A Case Study Comparing Different Big-Data Handling Approaches Using Hadoop-Hive VS Spark-Shark

Aparna Shikhare
Computer Science Department
San Jose State University
San Jose, CA 95192
408-924-1000
aparna.shikhare@gmail.com

Swapna Kulkarni
Computer Science Department
San Jose State University
San Jose, CA 95192
408-924-1000
kulkarni.swapna.m@gmail.com

ABSTRACT
We have worked on the implementation of a simple analytics engine to process huge amount of Wikipedia dump on two different cluster platforms that compares two big data technologies Hadoop-Hive vs. Spark Shark. MapReduce's greatest strength is processing lots of large text files. Hadoop's implementation is built around string processing, and it's very I/O heavy whereas Spark eliminates a lot of Hadoop's overheads, such as the reliance on I/O for everything. Instead it keeps everything in-memory provided there is enough memory storage and caching.

1. INTRODUCTION
Apache Hadoop is an open source software framework for storage and large scale processing of data set on clusters. Hadoop mainly consists of HDFS (Hadoop Distributed File System) and Mapreduce engine. HDFS is a distributed, scalable and portable file-system written in Java for Hadoop framework. Mapreduce is a programming model for large-scale data processing.

Apache Hive is a data warehouse infrastructure built on top of Hadoop for providing data summarization, query, and analysis. Hive supports analysis of large datasets stored in Hadoop's HDFS and compatible file systems. . In section 2, we will go into the details of Hadoop architecture, Hadoop file system, Hadoop Mapreduce, Yarn and Apache Hive.

Apache Spark is an open-source software framework for data analytics on cluster. Spark fits into the Hadoop open-source community, building on top of HDFS but Spark is not tied to two stages Mapreduce paradigm and promises a much better performance than Hadoop MapReduce for some applications. Spark provides primitives for in-memory cluster computing that allows user programs to load data into cluster's memory and query it repeatedly, making it a good match for machine learning algorithms.

Apache Shark is a large-scale data warehouse system for Spark designed to be compatible with Apache Hive. It can execute Hive QL queries up to much faster than Hive without any modification to the existing data or queries. Shark supports Hive's query language, metastore, serialization formats, and user-defined functions. In section 3, we provide the details of specific applications of Spark, its programming model and use cases.

In section 4, we provide the implementation details of our case study in Hadoop-Hive and Spark-Shark

In section 5, we shall see the comparison of these big data approaches in depth.

Hadoop provides an effective solution for analysis of large data sets by storing the data on HDFS but it requires lot of disk reads and is I/O intensive. It provides simplified data analysis but users need more than just large amount of data analysis. Users need applications that are more complex and require multiple passes over the data. More interactive adhoc queries are needed and real time processing of huge data is required. Spark can achieve these three objectives.

2. APACHE HADOOP
2.1 Hadoop framework
The term "Hadoop" refers not only to the base package, it also refers to the entire Hadoop ecosystem that contains of many software written on top of Hadoop such as query languages Hive, Pig etc. Hadoop framework consists of four broad modules [1].

1. Hadoop common: It contains the various libraries and utilities required by other Hadoop modules.

2. Hadoop distributed file system (HDFS): It is a large distributed file system to store huge amount of data for Hadoop framework.

3. Hadoop Mapreduce: It is a programming model for large-scale data processing.

4. Hadoop Yarn: It is the resource manager for Hadoop framework that is responsible for managing of the resources on various clusters and to schedule user applications.

2.2 Hadoop Architecture
Hadoop Common contains JAR files and scripts that are required to start and run Hadoop applications [1]. A Hadoop cluster usually contains one master node and multiple worker nodes. The master node contains of JobTracker, TaskTracker, NameNode and DataNode. The worker node behaves both as Datanode and TaskTracker. Hadoop requires JRE of 1.6 or higher [1]. NameNode is the most important component of Hadoop file system, which maintains the tree directory of all the files present in Hadoop file system. It does not store the data itself. Datanode is the service, which store the data on Hadoop file system across various clusters. JobTracker is the service in Hadoop file system that manages the Mapreduce tasks to specific nodes in the cluster that have the data within them. A TaskTracker in a cluster is the service that accepts predefined number of operations such as Map, reduce and shuffle from the JobTracker. Figure 1 shows the broad view of Hadoop architecture in multinode cluster.

2.3 Hadoop Distributed File System
Hadoop distributed file system is a large, scalable, distributed file system for Hadoop framework. It stores large file of the magnitude ranging from several Gigabytes to Terabytes. As

mentioned earlier, a cluster consists typically of a single NameNode and several Datanodes as shown in Figure 2. A DataNode stores several blocks of data using the block protocol specific to HDFS. The client applications communicate using RPC between themselves. The Hadoop file system communicates with the nodes using the TCP/IP sockets. Hadoop is highly scalable because it replicates the data among multiple hosts. The default replication value is three. Therefore, no RAID techniques are required to maintain multiple copies of data on various hosts. Data nodes that store multiple copies can talk to each other to rebalance the data, synchronize the data by circulating copies of data. HDFS also contains secondary namenode that is not the backup of NameNode, instead it connects with the NameNode and takes the snapshots of the directory information of the namenode and keeps it into the local directory. These saved snapshots can be used in case of the NameNode failure instead of replaying the actions again from the beginning. Sometimes, a single namenode to manage various namespaces or large number of files can be bottleneck. Providing various namenodes to manage various namespaces can solve this.

