A comparison Hadoop and Spark in terms of fault tolerance and the way the search is presented through a website

List of Abbreviations:

HPC	High Performance Computing
MPI	Message Passing Interface
API	Application Program Interfaces
RDMS	Relational Database Management System
SQL	Structured Query Language
SAN	Storage Area Network
HDFS	Hadoop Distributed File System
POSIX	Portable Operating System Interface
NFS	Network File System
MPP	Massively parallel processing
RAID	Redundant Array of Independent Disks
CLI	Command Line Interface
JVM	Java Virtual Machine
YARN	Yet Another Resource Negotiator
LRU	Least Recently Used
UDF	User-defined Functions
UDA	User-defined Aggregate
UDTF	User-defined Table Function
PDSH	Parallel Distributed Shell
QJM	Quorum Journal Manager
RDD	Resilient Distributed Data-set

ML	Machine Learning
LRU	Least Recently Used
DAG	Directed Acyclic Graph

Chapter one

Hadoop working platform

Hadoop framework

1-1 Introduction:

Apache Hadoop is an open source tool of ASF –Apache Software Foundation and it’s one of the most popular Big Data management techniques, Where it stores and processes different data what enables the Data-Driven Companies to extract the full value of all their data, And it is a software environment for writing and executing distributed applications that process large amounts of data, Nowadays Hadoop is an important part of the processing infrastructure in many internet companies like LinkedIn [3].

Hadoop provides a powerful acara to run tasks in parallel on multiple nodes connected across a local network.

The basic Hadoop programming language is Java, but this doesn’t mean that you can write the code just in Java but it is possible to program in C, C++, Perl, Python and Ruby extra… , But Java programming language would be more suitable.

Hadoop focuses on transferring the code to the location of the data instead of the continuous transfer of data, meaning that the calculations are made on the data on the same device and because data transmission take longer than performing calculations on the data[15], Among Hadoop advantages we can mention the following elements[13]:

1. **Size:** Hadoop runs of a large clusters of computing devices and computing services.
2. **Strength:** Hadoop assumes in its design the presence of frequent defects in the performance of the devices and sets appropriate scenarios to deal with such disturbances.
3. **Scalability:** Hadoop is linearly scalable to support handling of increasingly large data by adding more nodes to the cluster.
4. **Simplicity:** Hadoop allows the users to write their own parallel programs easily and efficiently.

1-2 Apache Hadoop Components[1]:

Hadoop consists of four main parts:

- **Map-Reduce:** A programming module that provides support for parallel processing, location aware scheduling, fault tolerance and scalability.

- **YARN (Hadoop 2.x):** A resource manager who schedules tasks and reserves resources within the cluster.
- **HDFS (Hadoop Distributed File Systems):** Hadoop distributed file system that stores files and links their blocks logically.
- **Hadoop Common:** Java libraries which needed to run other Hadoop modules.

Both HDFS (Storage) and Map-Reduce (processing) considers as the two basic components of the Apache Hadoop and the most important aspect of Hadoop is that both of HDFS and Map-Reduce are designed with each other and both of them are jointly published so that there is a single block, and thus provide the ability to transfer the account to the data and not the other way around. Therefore, the storage system is not actually separate from the processing system [2].

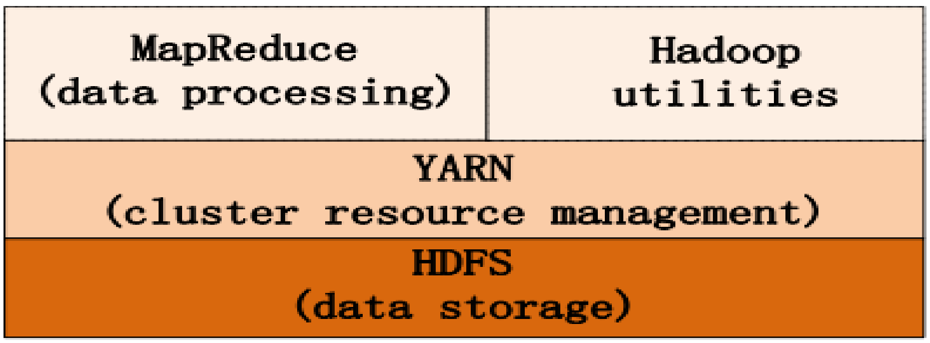


Figure (1-1) Components of Apache Hadoop [1].

1-3 HDFS (Hadoop Distributed File System) [3]:

HDFS is a Java-based file system that provides a scalable and reliable data store designed to cover large clusters and provide high-speed access to the data, It is a storage system for the Hadoop cluster, the file system divides it into parts and distributes these parts to the different servers participating in the cluster, Each server stores a small portion of the total data set and each piece of data will be copied to more than one server, and since this file system stores aggregate data in small pieces on a set of servers the analysis tasks distribute a branch to all the servers that contain part of the aggregate data [15].

Each server evaluates the value of the part of the data stored in it synchronously with the rest of the shared servers with the aggregate data, and it presents the result to be aggregated until we get a comprehensive answer to the question which wanted to be asked on the total data set , Map-Reduce take care of distributing the work and recollecting the result, HDFS is highly fail-tolerant and is designed to be deployed on low cost devices, HDFS creates multiple replicas of each data block and distributes them to computers across the cluster to enable fast and reliable access.

1-3-1 Advantages of Hadoop Distributed File System [3]:

- **Very Large Files:** Hadoop handles files hundreds of megabytes, gigabytes, or terabytes in size. There are Hadoop clusters in operation today that store petabytes of data.
- **Streaming Data Access:** HDFS is based around the idea that the most efficient data processing pattern is write once, and read multiple times. A data set is usually created or copied from the source, and then many analyzes are performed on that data set over time.
- **Commodity Hardware:** Hadoop does not require expensive and highly reliable hardware. They are designed to run on clusters of commodity devices (typically available hardware that can be obtained from multiple vendors) that have a high chance of node failure across the cluster, at least for large clusters, HDFS is designed to continue operating without noticeable user interruption in the face of such failure.
- **Low-latency Data Access:** Applications that require a short time to access data, in the tens of milliseconds, will not work well with HDFS, HDFS is optimized for high throughput data delivery, and this may be at the cost of delays.
- **Multiple Writers & Arbitrary File Modifications:** The file can be written and modified in HDFS by a single client at a certain moment in time i.e. there is no support for multiple writers and simultaneous modification to the file through a lease between the client and the master node and writing is always done at the end of the file as Hadoop does not support modifications with offsets of different places in the file.
- **Blocks:** When any file is written in HDFS, it is broken into small pieces of data known as blocks, HDFS has a default block size of 128MB which can be increased according to the requirements, These blocks are stored in the cluster in a way that is distributed over the different nodes, and this provides a Map-Reduce mechanism to process The data is parallel in the cluster [15].

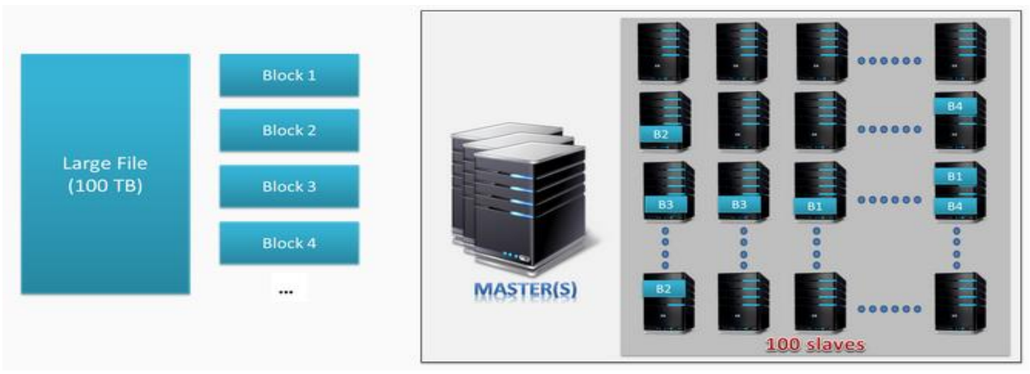


Figure (1-2) Storage in the Hadoop Distributed File System. [55]

- **Account Transfer is Cheaper Than Data Transfer:** The account requested by the application is more effective if it is performed close to the data it is working on. This is especially true when the size of the data set is large. This reduces network congestion and increases the overall throughput of the system.
- **Fault Tolerant:** Since HDFS uses a lot of hardware, it can continue to work even if some of them fail [3].

1-3-2 The Components of the Hadoop Distributed File System [55]:

The HDFS system contains two types of nodes that operate in the master-worker mode, where the HDFS cluster is composed
One NameNode (the master node) and a number of DataNodes (data nodes).

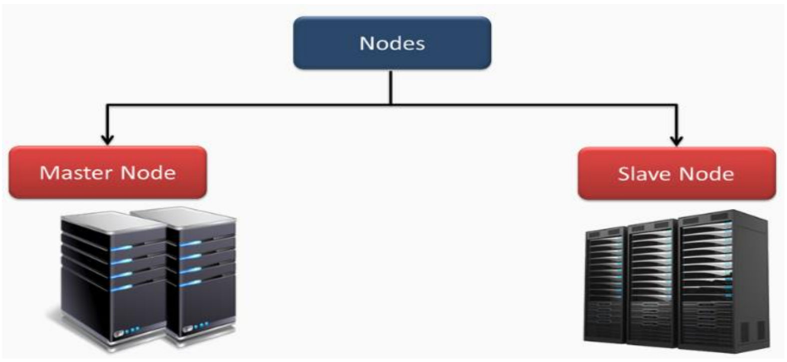


Figure (1-4) Types of Nodes in the Hadoop Distributed File System [55].

