learn from it—as it enters the memory.

4. Fog Computing:

While big data analytics may be getting a lot of attention, the concept that really sparks the tech community‘s imagination is the Internet of Things (IoT). The IoT embeds objects and devices with tiny sensors that communicate with each other and the user, creating a fully interconnected world. This world collects massive amounts of data, processes it, and delivers revolutionary new features and applications for people to use in their everyday lives. However, as the IoT expands so too does the need for distributed massively parallel processing of vast amounts and varieties of a machine and sensor data. All that processing, however, is tough to manage with the current analytics capabilities in the cloud. That‘s where fog computing and Apache Spark come in.

Fog computing decentralizes data processing and storage, instead of performing those functions on the edge of the network. However, Fog computing brings new complexities to processing decentralized data, because it increasingly requires low latency, massively parallel processing of machine learning, and extremely complex graph analytics algorithms. Fortunately, with key stack components such as Spark Streaming, an interactive real-time query tool (Shark), a machine learning library (MLib), and a graph analysis engine (GraphX), Spark more than qualifies as a fog computing solution. In fact, as the IoT industry gradually and inevitably converges, many industry experts predict that—compared to other open-source platforms— Spark has the potential to emerge as the de facto fog infrastructure.

5. Spark in the Real World:

As mentioned earlier, online advertisers and companies such as Netflix are leveraging Spark for insights and competitive advantage. Other notable businesses also benefiting from Spark are:

a) **Uber –** Every day this multinational online taxi dispatch company gathers terabytes of event data from its mobile users. By using Kafka, Spark Streaming, and HDFS, to build a continuous ETL pipeline, Uber can convert raw unstructured event data into structured data as it is collected, and then use it for further and more complex analytics.

b) **Pinterest –** Through a similar ETL pipeline, Pinterest can leverage Spark Streaming to gain immediate insight into how users all over the world are engaging with Pins— in real-time. As a result, Pinterest can make more relevant recommendations as people navigate the site and see related Pins to help those select recipes, determine which products to buy or plan trips to various destinations.

c) **Conviva –** Averaging about 4 million video feeds per month, this streaming video company is second only to YouTube. Conviva uses Spark to reduce customer churn by optimizing video streams and managing live video traffic—thus maintaining a consistently smooth, high-quality viewing experience.

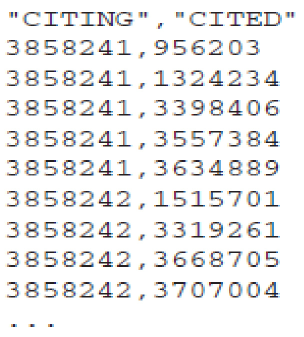
Note: Pinterest - Pinterest is an image sharing and social media service designed to enable saving and discovery of information on the internet using images, and on a smaller scale, animated GIFs (Graphics Interchange Format) and videos, in the form of pinboards.

2. MapReduce Programming

Writing basic Map Reduce programs:

The MapReduce programming model is unlike most programming models. To write and understand MapReduce program, the ***patent data set*** is chosen and is downloaded National Bureau of Economic Research (NBER) at http://www.nber.org/patents/.The data sets were originally compiled for the paper ―The NBER Patent Citation Data File: Lessons, Insights and Methodological Tools.‖ We use the citation data set cite75_99 (250 MB).txt and the patent description data set apat63_99.txt (250 MB).

The patent citation data set contains citations from U.S. patents issued between 1975 and1999. It has more than 16 million rows and the first few lines resemble the following:

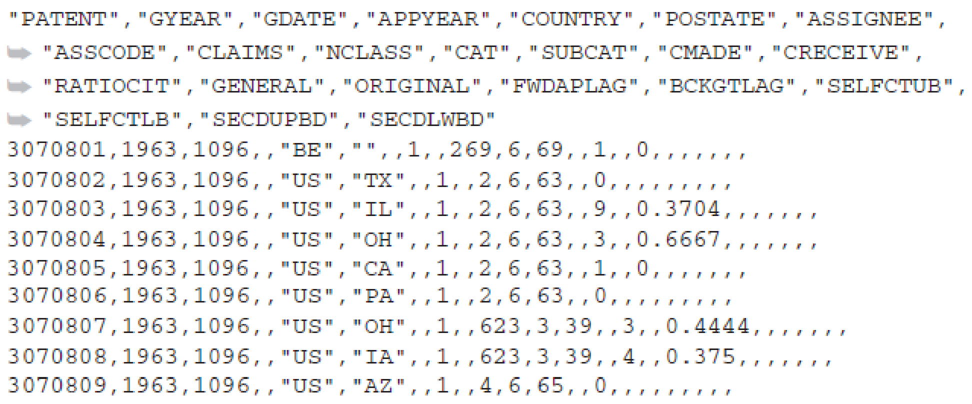


```
"CITING","CITED"
3858241,956203
3858241,1324234
3858241,3398406
3858241,3557384
3858241,3634889
3858242,1515701
3858242,3319261
3858242,3668705
3858242,3707004
...
```

The data set is in the standard comma-separated values (CSV) format, with the first line a description of the columns. Each of the other lines record one particular citation. For example, the second line shows that patent 3858241 cites patent956203. The file is sorted by the citing patent. We can see that patent 3858241 cites five patents in total. Analyzing the data more quantitatively will give us deeper awareness into it.

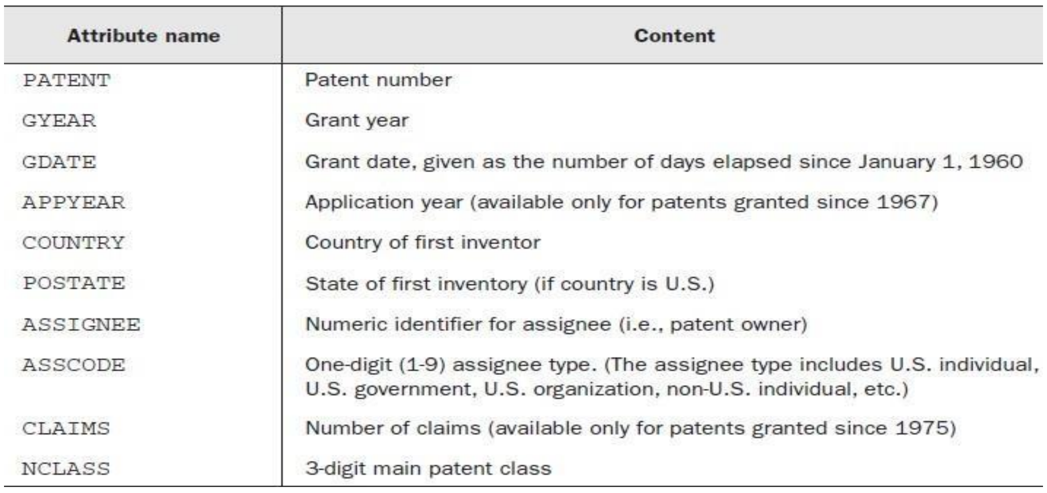
If we‘re only reading the data file, the citation data appears to be a bunch of numbers. Patents like 5936972 and 6009552 cite a similar set of patents (4354269, 4486882, 5598422) even though they don‘t cite each other. We use Hadoop to derive descriptive statistics about this patent data, as well as look for interesting patterns that aren‘t immediately obvious.

The patent description data: The other data set we use is the patent description data. It has the patent number, the patent application year, the patent grant year, the number of claims, and other metadata about patents. The following first few lines of this data set is similar to a table in a relational database, but in CSV format. This data set has more than 2.9 million records and it has many missing values.



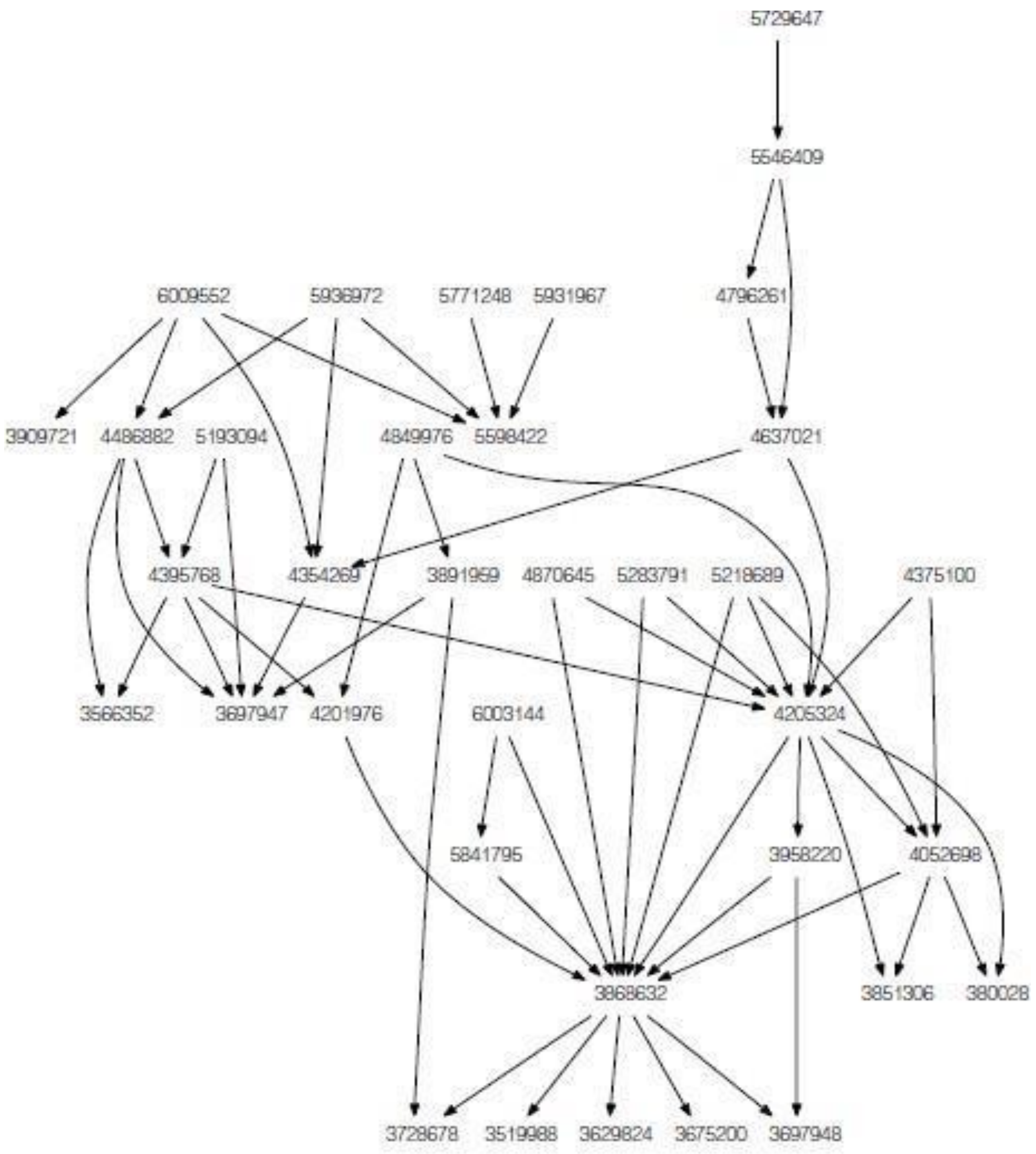
```
"PATENT","GYEAR","GDATE","APPYEAR","COUNTRY","POSTATE","ASSIGNEE",
➥ "ASSCODE","CLAIMS","NCLASS","CAT","SUBCAT","CMADE","CRECEIVE",
➥ "RATIOCIT","GENERAL","ORIGINAL","FWDAPLAG","BCKGTLAG","SELFCTUB",
➥ "SELFCTLB","SECDUPBD","SECDLWBD"
3070801,1963,1096,,"BE","",,1,,269,6,69,,1,,0,,,,,,
3070802,1963,1096,,"US","TX",,1,,2,6,63,,0,,,,,,,,
3070803,1963,1096,,"US","IL",,1,,2,6,63,,9,,0.3704,,,,,,
3070804,1963,1096,,"US","OH",,1,,2,6,63,,3,,0.6667,,,,,,
3070805,1963,1096,,"US","CA",,1,,2,6,63,,1,,0,,,,,,
3070806,1963,1096,,"US","PA",,1,,2,6,63,,0,,,,,,,,
3070807,1963,1096,,"US","OH",,1,,623,3,39,,3,,0.4444,,,,,,
3070808,1963,1096,,"US","IA",,1,,623,3,39,,4,,0.375,,,,,,
3070809,1963,1096,,"US","AZ",,1,,4,6,65,,0,,,,,,,,
```

The first row contains the name of a couple dozen attributes, which are meaningful only to patent specialists. Even without understanding all the attributes, it‘s still useful to have an idea of a few of them. The following table describes the first ten.



Attribute name	Content
PATENT	Patent number
GYEAR	Grant year
GDATE	Grant date, given as the number of days elapsed since January 1, 1960
APPYEAR	Application year (available only for patents granted since 1967)
COUNTRY	Country of first inventor
POSTATE	State of first inventory (if country is U.S.)
ASSIGNEE	Numeric identifier for assignee (i.e., patent owner)
ASSCODE	One-digit (1-9) assignee type. (The assignee type includes U.S. individual, U.S. government, U.S. organization, non-U.S. individual, etc.)
CLAIMS	Number of claims (available only for patents granted since 1975)
NCLASS	3-digit main patent class

Now we have two patent data sets. The following diagram describes partial view of the patent citation data set as a graph:



A partial view of the patent citation data set as a graph. Each patent is shown as a vertex (node), and each citation is a directed edge (arrow).

Constructing the basic template of a MapReduce program: When writing a new MapReduce program, we generally take an existing MapReduce program and modify it until it does what we want. We write our first MapReduce program and explain its different parts. This program can serve as a template for future MapReduce programs.

The first program will take the patent citation data and *invert* it. For each patent, we want to find and group the patents that cite it. Our output should be similar to the following:

```
1       3964859,4647229
10000   4539112
100000  5031388
1000006 4714284
1000007 4766693
1000011 5033339
1000017 3908629
1000026 4043055
1000033 4190903,4975983
1000043 4091523
1000044 4082383,4055371
1000045 4290571
1000046 5918892,5525001
1000049 5996916
1000051 4541310
1000054 4946631
1000065 4748968
1000067 5312208,4944640,5071294
1000070 4928425,5009029
```

By verifying the output, patents 5312208, 4944640, and 5071294 cited patent1000067.We need only one file for the entire program as the following:

Template for a typical Hadoop program

```
public class MyJob extends Configured implements Tool {

    public static class MapClass extends MapReduceBase
        implements Mapper<Text, Text, Text, Text> {

        public void map(Text key, Text value,
                        OutputCollector<Text, Text> output,
                        Reporter reporter) throws IOException {

            output.collect(value, key);
        }
    }

    public static class Reduce extends MapReduceBase
        implements Reducer<Text, Text, Text, Text> {

        public void reduce(Text key, Iterator<Text> values,
                           OutputCollector<Text, Text> output,
                           Reporter reporter) throws IOException {

            String csv = "";
            while (values.hasNext()) {
                if (csv.length() > 0) csv += ",";
                csv += values.next().toString();
            }
            output.collect(key, new Text(csv));
        }
    }

    public int run(String[] args) throws Exception {
        Configuration conf = getConf();

        JobConf job = new JobConf(conf, MyJob.class);

        Path in = new Path(args[0]);
        Path out = new Path(args[1]);
        FileInputFormat.setInputPaths(job, in);
        FileOutputFormat.setOutputPath(job, out);

        job.setJobName("MyJob");
        job.setMapperClass(MapClass.class);
        job.setReducerClass(Reduce.class);
        job.setOutputFormat(TextOutputFormat.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);
        job.set("key.value.separator.in.input.line", ",");

        JobClient.runJob(job);

        return 0;
    }

    public static void main(String[] args) throws Exception {
        int res = ToolRunner.run(new Configuration(), new MyJob(), args);

        System.exit(res);
    }
}
```

A single class, called MyJob completely defines each MapReduce job. Hadoop requires the Mapper and the Reducer have their own static classes. These classes are quite small, and the template includes them as inner classes to the MyJob class. The advantage is that everything fits in one file, simplifying code management. Various nodes with different JVMs clone and run the Mapper and the Reducer during job execution, whereas the rest of the job class is executed *only* at the client machine.

The core of the skeleton is within the run() method, also known as the *driver*. The driver instantiates configures, and passes a JobConf object named job to JobClient.Each job can reset the default job properties, such as InputFormat, OutputFormat, and so on. The following command is used to execute MyJob class:

$ bin/hadoop jar playground/MyJob.jar MyJob input/cite75_99.txt output

To see the only mapper‘s output (which we may want to do for debugging purposes), we could set the number of reducers to zero with the option -D mapred.reduce.tasks=0.

$ bin/hadoop jar playground/MyJob.jar MyJob -D mapred.reduce.tasks=0 input/cite75_99.txt output

The convention for the template is to call the Mapper class MapClass and the Reducer class Reduce. The naming would seem more symmetrical if we call the Mapper class Map, but Java already has a class (interface) named Map. Both the Mapper and the Reducer extend MapReduceBase, which is a small class providing no-op implementations to the configure() and close() methods required by the two interfaces. We use the configure() and close() methods to set up and clean up the map (reduce) tasks. We won‘t need to override them except for more advanced jobs. The signatures for the Mapper class and the Reducer class are:

```
public static class MapClass extends MapReduceBase
    implements Mapper<K1, V1, K2, V2> {

    public void map(K1 key, V1 value,
                    OutputCollector<K2, V2> output,
                    Reporter reporter) throws IOException { }
}

public static class Reduce extends MapReduceBase
    implements Reducer<K2, V2, K3, V3> {

    public void reduce(K2 key, Iterator<V2> values,
                       OutputCollector<K3, V3> output,
                       Reporter reporter) throws IOException { }
}
```

The key/value pairs generated by the mapper are outputted via the collect() method of the OutputCollector object. Somewhere in the map() method we need to call:

output.collect((K2) k, (V2) v);

The reduce() method will likely have a loop to go through all the values of type V2.

```
while (values.hasNext()) {
    V2? v = values.next();
    ...
}
```

Counting things: For the patent citation data, we may want the number of citations a patent has received. This too is counting. The desired output would look like this:

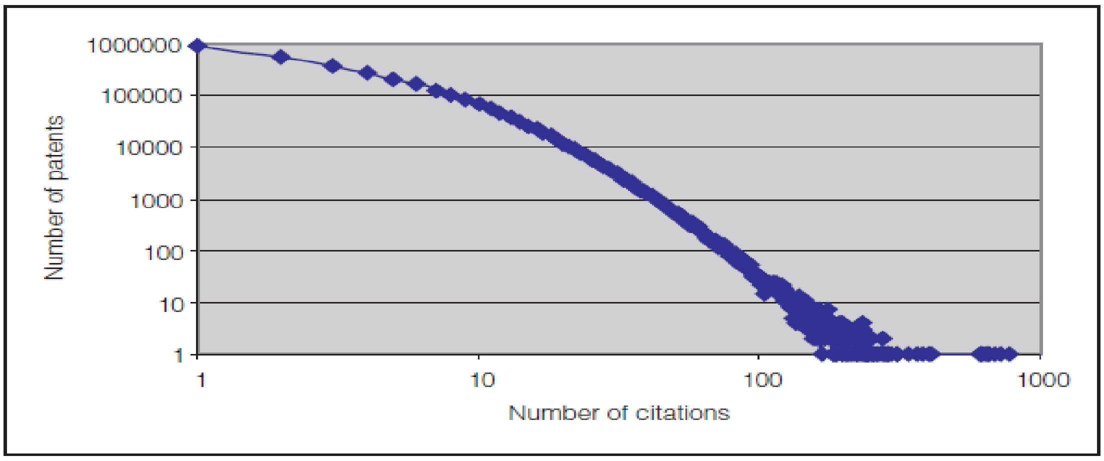
```
1       2   1000017 1   1000046 2
10000   1   1000026 1   1000049 1
100000  1   1000033 2   1000051 1
1000006 1   1000043 1   1000054 1
1000007 1   1000044 2   1000065 1
1000011 1   1000045 1   1000067 3
```

In each record, a patent number is associated with the number of citations it has received. We can write a MapReduce program as the following for this task.