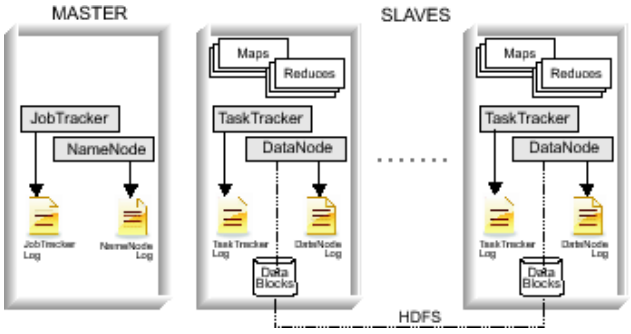


Figure 1: Hadoop Architecture consisting of multinode cluster [3]

Hadoop file system has an advantage of data awareness between JobTracker and TaskTracker. When a client application is submitted JobTracker co-ordinates with the DataNode to retrieve the location of the data and submits the map and reduces tasks to the TaskTracker. Different data nodes, which have different datasets, perform map and reduce tasks separately and prevent unnecessary data transfers over the network. This amplifies the time required for job completion greatly. However, it is not the same when Hadoop is used with other file systems. One disadvantage of HDFS is it is used with immutable file systems and cannot be used for concurrent write operations.

2.4 Hadoop Mapreduce

Above the file system is the Hadoop Mapreduce engine that consists of a JobTracker and multiple Tasktrackers. When a client application sends Mapreduce jobs to JobTracker, it submits the map and reduce jobs to Tasktrackers that are closer to the data [2]. It gets this information from the Namenode that gives the location of the data. If the jobs cannot be hosted on the same nodes where the actual data resides, the JobTracker allocated the jobs on the same track in order to avoid large amount data transfer on the network. The key aspect here is that map and reduce jobs can run in parallel across various TaskTracker nodes, provided the map and reduce jobs are independent datasets. Then the output of these TaskTracker nodes is output to the HDFS. Then the reducers can be run to merge the results. The JobTracker monitors the TaskTrackers, if the TaskTracker does not complete successfully, then the job is allocated to other TaskTracker. JobTracker manages the resources among various nodes, schedules the map and reduce jobs. To achieve maximum parallelism the data should be independent that is Maps and Reduces should be stateless. The order in which the mapping and reducing is done is not under the control of the programmer. At the very minimum, client applications specify the input/output locations and map and reduce function via some interface calls.

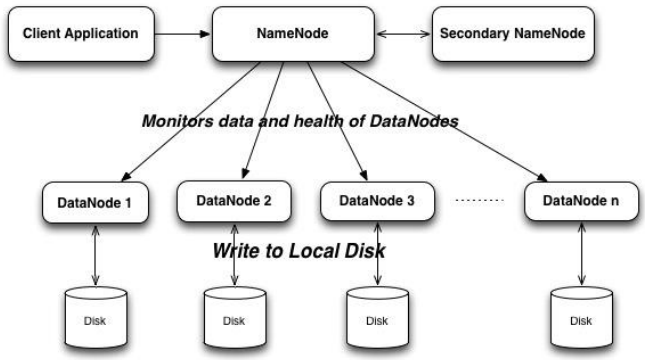


Figure 2: Hadoop file system

Map:

The mapper accepts the input and outputs key/value pairs. The output list can contain the same key more than once.

Input->Map(list(keys,values))

Reduce:

It takes the key/value pairs generated by Map and merges the values for the same key and produces the reduce key/value pairs.

Map(list(keys,values)) ->Reduce(key, <value1,value2>)

Figure 3 shows the MapReduce operations in Hadoop, which consists of three mappers and two reducers.

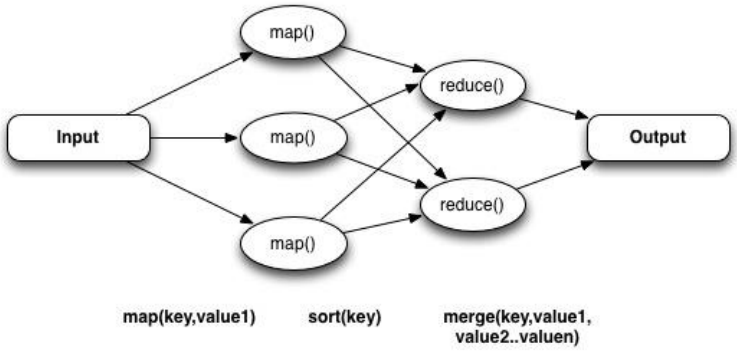


Figure 3: Hadoop MapReduce

Sample word count example:

Let us consider that there are two files.

File1 consists of :

Hello, World, Goodday, Bye

File2 consists of:

Hello, There, Goodnight, Bye

There can be two map jobs that can be run in parallel.

Map1 outputs

Hello: 1

World: 1

Goodday: 1

Bye: 1

Map2 outputs:

Hello: 1

There: 1

Goodnight: 1

Bye: 1

After this, the keys are sorted in ascending order and key value pairs are given to the reducer.

Both the mappers store the output in HDFS. Then, the reducer merges both the outputs.