The master node manages the namespace and persistently stores the file system tree and metadata of all files and folders on its local disk as two files: FsImage and EditLog, EditLog constantly logs every change to file system metadata and stores the entire namespace including linking blocks to files and file system properties in a file called FsImage.

The master node also has knowledge of the data nodes on which all blocks of a given file are located and regulates clients' access to the files [15].

Data nodes are considered as operating nodes in the file system, where they store and retrieve blocks when requested (by clients or the master node) the data node also creates a block, deletes it, and copies it to other data nodes based on instructions from the master node and the application can specify the number of file replicas to be maintained by HDFS The number of copies of a file is called the replication factor this information is stored by the master node, and the data node also sends reports to the master node periodically containing lists of the blocks it stores.

1-3-3 Block Caching [3]:

Normally the DataNode reads blocks from disk, but for frequently accessed files the blocks can be explicitly stored in the data node cache (outside the heap by default).

1-3-4 SafeMode [15]:

On startup, the master node enters a special state called safe mode. No data block copying occurs when the master node is in safe mode. The master node receives Heartbeat and Blockreport messages from the data nodes, The block report message contains the list of data blocks hosted by the data nodes, Each block has a specified minimum number of copies, when the minimum copy number condition for blocks is reached + 30 additional seconds the master node exits from the safe mode, then selects the list of data blocks that are still less than the specified number of replicas and copies them to other data nodes.

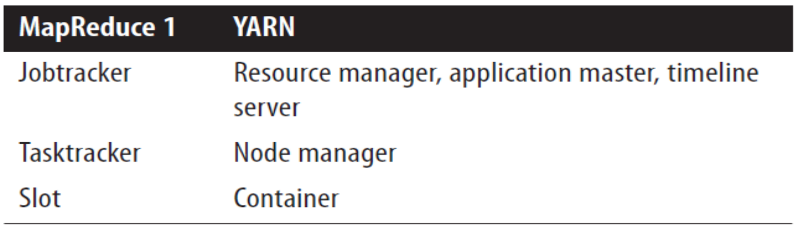
MetaData [55]:

HDFS uses a registry called the EditLog to constantly log every change that occurs to the file system metadata, for example creating a new file in HDFS makes the master node insert a record in the EditLog that indicates that, and changing the copy factor causes a new record to be inserted into the EditLog.

The entire namespace including file associations and file system properties is stored in a file called FsImage, EditLog and FsImage store files in the master node's local file system, and when it starts it loads the namespace from the last FsImage saved in its memory, applies and merges EditLog on FsImage to give a new namespace, then enter Safe Mode.

1-4 Comparison Between YARN1 & MapReduce [3]:

The distributed implementation of MapReduce in the original version of Hadoop (version 1 and earlier) is sometimes referred to as "MapReduce 1" to distinguish it from MapReduce 2 which is the representation that uses YARN in Hadoop 2 and later, There are two types of agents in MapReduce 1 which are the JobTracker which controls the task execution process and one or more TaskTrackers, The work tracker coordinates all the tasks that are running on the system by scheduling the tasks to run on the task trackers, TaskTrackers perform tasks and send progress reports to the work tracker, which keeps a record of the overall progress of each work, If a task fails, the work tracker can reschedule the tasks on another task tracker. In MapReduce 1 Work Tracker takes care of scheduling tasks (assigning tasks to task trackers) and monitoring task progress (tracking tasks, restarting failed or slow tasks). By contrast, at YARN these responsibilities are handled by separate entities: : Resource Manager and Application Manager (one for each MapReduce work) The work tracker is also responsible for storing the history of completed works, although it is possible to run a work history server as a separate entity to lighten the load on the work tracker. In YARN there is a server called Timeline Server that stores the application history.



MapReduce 1	YARN
Jobtracker	Resource manager, application master, timeline server
Tasktracker	Node manager
Slot	Container

Table (1-1) compare YARN with MapReduce1 [3].

YARN is designed to solve many of the limitations in MapReduce1 and the benefits of using YARN include:

1. **Scalability:** YARN can run on clusters larger than MapReduce1, MapReduce1 hits scalability bottlenecks at 4,000 nodes and 40,000 tasks stemming from the fact that the work tracker has to manage both tasks and works. YARN overcomes these limitations by virtue of its partition design for resource manager and application manager, it is designed to reach 10,000 nodes and 100,000 missions. In contrast to the work tracker, each version of the Work MapReduce application has its own dedicated application manager that runs for the duration of the application.
2. **Availability:** High Availability (HA) is usually achieved by copying or duplicating the requested state to another proxy to take over the work required to provide the service if the first proxy fails. However, the rapid change in the work tracker memory state (each important state is updated every few seconds) makes it very difficult

to achieve availability in the work tracker, With the division of job responsibilities between the resource manager and the application manager in YARN, the service became more accessible by following the principle of divide and conquer which meaning that the issue of availability became related to the availability of the resource manager and the availability of the application manager [17].

3. **Utilization:** In MapReduce1 each task tracker is set up with a fixed allocation of fixed-size "slots", which is divided into slots for corresponding tasks and slots for reduction tasks at initialization time. A corresponding slot can only be used to run a corresponding task, and the reduction slots can only be used for a reduction task. In YARN the node manager manages a set of resources, rather than a fixed number of slots. MapReduce running on YARN will not reach the state in which it must wait for the task to wait because only corresponding slots are available in the cluster as in MapReduce1 if the resources needed to run the task are available, Furthermore, the resources in YARN are fine grained so the application can make a request with what it needs, rather than having an indivisible slot, which may be too large (which wastes resources) or too small (which may causes a failure) for the specified task.
4. **Multitenancy:** In some ways, the biggest benefit of YARN is that it opens up Hadoop to other types of distributed applications outside of MapRadius, as MapRadius is just one of YARN's many applications, it is also possible for users to run different versions of MapReduce on the same YARN cluster, making the MapReduce upgrade process more manageable.

3-9 Hadoop MapReduce: MapReduce works by breaking the processing into two stages: the corresponding stage and the reduction stage. Each stage has key-value pairs as input and output, the types of which can be selected by the programmer, the programmer also defines two functions, the corresponding function represented by the Mapper class and then the abstract function map() is declared in it and the reduction function represented by the class Reducer Then the reduce() function is declared in addition to the last row responsible for doing the MapReduce work of By declaring the main() function inside it [3].

1-5 Data Flow [15] [3]:

Hadoop manages work by dividing it into tasks. There are two types of tasks, corresponding tasks and shorthand tasks. Tasks are scheduled using YARN and run on nodes in the cluster. If a task fails, it will be

automatically rescheduled to run on a different node. Hadoop splits MapReduce work income into fixed-size chunks called input splits, Hadoop creates one corresponding task for each input split, which runs the user-defined corresponding function for each record.
Hadoop makes every effort to run the corresponding job on the node where the HDFS input data resides because it does not use a valuable bandwidth, this is called data locality optimization. Another corresponding, so the task scheduler will look for an empty corresponding slot on a node in the same rack and sometimes even this is not possible so an off-rack node is used outside the rack which will move data between the racks the three possibilities are shown in the following figure:

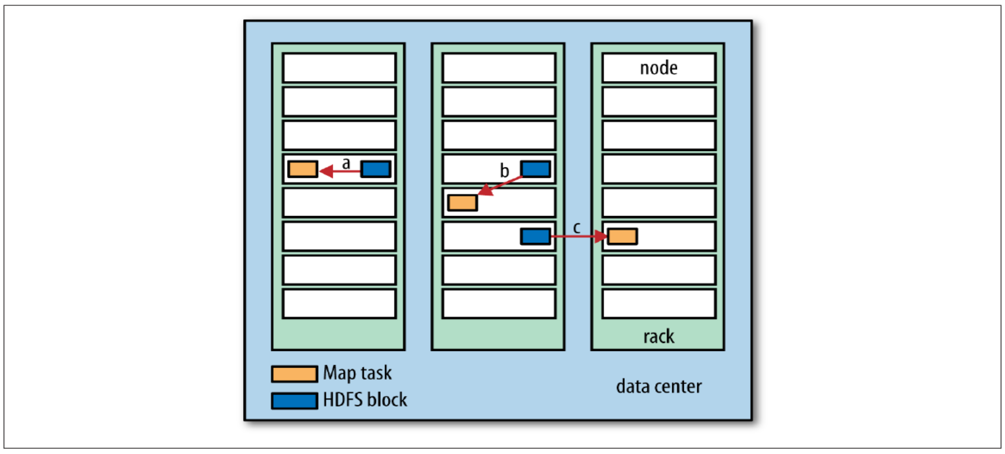


Figure (1-5) Data flow in HDFS [3].

Shorthand jobs do not have data locality the input to a single shorthand job is usually the output of all corresponding jobs. In the current example we have a single reduction job that is fed by all the corresponding jobs, so the sorted corresponding output must be moved across the network to the node where the reduction job is running on**,** where the results are combined and then passed to the user-defined reduction function The output of reduction is usually stored in HDFS for reliability where for each block of reduction function output its first replica is stored on the local node, with other replicas stored on outside nodes (off-rack nodes) for increased reliability, so writing the output of the reduction function consumes network bandwidth, but only as much as writing a new block on a normal HDFS pipeline.