CitationHistogram.java: count patents cited once, twice, and so on

```
public class CitationHistogram extends Configured implements Tool {

    public static class MapClass extends MapReduceBase
        implements Mapper<Text, Text, IntWritable, IntWritable> {

        private final static IntWritable uno = new IntWritable(1);
        private IntWritable citationCount = new IntWritable();

        public void map(Text key, Text value,
                        OutputCollector<IntWritable, IntWritable> output,
                        Reporter reporter) throws IOException {

            citationCount.set(Integer.parseInt(value.toString()));
            output.collect(citationCount, uno);
        }
    }
public static class Reduce extends MapReduceBase
    implements Reducer<IntWritable,IntWritable,IntWritable,IntWritable>
{

    public void reduce(IntWritable key, Iterator<IntWritable> values,
                       OutputCollector<IntWritable, IntWritable>output,
                       Reporter reporter) throws IOException {

        int count = 0;
        while (values.hasNext()) {
            count += values.next().get();
        }
        output.collect(key, new IntWritable(count));
    }
}

public int run(String[] args) throws Exception {
    Configuration conf = getConf();

    JobConf job = new JobConf(conf, CitationHistogram.class);

    Path in = new Path(args[0]);
    Path out = new Path(args[1]);
    FileInputFormat.setInputPaths(job, in);
    FileOutputFormat.setOutputPath(job, out);

        job.setJobName("CitationHistogram");
        job.setMapperClass(MapClass.class);
        job.setReducerClass(Reduce.class);

        job.setInputFormat(KeyValueTextInputFormat.class);
        job.setOutputFormat(TextOutputFormat.class);
        job.setOutputKeyClass(IntWritable.class);
        job.setOutputValueClass(IntWritable.class);

        JobClient.runJob(job);

        return 0;
    }

    public static void main(String[] args) throws Exception {
        int res = ToolRunner.run(new Configuration(),
                                 new CitationHistogram(),
                                 args);

        System.exit(res);
    }
}
```

The following Figure shows the number of patents at various citation frequencies. The plot is on a log-log scale. When a distribution shows as a line in a log-log plot, it‘s considered to be a ***power law*** distribution. The citation count histogram seems to fit the description, although its approximately parabolic curvature also suggests a ***lognormal*** distribution.



Plotting the number of patents at different citation frequencies. Many patents have one citation (or not at all, which is not shown on this graph). Some patents have hundreds of citations. On a log-log graph, this looks close enough to a straight line to be considered a power-law distribution.

Programming with RDDs (Resilient Distributed Dataset)-Basics:

An RDD is simply a distributed collection of elements. Each RDD is split into multiple partitions, which may be computed on different nodes of the cluster. RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes.

1. Users create RDDs in two ways: by loading an external dataset, or by distributing a collection of objects (e.g., a list or set) in their driver program. The following statement can load a text file as an RDD of strings using ***SparkContext.textFile()***.

Example-1. Creating an RDD of strings with textFile() in Python

```
>>> lines = sc.textFile("README.md")
```

2. Once created, RDDs offer two types of operations: *transformations and actions*. Transformations construct a new RDD from a previous one. For example, one common transformation is filtering data that matches a predicate. In the text file example, we can use this to create a new RDD holding just the strings that contain the word Python, as shown below.

Example-2. Calling the filter() transformation

```
>>> pythonLines = lines.filter(lambda line: "Python" in line)
```

3. Actions, on the other hand, compute a result based on an RDD, and either returns it to the driver program or save it to an external storage system (e.g., HDFS). One example of an action we called earlier is first(), which returns the first element in an RDD and is shown below.

Example-3. Calling the first() action

```
>>> pythonLines.first()
u'## Interactive Python Shell'
```

4. The persist() can load a subset of the data into memory and query it repeatedly. For example, to compute multiple results about the README lines that contain Python, we couldwrite the script as the following.

Example-4. Persisting an RDD in memory

```
>>> pythonLines.persist
>>> pythonLines.count()
2
>>> pythonLines.first()
u'## Interactive Python Shell'
```

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

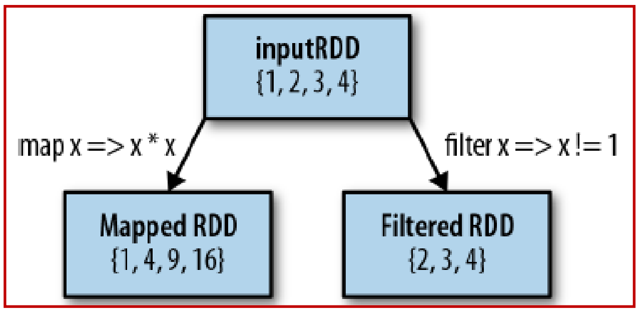
1. Create some input RDDs from external data.
2. Transform them to define new RDDs using transformations like filter().
3. Ask Spark to persist() any intermediate RDDs that will need to be reused. cache() is the same as calling persist() with the default storage level.
4. Launch actions such as count()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

Basic RDDs:

The following concepts explain what transformations and actions we can perform on all RDDs regardless of the data.

Element-wise transformations:

The two most common transformations will likely be using are map() and filter(). 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. The filter() transformation takes in a function and returns an RDD that only has elements that pass the filter() function.



Mapped and filtered RDD from an input RDD

We can use map() to do any number of things, from fetching the website associated with each URL in our collection to just squaring the numbers. It is useful to note that map()'s return type does not have to be the same as its input type, so if we had an RDD String and our map() function were to parse the strings and return a Double, our input RDD type would be RDD[String] and the resulting RDD type would be RDD[Double]. A basic example of map() that squares all of the numbers in an RDD is shown below.

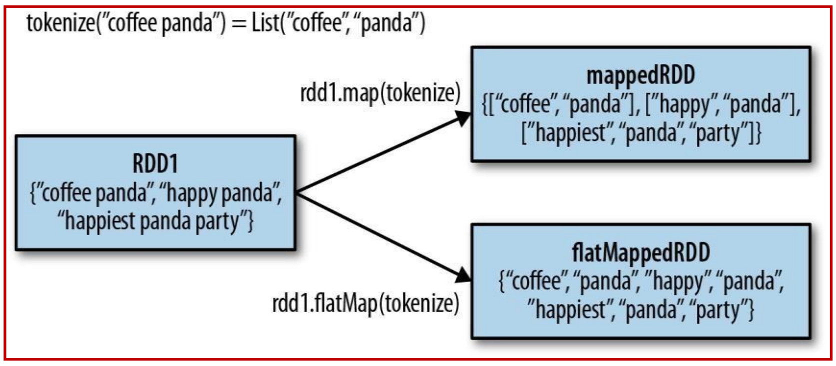
```
# Python squaring the values in an RDD
nums = sc.parallelize([1, 2, 3, 4])
squared = nums.map(lambda x: x * x).collect()
for num in squared:
print "%i " % (num)
# Scala squaring the values in an RDD
val input = sc.parallelize(List(1, 2, 3, 4))
val result = input.map(x => x * x)
println(result.collect().mkString(","))
// Java squaring the values in an RDD
JavaRDD<Integer> rdd = sc.parallelize(Arrays.asList(1, 2, 3, 4));
JavaRDD<Integer> result = rdd.map(new Function<Integer, Integer>() {
public Integer call(Integer x) { return x*x; }
});
System.out.println(StringUtils.join(result.collect(), ","));
```

Sometimes we want to produce multiple output elements for each input element. The operation to do this is called flatMap(). As with map(), the function we provide to flatMap() is called individually for each element in our input RDD. Instead of returning a single element, we return an iterator with our return values. Rather than producing an RDD of iterators, we get back an RDD that consists of the elements from all of the iterators. A simple usage of flatMap() is splitting up an input string into words, as shown in the following Examples.

```
# flatMap() in Python, splitting lines into words
lines = sc.parallelize(["hello world", "hi"])
words = lines.flatMap(lambda line: line.split(" "))
words.first() # returns "hello"
# flatMap() in Scala, splitting lines into multiple words
val lines = sc.parallelize(List("hello world", "hi"))
val words = lines.flatMap(line => line.split(" "))
words.first() // returns "hello"
```

```
// flatMap() in Java, splitting lines into multiple words
JavaRDD<String> lines = sc.parallelize(Arrays.asList("hello world", "hi"));
JavaRDD<String> words = lines.flatMap(new FlatMapFunction<String, String>() {
public Iterable<String> call(String line) {
return Arrays.asList(line.split(" "));
}
});
words.first(); // returns "hello"
```

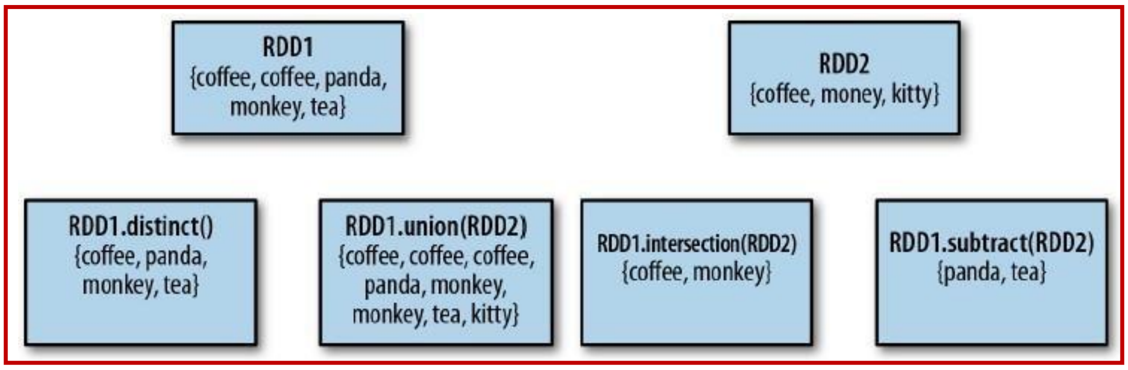
The difference between flatMap() and map() explained in the following Figure. We can think of flatMap() as ―flattening‖ the iterators returned to it, so that instead of ending up with an RDD of lists we have an RDD of the elements in those lists.


tokenize("coffee panda") = List("coffee", "panda")
rdd1.map(tokenize)
mappedRDD
{["coffee", "panda"], ["happy", "panda"], ["happiest", "panda", "party"]}
RDD1
{"coffee panda", "happy panda", "happiest panda party"}
rdd1.flatMap(tokenize)
flatMappedRDD
{"coffee", "panda", "happy", "panda", "happiest", "panda", "party"}

Difference between flatMap() and map() on an RDD

Pseudo set operations:

RDDs support many of the operations of mathematical sets, such as union and intersection, even when the RDDs themselves are not properly sets. Four operations are shown in the following Figure. It‘s important to note that all of these operations require that the RDDs being operated on are of the same type.



Some simple set operations

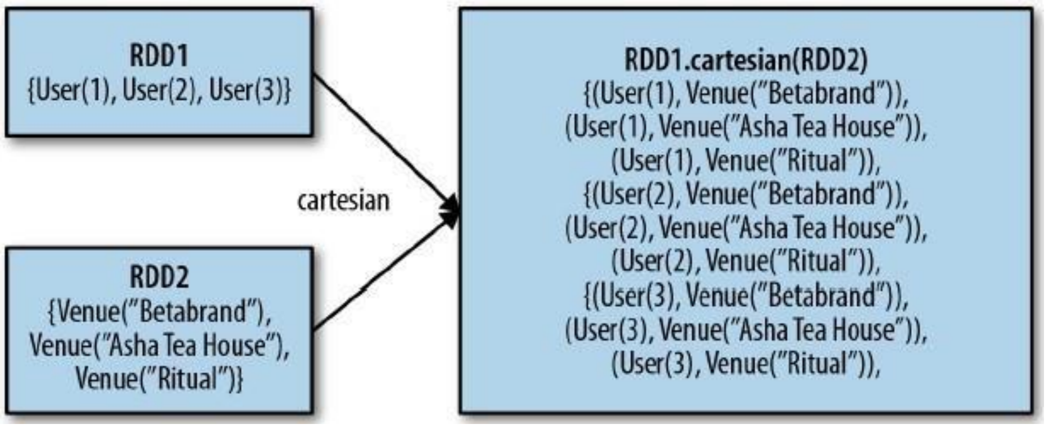
1. If we want only unique elements a set we can use the RDD.distinct() transformation to produce a new RDD with only distinct items.

2. The simplest set operation is union (other), which gives back an RDD consisting of the data from both sources. Unlike the mathematical union (), if there are duplicates in the input RDDs, the result of Spark‘s union () will contain duplicates.

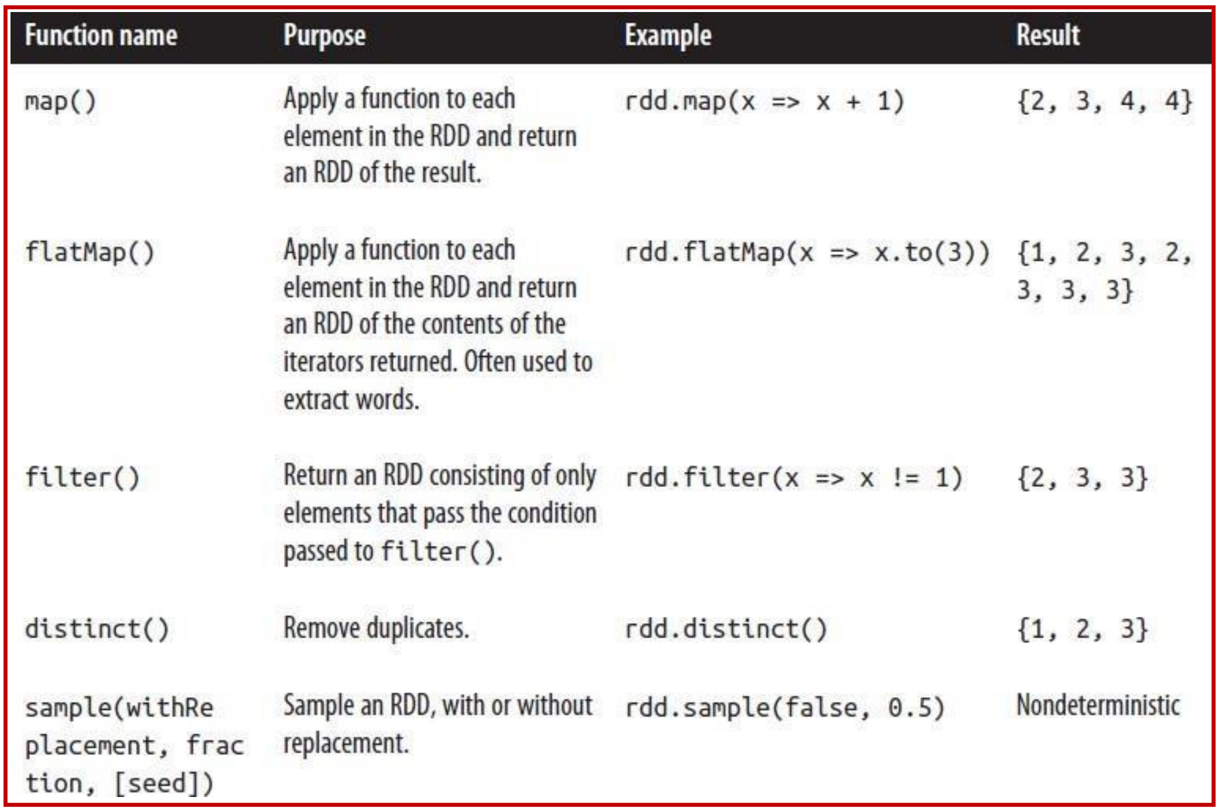
3. Spark also provides an intersection (other) method, which returns only elements in both RDDs. The intersection () also removes all duplicates while running.

4. The subtract(other) function takes in another RDD and returns an RDD that has only values present in the first RDD and not the second RDD.

5. We can also compute a Cartesian product between two RDDs, as shown in the following Figure. The Cartesian (other) transformation returns all possible pairs of (a, b) where _a‘ is in the source RDD and _b‘ is in the other RDD.

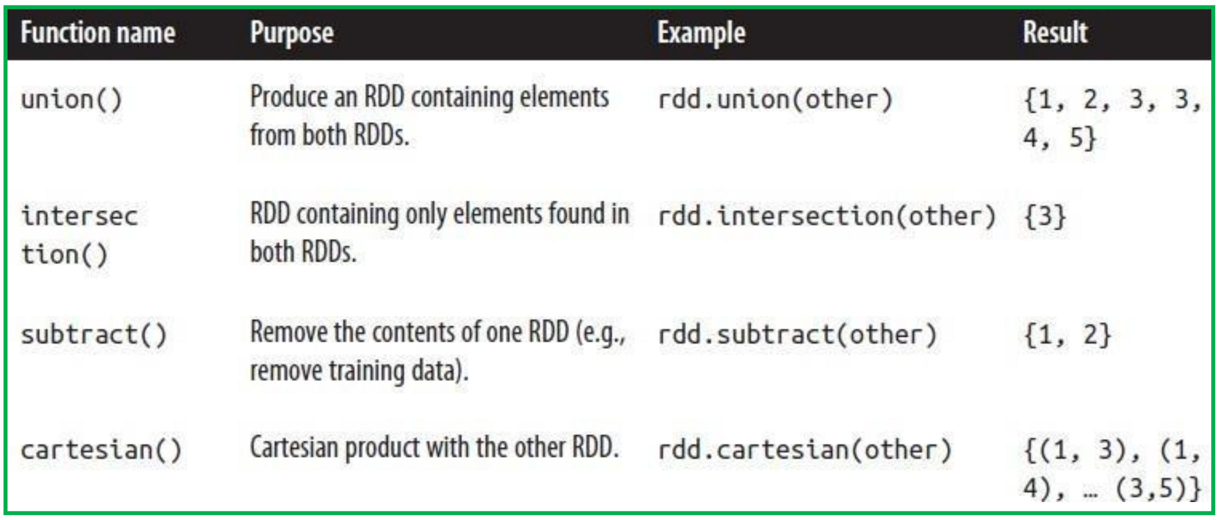

RDD1
{User(1), User(2), User(3)}
cartesian
RDD2
{Venue("Betabrand"), Venue("Asha Tea House"), Venue("Ritual")}
RDD1.cartesian(RDD2)
{(User(1), Venue("Betabrand")),
(User(1), Venue("Asha Tea House")),
(User(1), Venue("Ritual")),
{(User(2), Venue("Betabrand")),
(User(2), Venue("Asha Tea House")),
(User(2), Venue("Ritual")),
{(User(3), Venue("Betabrand")),
(User(3), Venue("Asha Tea House")),
(User(3), Venue("Ritual")),

Cartesian product between two RDDs



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}`
`sample(withReplacement, fraction, [seed])`	Sample an RDD, with or without replacement.	`rdd.sample(false, 0.5)`	Nondeterministic

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



Function name	Purpose	Example	Result
`union()`	Produce an RDD containing elements from both RDDs.	`rdd.union(other)`	`{1, 2, 3, 3, 4, 5}`
`intersec tion()`	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)}`