Bye: 2

Goodday: 1

Goodnight: 1

Hello: 2

There: 1

World: 1

2.5 Hadoop Yarn

Yarn is also called as MapReduce version 2. It is a new version of MapReduce. The idea in Yarn is to split the task of resources management and scheduling of jobs into different daemons [1]. It has a global ResouceManager (RM), per application ApplicationMaster, per slave NodeManager, and per application container running on NodeManager as shown in Figure 4. The ResourceManager runs on the master node and is responsible for scheduling the resources to various applications. The ApplicationMaster per application negotiates with the ResourceManager for the resources and co-ordinates with various NodeManagers to schedule and monitor individual tasks. The NodeManager runs on the slave nodes and is responsible for launching the containers which are allocated by ResourceManager when requested by application, monitor their resource usage such as CPU time, memory etc. and then it reports the same to the ResourceManager. The applications run in one or more containers.

With this, Yarn enhances the Hadoop processing power in the following ways [1]:

Yarn is more scalable because most of the functions of the JobTracker is now transferred to ApplicationMaster and there can be more ApplicationMasters per cluster whereas in old Mapreduce (Mapreduce 1) there can be only one JobTracker per cluster.

Unlike Mapreduce, Yarn supports both MR and non-MR jobs to run on the same cluster.

There is no concept of named and fixed slots that is separate slots for Map and Reduce that improves the cluster utilization. Nodes have the resources and are allocated to applications whenever requested.

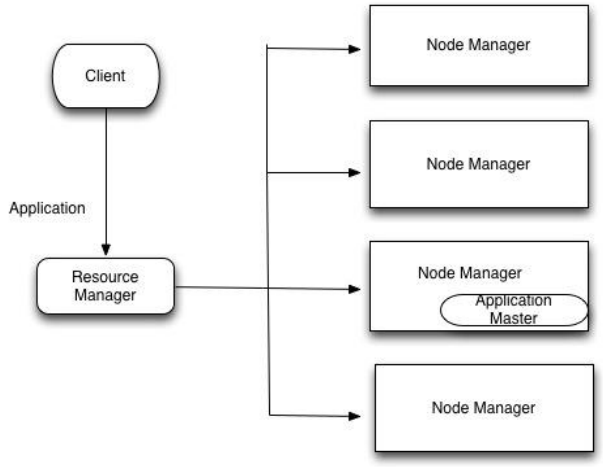


Figure 4: Yarn cluster consisting of Resource Manager, Node Manager, Application Master

2.6 Apache Hive

Apache Hive is a data warehouse infrastructure built on top of Hadoop, which provides analysis of large dataset, ability to summarize and query huge amount of data stored in HDFS or any other compatible file system. It provides SQL like language called HiveQL. It maintains full support for map/reduce. To accelerate data analysis it provides indexes. Hive stores the metadata on Apache Derby by default. It supports mainly four types of file formats namely Textfile, sequence file, ORC and RCFILE [1]. The SQL like queries are simply converted into map-reduce jobs underneath. Hive has various built in functionalities to manipulate various data types such as string, date etc. When the Hive queries are submitted, they are internally converted into directed acyclic graphs of MapReduce jobs, which are then submitted to Hadoop for execution. The broad view of Hive architecture is shown in Figure 5.

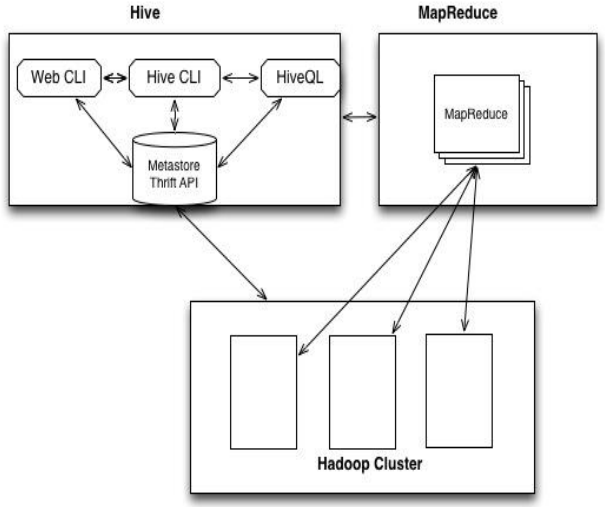


Figure 5: Hive Architecture

3. APACHE SPARK

Apache spark is an Open-Source data analytics cluster computing framework [1]. Spark belongs to Hadoop Open Source community. It is built on top of Hadoop Distributed File System. Although such similarities exists, Spark performs better than Hadoop in case certain specific applications. Spark is not restricted by two stage Map-Reduce paradigm. It delivers 100 times better performance than Hadoop for certain applications. Spark provides the capabilities for in-memory cluster computing which allows user programs to load data into cluster's memory.

The data loaded into main memory can be used repeatedly by subsequent database accesses and it speeds up the entire response time. This property makes Spark well suited for Machine Learning algorithms.