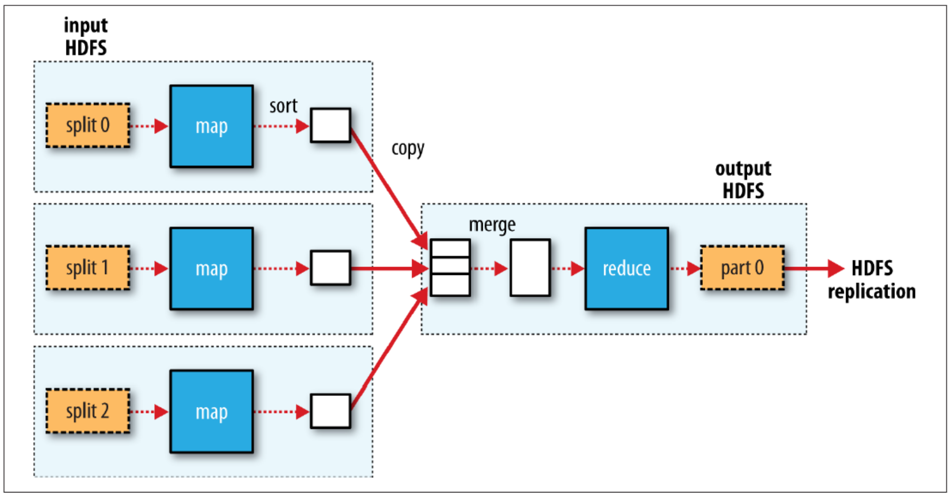


Figure (1-6) Data flow from several corresponding tasks to a single reduction task [3].

When there are multiple reductants, The corresponding tasks divide their outputs, each creating one section for each reduction task. There can be many keys (and their associated values) in each partition, but the records of any key reside within a single partition the partition can be controlled by the user-defined partition function but usually the default splitter works.

The following figure shows the data flow in the case of multiple reduction tasks. This graph shows why the data flow between corresponding and reduction tasks is called shuffle, as each reduction task is fed by several corresponding tasks.

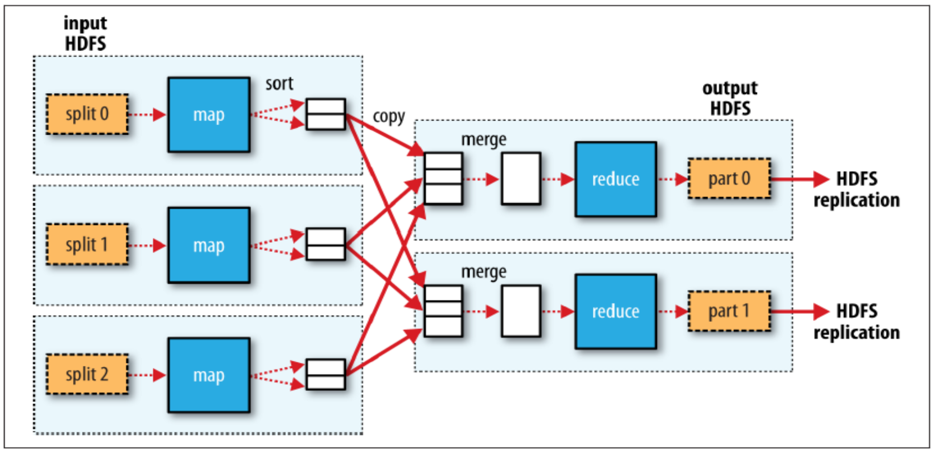


Figure (1-7) the flow of data from several corresponding tasks to several reduction tasks [3].

1-6 Combiner Functions:

Many MapReduce jobs are limited by the bandwidth available on the cluster, so in order to reduce the data transferred between the corresponding and reduction tasks, Hadoop allows the user to define an aggregation method to run on the outputs of the corresponding task, and the aggregation function's output constitutes the reduction function's input.

1-6-1 How MapReduce Work in Hadoop:

1- **The customer operates the MapReduce work through the following steps:**
   - The client asks the resource manager for a new application ID.
   - The directory in which the output will be saved is checked. For example, if it already exists, the job will not be sent and an error will appear in the program.
   - Work income is broken down into parts.
   - The resources needed to run the work as the work's JAR file, the setup file, and the input file portions are then copied to the Distributed File System (HDFS) in a directory named after the work identifier is specified.
   - Work is called.

2- **The YARN resource manager performs the process of allocating computing resources on the cluster [3]:**
   - When the application is called it hands the request to the YARN scheduler who allocates a container to work and then the resource manager launches the application manager to run under the node manager.
   - The application manager is a Java application that initializes the work and creates a set of objects that track the progress of the work. The application manager receives reports on the progress and completion of tasks, then retrieves the work income parts from the distributed file system and creates a corresponding task for each part. The number of reduction tasks is specified by the property mapreduce.job.reduces.
   - The application manager decides how to run the tasks that make up the work. If the work is small in size, it may choose to run it in the same JVM. This happens when the application manager sees that the overhead of allocating tasks and running them in new containers outweighs the benefits of running them in parallel compared to running them serially on a single node.

3- **The MapReduce Application Manager coordinates the running of tasks where the application manager and MapReduce tasks run in containers that are scheduled by the resource manager and managed by node managers:**

- When a job needs additional resources, the application manager requests new containers from the resource manager for corresponding and reduction.
- Corresponding assignment requests are made first and have a higher priority than those of the reduction assignments because all the corresponding assignments must be completed before the reduction phase begins.

4- Executing the task:

- Once container resources on a particular node are assigned to a task by the resource manager scheduler. The application manager starts the container by calling the node manager. The task is performed by a Java application whose main class is YarnChild and then corresponding or reduction tasks are run (step 11).
- YarnChild runs in a custom JVM so that a bug in the user-defined corresponding and reduction methods does not affect the node manager by causing it to crash or hang for example.
- Each job can perform setup and commit actions that run in the same JVM for file-based jobs, the commit action moves the job's output from its temporary location to its final location. The commit protocol ensures that when speculative execution is enabled, only one output is committed of the duplicate tasks and others are aborted.

5- Job Completion:

- When the application manager receives a notification that the last work task is finished, it changes the status of the task to 'Successful', a message is printed telling the user that the work has completed successfully, and the work stats and counters are printed.
- Finally the application manager and job containers upon completion of the work clean up their working state (so that the staged output is deleted), the commitJob() function is called and the job information is archived through the job history server for users to access later.

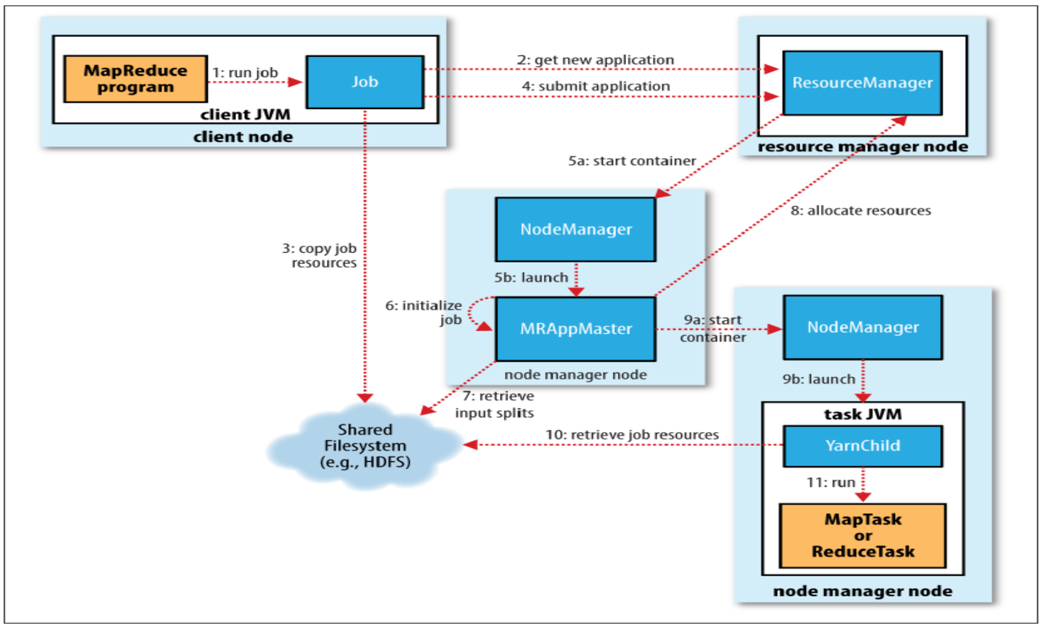


Figure (1-8) How to run Hadoop to make MapReduce [3].