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

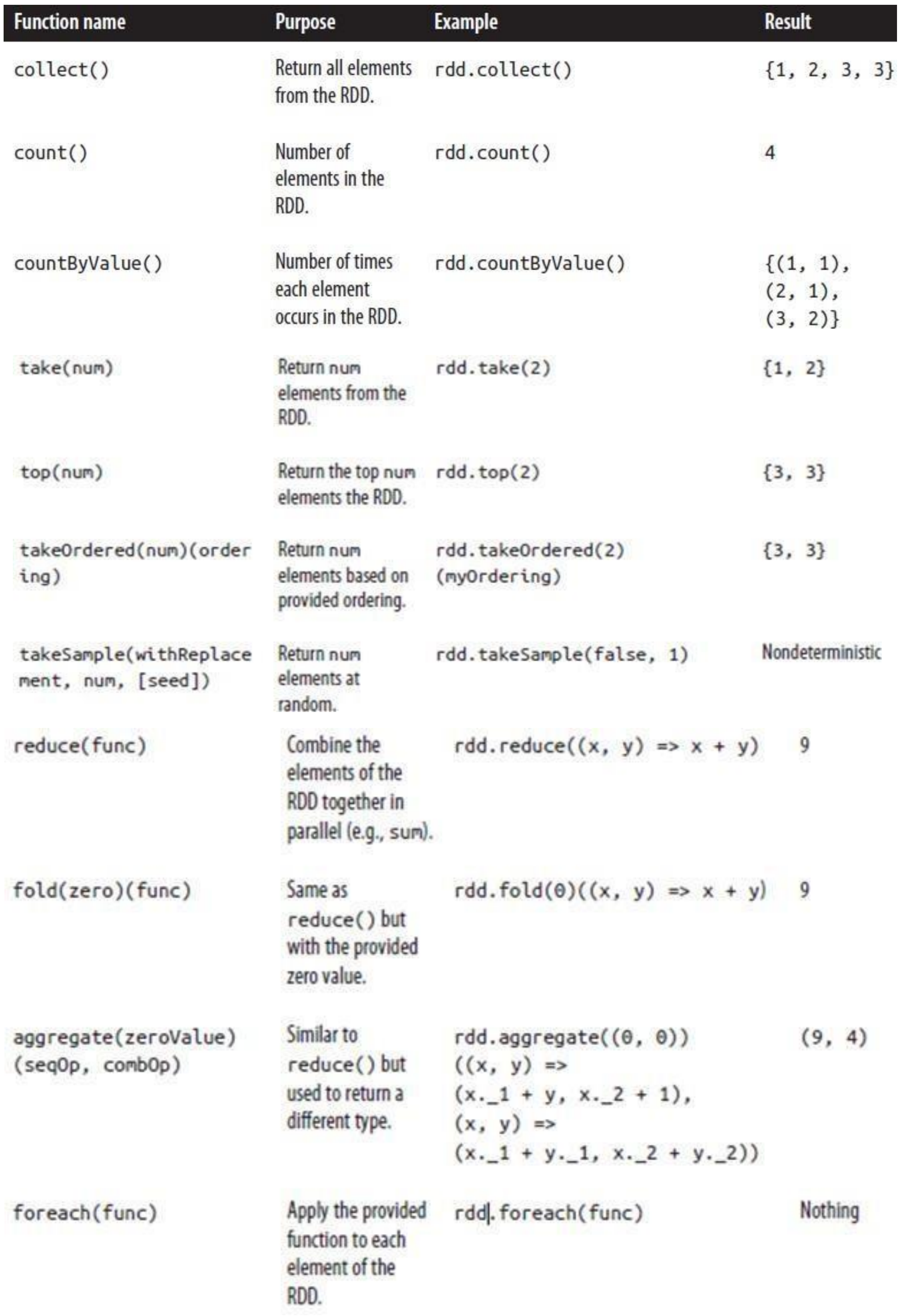
```
# reduce() in Python
sum = rdd.reduce(lambda x, y: x + y)
# reduce() in Scala
val sum = rdd.reduce((x, y) => x + y)
# reduce() in Java
Integer sum = rdd.reduce(new Function2<Integer, Integer, Integer>() {
public Integer call(Integer x, Integer y) { return x + y; }
});
```

We can use aggregate() to compute the average of an RDD, avoiding a map() before the fold(), as shown in Examples.

```
# aggregate() in Python
sumCount = nums.aggregate((0, 0),
(lambda acc, value: (acc[0] + value, acc[1] + 1),
(lambda acc1, acc2: (acc1[0] + acc2[0], acc1[1] + acc2[1]))))
return sumCount[0] / float(sumCount[1])
# aggregate() in Scala
val result = input.aggregate((0, 0))(
(acc, value) => (acc._1 + value, acc._2 + 1),
(acc1, acc2) => (acc1._1 + acc2._1, acc1._2 + acc2._2))
val avg = result._1 / result._2.toDouble
# aggregate() in Java
class AvgCount implements Serializable {
public AvgCount(int total, int num) {
this.total = total;
this.num = num;
}
public int total;
public int num;
public double avg() {
return total / (double) num;
}
}
Function2<AvgCount, Integer, AvgCount> addAndCount =
new Function2<AvgCount, Integer, AvgCount>() {
public AvgCount call(AvgCount a, Integer x) {
a.total += x;
a.num += 1;
return a;
}
};
Function2<AvgCount, AvgCount, AvgCount> combine =
new Function2<AvgCount, AvgCount, AvgCount>() {
public AvgCount call(AvgCount a, AvgCount b) {
a.total += b.total;
a.num += b.num;
return a;
}
};
AvgCount initial = new AvgCount(0, 0);
AvgCount result = rdd.aggregate(initial, addAndCount, combine);
System.out.println(result.avg());
```

Some actions on RDDs return some or all of the data to our driver program in the form of a regular collection or value.

***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)}
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}
takeSample(withReplacement, num, [seed])	Return num elements at random.	rdd.takeSample(false, 1)	Nondeterministic
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

Converting Between RDD Types:

Some functions are available only on certain types of RDDs, such as mean() and variance() on numeric RDDs or join() on key/value pair RDDs. In Scala and Java, these methods aren‘t defined on the standard RDD class, so to access this additional functionality we have to make sure we get the correct specialized class.

Scala:

In Scala the conversion to RDDs with special functions (e.g., to expose numeric functions on an RDD[Double]) is handled automatically using implicit conversions. We need to add "import org.apache.spark.SparkContext._" for these conversions to work. These implicits turn an RDD into various wrapper classes, such as DoubleRDDFunctions (for RDDs of numeric data) and PairRDDFunctions (for key/value pairs), to expose additional functions such as mean() and variance().

Java:

In Java the conversion between the specialized types of RDDs is a bit more explicit. In particular, there are special classes called JavaDoubleRDD and JavaPairRDD for RDDs of these types, with extra methods for these types of data. If we want to create a DoubleRDD from an RDD of type T, rather than using Function<T, Double> we use DoubleFunction<T>. When we want a DoubleRDD back, instead of calling map(), we need to call mapToDouble() with the same pattern all of the other functions follow.

Function name	Equivalent function*<A, B,…>	Usage
DoubleFlatMapFunction<T>	Function<T, Iterable<Double>>	DoubleRDD from a flatMapToDouble
DoubleFunction<T>	Function<T, double>	DoubleRDD from map ToDouble
PairFlatMapFunction<T, K, V>	Function<T, Iterable<Tuple2<K, V>>>	PairRDD<K, V> from a flatMapToPair
PairFunction<T, K, V>	Function<T, Tuple2<K, V>>	PairRDD<K, V> from a mapToPair

Java interfaces for type-specific functions

We can modify the above Example, where we squared an RDD of numbers, to produce a JavaDoubleRDD, as shown below. This gives us access to the additional DoubleRDD specific functions like mean() and variance().

```
# Creating DoubleRDD in Java
JavaDoubleRDD result = rdd.mapToDouble(
new DoubleFunction<Integer>() {
public double call(Integer x) {
return (double) x * x;
}
});
System.out.println(result.mean());
```

Python:

The Python API is structured differently than Java and Scala. In Python all of the functions are implemented on the base RDD (Resilient Distributed Datasets) class but will fail at runtime if the type of data in the RDD is incorrect.

&&&&&&&&&

UNIT - III

Streaming in Spark, Streaming features, Streaming Fundamentals. Usecase on streaming. Machine Learning, Spark MLlib Overview, Tools, Algorithms-Classification, Regression, Clustering, Dimensionality Reduction, Feature Extraction. MapReduce Advanced Programming-Chaining Map Reduce jobs, joining data from different sources. Usecase

Streaming in Spark:

Spark Streaming is an extension of the core Spark API that allows data engineers and data scientists to process real-time data from various sources including (but not limited to) Kafka, Flume, and Amazon Kinesis. This processed data can be pushed out to file systems, databases, and live dashboards.

Spark is built on the concept of RDDs, Spark Streaming provides an abstraction called DStreams, or discretized streams. A DStream is a sequence of data arriving over time. Internally, each DStream is represented as a sequence of RDDs arriving at each time step (hence the name –discretized‖). DStreams can be created from various input sources, such as Flume, Kafka, or HDFS. Once built, they offer two types of operations: ***transformations***, which yield a new DStream, and ***output operations***, which write data to an external system. DStreams provide many of the same operations available on RDDs, plus new operations related to time, such as sliding windows.

Spark Streaming is available only in Java and Scala. Experimental Python support was added in Spark 1.2, though it supports only text data.

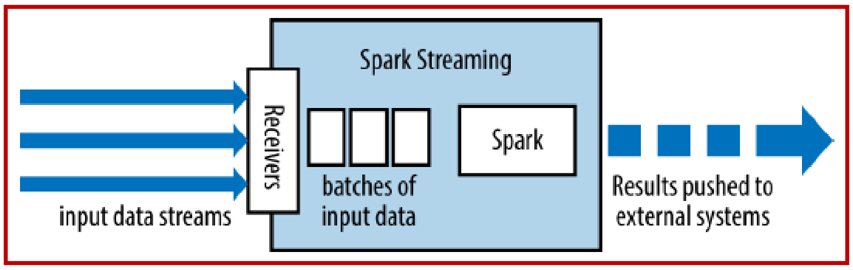
Streaming features:

1. Fast and general-purpose engine for large-scale data processing
   a. Not a modified version of Hadoop
   b. The leading candidate for –successor to Map Reduce‖
2. Spark can efficiently support **more types of computations**
   a. For example, interactive queries, stream processing
3. Can read/write to any Hadoop-supported system (e.g., HDFS)
4. **Speed:** in-memory data storage for very fast iterative queries

a. the system is also more efficient than MapReduce for complex applications running on disk
b. up to 40x faster than Hadoop
c. Ingest data from many sources: Kafka, Twitter, HDFS, TCP sockets
d. Results can be pushed out to file-systems, databases, live dashboards, but not only

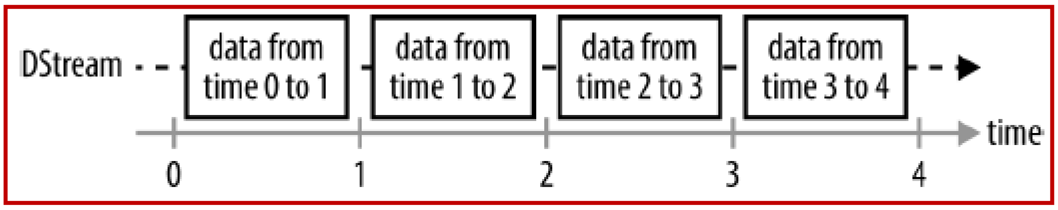
Architecture and Abstraction:

1. Spark Streaming uses a –micro-batch‖ architecture, where the streaming computation is treated as a continuous series of batch computations on small batches of data.
2. Spark Streaming receives data from various input sources and groups it into small batches. New batches are created at regular time intervals.
3. At the beginning of each time interval a new batch is created, and any data that arrives during that interval gets added to that batch. At the end of the time interval the batch is done growing.
4. The size of the time intervals is determined by a parameter called the batch interval. The batch interval is typically between 500 milliseconds and several seconds, as configured by the application developer.
5. Each input batch forms an RDD, and is processed using Spark jobs to create other RDDs. The processed results can then be pushed out to external systems in batches. This high-level architecture is shown in the following Figure.



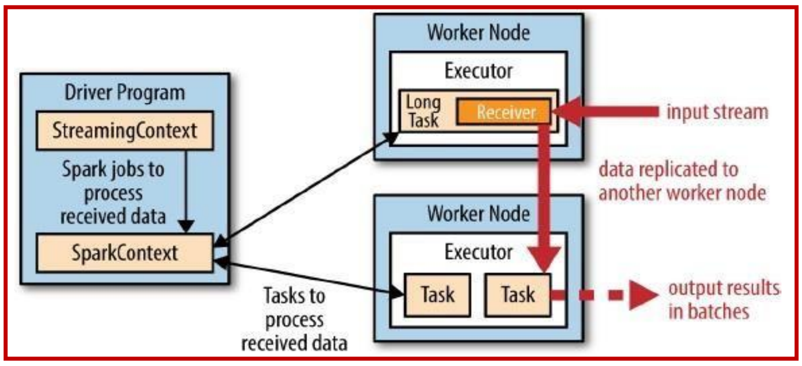
High-level architecture of Spark Streaming

The programming abstraction in Spark Streaming is a discretized stream or a DStream, which is a sequence of RDDs, where each RDD has one time slice of the data in thestream and is shown in the following Figure.



DStream as a continuous series of RDDs

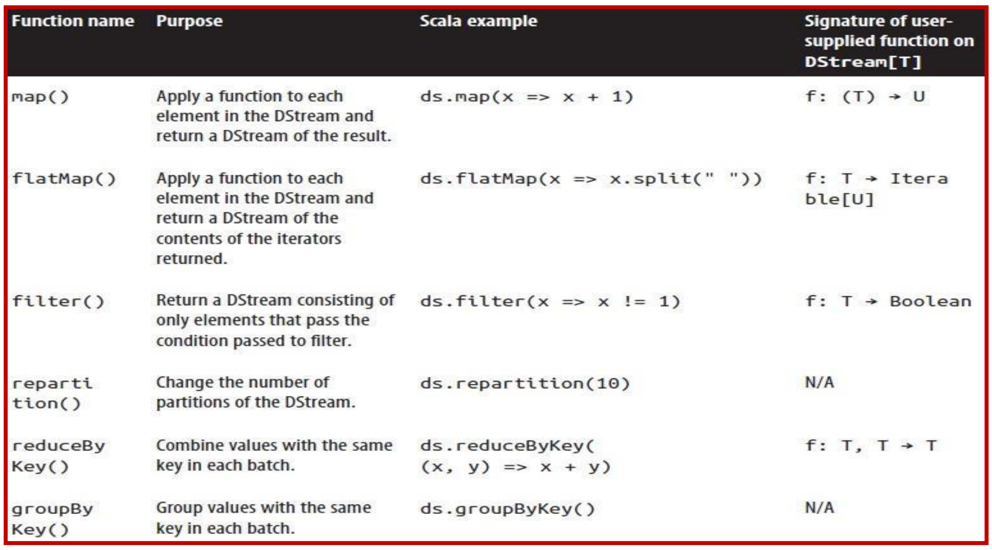
The execution of Spark Streaming within Spark‘s driver-worker components is shown in the following Figure. For each input source, Spark Streaming launches receivers, which are tasks running within the application‘s executors that collect data from the input source and save it as RDDs. These receive the input data and replicate it (by default) to another executor for fault tolerance. This data is stored in the memory of the executors in the same way as cached RDDs. The –StreamingContext‖ in the driver program then periodically runs Spark jobs to process this data and combine it with RDDs from previous time steps.


Driver Program
StreamingContext
Spark jobs to process received data
SparkContext
Worker Node
Executor
Long Task
Receiver
input stream
data replicated to another worker node
Worker Node
Executor
Task
Task
output results in batches
Tasks to process received data

Execution of Spark Streaming within Spark ‘s components

Transformations: Transformations on DStreams can be grouped into either stateless or stateful.

1. In stateless transformations the processing of each batch does not depend on the data of its previous batches. They include the common RDD transformations like map(), filter(), and reduceByKey().
2. Stateful transformations, in contrast, use data or intermediate results from previous batches to compute the results of the current batch. They include transformations based on sliding windows and on tracking state across time.



Function name	Purpose	Scala example	Signature of user-supplied function on DStream[T]
map()	Apply a function to each element in the DStream and return a DStream of the result.	ds.map(x => x + 1)	f: (T) → U
flatMap()	Apply a function to each element in the DStream and return a DStream of the contents of the iterators returned.	ds.flatMap(x => x.split(" "))	f: T → Iterable[U]
filter()	Return a DStream consisting of only elements that pass the condition passed to filter.	ds.filter(x => x != 1)	f: T → Boolean
repartition()	Change the number of partitions of the DStream.	ds.repartition(10)	N/A
reduceByKey()	Combine values with the same key in each batch.	ds.reduceByKey((x, y) => x + y)	f: T, T → T
groupByKey()	Group values with the same key in each batch.	ds.groupByKey()	N/A

Examples of stateless DStream transformations

Stateful Transformations: These are operations on DStreams that track data across time; that is, some data from previous batches is used to generate the results for a new batch. The two main types are **windowed operations** and **UpdateStateByKey transformation**.

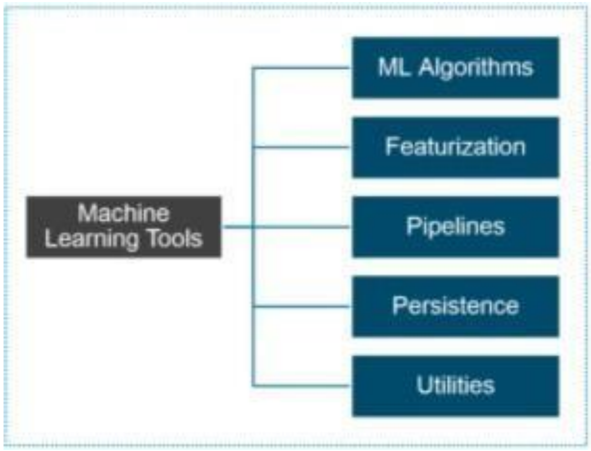
Windowed operations compute results across a longer time period than the StreamingContext‘s batch interval, by combining results from multiple batches. The updateStateByKey(), which is used to track state across events for each key (e.g., to build up an object representing each user session).

Output Operations:

1. Output operations specify what needs to be done with the final transformed data in a stream (e.g., pushing it to an external database or printing it to the screen).
2. A common debugging output operation is print(). This grabs the first 10 elements from each batch of the DStream and prints the results.
3. Once we‘ve debugged our program, we can also use output operations to save results.
4. Spark Streaming has similar save() operations for DStreams, each of which takes a directory to save files into and an optional suffix.
5. The results of each batch are saved as subdirectories in the given directory, with the time and the suffix in the filename.

Machine Learning with Spark MLlib:

Machine learning is a method of data analysis that automates analytical model building. It is a branch of artificial intelligence based on the idea that systems can learn from data, identify patterns and make decisions with minimal human intervention. The tools of Machine Learning are shown in the following Figure.



Machine Learning tools

Machine learning is closely related to computational statistics, which also focuses on prediction-making through the use of computers. It has strong ties to mathematical optimization, which delivers methods, theory and application domains to the field. Within the field of data analytics, machine learning is a method used to devise complex models and algorithms that lend themselves to a prediction which in commercial use is known as predictive analytics. There are three categories of Machine learning tasks:

1. Supervised Learning: Supervised learning is where you have input variables (x) and an output variable (Y) and you use an algorithm to learn the mapping function from the input to the output.

2. Unsupervised Learning: Unsupervised learning is a type of machine learning algorithm used to draw inferences from datasets consisting of input data without labeled responses.

3. Reinforcement Learning: A computer program interacts with a dynamic environment in which it must perform a certain goal (such as driving a vehicle or playing a game against an opponent). The program is provided feedback in terms of rewards and punishments as it navigates its problem space. This concept is called reinforcement learning.

Spark MLlib Overview:

Spark MLlib is used to perform machine learning in Apache Spark. MLlib consists popular algorithms and utilities.

1. spark.mllib contains the original API built on top of RDDs. It is currently in maintenance mode.
2. spark.ml provides higher level API built on top of DataFrames for constructing ML pipelines. spark.ml is the primary Machine Learning API for Spark at the moment.

Spark MLlib Tools: This provides the following tools:

1. **ML Algorithms:** ML Algorithms form the core of MLlib. These include common learning algorithms such as classification, regression, clustering and collaborative filtering.
2. **Featurization:** Featurization includes feature extraction, transformation, dimensionality reduction and selection.
3. **Pipelines:** Pipelines provide tools for constructing, evaluating and tuning ML Pipelines.
4. **Persistence:** Persistence helps in saving and loading algorithms, models and Pipelines.
5. **Utilities:** Utilities for linear algebra, statistics and data handling.

Data Types:

MLlib contains a few specific data types, located in the org.apache.spark.mllib package (Java/Scala) or pyspark.mllib (Python). The main ones are:

1. Vector: A mathematical vector. MLlib supports both dense vectors, where every entry is stored, and sparse vectors, where only the nonzero entries are stored to save space. We will discuss the different types of vectors shortly. Vectors can be constructed with the mllib.linalg.Vectors class.

2. LabeledPoint: A labeled data point for supervised learning algorithms such as classification and regression. It includes a feature vector and a label (which is a floating-point value). Located in the mllib.regression package.

3. Rating: A rating of a product by a user, used in the mllib.recommendation package for product recommendation.

4. Various Model classes: Each Model is the result of a training algorithm, and typically has a predict() method for applying the model to a new data point or to an RDD of new data points.

MLlib Algorithms: The popular algorithms and utilities in Spark MLlib are:

1. Basic Statistics
2. Classification and Regression
3. Clustering
4. Collaborative Filtering and Recommendation
5. Dimensionality Reduction
6. Feature Extraction
7. Optimization

1. Basic Statistics: This includes the most basic of machine learning techniques. These include:

a. **Summary Statistics:** Examples include mean, variance, count, max, min and numNonZeros.
b. **Correlations:** Spearman and Pearson are some ways to find correlation.
c. **Stratified Sampling:** These include sampleBykey and sampleByKeyExact.
d. **Hypothesis Testing:** Pearson‘s chi-squared test is an example of hypothesis testing.
e. **Random Data Generation:** RandomRDDs, Normal and Poisson are used to generate random data.