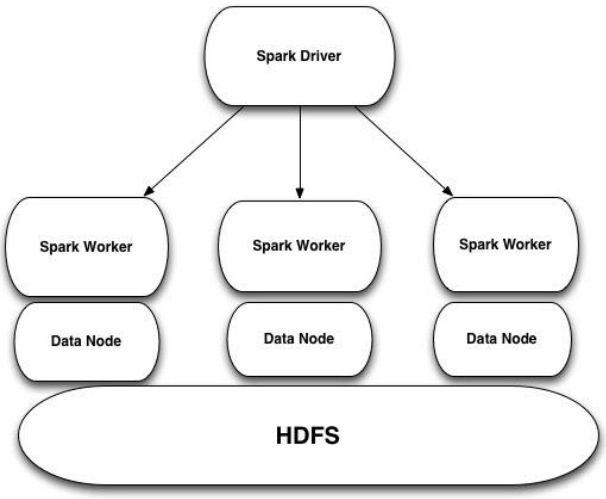


Figure 6: Spark Architecture

3.1 Spark Specific Applications

Hadoop Users have specified over and over again two main types of jobs/applications in which case they found Map-Reduce to be deficient. [4]

Iterative Jobs: Many machine-learning algorithms follow an iterative model, wherein a function is applied repeatedly on same dataset. Each iteration can be expressed as Map-Reduce job. But, each iteration/job must reload the data from disk. This reduces the performance significantly.

Interactive Analytics: Hadoop is often used to run queries on large datasets using Hive and Pig. In such cases, user should be able to load the data at once in main memory and query it over and over again. In case of Hadoop, each query is executed as a separate job. Each job will again access the disk for loading data in memory, slowing down the entire system.

To achieve these goals, Spark introduces an abstraction called resilient distributed datasets (RDDs). An RDD is a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Spark can outperform Hadoop by up to 100x in iterative machine learning jobs, and can be used to interactively query a 39 GB dataset with sub-second response time [4].

3.2 Spark Programming Model

3.2.1 *Resilient Distributed Datasets (RDDs)*

A resilient distributed dataset (RDD) is a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. The elements of an RDD need not exist in physical storage; instead, a handle to an RDD contains enough information to compute the RDD starting from data in reliable storage. This means that RDDs can always be reconstructed if nodes fail [4]. As elements of the RDD need not exist in physical storage, and only a handle is enough for reconstructing it, the iterative and or interactive jobs will execute faster than that in Hadoop.

In Spark, each RDD is represented by a Scala object. Spark lets programmers construct RDDs in four ways: [4]

File: From the shared file system, for example Hadoop Distributed File System

By parallelizing collection like an array: By dividing the array into number of slices. These slices can be sent to multiple nodes.

By transforming an existing RDD: A dataset with elements of type A can be transformed into a dataset with elements of type B using an operation called flatMap, which passes each element through a user-provided function of type A => List [B].1 Other transformations can be expressed using flatMap, including map [4].

By Changing the Persistence of an existing RDD: The RDDs are by default lazy. The RDDs are not immediately reflected onto the disk. They are stored into main memory as long as they can be stored. Unless driven by any need from the user or application, the RDDs are not written back to the disk. Although, user can change the persistence properties of an RDD and change it to cache action and save action.

The **Cache Action** will mark the dataset that it will be referred to in future and should be kept in memory. The dataset marked with cache action will not be immediately written onto the file system as HDFS but it would be kept in main memory. It indicates the immediate future reference.

The cache action is just a hint that the data can be used in immediate future. But if there is not enough memory on the cluster to keep the data marked cache, Spark will recompute the data as and when it will be referenced.

The **Save Action** will not leave the dataset lazy. It will save the dataset and write onto the file system. The saved version of the file will be referred to in future operations.

The concepts of save and cache actions is fairly similar to virtual memory and it is used to impose the fault tolerance in Spark.

3.2.2 Parallel Operations

Parallel operations need to be performed for scalability. Following parallel operations are possible in case of resilient distributed datasets-

Reduce: Reduce operation is fairly similar to reduce in Hadoop. Reduce will combine the dataset elements to produce the result. The dataset elements will be combined associatively to produce a cumulative result.

Collect: The user program will be able to get the results from all the nodes using collect operation. Collect operation will send the all the elements of the dataset to the driver or user program. If we consider of processing an array, user can update the array in

parallel. He can parallelize the array, map the array to the different nodes and collect the results. The collect operation here will give the user the independence to collect the results at once.

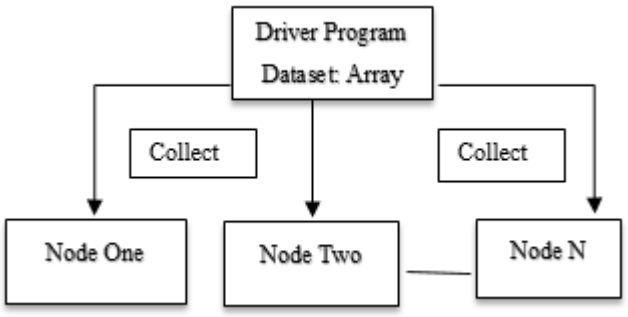


Figure 7: Figure illustrating the collect operation

As shown in figure 7, the dataset elements will be mapped, parallelized and processed, and collected back by the driver program. Currently the Spark Programming Model does not support the grouped Reduce operations as Hadoop does. In case of Hadoop the reduced data is collected by multiple threads. In case of Spark, that data will be collected by one single thread. However, according to Spark documentation, the one thread reducer is enough for implementation of multiple algorithms that Spark aims for.

For-Each: For-each passes the elements via the functions provided by users. This is used for some repetitive kind of functions which will collect the data and produce some kind of cumulative result by processing the collected data.

3.2.3 Shared Variables

In Spark, the operations like map, reduce, filter are invoked by users or programmers. Users generally invoke these functions by passing functions to Spark. Hence the variables that these functions use should be within the scope where these functions will get executed. If a worker node is supposed to execute a function, then the variables needed by that function should be copied onto worked node. Spark framework does exactly the same. Although this is what happens generally in case of Spark, it also lets users choose two special types of variables. These variables are named as **Broadcast Variables** and **Accumulators** [4].