Chapter Two

Hadoop Fault Tolerance

2-1 Introduction:

The concept of failure tolerance is defined as the ability of a system to continue to function properly without losing any data even if some components of the system fail to perform properly. 100% tolerance is very difficult to achieve but failures can be tolerated to some extent. Both MapReduce and HDFS are designed to keep up with any system failure, as Hadoop continuously monitors the data stored in the cluster. If any of the servers become unavailable, the drive fails, or the data becomes corrupted due to hardware or software issues, HDFS automatically restores the same data from one of the other servers on which it was backed up when partitioning. Similarly, when an analysis is in progress, MapReduce monitors the progress on all servers participating in the process, and if one of these servers is slow to return the result or fails to complete its task, MapReduce immediately routes another server on which the same piece of data is stored on the first. As a result, because of the way HDFS and MapReduce work, Hadoop offers reliable, fail-safe and scalable servers for storing and analyzing data at very low cost.

2-2 HDFS Fault Tolerance:

HDFS is very fault tolerant and is designed to be deployed on low-cost hardware, so hardware failure is the rule rather than the exception. HDFS may contain hundreds or thousands of devices, each of which stores a piece of file system data, and the fact that there are so many components and that each component has an irreversible failure probability means that some HDFS components may be ineffective [1], Therefore, fault detection and fast automatic recovery from it is a primary goal of HDFS and the main purpose of the system is to remove common failures, which occur frequently and the system stops working, and the most important advantages of using Hadoop is that there are two main methods that are used to produce the tolerance With failures in data redundancy and checkpoints [55] **the three types of failures in HDFS are:**

- Failed master node (NameNode).
- Data node failure (DataNode).
- Network partitions fail.

2-2-1 NameNode Failure and Fault Tolerance:

Without the NameNode the file system cannot be used so if the device running NameNode is wiped all files on the file system will be lost since there is no way to know how to rebuild the files from the blocks they are on DataNodes.

The master node represents a single point of failure (SPOF) and if it fails, all clients, including the MapReduce work, will not be able to read, write or list files because NameNode is the only repository of metadata and the link between each file and its block, in Such case the entire Hadoop system will be out of service until a new NameNode is fetched. This is why it is important to make the master node resilient to failure. Hadoop provides two mechanisms for this **Tolerance for master node failure is achieved in HDFS by:**

1. **Secondary Name Node (Hadoop1) [49]:** The first method is to back up the files that make up the persistent state of the file system metadata. Hadoop can be set up so that the name node writes metadata to multiple file systems the setting that is usually chosen is to write to the local disk as well as to a remote NFS system. It is also possible to run a Secondary NameNode, Despite its name, it does not act as a name node. Its main role is to periodically merge the NameSpace Image with the EditLog to prevent the EditLog from becoming too large. The secondary name node is usually run on a physical machine Separate because it requires the same CPU and memory requirements as the name node to perform the merge and keeps a copy of the NameSpace Image that the changes were merged with, which can be used if the name node fails, However, the information contained in the secondary node is considered incomplete, so when the name node fails, only changes since the last check point of the secondary name node will be lost. The new name node will not be able to serve requests until (i) loads image its NameSpace in memory, (ii) re-apply its modification log, and (iii) receive enough block reports from name nodes to leave safe mode. In large clusters that contain many files and clusters, the start time of the name node can take about 30 minutes or more, which causes a major problem with the availability of the Hadoop distributed file system.
2. **Standby Name Node(Haoop2) [39]:** The combination of copying metadata on the name node on multiple file systems and using a secondary name node to create checkpoints protects against data loss, but does not give high file system availability [10] Hadoop2 dealt with this situation by providing support for high availability of HDFS By putting a pair of name nodes in active-standby mode, and in case the active name node fails, the standby node assumes its duties to continue serving customer requests without significant interruption, and there are some infrastructure changes necessary for this to happen, namely [40] :

- The two name nodes must use high-availability shared storage to share the EditLog When any namespace modification is performed by the active node, it adds a new edit log to the edit log file stored in the shared directory. It applies it to its own namespace. In a failover case, the standby node ensures that all modifications are read from the shared volume before it is upgraded to the active state, and this ensures that the namespace state is fully synchronized before the failover occurs.
- Data nodes must send block reports to both the name node and the standby node because the links between block mappings are stored in the master node's memory and not on disk.
- Clients should be set up to handle failures at the name node using a mechanism that is transparent to users.

2-2-1-1 Failover & Fencing [3]:

The transition from the active name node to the standby node is managed by a new entity in the system called a failover controller. There are many failover controllers, but by default Hadoop uses ZooKeeper to ensure that only one active name node is running at any one time. Each name node runs a single failover controller whose job it is to monitor it (using a simple heartbeat mechanism) and trigger a failover when it fails
The failure can also be initiated manually by the administrator, for example in the case of routine maintenance and this type of failure is called a desired failure. In the event of an unwanted failure it is impossible to confirm that the failure is caused by a shutdown of the active name node, for example a poor network can cause a failover transition [42] even though the active name node it thinks is still running it is still the active name node. The process of ensuring that the previously active name node will not harm or corrupt the system is called fencing. QJM only allows one name node to write to the edit log at a time, however the previously active name node can still serve read requests. So, it is a good idea to set up an SSH fencing command that will kill the name node and a previously active node's access to the shared file system can be revoked by disabling its network port by remote commands.

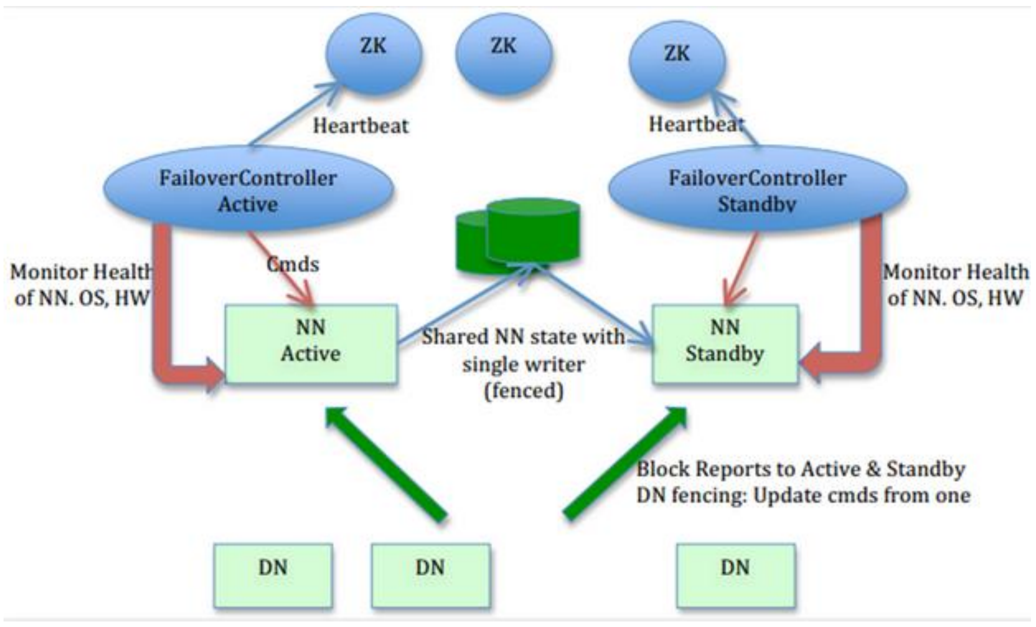

ZK
ZK
ZK
Heartbeat
Heartbeat
FailoverController Active
FailoverController Standby
Monitor Health of NN. OS, HW
Monitor Health of NN. OS, HW
Cmds
NN Active
NN Standby
Shared NN state with single writer (fenced)
Block Reports to Active & Standby DN fencing: Update cmds from one
DN
DN
DN

Figure (2-1) Failover and fencing in HDFS [3].

2-2-2 DataNode Failure and Fault Tolerance [49][55]:

Each data node sends a Heartbeat message to the name node periodically (every 3 seconds) which means it is working properly, network failure or data node failure can cause the name node not to receive the Heartbeat message, if the name node does not receive a heartbeat message for Ten minutes The data node is dead and its data is unavailable to HDFS and the name node does not forward any new IO requests to it, The death of a data node may cause the replication factor of some blocks to drop below the specified value, the name node constantly tracks which blocks need to be replicated and starts copying whenever needed, the need for replication may arise due to many reasons: The data node may become unavailable , the replica may get corrupted, the data node's hard disk may fail, or the file replication factor may increase.

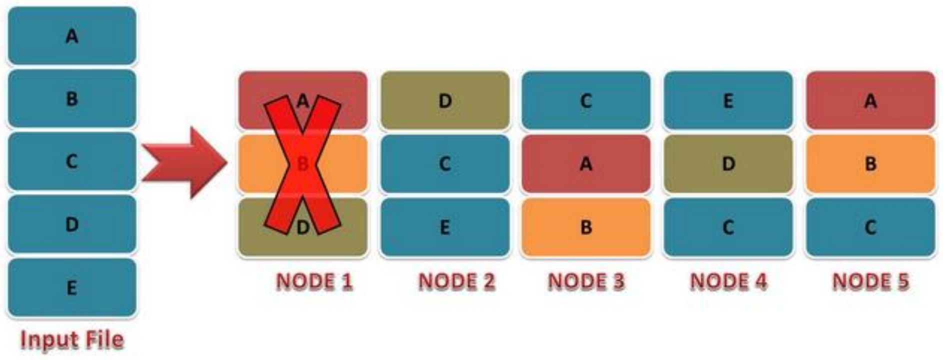


Figure (2-2) A data node failure in the Hadoop distributed file system [49].