2. Classification and Regression: Classification is the problem of identifying to which of a set of categories (sub-populations) a new observation belongs, on the basis of a training set of data containing observations (or instances) whose category membership is known. It is an example of pattern recognition.

Regression analysis is a statistical process for estimating the relationships among variables. It includes many techniques for modeling and analyzing several variables when the focus is on the relationship between a dependent variable and one or more independent variables.

Classification and regression are two common forms of supervised learning, where algorithms attempt to predict a variable from features of objects using labeled training data (i.e., examples where we know the answer). The difference between them is the type of variable predicted: in classification, the variable is discrete (i.e., it takes on a finite set of values called classes); for example, classes might be spam or non-spam for emails, or the language in which the text is written. In regression, the variable predicted is continuous (e.g., the height of a person given her age and weight).

Both classification and regression use the LabeledPoint class in MLlib which resides in the mllib.regression package. A Label edPoint consists simply of a label (which is always a Double value, but can be set to discrete integers for classification) and a features vector. MLlib includes a variety of methods for classification and regression, including simple linear methods and decision trees and forests.

a. **Linear regression:** Linear regression is one of the most common methods for regression, predicting the output variable as a linear combination of the features. MLlib also supports L^1 and L2 regularized regression, commonly known as Lasso and ridge regression.

The linear regression algorithms are available through the mllib.regression.LinearRegressionWithSGD, LassoWithSGD, and RidgeRegressionWithSGD classes. These follow a common naming pattern throughout MLlib, where problems involving multiple algorithms have a –With‖ part in the class name to specify the algorithm used. Here, SGD is Stochastic Gradient Descent. These classes all have several parameters to tune the algorithm:

i) numIterations - Number of iterations to run (default: 100).

ii) stepSize - Step size for gradient descent (default: 1.0).

iii) intercept - Whether to add an intercept or bias feature to the data—that is, another feature whose value is always 1 (default: false).

iv) regParam - Regularization parameter for Lasso and ridge (default: 1.0).

```
# Linear regression in Python
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.regression import LinearRegressionWithSGD
points = # (create RDD of LabeledPoint)
```

```
model = LinearRegressionWithSGD.train(points, iterations=200, intercept=True)
print "weights: %s, intercept: %s" % (model.weights, model.intercept)
# Linear regression in Scala
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.regression.LinearRegressionWithSGD
val points: RDD[LabeledPoint] = // ...
val lr = new LinearRegressionWithSGD().setNumIterations(200).setIntercept(true)
val model = lr.run(points)
println("weights: %s, intercept: %s".format(model.weights, model.intercept))

// Linear regression in Java
import org.apache.spark.mllib.regression.LabeledPoint;
import org.apache.spark.mllib.regression.LinearRegressionWithSGD;
import org.apache.spark.mllib.regression.LinearRegressionModel;
JavaRDD<LabeledPoint> points = // ...
LinearRegressionWithSGD lr =
new LinearRegressionWithSGD().setNumIterations(200).setIntercept(true);
LinearRegressionModel model = lr.run(points.rdd());
System.out.printf("weights: %s, intercept: %s\n",
model.weights(), model.intercept());
```

b. **Logistic regression:** Logistic regression is a binary classification method that identifies a linear separating plane between positive and negative examples. In MLlib, it takes LabeledPoints with label 0 or 1 and returns a LogisticRegressionModel that can predict new points.

The logistic regression algorithm has a very similar API to linear regression, covered in the previous section. One difference is that there are two algorithms available for solving it: SGD and LBFGS LBFGS is generally the best choice, but is not available in some earlier versions of MLlib (before Spark 1.2). These algorithms are available in the mllib.classification.LogisticRegressionWithLBFGS and WithSGD classes, which have interfaces similar to LinearRegressionWithSGD. They take all the same parameters as linear regression.

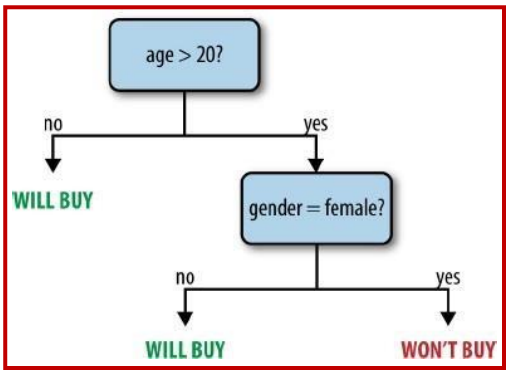
The LogisticRegressionModel from these algorithms computes a score between 0 and 1 for each point, as returned by the logistic function. It then returns either 0 or 1 based on a threshold that can be set by the user: by default, if the score is at least 0.5, it will return 1. We can change this threshold via setThreshold(). You can also disable it altogether via clearThreshold(), in which case predict() will return the raw scores.

c. **Support Vector Machines:** Support Vector Machines, or SVMs, are another binary classification method with linear separating planes, again expecting labels of 0 or 1. They are available through the SVMWithSGD class, with similar parameters to linear and logisitic regression. The returned SVMModel uses a threshold for prediction like LogisticRegressionModel.

d. **Naive Bayes:** Naive Bayes is a multiclass classification algorithm that scores how well each point belongs in each class based on a linear function of the features. It is commonly used in text classification with TF-IDF features, among other applications. MLlib implements Multinomial Naive Bayes, which expects nonnegative frequencies (e.g., word frequencies) as input features.

In MLlib, you can use Naive Bayes through the mllib.classification.NaiveBayes class. It supports one parameter, lambda (or lambda_ in Python), used for smoothing. We can call it on an RDD of LabeledPoints, where the labels are between 0 and C–1 for C classes. The returned NaiveBayesModel can predict() the class in which a point best belongs.

e. **Decision trees and random forests:** Decision trees are a flexible model that can be used for both classification and regression. They represent a tree of nodes, each of which makes a binary decision based on a feature of the data (e.g., is a person‘s age greater than 20?), and where the leaf nodes in the tree contain a prediction (e.g., is the person likely to buy a product?). Decision trees are attractive because the models are easy to inspect and because they support both categorical and continuous features. The following Figure shows an example tree.



An example decision tree predicting whether a user might buy a product

In MLlib, you can train trees using the mllib.tree.DecisionTree class, through the static methods trainClassifier() and trainRegressor(). Unlike in some of the other algorithms, the Java and Scala APIs also use static methods instead of a DecisionTree object with setters. The training methods take the following parameters:

i) data - RDD of LabeledPoint.

ii) numClasses (classification only) - Number of classes to use.

iii) impurity - Node impurity measure; can be gini or entropy for classification, and must be variance for regression.

iv) maxDepth - Maximum depth of tree (default: 5).

v) maxBins- Number of bins to split data into when building each node (suggested value: 32).

vi) categoricalFeaturesInfo - A map specifying which features are categorical, and how many categories they each have. For example, if feature 1 is a binary feature with labels 0 and 1, and feature 2 is a three-valued feature with values 0, 1, and 2, you would pass {1: 2, 2: 3}. Use an empty map if no features are categorical.

In Spark 1.2, MLlib adds an experimental **RandomForest** class in Java and Scala to build ensembles of trees, also known as random forests. It is available through RandomForest.trainClassifier and trainRegressor. Apart from the pertree parameters just listed, RandomForest takes the following parameters:

i) numTrees - How many trees to build. Increasing numTrees decreases the likelihood of overfitting on training data.

ii) featureSubsetStrategy - Number of features to consider for splits at each node; can be auto (let the library select it), all, sqrt, log2, or onethird; larger values are more expensive.

iii) seed - Random-number seed to use.

Random forests return a WeightedEnsembleModel that contains several trees (in the weakHypotheses field, weighted by weakHypothesisWeights) and can predict() an RDD or Vector. It also includes a toDebugString to print all the trees.

3. Clustering: Classification is the problem of identifying to which of a set of categories (sub-populations) a new observation belongs, on the basis of a training set of data containing observations (or instances) whose category membership is known. It is an example of pattern recognition.

K-means: MLlib includes the popular K-means algorithm for clustering, as well as a variant called K-means|| that provides better initialization in parallel environments. Kmeans||is similar to the K-means++ initialization procedure often used in single node settings. The most important parameter in K-means is a target number of clusters to generate, K. Apart from K, K-means in MLlib takes the following parameters:

i) initializationMode - The method to initialize cluster centers, which can be either –k-means||‖ or –random‖; k-means|| (the default) generally leads to better results but is slightly more expensive.

ii) maxIterations - Maximum number of iterations to run (default: 100).

iii) runs - Number of concurrent runs of the algorithm to execute. MLlib‘s K-means supports running from multiple starting positions concurrently and picking the best result, which is a good way to get a better overall model (as K-means runs can stop in local minima).

4. Collaborative Filtering and Recommendation: Collaborative filtering is a technique for recommender systems wherein users‘ ratings and interactions with various products are used to recommend new ones. Collaborative filtering is attractive because it only needs to take in a list of user/product interactions: either –explicit‖ interactions (i.e., ratings on a shopping site) or –implicit‖ ones (e.g., a user browsed a product page but did not rate the product). Based solely on these interactions, collaborative filtering algorithms learn which products are similar to each other (because the same users interact with them) and which users are similar to each other, and can make new recommendations.

While the MLlib API talks about –users‖ and –products,‖ you can also use collaborative filtering for other applications, such as recommending users to follow on a social network, tags to add to an article, or songs to add to a radio station.

Alternating Least Squares: MLlib includes an implementation of Alternating Least Squares (ALS), a popular algorithm for collaborative filtering that scales well on clusters.6 It is located in the mllib.recommendation.ALS class.

ALS works by determining a feature vector for each user and product, such that the dot product of a user‘s vector and a product‘s is close to their score. It takes the following parameters:

i) rank - Size of feature vectors to use; larger ranks can lead to better models but are more expensive to compute (default: 10).

ii) iterations - Number of iterations to run (default: 10).

iii) lambda - Regularization parameter (default: 0.01).

iv) alpha - A constant used for computing confidence in implicit ALS (default: 1.0).

numUserBlocks, numProductBlocks Number of blocks to divide user and product data in, to control parallelism; we can pass –1 to let MLlib automatically determine this (the default behavior).

5. Dimensionality Reduction: Dimensionality Reduction is the process of reducing the number of random variables under consideration, via obtaining a set of principal variables. It can be divided into feature selection and feature extraction.

a. Feature Selection: Feature selection finds a subset of the original variables (also called features or attributes).
b. Feature Extraction: This transforms the data in the high-dimensional space to a space of fewer dimensions. The data transformation may be linear, as in Principal Component Analysis (PCA), but many nonlinear dimensionality reduction techniques also exist.

PCA is currently available only in Java and Scala (as of MLlib 1.2). To invoke it, we must first represent your matrix using the mllib.linalg.distributed. RowMatrix class, which stores an RDD of Vectors, one per row. We can then call PCA as shown in the following Example.

```
# PCA in Scala
import org.apache.spark.mllib.linalg.Matrix
import org.apache.spark.mllib.linalg.distributed.RowMatrix
val points: RDD[Vector] = // ...
val mat: RowMatrix = new RowMatrix(points)
val pc: Matrix = mat.computePrincipalComponents(2)
// Project points to low-dimensional space
val projected = mat.multiply(pc).rows
// Train a k-means model on the projected 2-dimensional data
val model = KMeans.train(projected, 10)
```

In this example, the projected RDD contains a two-dimensional version of the original points RDD, and can be used for plotting or performing other MLlib algorithms, such as clustering via K-means. The computePrincipalComponents() returns a mllib.linalg.Matrix object, which is a utility class representing dense matrices, similar to Vector. You can get at the underlying data with _toArray‘.

Singular value decomposition: MLlib also provides the lower-level singular value decomposition (SVD) primitive. The SVD factorizes an $m \times n$ matrix A into three matrices $A \approx U\Sigma V^T$, where:

a. U is an orthonormal matrix, whose columns are called left singular vectors.

b. Σ is a diagonal matrix with nonnegative diagonals in descending order, whose diagonals are called singular values.

c. V is an orthonormal matrix, whose columns are called right singular vectors.

For large matrices, usually we don‘t need the complete factorization but only the top singular values and its associated singular vectors. This can save storage, denoise, and recover the low-rank structure of the matrix. To achieve the decomposition, we call computeSVD on the RowMatrix class, as shown in following Example.

# SVD in Scala

// Compute the top 20 singular values of a RowMatrix mat and their singular vectors.

val svd**: SingularValueDecomposition**[**RowMatrix**, **Matrix**] **=**

mat.computeSVD(20, computeU**=true**)

val U**: RowMatrix** = svd.U *// U is a distributed RowMatrix.*

val s**: Vector** = svd.s *// Singular values are a local dense vector.*

val V**: Matrix** = svd.V *// V is a local dense matrix.*

6. Feature Extraction: This starts from an initial set of measured data and builds derived values (features) intended to be informative and non-redundant, facilitating the subsequent learning and generalization steps, and in some cases leading to better human interpretations. This is related to dimensionality reduction.

7. Optimization: Optimization is the selection of the best element (with regard to some criterion) from some set of available alternatives.

In the simplest case, an optimization problem consists of maximizing or minimizing a real function by systematically choosing input values from within an allowed set and computing the value of the function. The generalization of optimization theory and techniques to other formulations comprises a large area of applied mathematics. More generally, optimization includes finding –best available‖ values of some objective function given a defined domain (or input), including a variety of different types of objective functionsand different types of domains.

Chaining MapReduce jobs:

Many complex tasks need to be broken down into simpler subtasks, each accomplished by an individual MapReduce job. For example, from the citation data set to find the ten most cited patents require sequence of two MapReduce jobs. The first one createsthe –inverted‖ citation data set and counts the number of citations for each patent, and the second job finds the top ten in that ―inverted‖ data.

Chaining MapReduce jobs in a sequence: Though we can execute the two jobs manually one after the other, it‘s more convenient to automate the execution sequence. We can chain MapReduce jobs to run sequentially, with the output of one MapReduce job being the inputto the next. This is done using Unix pipes as the following.

```
mapreduce-1 | mapreduce-2 | mapreduce-3 | ....
```

Chaining MapReduce jobs with complex dependency: Hadoop has a mechanism to simplify the management of nonlinear job dependencies via the Job and `JobControl` classes. A Job object is a representation of a MapReduce job. We instantiate a Job object by passing a `JobConf` object to its constructor. In addition to holding job configuration information, Job also holds dependency information, specified through the `addDependingJob()` method. For Job objects x and y,

```
x.addDependingJob(y)
```

means `x` will not start until `y` has finished.

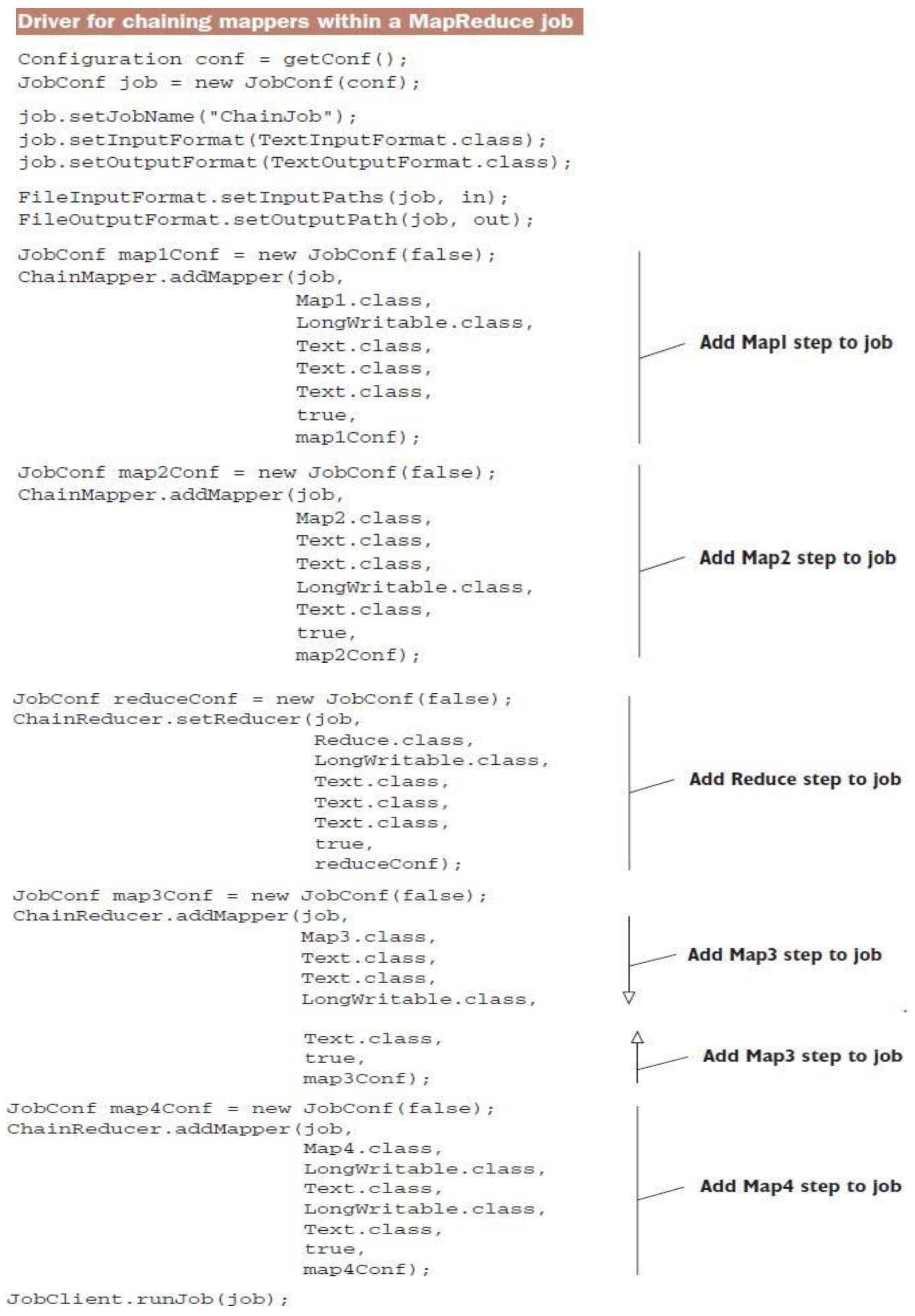
Chaining preprocessing and postprocessing steps: A lot of data processing tasks involve record-oriented preprocessing and postprocessing. For example, in processing documents for information retrieval , we may have one step to remove *stop words* (words like *a*, *the*, and *is* that occur frequently but aren‘t too meaningful), and another step for *stemming* (converting different forms of a word into the same form, such as *finishing* and *finished* into *finish*.) We can write a separate MapReduce job for each of these pre- and postprocessing steps and chain them together, using `IdentityReducer` (or no reducer at all) for these steps. This approach is inefficient as each step in the chain takes up I/O and storage to process the intermediate results.

Another approach is to write a mapper such that it calls all the preprocessing steps beforehand and the reducer to call all the postprocessing steps afterward. Hadoop introduced the `ChainMapper` and the `ChainReducer` classes in version 0.19.0 to simplify the composition of pre- and postprocessing.

For example, four mappers (Map1, Map2, Map3, and Map4) and one reducer (Reduce), and they‘re chained into a single MapReduce job in this sequence:

```
Map1 | Map2 | Reduce | Map3 | Map4
```

We need to make sure the key and value outputs of one task have matching types (classes) with the inputs of the next task. This is explained in the following code:



Driver for chaining mappers within a MapReduce job

```
Configuration conf = getConf();
JobConf job = new JobConf(conf);

job.setJobName("ChainJob");
job.setInputFormat(TextInputFormat.class);
job.setOutputFormat(TextOutputFormat.class);

FileInputFormat.setInputPaths(job, in);
FileOutputFormat.setOutputPath(job, out);

JobConf map1Conf = new JobConf(false);
ChainMapper.addMapper(job,
                      Map1.class,
                      LongWritable.class,
                      Text.class,
                      Text.class,
                      Text.class,
                      true,
                      map1Conf);

JobConf map2Conf = new JobConf(false);
ChainMapper.addMapper(job,
                      Map2.class,
                      Text.class,
                      Text.class,
                      LongWritable.class,
                      Text.class,
                      true,
                      map2Conf);

JobConf reduceConf = new JobConf(false);
ChainReducer.setReducer(job,
                        Reduce.class,
                        LongWritable.class,
                        Text.class,
                        Text.class,
                        Text.class,
                        true,
                        reduceConf);

JobConf map3Conf = new JobConf(false);
ChainReducer.addMapper(job,
                       Map3.class,
                       Text.class,
                       Text.class,
                       LongWritable.class,

                       Text.class,
                       true,
                       map3Conf);

JobConf map4Conf = new JobConf(false);
ChainReducer.addMapper(job,
                       Map4.class,
                       LongWritable.class,
                       Text.class,
                       LongWritable.class,
                       Text.class,
                       true,
                       map4Conf);

JobClient.runJob(job);
```

The driver first sets up the ―global‖ `JobConf` object with the job‘s name, input path, output path, and so forth.