Broadcast variables: As explained above, with function, the required variables are copied onto the worker node where that function is being executed. But there might be a use-case wherein many of the functions might be using a single data variable such as lookup table. In this case, it is not performance beneficial if we copy the look up table onto each of the nodes over and over again. Instead, make the common data as broadcast variable. By doing so, programmer can rest assured that the variable value is only copied to each worker once.

Accumulators: Accumulators are like counters. They have add operation and zero value. They can be added to by different nodes. They are fault tolerant due to their add-only property. They can be used in case of parallel sums. Map-Reduce paradigm also uses them.

3.3 Apache Shark

Shark is nothing but Hive on top of Spark. A shark cluster consists of masters and workers. The lifetime of the master can be equal to one query or multiple queries. The workers are the processes which are long lived. Workers can store dataset partitions in-memory across operations. This property of workers helps to reduce the latency in query processing. Figure 8 shows the general architecture of Shark.

Data are stored physically in the underlying distributed file system HDFS. The Hive meta-store is used without modification in Shark and tracks the metadata and statistics about tables, much like the system catalog found in traditional RDBMS [6].

From a higher level, Shark's query execution consists of three steps similar to traditional RDBMS: query parsing, logical plan generation, and physical plan generation.

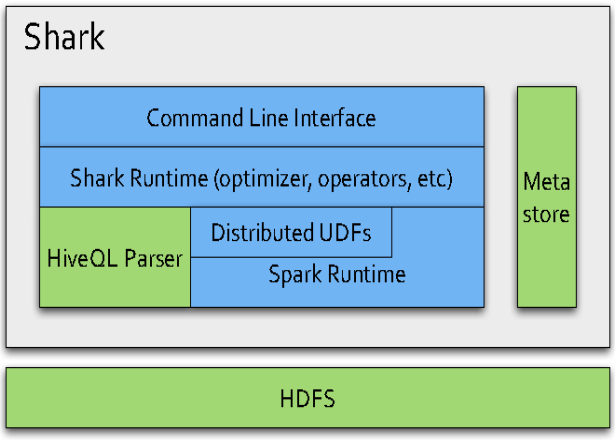


Figure 8: Shark Architecture [6]

Hive provides an SQL-like declarative query language, Hive-QL, which gets compiled into lower level operators that are executed in a sequence of Map-Reduce programs. Shark uses Hive as a third-party Java library for query parsing and logical plan generation. The main difference is in the physical plan execution, as Shark has its own operators written specifically to exploit the benefits provided by RDDs [6].

3.3.1 Query Processing

A major advantage of Shark over Hive is the inter-query caching of data. The Spark framework provides a simple mechanism to cache RDDs in memory across clusters and recomputed RDDs in the event of failures.

Shark configuration file is used by users to customize the query processing to a certain extent according to their needs. Users can tell Shark that these many specific objects need to be cached during query execution. Also they can specify in the configuration file the types of operations whose outputs will be cached by Shark. Each output is cached as an in-memory RDD whose signature will be computed by Shark.

In addition, users can also choose to explicitly cache specific tables by providing the option as CREATE TABLE AS SELECT. If the name of the table being created using this statement ends with _cached, the table will be cached as in-memory RDD. If the subsequent query comes on the same table, the table need not be

loaded to main memory. The table will be present in the main memory as it was cached. For example –

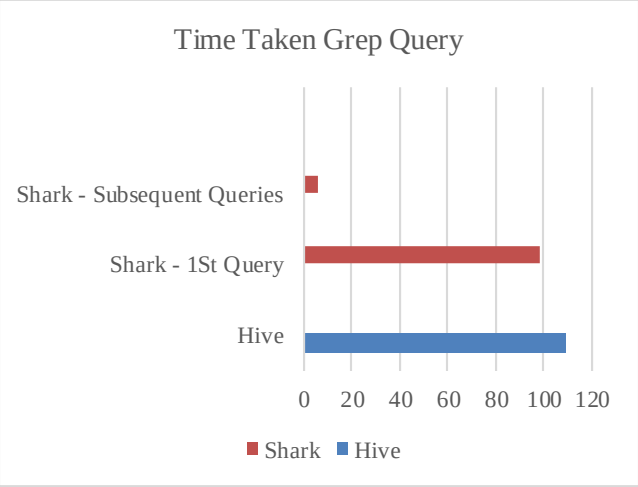
```
CREATE TABLE top_student_cached AS
SELECT student_id, name, age
FROM  student_data WHERE age > 20;
```

The above-specified query has '_cached' as the suffix to the table name. Hence the student table will be created as an in-memory RDD.

3.32 Performance: Query Processing

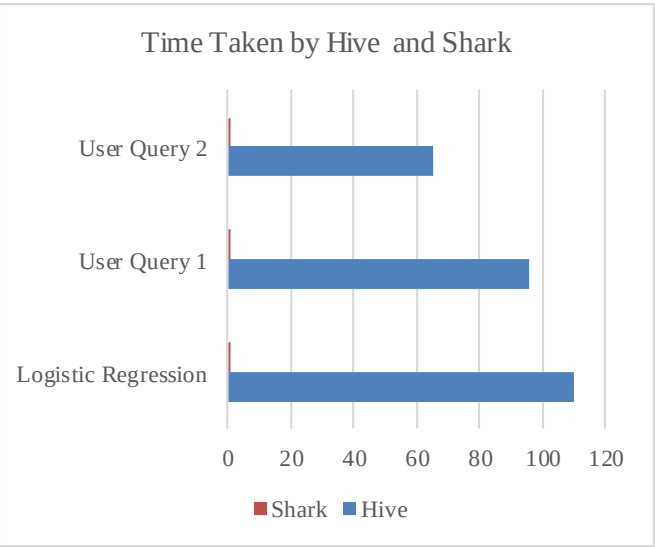
Shark has reported to perform considerably well when it comes to processing the queries like using 'grep'.