2-2-2-1 Data Block Replication [55][15]:

HDFS is designed to reliably store very large files across devices in a large cluster. It stores each file as a sequence of blocks. All blocks in the cluster are the same size except for the last block in the file. File blocks are replicated in order to achieve failure tolerance, reliability, and high availability. The block size and replication factor can be specified for each file, the application can specify the number of file replicas, and the replication factor can be specified at the time of file creation and can be changed later.
By default, the HDFS replication factor is 3. The name node makes all decisions about block redundancy and periodically receives heartbeat messages and block reports from each of the data nodes in the cluster. A heartbeat is received indicating that the data node is working properly as a block report contains a list of all blocks on the data node [2]. Data redundancy provides instant recovery from failure but to achieve such tolerance a large amount of memory is consumed in storing data on different nodes meaning a large amount of memory and resources is wasted. But because this technology provides instant and quick recovery from failures, so it is used more frequently compared to checkpoints. [55] It is possible that the data block fetched from the DataNode was corrupted due to:

- Storage device malfunctions.
- Network malfunctions.
- Buggy software.

When the client creates an HDFS file, it calculates a Checksum for each block and stores it in a separate hidden file in the same HDFS namespace. When the client retrieves the contents of the file, it verifies that the data it received from each DataNode matches the stored Checksum. If not, the client can choose to retrieve this block from another DataNode that contains a copy of that block.

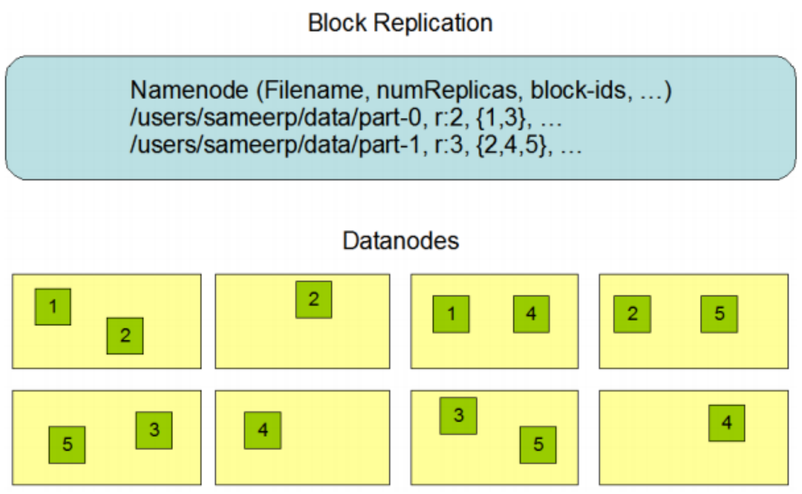

Block Replication
Namenode (Filename, numReplicas, block-ids, …)
/users/sameerp/data/part-0, r:2, {1,3}, …
/users/sameerp/data/part-1, r:3, {2,4,5}, …
Datanodes
1
2
2
1
4
2
5
5
3
4
3
5
4

Figure (2-3) Duplicate data blocks across the HDFS cluster [45].

Large HDFS clusters contain a group of computers that are commonly spread across many racks and communication between two nodes in different shelfs must pass through network switches. Switches In most cases the network bandwidth between devices in the same shelf is greater than the network bandwidth between devices in different shelfs. The name node specifies the ID of the shelf on which each data node is located.

The policy of placing replicas on unique racks is simple but not perfect as it prevents data loss when the entire shelf fails and allows bandwidth from multiple shelfs to be used when reading data. This policy distributes replicas evenly in the cluster, making it easier to balance the load. But it increases the cost of writing because writing needs to move the blocks to multiple racks. So in the common case when the replication factor is three:

- A copy is placed on a node in the local shelf.
- Another version is placed on a knot in a different remote shelf.
- The last are placed in a different node on the same remote shelf.

This policy does not affect data reliability because the chance of a node failure is much greater than a shelf failure.

HDFS attempts to fulfill the read request from the copy closest to the reader in order to reduce overall bandwidth consumption and read delay. If there is a copy on the same shelf of the reading node, it should be the copy that satisfies the read request.

2-3 Hadoop MapReduce Fault Tolerance:

In fact, user code may contain errors, processes may crash, and devices may fail. Hadoop has the advantage of handling such failures and allowing work to be completed successfully despite the failure. Failure includes any of the following entities: task, application manager, node manager and resource manager [3].

2-3-1 Task Failure:

The first and most common case of task failure is when user code throws a runtime exception. If this happens, its JVM returns the error to its application manager before it exits. The error is logged into the user logs. Then the application manager marks the task as a failed task, and frees the container so that its resources are available for another task.

The second case of failure is the JVM's abrupt exit. There may be a JVM error that is causing the JVM to terminate due to a certain set of circumstances for which the MapReduce user code is exposed. In this case, the node manager

notices that the process has exited and reports the main application manager to determine that this attempt has failed.
Suspended tasks are treated differently as the application manager notices that it has not received a progress update for some time and proceeds to determine the task as failed, and the JVM process of the tasks will be killed automatically after this period, the timeout period after which the failed tasks are usually 10 minutes and can be configured on All-action basis (or group basis) by setting the mapreduce.task.timeout property to a value in milliseconds.
Giving the timeout a value of zero disables the timeout so that long-running tasks are not flagged as failed. In this case, suspending the task will never release its container, and over time the cluster may slow down as a result. So this approach should be avoided, and ensure that the task reports its progress periodically.
When the application manager is notified that a task attempt has failed, it will reschedule the execution of the task, the application manager will try to avoid rescheduling the task on the manager of a node on which it has already failed. Moreover, if a task fails four times, it will not be rescheduled again. This value is configurable as the maximum number of attempts to run a task is controlled by the MapReduce property. map.maxattempts for corresponding tasks and mapreduce.reduce.maxattempts for reduction tasks and by default if any task fails four times the whole work will fail.

2-3-2 Application Master Failure:

As in MapReduce tasks that are given multiple attempts to succeed (in the face of hardware or network failure) the applications are retried in YARN in case of failure the maximum number of attempts to run the MapReduce application manager is controlled by the property mapreduce.am.max -attempt default value is 2. So if a master application manager fails twice it will not be tried again and the work will fail, the recovery process is as follows: Application manager sends periodic pulses to the resource manager and in the event of a failure in the application manager, the resource manager will detect the failure and start a new instance of the application manager under a new container (They are managed by the node manager).

2-3-3 Node Manager Failure:

If the node manager fails by crashing or running too slowly it will stop sending the heartbeat to the resource manager (or send it infrequently), the resource manager will notice that the node manager has stopped sending heartbeats if it does not receive a heartbeat message for ten minutes it will delete it from List of nodes to which containers will be scheduled, any task or application manager running on the failed node manager will be retrieved using the above mechanisms. Additionally the application manager arranges the restart of corresponding tasks that have been successfully run and completed on the failed node manager to be restarted if they belong to uncompleted works since their staging outputs on the local file system of the failed node manager may not be accessed by tasks reduction, a node manager may be blacklisted if the number of application failures is high, even if the node manager itself has not failed. The blacklist is accomplished by the application manager.

2-3-4 Resources Manager Failure:

The failure of the resource manager is dangerous because without it it is not possible to launch works or task containers before Hadoop 2.4 The resource manager was considered a single point of failure, since in the (unlikely) event of a hardware failure all running works fail and cannot be recovered. To achieve high availability (HA) it is necessary to run a pair of standby and active resource managers, if the active resource manager fails, the standby manager will take his turn as quickly as possible so that there is no significant interruption of the client.

Information about all running applications is stored in a highly available state store (powered by ZooKeeper or HDFS) so that a prepared resource manager can retrieve the failed active resource manager's base state. It can be rebuilt relatively quickly by the new resource manager when node managers send their first heartbeats. Also note that tasks are not part of the resource manager state, since they are managed by the application manager.

Chapter Three
Apache Spark

3-1 Introduction [63]:

In a very short time, Apache Spark has emerged as a new generation of big data processing and is being implemented faster than ever before. Spark runs the batch applications supported by Hadoop and supports a variety of workloads including interactive queries, streaming, machine learning, and graphic processing. With the rapid rise in popularity of Spark, The lack of good reference material you speak of is a major concern. Spark offers three main benefits, first, it is easy to use, second, Spark is fast, enables the user to interactively use and implement complex algorithms, and third, Spark is a generic working platform allowing the combination of multiple types of computations (such as SQL queries, word processing, and machine learning) which required Previously different work platforms.

3-2 A Unified of Spark [63][59]:

Spark project contains many closely integrated components. At its core, Spark is a "computer engine" responsible for scheduling, distributing, and monitoring applications that consist of many computational tasks across many operating devices.