Joining Data from Different Sources:

Unfortunately, joining data in Hadoop is more complex, and there are several possible approaches with different trade-offs. We use a couple `toy datasets` to better illustrate joining in Hadoop. Let‘s take a comma-separated Customers file where each record has three fields: `Customer ID,` `Name`, and `Phone Number`. We put four records in the file for illustration:

```
1,Stephanie Leung,555-555-5555
2,Edward Kim,123-456-7890
3,Jose Madriz,281-330-8004
4,David Stork,408-555-0000
```

We store Customer orders in a separate file, called `Orders`. It‘s in CSV format, with four fields: `Customer ID`, `Order ID`, `Price`, and `Purchase Date`.

```
3,A,12.95,02-Jun-2008
1,B,88.25,20-May-2008
2,C,32.00,30-Nov-2007
3,D,25.02,22-Jan-2009
```

If we applied an inner join of the two data sets above, the desired output is:

Desired output of an inner join between Customers and Orders data

```
1,Stephanie Leung,555-555-5555,B,88.25,20-May-2008
2,Edward Kim,123-456-7890,C,32.00,30-Nov-2007
3,Jose Madriz,281-330-8004,A,12.95,02-Jun-2008
3,Jose Madriz,281-330-8004,D,25.02,22-Jan-2009
```

Hadoop can also perform outer joins. But we focus on inner joins.

Reduce-side joining: Hadoop has a `contrib` package called *`datajoin`* that works as a generic framework for data joining in Hadoop. Its jar file is at `contrib/datajoin/hadoop-*-datajoin.jar`. To distinguish it from other joining techniques, it‘s called the ***`reduce-side join`***, as we do most of the processing on the reduce side. It‘s also known as the *`repartitioned join`* (or the *`repartitioned sort-merge join`*), as it‘s the same as the database technique of the same name. Although it‘s notthe most efficient joining technique, it‘s the most general and forms the basis of some more advanced techniques (such as the **`semijoin`**).

Reduce-side join introduces some new terminologies and concepts, namely, datasource, tag, and group key. A *`data source`* is similar to a table in relational databases. We have two data sources in our `toy` example: `Customers` and `Orders`. A data source can be asingle file or multiple files. The important point is that all the records in a data source have the same structure, equivalent to a schema.

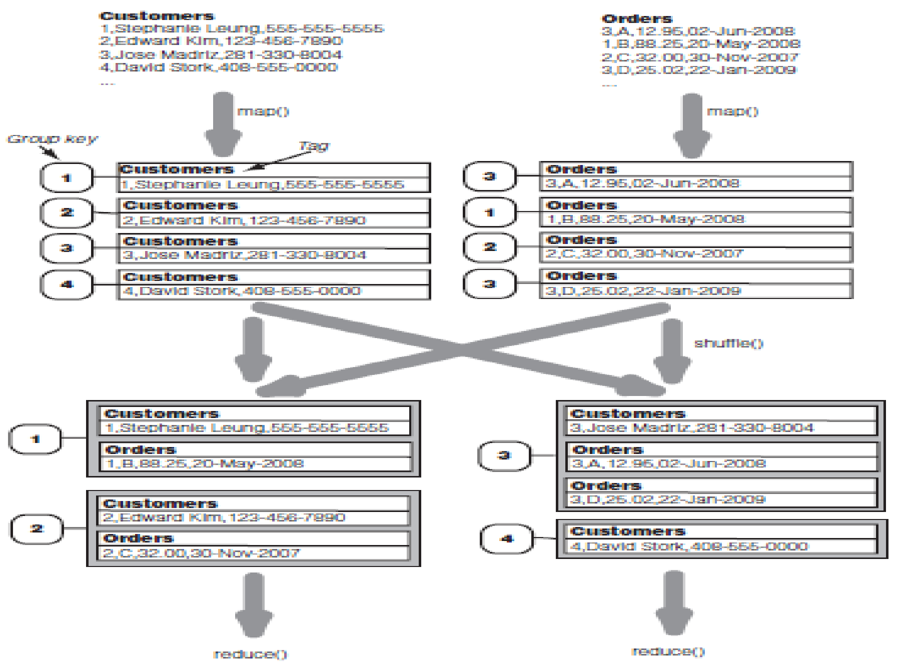
The MapReduce paradigm calls for processing each record one at a time in a stateless manner. If we want some state information to persist, we have to *tag* the record with such state. For example, given our two files, a record may look to a mapper like this:

```
3,Jose Madriz,281-330-8004
or:
3,A,12.95,02-Jun-2008
```

where the record type (`Customers or Orders`) is dissociated from the record itself. Tagging the record will ensure that specific metadata will always go along with the record. For the purpose of data joining, we want to tag each record with its *`data source`*.

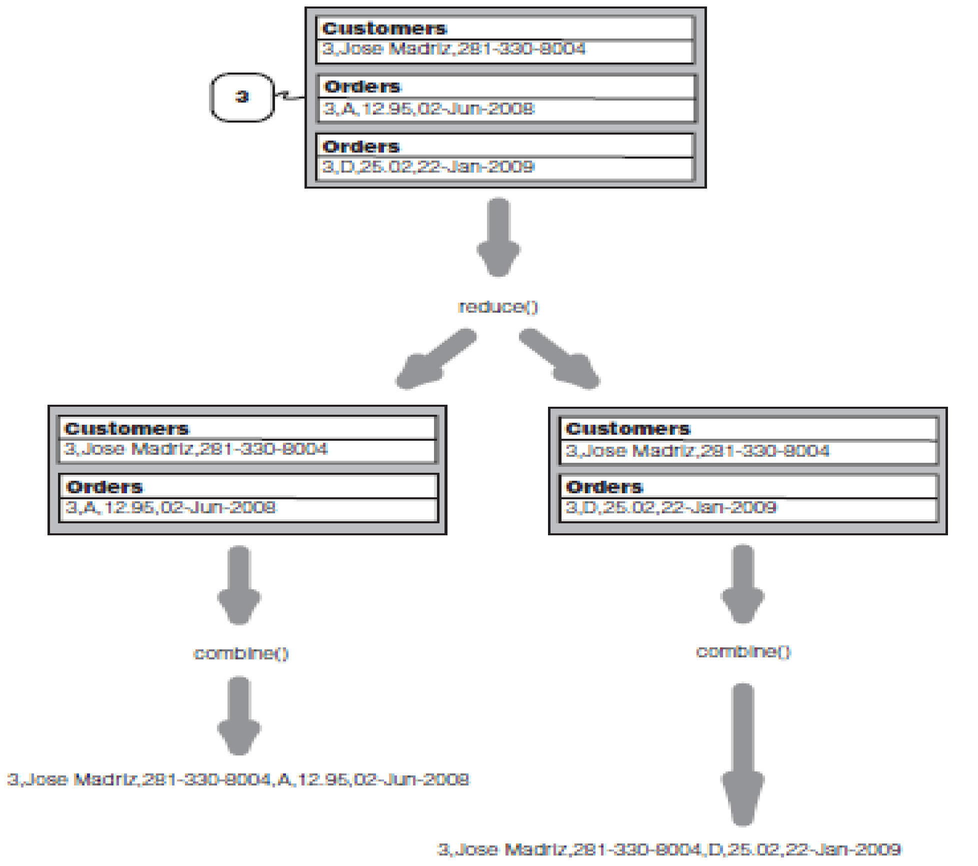
The *`group key`* functions like a join key in a relational database. For our example, the group key is the `Customer ID`.

DATA FLOW OF A REDUCE-SIDE JOIN: The following Figure illustrates the data flow of a repartitioned join on the `toy data sets Customers and Orders`, up to the reduce stage.



In repartitioned join, the mapper first wraps each record with a group key and a tag. The group key is the joining attribute, and the tag is the data source (*table* in SQL parlance) of the record. The partition and shuffle step will group all the records with the same group key together. The reducer is called on the set of records with the same group key.

The function `reduce()` will take its input and do a full *`cross-product`* on the values. `Reduce()` creates all combinations of the values with the constraint that a combination will not be tagged more than once. In cases where `reduce()` sees values of distinct tags, the cross-product is the original set of values. In our example, this is the case for`group keys 1, 2, and 4`. The following Figure illustrates cross product for group key 3.We have three values, one tagged with Customers and two tagged with Orders. The cross-product creates two combinations. Each combination consists of the Customers value and one of the Orders value.


Customers
3,Jose Madriz,281-330-8004
3
Orders
3,A,12.95,02-Jun-2008
Orders
3,D,25.02,22-Jan-2009
reduce()
Customers
3,Jose Madriz,281-330-8004
Orders
3,A,12.95,02-Jun-2008
Customers
3,Jose Madriz,281-330-8004
Orders
3,D,25.02,22-Jan-2009
combine()
combine()
3,Jose Madriz,281-330-8004,A,12.95,02-Jun-2008
3,Jose Madriz,281-330-8004,D,25.02,22-Jan-2009

The reduce side of a repartitioned join. For a given join key, the reduce task performs a full cross-product of values from different sources. It sends each combination to *combine()* to create an output record. The *combine()* function can choose to not output any particular combination.

IMPLEMENTING JOIN WITH THE DATAJOIN PACKAGE: Hadoop‘s datajoin package has three abstract classes that we inherit and make concrete: `DataJoinMapperBase`, `DataJoinReducerBase`, and `TaggedMapOutput`. As the names suggest, our `MapClass` will extend `DataJoinMapperBase`, and our `Reduce class` will extend `DataJoinReducerBase`. The `datajoin` package has already implemented the `map()` and `reduce()` methods in these respective base classes to perform the join dataflow.

Replicated joins using DistributedCache: Hadoop has a mechanism called *`distributed cache`* that‘s designed to distribute files to all nodes in a cluster. Distributed cache is handled by the appropriately named class DistributedCache.

```
public static class MapClass extends MapReduceBase
    implements Mapper<Text, Text, Text, Text> {

    private Hashtable<String, String> joinData =
                                new Hashtable<String, String>();

    @Override
    public void configure(JobConf conf) {
        try {

            Path [] cacheFiles = DistributedCache.getLocalCacheFiles(conf);
            if (cacheFiles != null && cacheFiles.length > 0) {
                String line;
                String[] tokens;

                BufferedReader joinReader = new BufferedReader(
                                new FileReader(cacheFiles[0].toString()));
                try {
                    while ((line = joinReader.readLine()) != null) {
                        tokens = line.split(",", 2);
                        joinData.put(tokens[0], tokens[1]);
                    }
                } finally {
                    joinReader.close();
                }
            }
        } catch (IOException e) {
            System.err.println("Exception reading DistributedCache: " + e);
        }
    }

    public void map(Text key, Text value,
                    OutputCollector<Text, Text> output,
                    Reporter reporter) throws IOException {

        String joinValue = joinData.get(key);
        if (joinValue != null) {
            output.collect(key,
                           new Text(value.toString() + "," + joinValue));
        }
    }
}
```

Semijoin: reduce-side join with map-side filtering: When processing records from Customers and Orders, the mapper will drop any record whose key is not in the set `CustomerID415` (415 is the area code). This is sometimes called a ***`semijoin`***, taking the terminology from the database world.

Last but not least, what if the file `CustomerID415` is still too big to fit in memory? Or maybe `CustomerID415` does fit in memory but its size makes replicating it across all the mappers inefficient. This situation calls for a data structure called a ***`Bloom filter`***. A Bloom filter is a compact representation of a set that supports only the ***contain*** query.

&&&&&&&&&

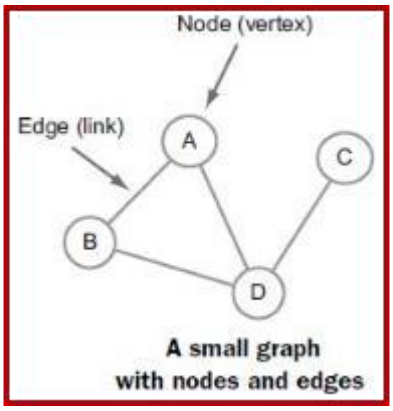
UNIT - IV

Graph Representation in MapReduce: Modeling data and solving problems with graphs, Shortest Path Algorithm, Friends-of-Friends Algorithm, PageRank Algorithm, Bloom Filters. Graph Analytics in Spark, Spark GraphX, GraphX features, GraphX Examples, Usecase. Creating RDDs, Operations, Passing Functions to Spark, Common Transformations andActions, Persistence, Adding Schemas to RDDs, RDDs as Relations, Creating Pairs in RDDs, Transformations and actions on RDDs. Spark SQL, Overview, Libraries, Features, Querying using Spark SQL.

1. Graph Representation in MapReduce

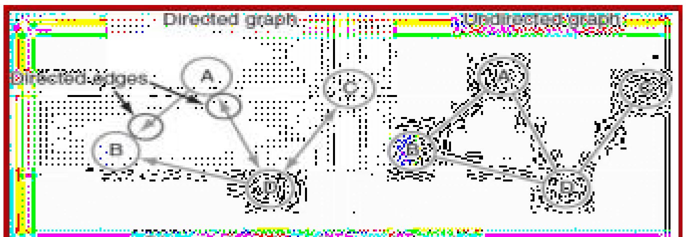
Modeling Data and Solving Problems with Graphs:

A ***`graph`*** consists of a number of nodes (formally called *vertices*) and links (informally called *edges*) that connect nodes together. The following Figure shows a graph with nodes and edges.



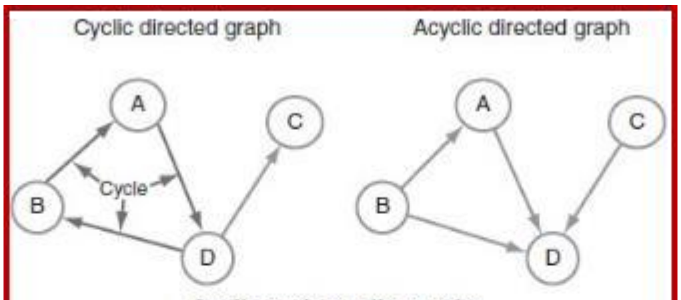
Graphs are mathematical constructs that represent an interconnected set of objects. They"re used to represent data such as the hyperlink structure of the internet, social networks (where they represent relationships between users), and in internet routing to determine optimal paths for forwarding packets.

The edges can be `directed (implying a one-way relationship), or undirected`. For example, we can use a directed graph to model relationships between users in a social network because relationships are not always bidirectional. The following Figure shows examples of directed and undirected graphs.



Directed and undirected graphs

Graphs can be `cyclic` or `acyclic`. In cyclic graphs it‟s possible for a vertex to reach itself by traversing a sequence of edges. In an acyclic graph it‟s not possible for a vertex to traverse a path to reach itself. The following Figure shows examples of cyclic and acyclic graphs.

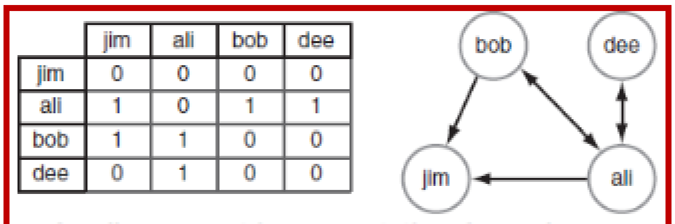


Cyclic and acyclic graphs

Modeling Graphs: There are two common ways of representing graphs are with *`adjacency matrices`* and with *`adjacencylists`*.

ADJACENCY MATRIX: In this matrix, we represent a graph as an *N x N* square matrix *M*, where *N* is the number of nodes, and M_{ij} represents an edge between nodes *i* and *j*.

The following Figure shows a directed graph representing connections in a social graph. The arrows indicate a one-way relationship between two people. The adjacency matrixshows how this graph would be represented.

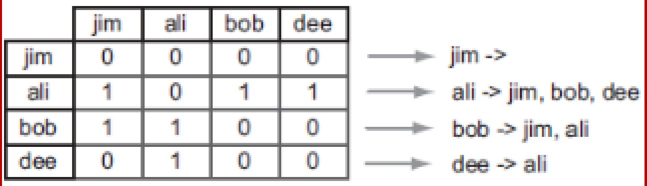


	jim	ali	bob	dee
jim	0	0	0	0
ali	1	0	1	1
bob	1	1	0	0
dee	0	1	0	0

An adjacency matrix representation of a graph

The disadvantage of adjacency matrices are that they model both the existence and lack of a relationship, which makes them a dense data structure.

ADJACENCY LIST: Adjacency lists are similar to adjacency matrices, other than the fact that they don‟t model the lack of relationship. The following Figure shows an adjacency list to theabove graph.



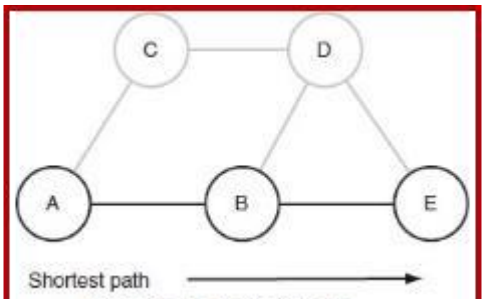
	jim	ali	bob	dee
jim	0	0	0	0
ali	1	0	1	1
bob	1	1	0	0
dee	0	1	0	0

jim ->
ali -> jim, bob, dee
bob -> jim, ali
dee -> ali

An adjacency list representation of a graph

The advantage of the adjacency list is that it offers a sparse representation of the data, which is good because it requires less space. It also fits well when representing graphs in MapReduce because the key can represent a vertex, and the values are a list of vertices that denote a directed or undirected relationship node.

Shortest path algorithm: This algorithm is a common problem in graph theory, where the goal is to find the shortest route between two nodes. The following Figure shows an example of this algorithm on a graph where the edges don‟t have a weight, in which case the shortest path is the path with the smallest number of hops, or intermediary nodes between the source and destination.



Example of shortest path between nodes A and E

Applications of this algorithm include traffic mapping software to determine the shortest route between two addresses, routers that compute the shortest path tree for each route, and social networks to determine connections between users.

Find the shortest distance between two users: Dijkstra‟s algorithm is a shortest path algorithm and its basic implementation uses a sequential iterative process to traverse the entire graph from the starting node.

Problem:

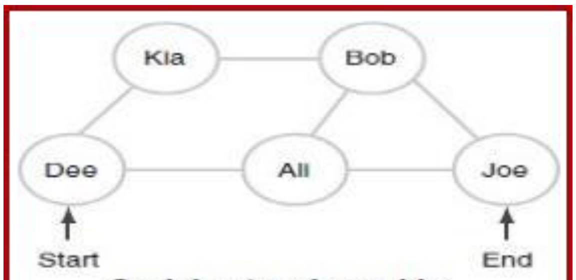
We need to use MapReduce to find the shortest path between two people in a social graph.