Table 1: Query Performance Timing Graph



Shark is compatible with Hive data, meta-stores, and queries. It provides speedups ranging between 10X to 100X better than Hive. One study conducted on usage of Hive and Shark by a Silicon Valley based company shows that Shark outperforms Hive/Hadoop on both disk and memory based datasets [6]

Table 2: Hive and Shark Query Performance



3.4 Use Cases of Spark

3.4.1 Iterative Algorithms:

As explained in section 3.2.1, Spark allows users and applications to use cache operation for explicitly caching a dataset. This leads to the fact that the applications will now be able to use the data from RAM and not from physical storage. This can increase the performance of iterative algorithms which access the same dataset many number of times.

4. CASE STUDY IMPLEMENTATION

We worked on a simple analytics engine to parse huge amount of XML Wikipedia dump. The XML Wikipedia dump consists of many tags such as pageid, title for the page, and revisions with respect to the page. Each page can consist of many revisions. Each revision again consists of many tags such as revision id, parent id (the previous revision), timestamp, contributor, and text. We needed only some data from the XML file to produce a # separated dataset. As a result, we wrote a python script, which would parse the XML dump to produce two files, which consisted of the following datasets. The first dataset consists of pageid, title and number of revisions each page has. The second dataset consists of pageid, revisionid, timestamp, and contributor.

File1 (Dataset 1)

pageid#title#revcount

AccessibleComputing#10#9

Anarchism#12#16851

AfghanistanHistory#13#5

AfghanistanGeography#14#6

AfghanistanPeople#15#6

AfghanistanCommunications#18#4

AfghanistanTransportations#19#8

…..

File2 (Dataset 2)

pageid#revisionid#timestamp#contributor

10#10#2001-01-21 02:12:21#RoseParks

10#862220#2002-02-25 15:43:11#Conversion script

10#15898945#2003-04-25 22:18:38#Ams80

10#56681914#2006-06-03 16:55:41#Nzd

10#74466685#2006-09-08 04:16:04#Rory096

10#133180268#2007-05-24 14:41:58#Ngaiklin

10#133452289#2007-05-25 17:12:12#Gurch

10#381200179#2010-08-26 22:23:51#76.28.186.133

10#381202555#2010-08-26 22:38:36#OlEnglish

12#12#2002-02-25 15:00:22#Conversion script

12#19746#2002-02-25 15:43:11#140.232.153.45

12#19749#2002-02-27 17:34:09#24.188.31.147

12#20514#2002-02-27 17:36:41#24.188.31.147

12#42733#2002-03-01 00:13:17#213.253.39.175

12#42738#2002-04-02 09:51:25#206.82.16.35

12#42740#2002-04-02 09:53:06#206.82.16.35

12#42743#2002-04-02 09:54:12#206.82.16.35

12#43618#2002-04-02 09:55:36#Lee Daniel Crocker

12#59361#2002-07-18 14:33:46#0

12#61039#2002-04-26 07:02:43#80.65.225.191

12#61179#2002-05-01 14:18:43#Eclecticology

12#61193#2002-05-01 20:54:35#Eclecticology

…

We implemented the same case study on different file sizes. The first time, we implemented it on file size of 2GB. This is the file size that is obtained after processing the Wikipedia dump using the python script and contains the required data set. We noted the time required for the analysis using Hadoop-Hive and Spark-Shark approaches. We then implemented the same analysis for 50GB of dataset file. For better understanding and readability, implementation of case study for dataset of 2GB is explained in detail.

4.1 Implementation using Hadoop-Hive

The Hadoop cluster set up for this implementation has two Datanodes and one NameNode.

4.1.1 Loading the datasets in HDFS

Step1: Create a directory in the HDFS

cd ~/install/hadoop-0.23.9/bin

./hadoop fs -mkdir /test

Step2: Put the file in the HDFS

./hadoop fs –put

/Users/aparna/projects/wiki/src/Processed_XML1.xml /test/

./hadoop fs –put

/Users/aparna/projects/wiki/src/Processed_XML2.xml /test/

4.1.2 Creating tables in Hive

Since we have two datasets we created two tables in hive. The first table WikiPage consists of three fields, pageid (int), title (string), revcount (int). The second table WikiRevision consists of four fields, pageid (int), revisionid (int), time (timestamp), and contributor (string). Contributor can either be the username or the ip address of the contributor.

When creating the table we specify that the fields are terminated or separated by # and the lines or rows are terminated by newline (“\n”) character.

The syntax for table creation:

WikiPage Table:

create table WikiPage (title string, pageid int, revcount int)

row format delimited

fields terminated by '#'

lines terminated by '\n';

WikiRevision Table:

create table WikiRevision (pageid int, revisionid int, time timestamp, contributor string)

row format delimited

fields terminated by '#'

lines terminated by '\n';

4.1.3 Importing data into tables from HDFS

After creating the tables, we need to import the data from the files in HDFS to the tables.