At its core, Spark is fast and comprehensive with many components running high-level workloads such as SQL or machine learning. These components are designed to interact with each other, allowing the user to combine them similarly to adding libraries to a software project. The following are the components of Spark in brief:

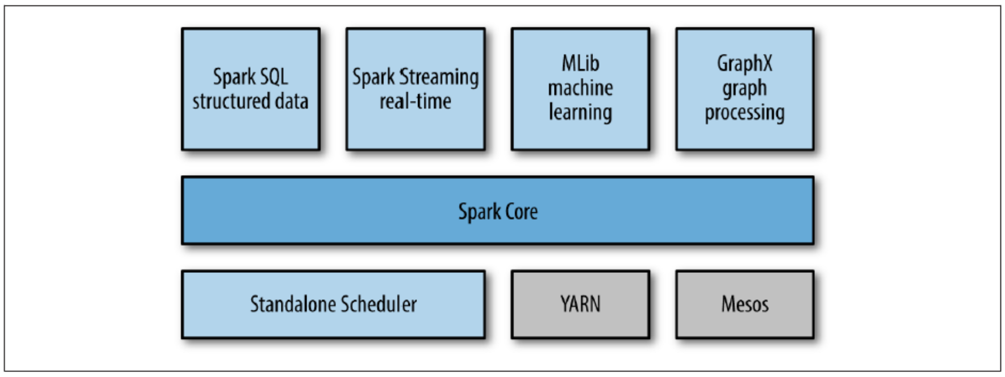


Figure (3-1) Spark standard stack [63].

1. **Spark Core:** It contains the main functions of Spark. The kernel contains components for task scheduling, memory management, failure recovery, and interaction with storage systems. The Spark kernel contains a set of APIs that contain the definition of the Resilient Distributed Data-   sets (RDDs) of Spark. distributed over multiple nodes which are processed in parallel.

2. **Spark SQL:** It is a package dedicated to dealing with structured data and enables developers to integrate SQL database queries and operations on RDD.
3. **Spark Streaming:** A Spark component that enables processing of live broadcasts of data such as log files generated by web servers or message queues containing status updates posted by users of a web service. Spark Streaming provides an API for processing data streams that closely matches the Spark Core RDD API, making it easier for programmers to know the project and transition between applications that process data stored in memory, on disk, or data that arrives in real time. Spark Streaming is designed to provide the same degree of failure tolerance, throughput and scalability as a Spark core.
4. **MLib:** Spark library to support machine learning algorithms (ML) MLlib provides multiple types of machine learning algorithms, including classification, regression, clustering, and collaborative filtering, as well as support functions such as model evaluation and data import.
5. **GraphX:** Libraries specializing in distributed processing on graphs (graphs).
6. **Cluster Managers [59]:** Spark is designed to operate efficiently on a cluster of one to several thousand computer nodes, and to achieve this while maximizing flexibility, Spark can run a variety of cluster managers including YARN and is a viable option if Hadoop is pre-installed and Apache Mesos If we already have Apache Mesos on the cluster and a simple cluster manager built into Spark itself is called Standalone Scheduler and is appropriate if Spark is installed on a cluster of empty devices [35][63].

3-3 Storage Layers for Spark:

Spark can create distributed datasets from any file stored in the Distributed File System (HDFS) or other storage systems (Amazon S3, Cassandra, Hive, HBase, etc.). It is important to stress that Spark does not require Hadoop because it simply has support for other storage systems not supported by Hadoop. It supports text files, sequential files, Avro, Parquet etc...

3-4 Resilient Distributed Dataset (RDD):

RDD is the basic data structure of Apache Spark, which is simply a distributed set of immutable and static elements that are logically partitioned and computed on the different nodes in the cluster. Users create RDDs by loading an external data set into their driver program and Spark automatically divides each RDD into multiple partitions and distributes the data in the RDDs across the cluster and parallelizes your operations on them.

Once created, it can be performed with two types of operations: transformations and actions. Transformations create a new RDD from a previous one by making modifications to it (one common transformation is to filter data that matches the value passed to the predicate) Example we can use the filter() transformation to create a new RDD that holds only strings containing a particular word.
RDDs are recalculated by default every time an action is run on them, so when the RDD needs to be reused in multiple actions, they can be kept in different user defined locations via the RDD.persist() function. After it is first computed, Spark will store the contents of the RDD in memory divided across devices in the cluster. RDDs can also be saved to disk instead of memory.
The behavior of not keeping RDD by default may seem useful when dealing with large data sets. When you deduce new RDDs from each other using transformations, Spark keeps track of the set of dependencies between the different RDDs, called a lineage graph, and uses this information to calculate each RDD on demand and recover lost data if part of the RDD is missing (Fig. 5-2) Graph of Ratios:

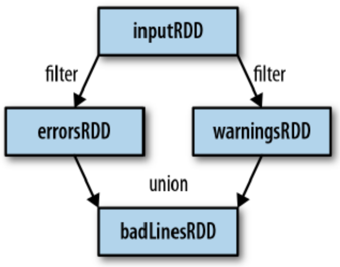


Figure (3-2) is an example of a graph of ratios [63].

RDD achieves partial fault tolerance through the feature of tracking the transfer rates applied to RDD, where the transfers applied to RDD are recorded to be repeated when data is lost or the cluster node fails, and high efficiency is achieved through parallel processing across multiple nodes in the cluster, reducing Duplicate data between those nodes.

3-4-1 Lazy Evaluation [59]:

Transformations on RDDs are evaluated lazy which means that they will not be executed until the subsequent action needs its result. Lazy evaluation means that when we invoke a transform on an RDD (for example, calling map()) the operation is not executed immediately, instead Spark internally logs metadata that indicates that this transformation has been requested on that RDD, That is, instead of retaining the data contained in RDD, how or what transformations needed to compute the data are preserved, which avoids the need to process the data unnecessarily. Likewise for the process of loading data into RDD, when sc.textFile() is called, the data is not loaded until it is necessary , Spark uses lazy evaluation to reduce the number of times data is accessed by grouping processes together, In systems like Hadoop, developers often have to spend a lot of time thinking about how to group processes together to reduce the number of times MapReduce accesses data. Organize their programs into smaller, more manageable operations.

3-5 Distributed Execution in Spark:

Work in Spark consists of a direct acyclic (DAG) scheme for a group of stages, each roughly equivalent to a corresponding or reduction phase in MapReduce, the stages are divided into tasks and run in parallel on portions of an RDD distributed across the cluster — just like the tasks in MapReduce - The work runs as an application (represented by a copy of SparkContext) and the application can run more than one work, sequentially or in parallel, and provides a mechanism for accessing the work to an RDD that was cached by a previous work in the same application.

Each Spark application consists of a driver program that launches different parallel operations on the cluster. The driver contains the application's main function, defines datasets distributed on the cluster, and then applies the operations to them. The driver accesses Spark through an intent called SparkContext. which represents the connection with the cluster and through which we can build a new RDD, to run operations on RDD the driver usually manages a number of nodes called executors, for example if we are running a count() operation on the cluster, different devices may perform the computation on lines in different ranges of the file, Spark takes the desired function It is executed automatically and sent to the executing nodes, so you can write code in one driver and parts of it are automatically run on multiple nodes, Figure (5-3) shows how Spark works.

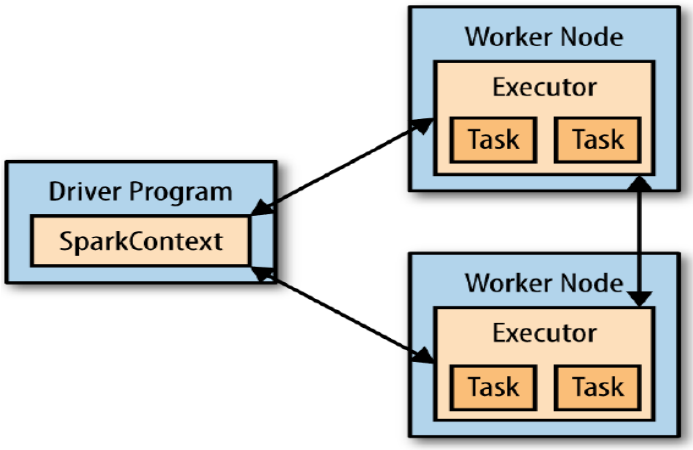


Figure (3-3) Components of distributed processing in Spark [63].

3-6 Spark Cluster Components [59][63]:

In distributed execution mode, Spark uses a master/slave architecture with one central coordinator and many distributed worker nodes called the driver, The launcher communicates with many distributed workers, each of which is called an executor. The launcher program runs in its own Java process, and each implementer runs in a separate Java process as well, both operator and its implementers make up the Spark application. The Spark application is launched on a group of devices using an external service called a cluster manager, which knows the workers, where they are located, the size of memory, and the number of cores of each working node.

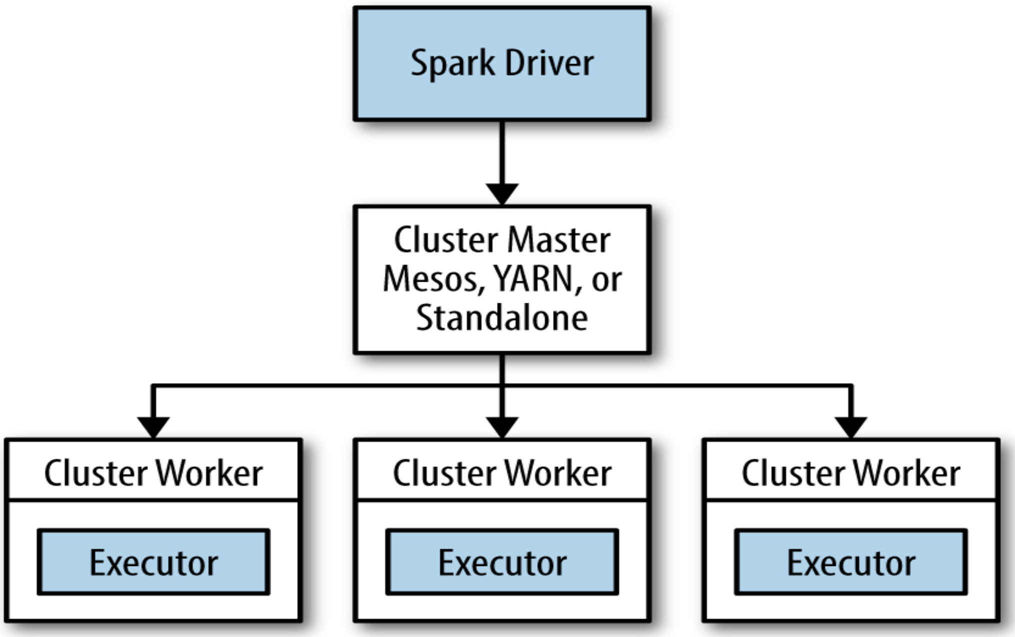


Figure 3-4 Components of a distributed Spark application [63].