Solution:

Use an adjacency list to model a graph, and for each node store the distance from the original node, as well as a backpointer to the original node. Use the mappers to propagate the

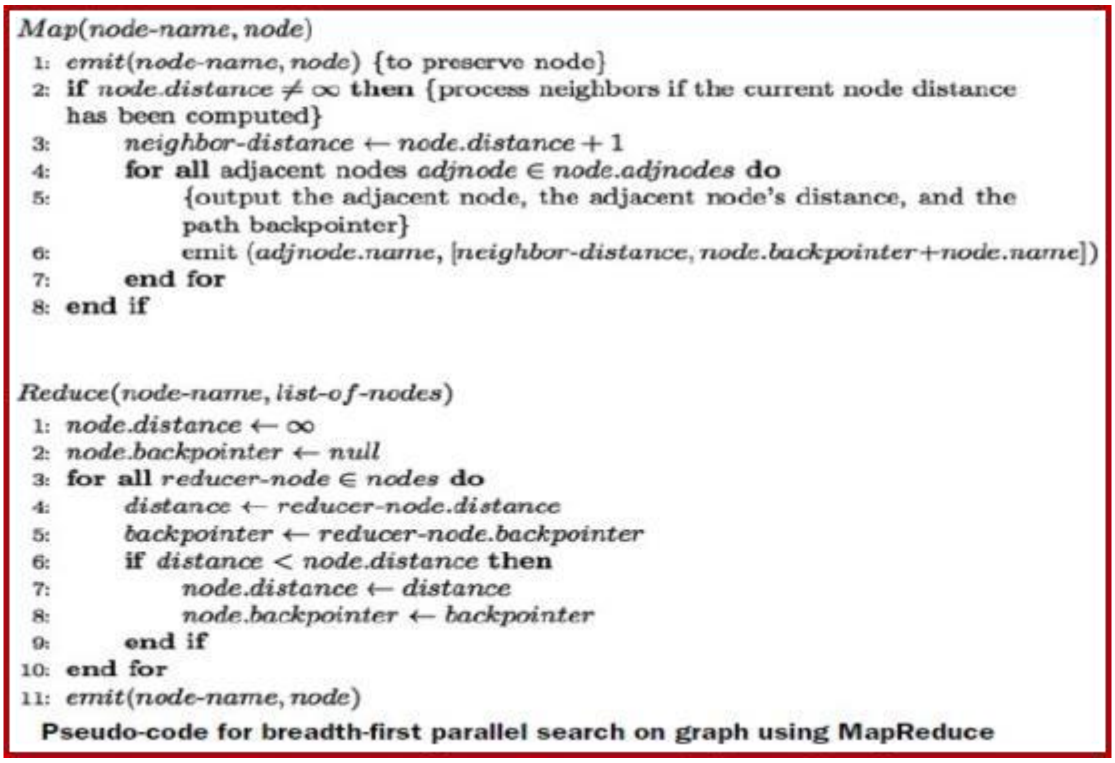
distance to the original node, and the reducer to restore the state of the graph. Iterate until the target node has been reached.

Discussion: The following Figure shows a small social network, which we‟ll use for this technique. Our goal is to find the shortest path between Dee and Joe. There are four paths that we can take from Dee to Joe, but only one of them results in the fewest number of hops.



Social network used in this technique

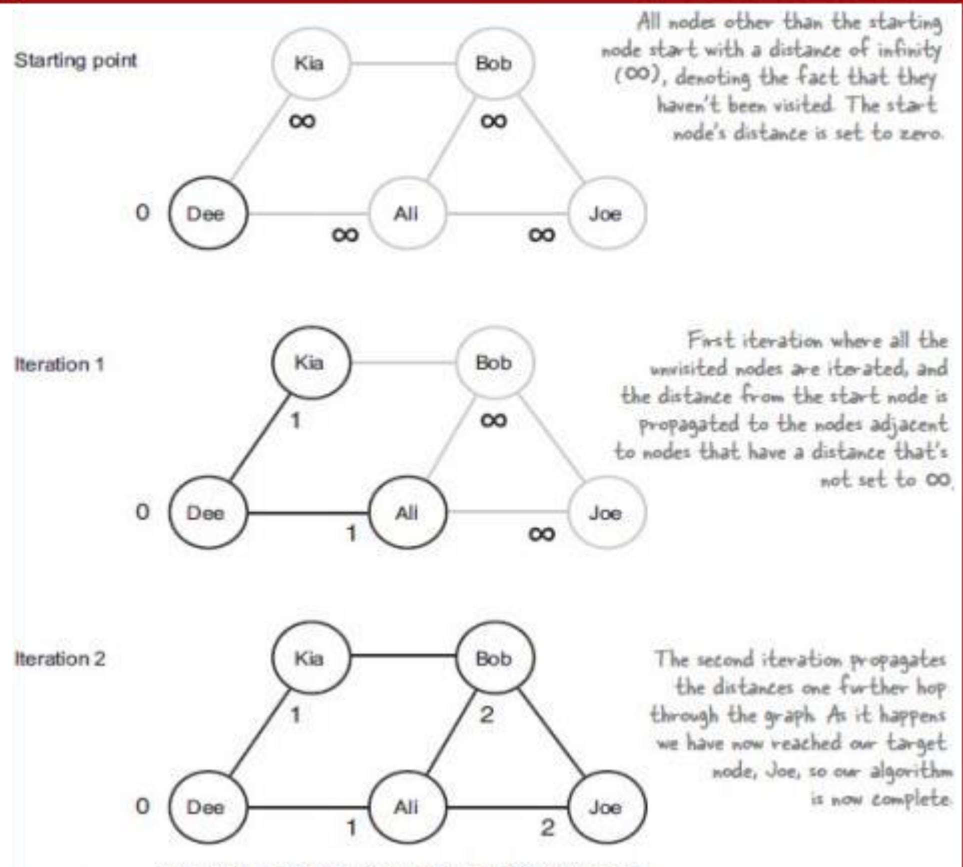
We‟ll implement a parallel breadth-first search algorithm to find the shortest path between two users. Because we‟re operating on a social network, we don‟t need to care about weights on our edges. The pseudo-code for the algorithm is described as the following:



```
Map(node-name, node)
1: emit(node-name, node) {to preserve node}
2: if node.distance $\neq \infty$ then {process neighbors if the current node distance
   has been computed}
3:     neighbor-distance $\leftarrow$ node.distance + 1
4:     for all adjacent nodes adjnode $\in$ node.adjnodes do
5:         {output the adjacent node, the adjacent node's distance, and the
           path backpointer}
6:         emit (adjnode.name, [neighbor-distance, node.backpointer+node.name])
7:     end for
8: end if


Reduce(node-name, list-of-nodes)
1: node.distance $\leftarrow \infty$
2: node.backpointer $\leftarrow$ null
3: for all reducer-node $\in$ nodes do
4:     distance $\leftarrow$ reducer-node.distance
5:     backpointer $\leftarrow$ reducer-node.backpointer
6:     if distance $<$ node.distance then
7:         node.distance $\leftarrow$ distance
8:         node.backpointer $\leftarrow$ backpointer
9:     end if
10: end for
11: emit(node-name, node)
```

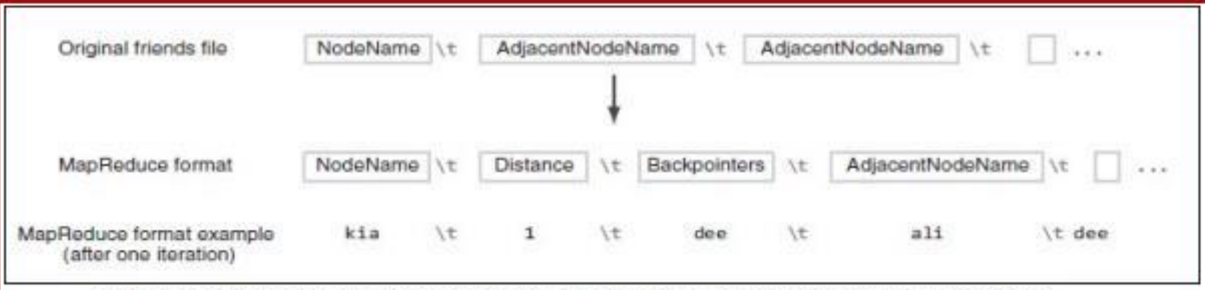
Pseudo-code for breadth-first parallel search on graph using MapReduce

The following Figure shows the algorithm iterations in play with our social graph. Just like Dijkstra‟s algorithm, we‟ll start with all the node distances set to infinite, and set the distance for the starting node, Dee, at zero. With each MapReduce pass, we‟ll determine nodes that don‟t have an infinite distance and propagate their distance values to their adjacentnodes. We continue this until we reach the end node.


Starting point
Kia
Bob
∞
∞
0
Dee
All
Joe
∞
∞
All nodes other than the starting node start with a distance of infinity (∞), denoting the fact that they haven't been visited. The start node's distance is set to zero.
Iteration 1
Kia
Bob
1
∞
0
Dee
All
Joe
1
∞
First iteration where all the unvisited nodes are iterated, and the distance from the start node is propagated to the nodes adjacent to nodes that have a distance that's not set to ∞.
Iteration 2
Kia
Bob
1
2
0
Dee
Ali
Joe
1
2
The second iteration propagates the distances one further hop through the graph. As it happens we have now reached our target node, Joe, so our algorithm is now complete.

Shortest path iterations through the network

We first need to create the starting point. This is done by reading in the social network (which is stored as an adjacency list) from file and setting the initial distance values. The following Figure shows the two file formats, the second being the format that‟s used iteratively in our MapReduce code.


Original friends file
NodeName \t AdjacentNodeName \t AdjacentNodeName \t ...
MapReduce format
NodeName \t Distance \t Backpointers \t AdjacentNodeName \t ...
MapReduce format example (after one iteration)
kia \t 1 \t dee \t ali \t dee

Original social network file format and MapReduce form optimized for algorithm

The reducer calculates the minimum distance for each node and to output the minimum distance, the backpointer, and the original adjacent nodes.

Friends-of-friends (FoFs):

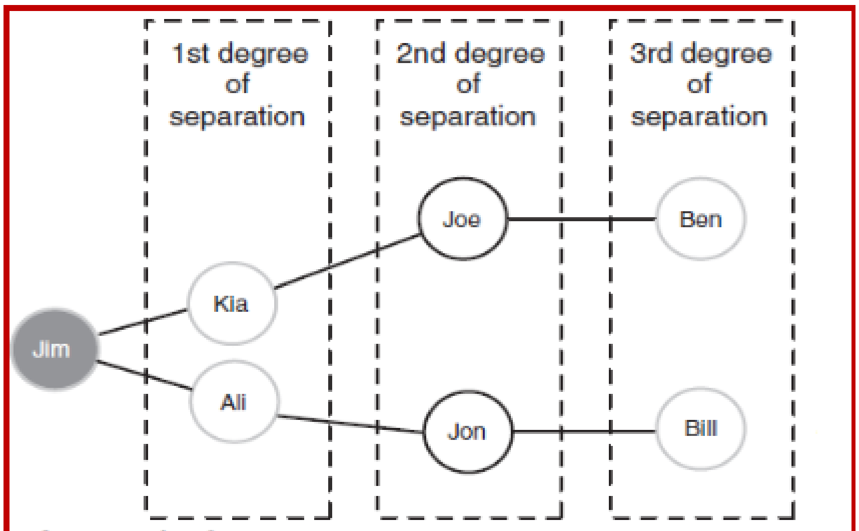
The FoF algorithm is using by Social network sites such as LinkedIn and Facebook to help users broaden their networks. The Friends-of-friends (FoF) algorithm suggests friends that a user may know that aren‟t part of their immediate network. The following Figure shows FoF to be in the 2nd degree network.

Problem

We want to implement the FoF algorithm in MapReduce.

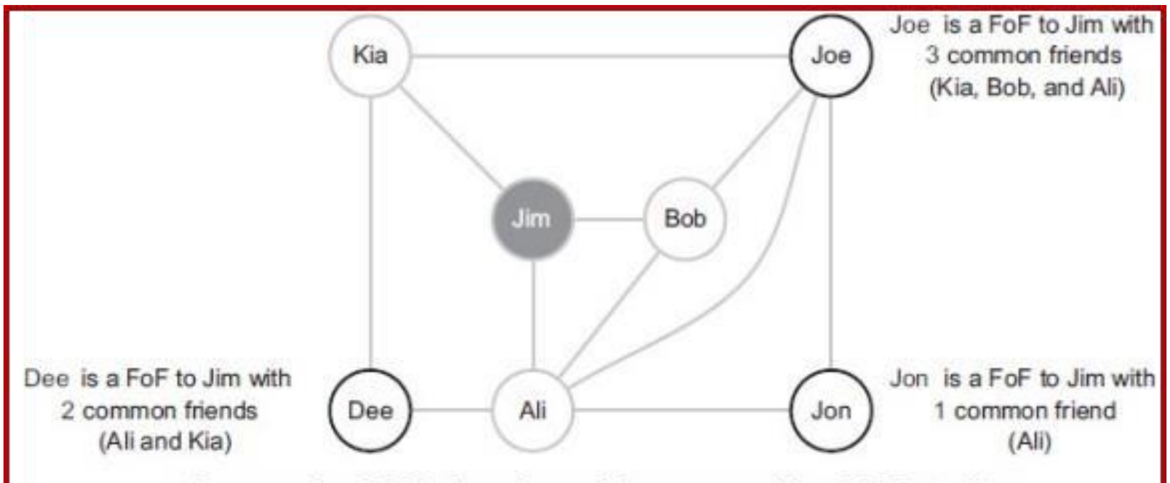
Solution

Two MapReduce jobs are required to calculate the FoFs for each user in a social network. The first job calculates the common friends for each user, and the second job sorts the common friends by the number of connections to our friends.



An example of FoFs where Joe and Jon are considered FoFs to Jim

The following Figure shows a network of people with Jim, one of the users, highlighted.


Kia
Joe
Joe is a FoF to Jim with 3 common friends (Kia, Bob, and Ali)
Jim
Bob
Dee is a FoF to Jim with 2 common friends (Ali and Kia)
Dee
Ali
Jon
Jon is a FoF to Jim with 1 common friend (Ali)

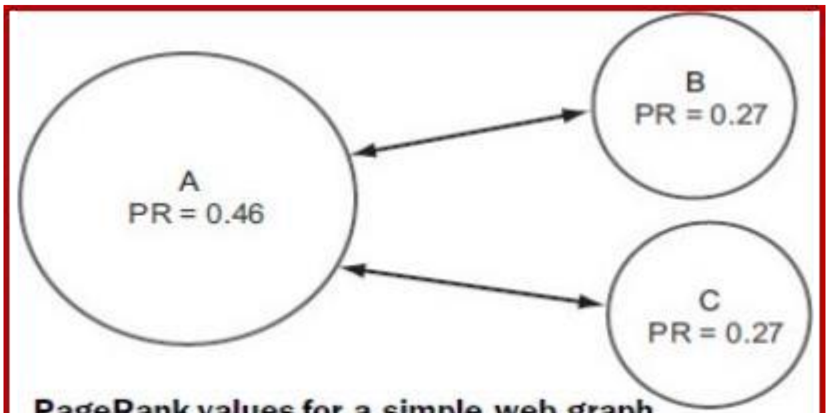
An example of FoF where Joe and Jon are considered FoFs to Jim

In above graph Jim‟s FoFs are represented in bold (Dee, Joe, and Jon). Next to Jim‟s FoFs is the number of friends that the FoF and Jim have in common. Our goal here is to determine all the FoFs and order them by the number of fiends in common. Therefore, our expected results would have Joe as the first FoF recommendation, followed by Dee, and then Jon.

PageRank:

PageRank was a formula introduced by the founders of Google during their Stanford years in 1998. *PageRank*, which gives a score to each web page that indicates the page‟s importance.

Calculate PageRank over a web graph: PageRank uses the scores for all the inbound links to calculate a page‟s PageRank. But it disciplines individual inbound links from sources that have a high number of outbound links by dividing that outbound link PageRank by the number of outbound links. The following Figure presents a simple example of a web graph with three pages and their respective PageRank values.



PageRank values for a simple web graph

$$PageRank(n) = \frac{1-d}{|webGraph|} + d \sum_{i \in InboundLinks(n)} \frac{PageRank(i)}{|i.outboundLinks|}$$

The PageRank formula

In the above formula, |*webGraph*| is a count of all the pages in the graph, and *d*, set to 0.85, is a constant damping factor used in two parts. First, it denotes the probability of a random surfer reaching the page after clicking on many links (this is a constant equal to 0.15 divided by the total number of pages), and, second, it dampens the effect of the inbound link PageRanks by 85 percent.

***Problem*:**

We want to implement an iterative PageRank graph algorithm in MapReduce.

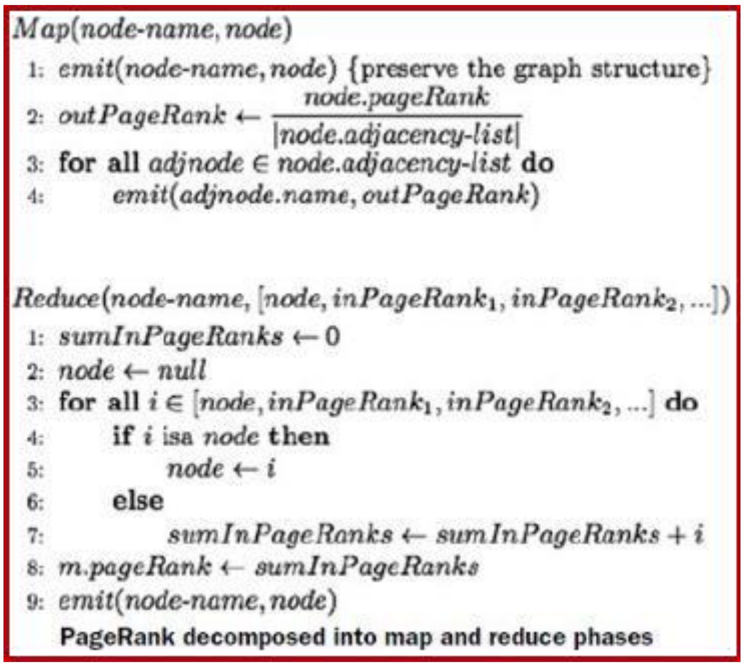
***Solution*:**

PageRank can be implemented by iterating a MapReduce job until the graph has converged. The mappers are responsible for propagating node PageRank values to their adjacent nodes, and the reducers are responsible for calculating new PageRank values for each node, and for re-creating the original graph with the updated PageRank values.

Discussion*:*

One of the advantages of PageRank is that it can be computed iteratively and applied locally. Every vertex starts with a seed value, with is 1 divided by the number of nodes, and with each iteration each node propagates its value to all pages it links to. Each vertex in turn sums up the value of all the inbound vertex values to compute a new seed value. This iterative process is repeated until such a time as convergence is reached. Convergence is a measure of how much the seed values have changed since the last iteration. If the convergence value is below a certain threshold, it means that there‟s been minimal change and we can stop the iteration. It‟s also common to limit the number of iterations in cases of large graphs where convergence takes too much iteration

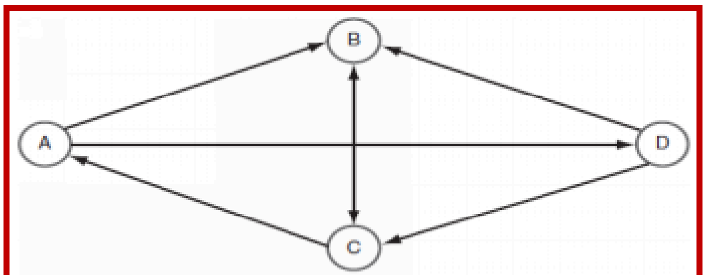
The following PageRank algorithm expressed as map and reduces parts. The map phase is responsible for preserving the graph as well as emitting the PageRank value to all the outbound nodes. The reducer is responsible for recalculating the new PageRank value for each node and including it in the output of the original graph.



```
Map(node-name, node)
1: emit(node-name, node) {preserve the graph structure}
2: outPageRank ← $\frac{node.pageRank}{|node.adjacency\text{-}list|}$
3: for all adjnode ∈ node.adjacency-list do
4:     emit(adjnode.name, outPageRank)


Reduce(node-name, [node, $inPageRank_1$, $inPageRank_2$, ...])
1: sumInPageRanks ← 0
2: node ← null
3: for all i ∈ [node, $inPageRank_1$, $inPageRank_2$, ...] do
4:     if i isa node then
5:         node ← i
6:     else
7:         sumInPageRanks ← sumInPageRanks + i
8: m.pageRank ← sumInPageRanks
9: emit(node-name, node)
```

PageRank decomposed into map and reduce phases

This technique is applied on the following graph. All the nodes of the graph have both inbound and output edges.



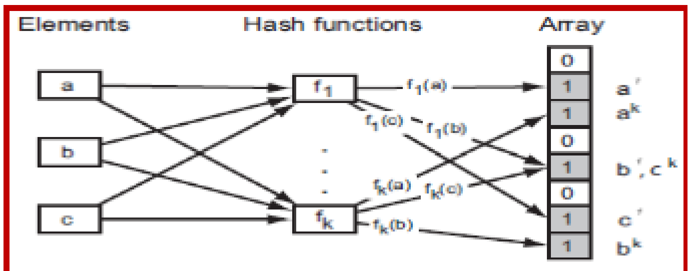
Sample web graph for this technique

Bloom Filters:

A Bloom filter is a data structure that offers a membership query mechanism where the answer to a lookup is one of two values: a definitive *no*, meaning that the item being looked up doesn‟t exist in the Bloom filter, or a *maybe*, meaning that there‟s a probability thatthe item exists.