Syntax to import the data into the tables:

LOAD DATA INPATH '/test/Processed_XML1.xml' OVERWRITE INTO TABLE WikiPage;

LOAD DATA INPATH '/test/Processed_XML2.xml' OVERWRITE INTO TABLE WikiRevision;

4.1.4 Executing the Queries on Hive

Once the data is imported into the tables, we need to execute the queries and analyze the data. The hive queries are converted to the map reduce jobs underneath.

Query One: This query below picks the pageid and contributor and groups the data with these two columns, then it lists the pageid, contributor and number of revisions for each of this group provided the revision count is more than 1000.

select pageid, count(revisionid), contributor from WikiRevision group by pageid, contributor having count(revisionid) > 1000;

The result of this query:

12	1544	RJII
25	1222	Eubulides
307	1339	Hoppyh
863	1132	Jimmuldrow
2174	1663	Qwghlm

Time taken: 31.113 seconds, Fetched: 5 row(s)

Query Two: This query lists the pageid grouped by pageid and the count of the contributors in the ascending order with respect to that pageid.

select pageid, count(contributor) as count from WikiRevision group by pageid order by count;

The result of this query:

…

737	11492
738	11530
765	11696
307	14798
736	14901
12	16849

Time taken: 45.799 seconds, Fetched: 2154 row(s).

4.2 Implementation using Spark-Shark

In case of querying the data using Shark, we are using the HDFS as the file system. In order to compare the data better, we used the exact same data for querying. As Spark can use the data loaded in

HDFS and Spark lies on top of HDFS, the data loading steps are same as explained in sections 4.1.1 through 4.1.3 are same for Hive and Shark. Let me explain the query processing time required in case of Shark.

4.2.1 Executing the queries on Shark

Unlike Hadoop query, Spark query does not translate into a job, each time an iteration is executed. It converts into an object of RDD, as explained earlier.

Query One: This query below picks the pageid and contributor and groups the data with these two columns, then it lists the pageid, contributor and number of revisions for each of this group provided the revision count is more than 1000.

shark> select pageid, count(revisionid), contributor from WikiRevision group by pageid, contributor having count(revisionid) > 1000;

The result of this query:

12	1544	RJII
25	1222	Eubulides
307	1339	Hoppyh
863	1132	Jimmuldrow
2174	1663	Qwghlm

Time taken: 3.59 seconds, Fetched: 5 row(s)

Query Two: This query lists the pageid grouped by pageid and the count of the contributors in the ascending order with respect to that pageid.

shark> select pageid, count(contributor) as count from WikiRevision group by pageid order by count;

The result of this query:

…

737	11492
738	11530
765	11696
307	14798
736	14901
12	16849

Time taken: 4.3535 seconds, Fetched: 2154 row(s).

Hence, in our experiment, we could notice that there is about 10x times speedup in case of processing times of Hadoop and Spark for both the datasets sizes. In case of machine learning algorithms or iterative algorithms the performance speedup from Spark can be gained between 10X to 100X than that from Hadoop.

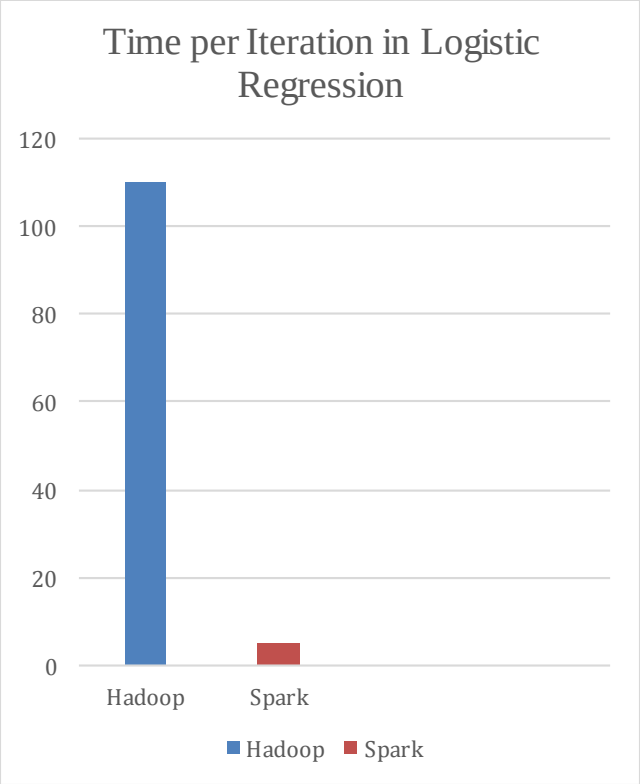
5. COMPARISON OF HADOOP-HIVE AND SPARK-SHARK

5.1.1 Performance Comparison Based on Iterative Algorithms:

Iterative algorithms such as logistic regression have run 100x faster on a Spark cluster than on Hadoop cluster. Other iterative algorithms such as Collaborative Filtering and alternating direction have executed 15 times faster on Spark. This success lies in the fact that Spark lets user keep data in RAM instead of disk by making use of methodologies such as RDDs and Cache operations. Table 3 illustrates the performance benefits of Spark over Hadoop when machine-learning algorithms are into consideration.