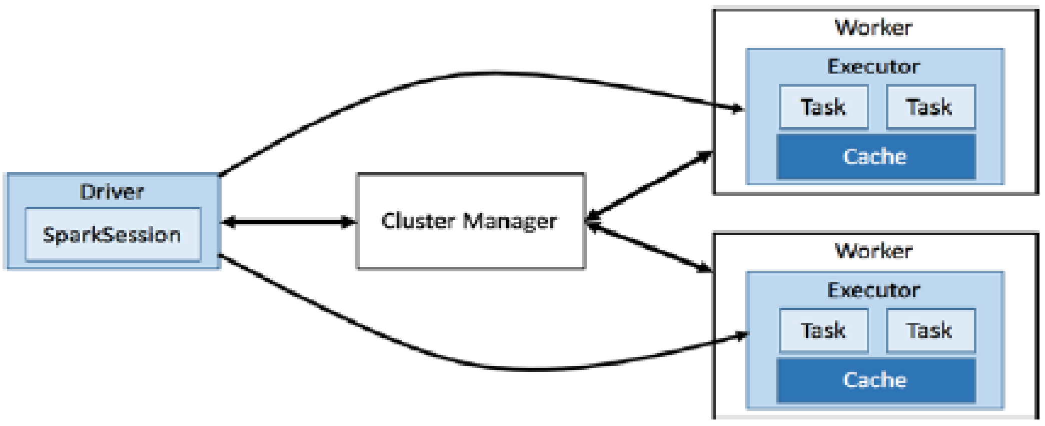


Figure 3-5: Spark's distributed application implementation mechanism [59].

3-6-1 The Driver [59][63]:

The trigger is the process in which the main() method is run. It is the process that runs the user code that creates the SparkContext, creates the RDDs, performs transformations and actions, and once the trigger finishes, the application terminates. When the trigger is running it does two things:

1. Converting a user program into tasks [59]: Spark is responsible for converting a user program into actual execution units called tasks. All Spark programs follow the same architecture: it generates RDDs from some input, derives new RDDs from the originator using transformations, and performs actions to collect or save data, Spark generates a directed acyclic graph DAG from the operations, When the driver runs, it converts this logical graph into an actual execution plan and transforms a Spark (DAG) into a group of stages, each stage in turn consisting of multiple tasks and tasks are grouped and prepared to be sent to the cluster. Tasks are the smallest unit of work in Spark. The user program can Typical launch hundreds or thousands of individual missions.
2. Scheduling of tasks on the two implementers: Depending on the actual execution plan, the operator must coordinate the scheduling of individual tasks on the two implementers so that when the two implementers start the operator requests to reserve resources for implementers on working nodes and release them. Then the implementers register themselves with the operator so that it always has full visibility of the application implementers.

   Each executor represents a process capable of running tasks and storing RDD data. The Spark driver will look at the current set of executors and try to schedule each task to a suitable location based on the position of the data.

   Task execution may be affected by cached data, so the driver also tracks the location of the cached data and uses it to schedule future tasks that access that data.

3-6-2 Executors [59]:

Implementers in Spark are working processes responsible for running individual tasks in a Spark business. Executors run once at the start of a Spark application and usually run for the life of the application, although Spark applications can continue if the implementers fail.
Executors have two roles, first the implementers run the tasks that make up the application and return the results to the launcher. Second, it provides in-memory storage for RDDs that are cached by user programs, through a service called Block Manager that lives inside each port, and since RDDs are cached directly inside the two ports can run tasks along with the cached data.

3-7 Spark on YARN [3]:

Running Spark on YARN provides integration with other Hadoop components and is the most convenient way to use Spark When you already have an existing Hadoop system, Spark provides two deployment modes to run on YARN:

- **YARN Client Mode [3]:**
  This pattern is suitable for programs that need an interaction that starts yarn when a new SparkContext is built by the driver program that sends an application yarn to the resource manager that starts a new container on the node manager in the cluster and runs on it AM ExecutorLauncher whose job is to launch executors within yarn containers By requesting new resources by the cluster manager and launching ExecutorBackend processes within the reserved containers, When the executor starts, it registers itself with the SparkContext, which helps it know how many executors are available to run tasks on and where they are, The default number of implementers is 2, the number of available cores for each is one, the default memory is 1024 MB, and the values can be changed by the spark-submit parameters while the program is running.
- **YARN Cluster Mode [3]:**
  In this pattern, the spark driver program is run inside the process assigned to the yarn application manager and not on the client node as in the previous pattern. driver inside YARN Application Master.

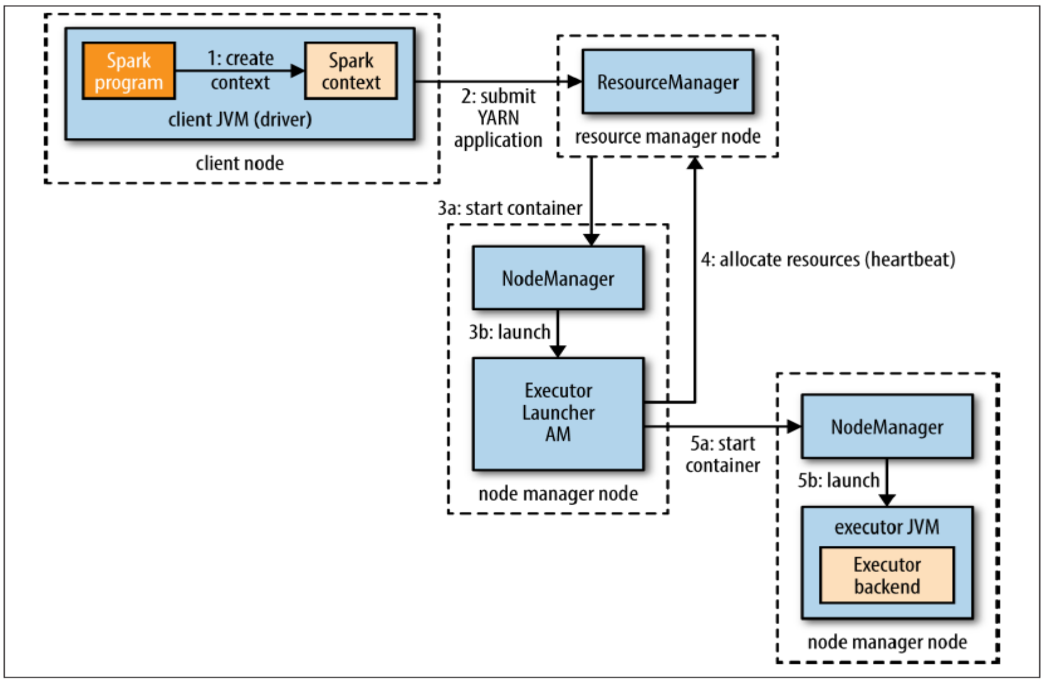


Figure (3-6) Client Mode [3].

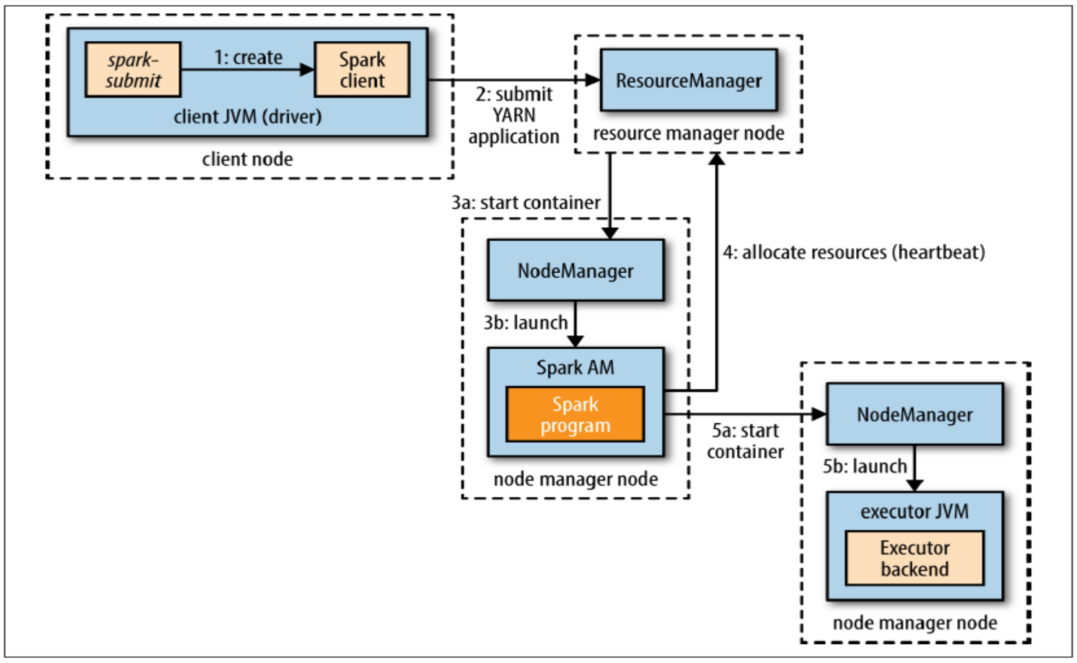


Figure (3-7) Cluster Mode [3].