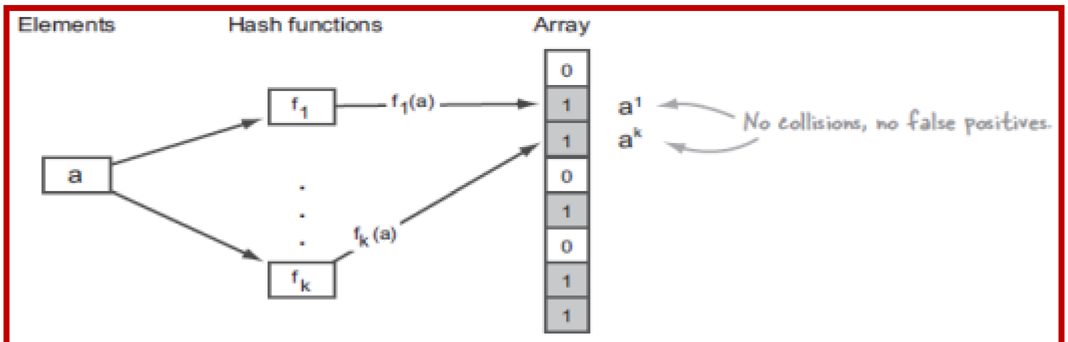
Bloom filters are used in BigTable and in HBase to remove the need to read blocks from disk to determine if they contain a key. The implementation of Bloom filters is simple. They use a bit array of size *m* bits, where initially each bit is set to 0 (zero). They also contain *k* hash functions, which are used to map elements to *k* locations in the bit array.

To add an element into a Bloom filter, it‟s hashed *k* times, and a modulo of the hashed value and the size of the bit array is used to map the hashed value to a specific bit array location. That bit in the bit array is then toggled to 1 (one). The following Figure shows three elements being added to a Bloom filter and their locations in the bit array.



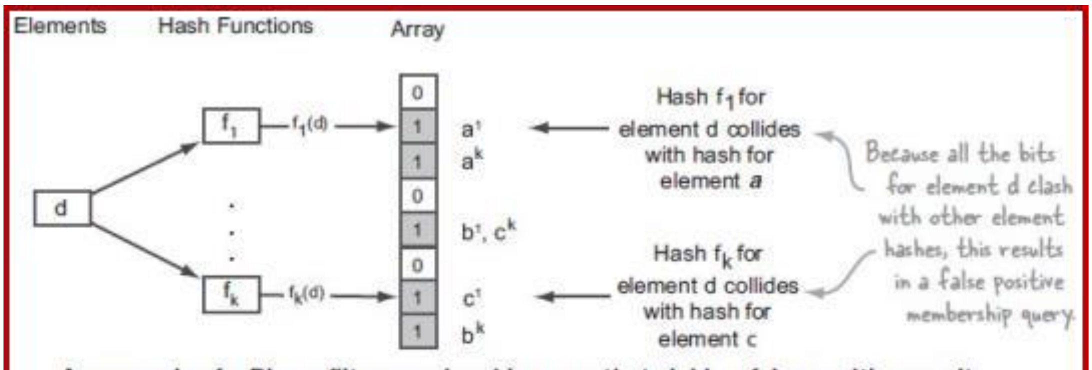
Adding elements to a Bloom filter

To check the membership of an element in the Bloom filter, just like with the add operation, the element is hashed *k* times and each hash key is used to index into the bit array. A true response to the membership query is only returned in cases where all *k* bit array locations are set to 1. Otherwise, the response to the query is false. The following Figure shows an example of a membership query where the item was previously added to the Bloom filter, and therefore all the bit array locations contained a 1. This is an example of a true positive membership result because the element had been previously added to the Bloom filter.



An example of a Bloom filter membership query that yields a true positive result

The next example shows how we can get a false positive result for a membership query. The element being queried is *d*, which hadn‟t been added to the Bloom filter. As it happens, all *k* hashes for *d* are mapped to locations that are set to 1 from other elements. The following Figure is an example of collision in the Bloom filter where the result is a false positive.


Elements
Hash Functions
Array
d
f_1
$f_1(d)$
f_k
$f_k(d)$
0
1
1
0
1
0
1
1
a^1
a^k
b^1, c^k
c^1
b^k
Hash f_1 for element d collides with hash for element *a*
Hash f_k for element d collides with hash for element c
Because all the bits for element d clash with other element hashes, this results in a false positive membership query.

An example of a Bloom filter membership query that yields a false positive result

The probability of false positives can be tuned based on two factors: *m*, the number of bits in the bit array, and *k*, the number of hash functions. Or expressed another way, if we have a desired false positive rate in mind, and we know how many elements will be added to the Bloom filter. We can calculate the number of bits needed in the bit array with the following equation.

$$m = -\frac{n \ln p}{(\ln(2))^2}$$

m is required number of bits in the bit array to achieve the desired false probability rate of p for n inserted elements

n is the number of elements inserted

p is the desired false positive rate (0.01 means 1%)

Equation to calculate the desired number of bits for a Bloom filter bit array

MapReduce semi-join with Bloom filters:

The semi-join was performed using a HashMap to cache the smaller of the datasets. Bloom filters can replace a HashMap in the semi-join if we don‟t care about storing the contents of the dataset being cached, and we only care about whether the element on which we‟re performing the join exists or doesn‟t exist in the other dataset.

Problem

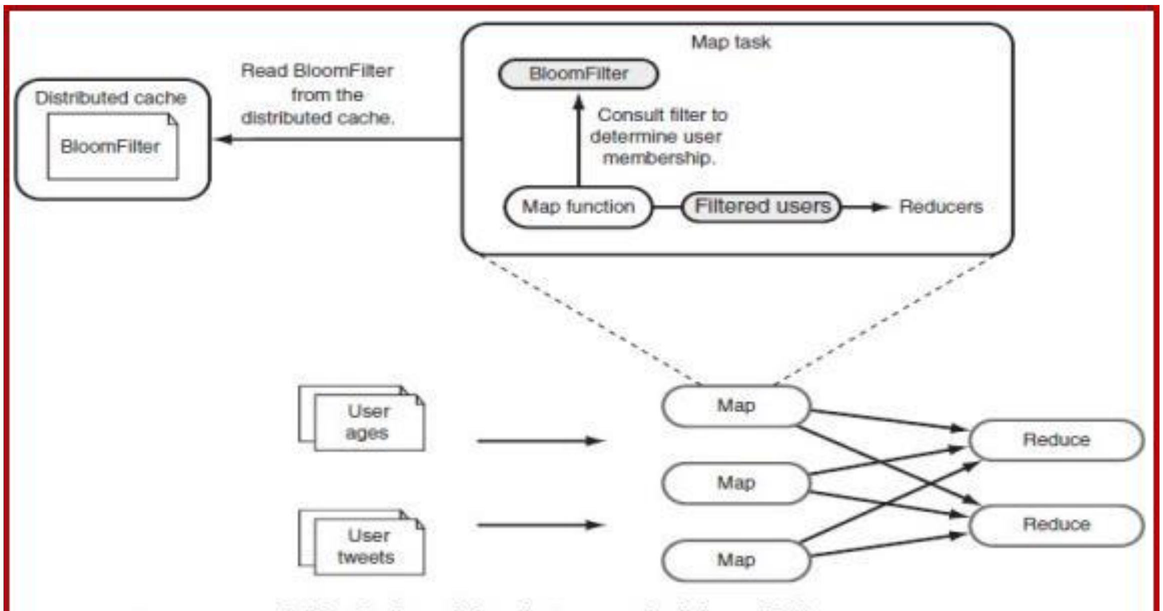
We want to perform an efficient semi-join in MapReduce.

Solution

Perform a semi-join by having the mappers load a Bloom filter from the Distributed Cache, and then filter results from the actual MapReduce data source by performing membership queries against the Bloom filter to determine which data source records should be emitted to the reducers.

Discussion

The key to this technique is to create a Bloom filter on a reasonably sized dataset. We‟ll extend the work we performed in the previous technique, where we created a Bloom filter on all users of a certain age. Imagine we wanted to join that set of users with all their tweets, which is a much larger dataset than the set of users. The Figure shows how we would filter data in a mapper such that the join in MapReduce operates on a smaller set of the overall data.



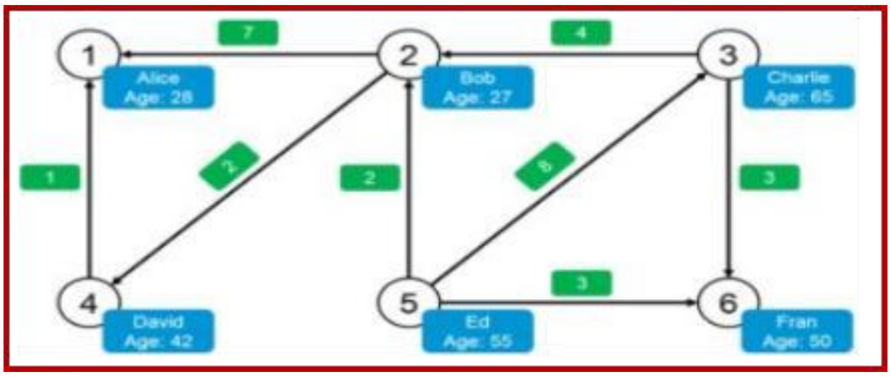
A MapReduce job using a map to join and filter

****

2. Graph Analytics in Spark

What is Spark GraphX?

GraphX is the Spark API for graphs and graph-parallel computation. It includes a growing collection of graph algorithms and builders to simplify graph analytics tasks.



Spark GraphX – Graph Example

Spark GraphX Features: The following are the features of Spark GraphX.

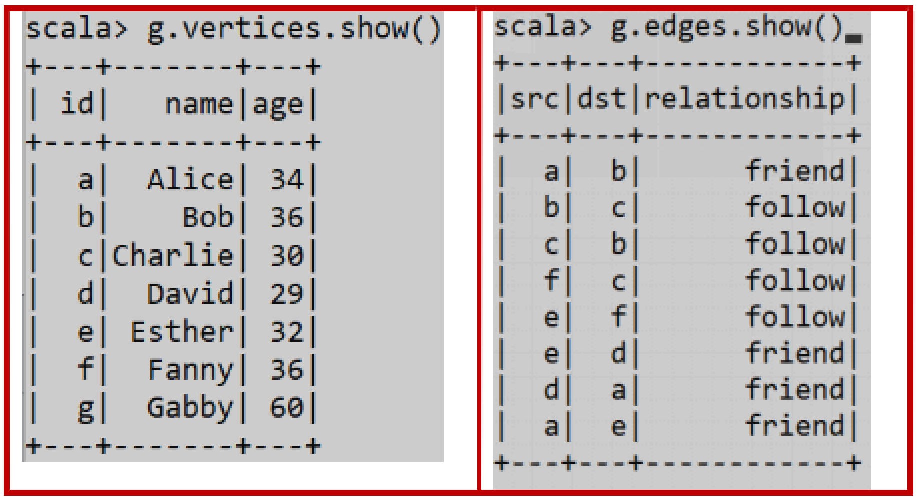
1. **Flexibility:** Spark GraphX works with both graphs and computations. GraphX unifies ETL (Extract, Transform & Load), exploratory analysis and iterative graph computation within a single system. We can view the same data as both graphs and collections, transform and join graphs with RDDs efficiently and write custom iterative graph algorithms using the Pregel API.
2. **Speed:** Spark GraphX provides comparable performance to the fastest specialized graph processing systems. It is comparable with the fastest graph systems while retaining Spark‟s flexibility, fault tolerance and ease of use.
3. **Growing Algorithm Library:** We can choose from a growing library of graph algorithmsthat Spark GraphX has to offer. Some of the popular algorithms are page rank, connectedcomponents, label propagation, SVD++, strongly connected components and triangle count. **GraphX with Examples:** We will now understand the concepts of Spark GraphX using thefollowing graph example.

We can represent the network as a graph, which is a set of vertices (users) and edges (connections between users). The graph here represents the Twitter users and whom they follow on Twitter. For e.g. Bob follows David and Alice on Twitter. This is implemented in Apache Spark as the following.

```
import org.apache.spark.sql.SparkSession
import org.graphframes.GraphFrame

object GraphFrameDemo {
```

```
def main (args: Array[String]): Unit ={

     val spark:SparkSession =
 SparkSession.builder().master("local[1]").appName("GraphFrameDemo
 ").getOrCreate()

     //Create the vertices and edges
     // Vertex DataFrame
     val v = spark.createDataFrame(List(
       ("a", "Alice", 34),
       ("b", "Bob", 36),
       ("c", "Charlie", 30),
       ("d", "David", 29),
       ("e", "Esther", 32),
       ("f", "Fanny", 36),
       ("g", "Gabby", 60)
     )).toDF("id", "name", "age")
     // Edge DataFrame
     val e = spark.createDataFrame(List(
       ("a", "b", "friend"),
       ("b", "c", "follow"),
       ("c", "b", "follow"),
       ("f", "c", "follow"),
       ("e", "f", "follow"),
       ("e", "d", "friend"),
       ("d", "a", "friend"),
       ("a", "e", "friend")
     )).toDF("src", "dst", "relationship")

     val g = GraphFrame(v, e)

    // Display the vertex and edge DataFrames
     g.vertices.show()

     g.edges.show()
   }
 }
```

O/P:



```
scala> g.vertices.show()
+---+-------+---+
| id|   name|age|
+---+-------+---+
|  a|  Alice| 34|
|  b|    Bob| 36|
|  c|Charlie| 30|
|  d|  David| 29|
|  e| Esther| 32|
|  f|  Fanny| 36|
|  g|  Gabby| 60|
+---+-------+---+
```

```
scala> g.edges.show()
+---+---+------------+
|src|dst|relationship|
+---+---+------------+
|  a|  b|      friend|
|  b|  c|      follow|
|  c|  b|      follow|
|  f|  c|      follow|
|  e|  f|      follow|
|  e|  d|      friend|
|  d|  a|      friend|
|  a|  e|      friend|
+---+---+------------+
```

Creating RDDs:

Spark provides two ways to create RDDs: loading an external dataset and parallelizing a collection in your driver program. The simplest way to create RDDs is to take an existing collection in your program and pass it to SparkContext‟s parallelize() method. This approach is very useful when you are learning Spark, since you can quickly create your own RDDs in the shell and perform operations on them.

Ex:

# parallelize() method in Scala

val lines = sc.parallelize(**List**("pandas", "i like pandas"))

A more common way to create RDDs is to load data from external storage using SparkContext.textFile(), which is shown in the following examples.

# textFile() method in Scala

val lines = sc.textFile("/path/to/README.md")

RDD Operations:

RDDs support two types of operations: **transformations and actions**. Transformations are operations on RDDs that return a new RDD, such as map() and filter(). Actions are operations that return a result to the driver program or write it to storage, and kick off a computation, such as count() and first(). Spark treats transformations and actions very differently.

As an example, suppose that we have a logfile, log.txt, with a number of messages, and we want to select only the error messages. We can use the filter() transformation in all three of Spark‟s language APIs.

// filter() transformation in Scala

val inputRDD = sc.textFile("log.txt")

val errorsRDD = inputRDD.filter(line => line.contains("error"))

Note that the filter() operation does not mutate the existing inputRDD. Instead, it returns a pointer to an entirely new RDD.

Actions:

To print out some information about the badLinesRDD of log file can use two actions, count(), which returns the count as a number, and take(), which collects a number of elementsfrom the RDD, as shown in the following Examples.

// Scala error count using actions

```
println("Input had " + badLinesRDD.count() + " concerning
lines")println("Here are 10 examples:")
badLinesRDD.take(10).foreach(println)
```

In the above example, the take() function can retrieve a small number of elements in the RDD at the driver program and then iterate over them locally to print out information at the driver. RDDs also have a collect() function to retrieve the entire RDD. This can be useful if our program filters RDDs down to a very small size and we wouldlike to deal with it locally. Keep in mind that our entire dataset must fit in memory on a single machine to use collect() on it, so collect() shouldn‟t be used on large datasets.

Lazy Evaluation:

Lazy evaluation means that when we call a transformation on an RDD (for instance, calling map()), the operation is not immediately performed. Instead, Spark internally records metadata to indicate that this operation has been requested. Loading data into an RDD is lazily evaluated in the same way transformations are. So, when we callsc.textFile(), the data is not loaded until it is necessary. As with transformations, the operation (in this case, reading the data) can occur multiple times. Spark uses lazy evaluation to reduce the number of passes it has to take over our data by grouping operations together.

Passing Functions to Spark:

Most of Spark‟s transformations, and some of its actions, depend on passing in functions that are used by Spark to compute data. Each of the core languages has a slightly different mechanism for passing functions to Spark.

Scala:

In Scala, we can pass in functions defined inline, references to methods, or static functions as we do for Scala‟s other functional APIs.

# Scala function passing

```
class SearchFunctions(val query:
```

```
String) {def isMatch(s: String): Boolean
= { s.contains(query)
}
def getMatchesFunctionReference(rdd: RDD[String]): RDD[String] = {
// Problem: "isMatch" means "this.isMatch", so we pass all of "this"
rdd.map(isMatch)
}
```

Persistence (Caching):

To avoid computing an RDD multiple times, we can ask Spark to persist the data. When we ask Spark to persist an RDD, the nodes that compute the RDD store their partitions. If a node that has data persisted on it fails, Spark will recomputed the lost partitions of the data when needed. We can also replicate our data on multiple nodes if we want to be able to handle node failure without slowdown. Spark has many levels of persistence to choose from based on what our goals are, as shown in the following table.

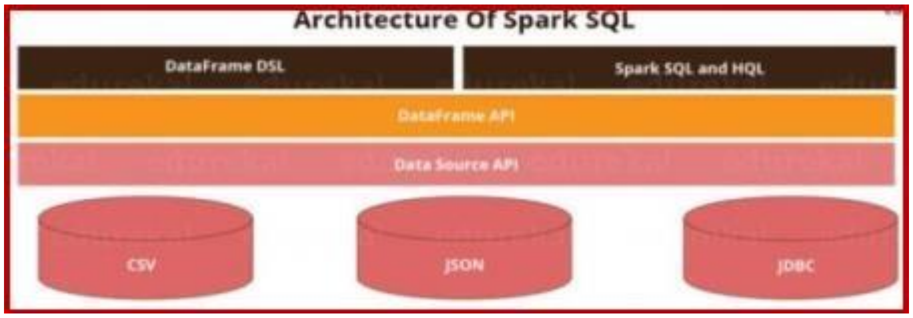
Without persist()	With persist()

# Double execution in Scala 	
 	
val result = input.map(x => x*x)
println(result.count())
println(result.collect().mkString(","))
	# persist() in Scala
	
	val result = input.map(x => x * x)
	result.persist(**StorageLevel**.**DISK_ONLY**)
	println(result.count())
	
	println(result.collect().mkString(","))

Spark SQL Overview:

Spark SQL integrates relational processing with Spark‟s functional programming. It provides support for various data sources and makes it possible to weave SQL queries with code transformations thus resulting in a very powerful tool.

With Spark SQL, Apache Spark is accessible to more users and improves optimization for the current ones. Spark SQL provides DataFrame APIs which perform relational operations on both external data sources and Spark‟s built-in distributed collections. It introduces an extensible optimizer called Catalyst as it helps in supporting a wide range of data sources and algorithms in Big-data.



Spark SQL Libraries:

Spark SQL has the following four libraries which are used to interact with relational and procedural processing:

a) Data Source API (Application Programming Interface): This is a universal API for loadingand storing structured data.

i. It has built in support for Hive, Avro, JSON, JDBC, Parquet, etc.
ii. Supports third-party integration through Spark packages.
iii. Support for smart sources.
iv. It is a Data Abstraction and Domain Specific Language (DSL) applicable on thestructure and semi-structured data.

b) DataFrame API: DataFrame API is a distributed collection of data in the form of namedcolumn and row.

i. It is lazily evaluated like Apache Spark Transformations and can be accessed throughSQL Context and Hive Context.
ii. It processes the data in the size of Kilobytes to Petabytes on a single-node cluster tomulti-node clusters.
iii. Supports different data formats (Avro, CSV, Elastic Search, and Cassandra) and storage systems (HDFS, HIVE Tables, MySQL, etc.).
iv. Can be easily integrated with all Big Data tools and frameworks via Spark-Core.
v. Provides API for Python, Java, Scala, and R Programming.
vi. A DataFrame is a distributed collection of data organized into a named column. It is equivalent to a relational table in SQL used for storing data into tables.