Hence in case of machine learning algorithms or logistic regression it has been proved experimentally that Hadoop takes more time that spark to execute the same algorithms. With Hadoop each iteration for above experiment takes 127 seconds because each iteration runs as independent map-reduce job. And with Spark it takes only 6 seconds as Spark will use the cached data.

Table 3: Hadoop and Spark Performance



5.1.2 Broadcast Variables and Alternating Least Squares:

Broadcast variables are used for iterative jobs that copy a shared dataset to multiple nodes. If the broadcast variables are not used in such kind of iterative jobs, the time to resend the shared dataset to each node is greater than the job's running time. If any preliminary implementation method is used for broadcasting the broadcast time grows linearly with the number of nodes. This limits the scalability of the system and the job. Even if with a little advanced system of broadcast implementation, resending the reused dataset with each iteration is costly.

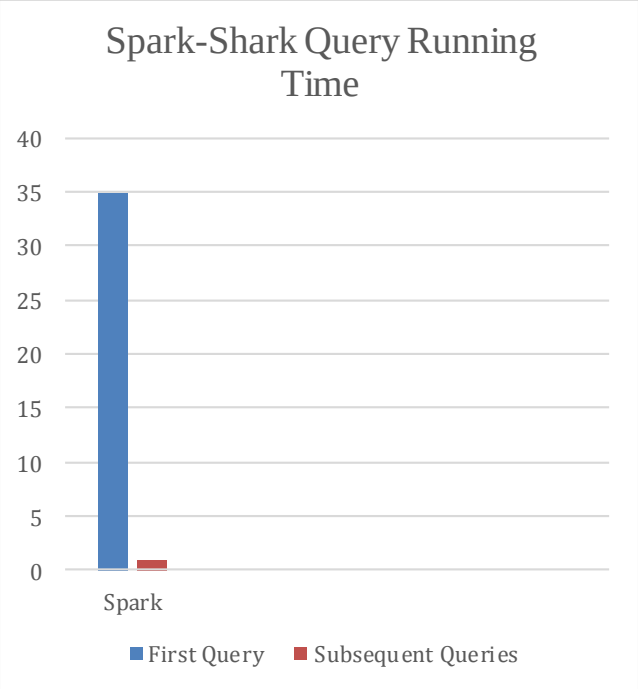
On the other hand, if broadcast variable is used and the reused dataset is copied to the node only once, performance increases significantly.

5.1.3 Interactive Spark:

In an experiment conducted by Matei Zaharia and others [5] they used the Spark interpreter to load a 39 GB dump of Wikipedia in

memory across 15 EC2 machines and query it interactively. The first time the dataset is queried, it takes roughly 35 seconds, comparable to running a Hadoop job on it. However, subsequent queries take only 0.5 to 1 seconds, even if they scan all the data. This provides a qualitatively different experience, comparable to working with local data.

Table 4: Spark-Shark Queries Running Time



6. CONCLUSION

Hadoop is highly scalable, that is it provides cheap and effective means for analysis of huge amount of data ranging from gigabytes to terabytes by storing the data across multiple nodes in HDFS. Another good feature of Hadoop is its fault tolerance [7]. When data is sent to an individual node, it is also replicated on other nodes eliminating the RAID techniques. In case of a failure, another copy can be picked easily. However, sometimes the users require more than just analysis of huge amount of data. Users need applications that are more complex and require multiple passes over the data. More interactive adhoc queries are needed and real time processing of huge data is required.

Though Spark is compatible with Hadoop and HDFS in many ways of its usage, it addresses certain applications in a specific manner and outperforms Hadoop. In case of Hadoop and Hive, each Map-Reduce job is executed as separate entity and hence the number of disk accesses are more. In case of most of the machine learning algorithms and interactive queries, each iteration of execution leads to a new Map-Reduce job and slows down the performance of the system. We explain certain studies to illustrate the latency involved in such operation when they are executed on Hadoop based Map-Reduce paradigm.

Spark provides three simple data abstractions for programming clusters: resilient distributed datasets (RDDs), and two restricted types of shared variables: broadcast variables and accumulators. The RDDs are Scala objects whose references can be stored in memory and the object can be recomputed from these references. This property of Spark makes it able to address the latency involved in iterative or machine learning algorithms.

Paper also presents a case study which testifies the fact that Spark and its Query Engine Shark outperforms Hadoop and Hive when HDFS is queried.

We sincerely thank Prof. Robert Chun for providing this opportunity, for his guidance, motivation and support.

7. REFERENCES

[1] Bappalige S, An introduction To Apache Hadoop for big data. http://opensource.com/life/14/8/intro-apache-hadoop-big-data, 2014

[2] Mittra D, Apache Hadoop http://wiki.apache.org/hadoop, 2014

[3] Architecture of Hadoop. http://www.kopo.com/bd/architecture-of-hadoop/

[4] Zaharia, M., Chowdhury M., Franklin M., Shenker S., Stoica I. Spark: Cluster Computing with Working Sets, ACM, 2010.

[5] Joshrosen. Low-Latency SQL at Scale: A Performance Analysis of Shark, AMPLAB, 2012.

[6] Cliff E, Antonio L, Reynold X, Matei Z, Michael J. Franklin, Scott S, and Ion S. Shark: Fast Data Analysis Using Coarse-grained Distributed Memory. AMPLAB, 2012.

[7] Nemschoff M, Big Data: 5 major advantages of Hadoop. http://www.itproportal.com/2013/12/20/big-data-5-major-advantages-of-hadoop/,2013