Features of Spark SQL: The following are the features of Spark SQL:

1. **Integrated** − Seamlessly mix SQL queries with Spark programs. Spark SQL lets you query structured data as a distributed dataset (RDD) in Spark, with integrated APIs in Python, Scala and Java. This tight integration makes it easy to run SQL queries alongside complex analytic algorithms.
2. **Unified Data Access** − Load and query data from a variety of sources. Schema- RDDs provide a single interface for efficiently working with structured data, includingApache Hive tables, parquet files and JSON files.
3. **Hive Compatibility** − Run unmodified Hive queries on existing warehouses. Spark SQL reuses the Hive frontend and MetaStore, giving you full compatibility with existing Hive data, queries, and UDFs. Simply install it alongside Hive.
4. **Standard Connectivity** − Connect through JDBC or ODBC. Spark SQL includes a server mode with industry standard JDBC and ODBC connectivity.
5. **Scalability** − Use the same engine for both interactive and long queries. Spark SQL takes advantage of the RDD model to support mid-query fault tolerance, letting it scale to large jobs too. Do not worry about using a different engine for historical data.

Querying Using Spark SQL:

1. Start the Spark Shell. Go to the Spark directory and execute ./bin/spark-shell in the terminal to being the Spark Shell.
2. Let two files, "employee.txt" and "employee.json". The images below show the content ofboth the files. Both these files are stored at

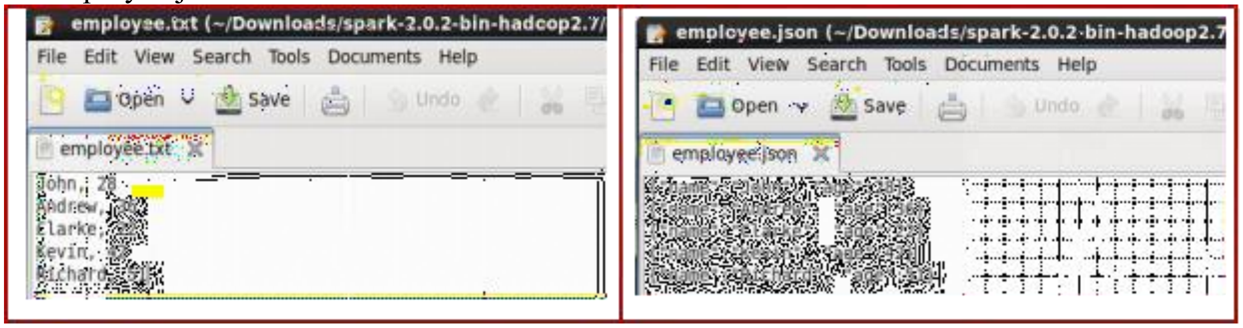
„examples/src/main/scala/org/apache/spark/examples/sql/SparkSQLExample.scala" inside the folder.

Code explanation:

1. First import a Spark Session into Apache Spark.
2. Creating a Spark Session „spark" using the „builder()" function.
3. Importing the Implicits class into our „spark" Session.

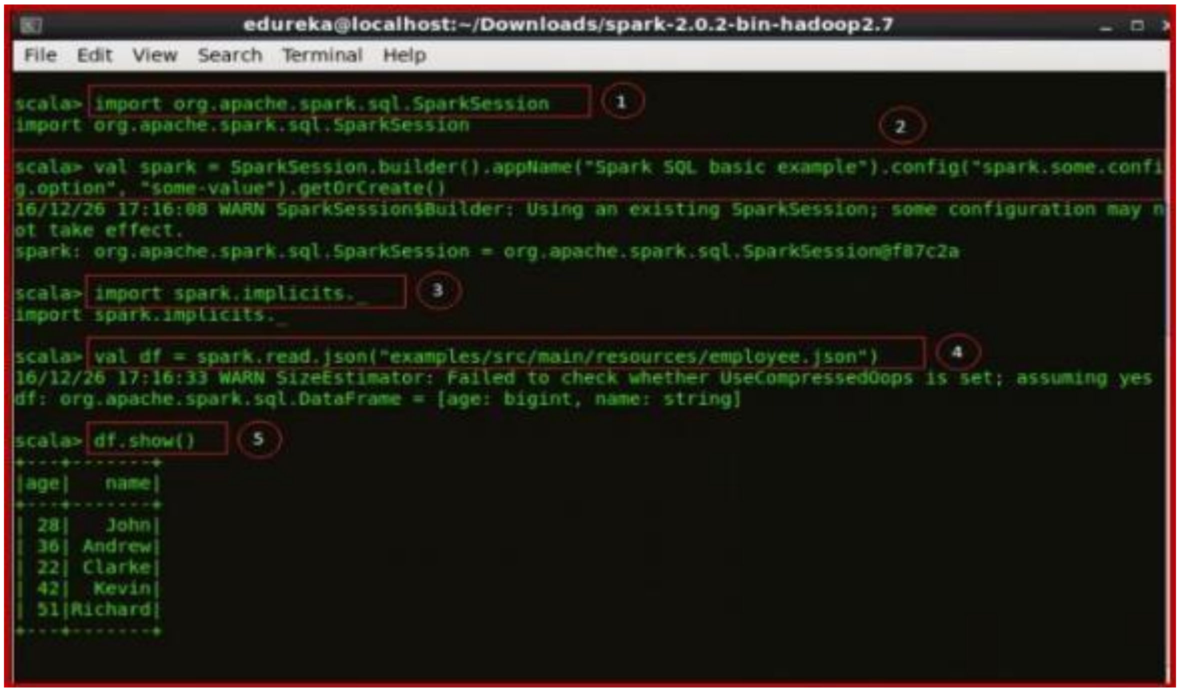
4. Now create a DataFrame „df" and import data from the "employee.json" file.

5. Displaying the DataFrame „df". The result is a table of 5 rows of ages and names from our "employee.json" file.


employee.txt (~/Downloads/spark-2.0.2-bin-hadoop2.7/
File Edit View Search Tools Documents Help
Open
Save
Undo
employee.txt
employee.json (~/Downloads/spark-2.0.2-bin-hadoop2.7
File Edit View Search Tools Documents Help
Open
Save
Undo
employee.json

```
John, 28
Andrew,
Clarke,
Kevin,
Richard,
```

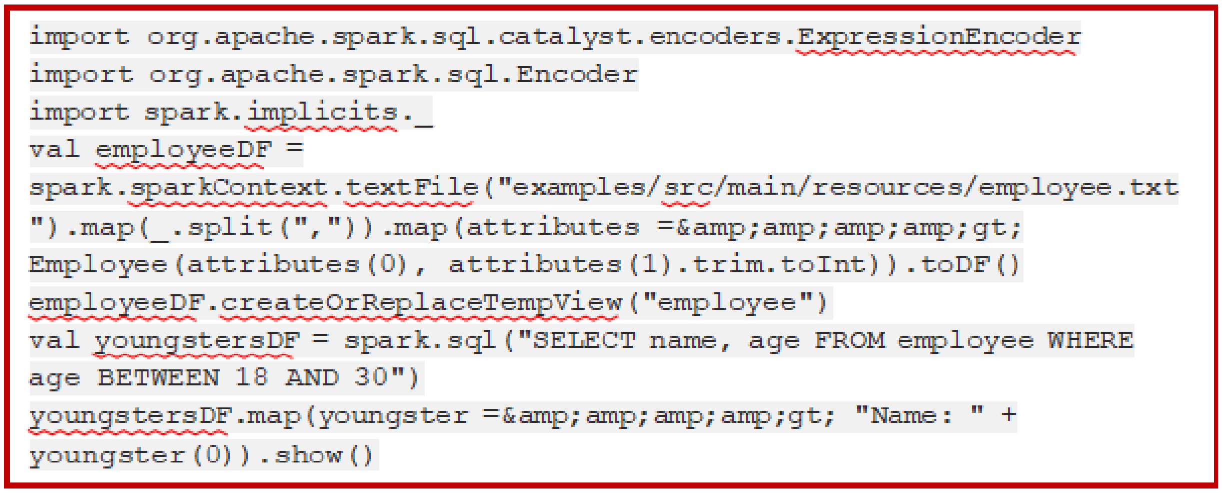
```
import org.apache.spark.sql.SparkSession
val spark = SparkSession.builder().appName("Spark SQL basic
example").config("spark.some.config.option", "some-
value").getOrCreate()
import spark.implicits._
val df =
spark.read.json("examples/src/main/resources/employee.json")
df.show()
```


edureka@localhost:~/Downloads/spark-2.0.2-bin-hadoop2.7
File Edit View Search Terminal Help
1
2
3
4
5

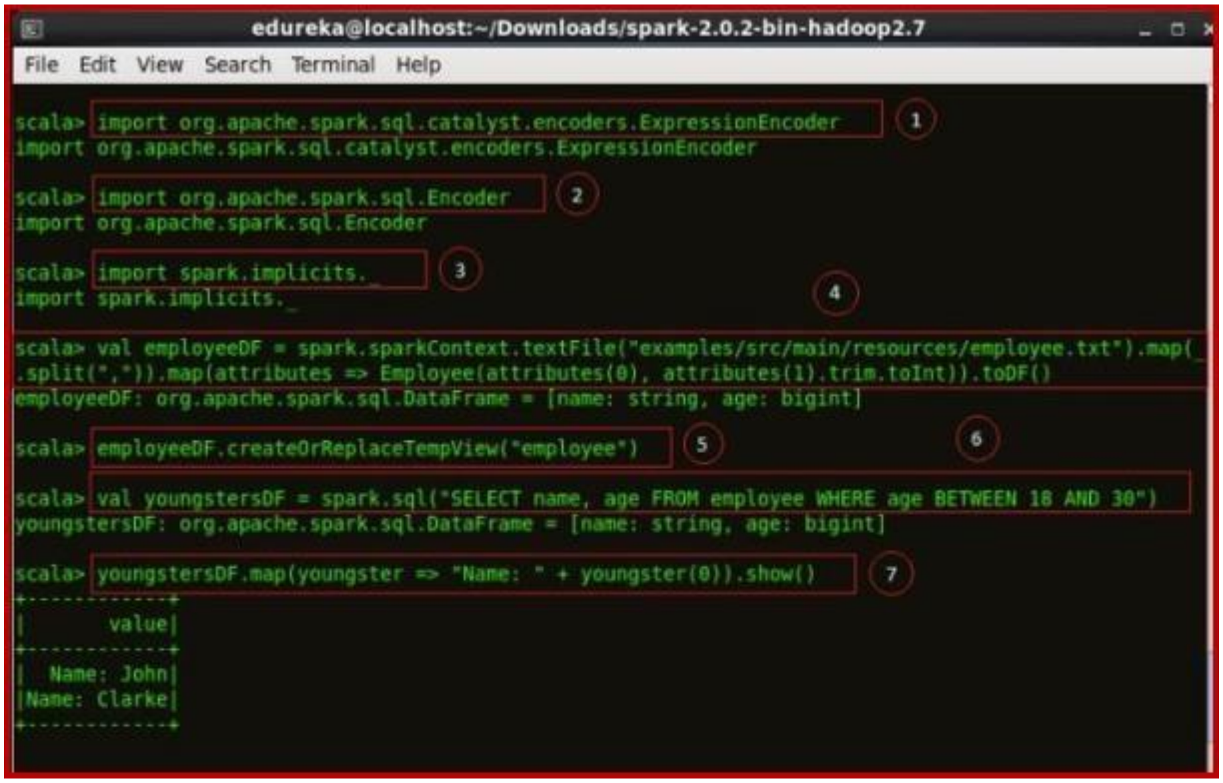
```
scala> import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.SparkSession

scala> val spark = SparkSession.builder().appName("Spark SQL basic example").config("spark.some.confi
g.option", "some-value").getOrCreate()
16/12/26 17:16:08 WARN SparkSession$Builder: Using an existing SparkSession; some configuration may n
ot take effect.
spark: org.apache.spark.sql.SparkSession = org.apache.spark.sql.SparkSession@f87c2a

scala> import spark.implicits._
import spark.implicits._

scala> val df = spark.read.json("examples/src/main/resources/employee.json")
16/12/26 17:16:33 WARN SizeEstimator: Failed to check whether UseCompressedOops is set; assuming yes
df: org.apache.spark.sql.DataFrame = [age: bigint, name: string]

scala> df.show()
+---+-------+
|age|   name|
+---+-------+
| 28|   John|
| 36| Andrew|
| 22| Clarke|
| 42|  Kevin|
| 51|Richard|
+---+-------+
```

Adding Schema to RDDs:

Spark introduces the concept of an RDD (Resilient Distributed Dataset), an immutable fault-tolerant, distributed collection of objects that can be operated on in parallel. An RDD can contain any type of object and is created by loading an external dataset or distributing a collection from the driver program. Schema RDD is a RDD where we can run SQL on. It is more than SQL. It is a unified interface for structured data.



```
import org.apache.spark.sql.catalyst.encoders.ExpressionEncoder
import org.apache.spark.sql.Encoder
import spark.implicits._
val employeeDF =
spark.sparkContext.textFile("examples/src/main/resources/employee.txt
").map(_.split(",")).map(attributes =&amp;amp;amp;amp;gt;
Employee(attributes(0), attributes(1).trim.toInt)).toDF()
employeeDF.createOrReplaceTempView("employee")
val youngstersDF = spark.sql("SELECT name, age FROM employee WHERE
age BETWEEN 18 AND 30")
youngstersDF.map(youngster =&amp;amp;amp;amp;gt; "Name: " +
youngster(0)).show()
```


edureka@localhost:~/Downloads/spark-2.0.2-bin-hadoop2.7
File Edit View Search Terminal Help
1
2
3
4
5
6
7

```
scala> import org.apache.spark.sql.catalyst.encoders.ExpressionEncoder
import org.apache.spark.sql.catalyst.encoders.ExpressionEncoder

scala> import org.apache.spark.sql.Encoder
import org.apache.spark.sql.Encoder

scala> import spark.implicits._
import spark.implicits._

scala> val employeeDF = spark.sparkContext.textFile("examples/src/main/resources/employee.txt").map(_
.split(",")).map(attributes => Employee(attributes(0), attributes(1).trim.toInt)).toDF()
employeeDF: org.apache.spark.sql.DataFrame = [name: string, age: bigint]

scala> employeeDF.createOrReplaceTempView("employee")

scala> val youngstersDF = spark.sql("SELECT name, age FROM employee WHERE age BETWEEN 18 AND 30")
youngstersDF: org.apache.spark.sql.DataFrame = [name: string, age: bigint]

scala> youngstersDF.map(youngster => "Name: " + youngster(0)).show()
+------------+
|       value|
+------------+
|  Name: John|
|Name: Clarke|
+------------+
```

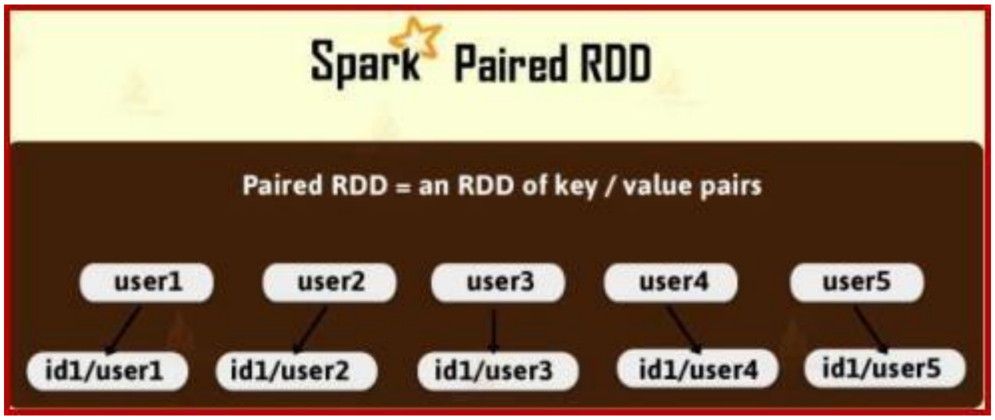
Code explanation:

1. Importing Expression Encoder for RDDs. RDDs are similar to Datasets but use encodersfor serialization.
2. Importing Encoder library into the shell.
3. Importing the Implicts class into our „spark‟ Session.
4. Creating an ‟employeeDF‟ DataFrame from ‟employee.txt‟ and mapping the columnsbased on the delimiter comma „,‟ into a temporary view ‟employee‟.
5. Creating the temporary view ‟employee‟.
6. Defining a DataFrame „youngstersDF‟ which will contain all the employees between theages of 18 and 30.
7. Mapping the names from the RDD into „youngstersDF‟ to display the names of youngsters.

Creating Pairs in RDDs:

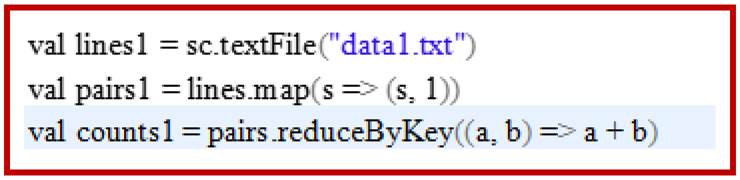
In Apache Spark, Key-value pairs are known as paired RDD. To understand operations on paired RDDs in spark requires knowledge in transformations and actions in Spark RDD.

The transformation operations such as groupByKey, reduceByKey, join, leftOuterJoin/rightOuterJoin, while, actions like countByKey.



1. Spark Paired RDDs are defined as the RDD containing a key-value pair. There is two linked data item in a key-value pair (KVP). We can say the key is the identifier, whilethe value is the data corresponding to the key value.
2. In addition, most of the Spark operations work on RDDs containing any type of objects. But on RDDs of key-value pairs, a few special operations are available.For example, distributed “shuffle” operations, such as grouping or aggregating the elements by a key.
3. These operations are automatically available on RDDs containing Tuple2 objects, in Scala. In the Pair RDD functions class, the key-value pair operations are available. That wraps around an RDD of tuples.

In the following code we are using the reduceByKey operation on key-value pairs. We will count how many times each line of text occurs in a file. There is one more method counts.sortByKey() we can use.



```
val lines1 = sc.textFile("data1.txt")
val pairs1 = lines.map(s => (s, 1))
val counts1 = pairs.reduceByKey((a, b) => a + b)
```

By running a map() function that returns key or value pairs, we can create spark pair RDDs. The procedure is different from one language to other to build the key-value RDDs differs. For the functions of keyed data to work, we need to return an RDD composed of tuples. To create a pair RDD in spark uses the first word.

val pairs = lines.map(x => (x.split(" ")(0), x)) **# In Scala**

Spark Paired RDD – Operations:

Transformation Operations: All the transformations available to standard RDDs, PairRDDs are allowed to use them. Even it can apply same rules from "passing functions to spark". As there are tuples available in spark paired RDDs, we need to pass functions that operate on tuples, rather than on individual elements. Some of the important transformationmethods are listed below.

1.

a) groupByKey - Basically, it groups all the values with the same key.

```
rdd.groupByKey()
```

b) reduceByKey(fun) - It uses to combine values with the same key.

```
add.reduceByKey( (x, y) => x +
y)
```

c) combineByKey(createCombiner, mergeValue, mergeCombiners, partitioner) - By using adifferent result type, combine values with the same key.

d) mapValues(func) - Without changing the key, apply a function to each value of a pair RDDof spark.

```
rdd.mapValues(x => x+1)
```

e) keys() - Basically, Keys() returns a spark RDD of just the keys.

```
rdd.keys()
```

f) values() - Generally, values() returns an RDD of just the values.

```
rdd.values()
```

g) sortByKey() - Basically, sortByKey returns an RDD sorted by the key.

```
rdd.sortByKey()
```

2. Action Operations: Like transformations, actions available on spark pair RDDs are similar to base RDD. Basically, there are some additional actions available on pair RDDs of spark. Some of the important operations are listed below.

a) countByKey() - For each key, it helps to count the number of elements.

```
rdd.countByKey()
```

b) collectAsMap() - Basically, it helps to collect the result as a map to provide easy lookup.

```
rdd.collectAsMap()
```

c) lookup(key) - Basically, lookup(key) returns all values associated with the provided key.

```
rdd.lookup()
```

****