Chapter Four
Apache Spark Fault Tolerance

4-1 Introduction [51]:

High-level cluster programming paradigms such as MapReduce have been widely adopted to process increasing amounts of data in industry and science. These systems simplify distributed programming by providing locality-aware scheduling, failure tolerance, and load balancing Load allowing a wide range of users to analyze their own large data sets using clusters of commodity devices. Most current cluster computing systems are based on an acyclic data flow model. That is, records are loaded from persistent storage (eg a distributed file system) passed through a DAG made up of specific processes and back to persistent storage. Knowledge of the data flow graph allows automatic scheduling of work and recovery from failures.

4-2 Spark Fault Tolerance [74]:

One of the main advantages of Spark is that it provides strong guarantees of failure, as long as the input data is stored reliably, Spark will always calculate the correct result from it despite the failure.

4-2-1 Resilient Distributed Datasets (RDDs) [51]:

Flexible Distributed Data Sets (RDDs) allow programmers to perform in-memory computations in large computer clusters while maintaining failure tolerance. Distributed fail-tolerant is creating restore points from data or logging updates made to them, In our target environment, creating data recovery points is expensive: it requires replicating large data sets across devices across a network of data centers, which typically have much less bandwidth than the memory bandwidth within a device that will also consume additional storage space (replicating data into access memory). RAM reduces the total amount that can be cached while accessing disk slows applications), Coarse-grained transformations are just in the sense that we can log a single operation to be applied to many logs, then remember the series of transformations used to build the RDD (that is, its proportions) and use it to recover the missing fragments. This is what RDD does to efficiently achieve failure tolerance as Provides a restricted form of shared memory based on deterministic coarse-grained transformations rather than fine-grained updates of the shared state. RDDs also rebuild the lost partitions through descent: RDD contains enough information about how it is derived from other RDDs to rebuild only the lost partition, without the need for a restore point. Duplicate data plays an important role in the self-recovery process as we can recover lost data from over duplicate data.

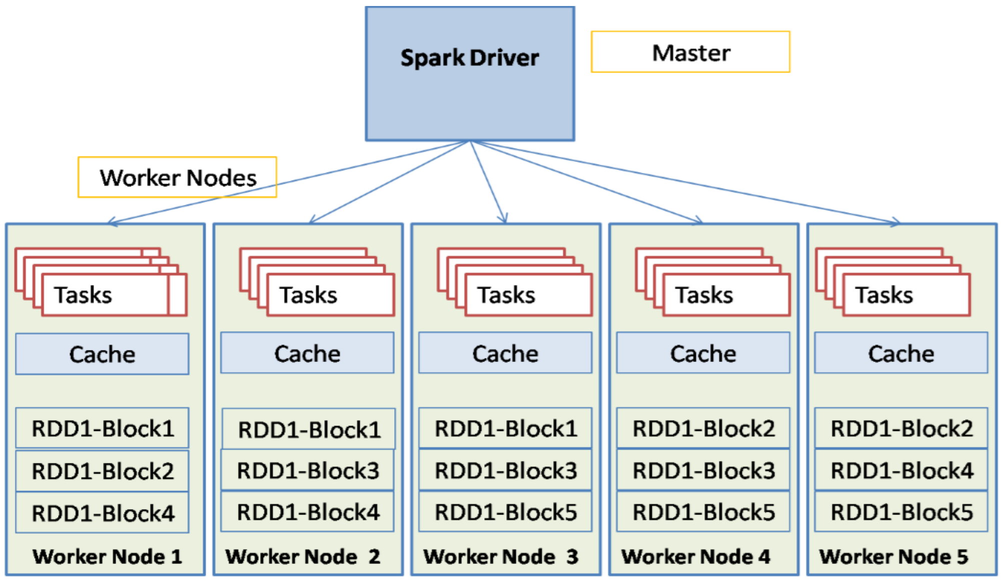


Figure (4-1) Repetition of RDD on working nodes [75].

How is failure tolerance achieved through a DAG? As we know that the DAG keeps a record of the operations applied to the RDD i.e. it keeps all the details of the tasks performed on the different sections of the RDD so in case of failure or in case of any RDD missing we can fetch it easily with the help of the DAG diagram, How is failure tolerance achieved through a DAG? As we know that the DAG keeps a record of the operations applied to the RDD meaning that it keeps all the details of the tasks performed on the different sections of the RDD so in case of failure or in case of any RDD missing we can fetch it easily with the help of the DAG diagram, For example, if there is any operation in progress and suddenly there is a loss of RDD data, then with the help of the cluster manager we will identify the partition in which the loss occurred, then through the DAG we will assign the work to a new node which will re-execute the necessary thread to get the missing part.

4-2-2 Check Pointing [51][63]:

Although the lineage information traced by RDDs always allows the program to recover from failures, this recovery can be time-consuming for RDDs with long lineage chains, for example in tasks that depend on past values it is useful to store the RDDs when Certain time points on permanent storage.

This feature periodically saves data about the application to a trusted storage system such as HDFS or Amazon S3 for use in failover recovery Checkpoints are useful in reducing instances that must be recalculated on failure When data is lost State is recalculated using the graph of conversions, but the checkpoints control how long we must get back in the account.

Spark currently provides an application programming interface (API) for the investigation of a restore point feature, where a check is performed to create restore points automatically, Since the scheduler knows the size of each data set as well as the time it took to process it, it should be able to select an optimal set of RDDs to create restore points on to reduce system recovery time.

4-2-3 Driver Fault Tolerance [63]:

When the launcher fails in batch programs, the implementers will be terminated and the whole Spark application will fail. Not only do RDDs lose their lineage, but the workers also lose their master from whom they receive orders, and all the results of the current calculations completed by the workers will be lost as well, leaving the user with the failure of the launcher with no solution but restart the application from the beginning.
Driver node failures require a special method to create a SparkContext. Instead of creating a new SparkContext, SparkContext.getOrCreate() is called, which takes the directory where the restore points have placed the data. Spark does not automatically restart the launcher if it crashes, so you need to monitor it with a tool like monit and restart it.

4-2-4 Worker Fault Tolerance [51][63]:

The DAG is generated and executed by the software operator The workers will process a portion of the DAG and they will send the results of the tasks to the software operator which keeps track of the tasks and block information of the running nodes which means that if the workers fail, It can reschedule tasks on other workers, knowing that the results of calculations not sent to the operator will be lost, for tolerance of working node failures Spark duplicates all data on working nodes, thus all RDDs generated through transfers on duplicate income data are tolerated With a working node failing, the RDD line of descent allows the system to recalculate the missing data along the way from the remaining replica of the input data.

4-2-5 Executer Fault Tolerance:

Each port sends a pulse message to the trigger periodically every 10 seconds By default, when a port failure is detected, the trigger tells the task scheduler about the loss of the port which later handles the loss of the tasks executed on the port and also tells the driver the DAG scheduler to remove all traces (such as shuffle blocks) of the missing port, Moreover the trigger also asks SparkContext to replace the missing port with another through its communication with the cluster manager, when the port fails in batch programs the cached data and partially computed RDDs will be lost however Spark RDDs are tolerant of this failure and Spark will start a new port to recompute This data from the original data source will degrade performance when the data is recalculated but failure of the executor will not result in work failure.

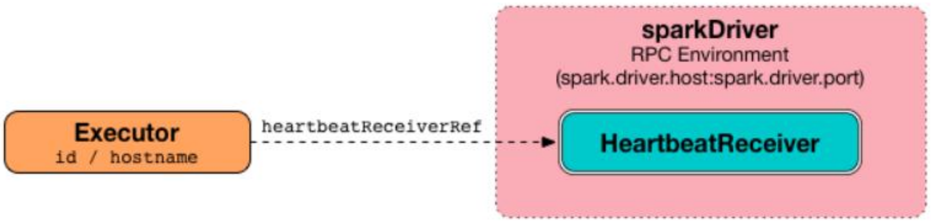


Figure (4-2) The mechanism of heartbeat in Spark [75].

4-2-6 Processing Guarantees [75]:

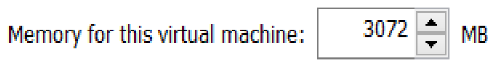
Because of its worker fail-tolerance guarantees, Spark can provide exact semantics for all transformations (once semantics exactly) meaning that even if a worker fails and some data is reprocessed, the transformed end result (meaning the transformed RDDs) will be the same as if the data had been processed Exactly once.

Chapter Five
The Practical Part

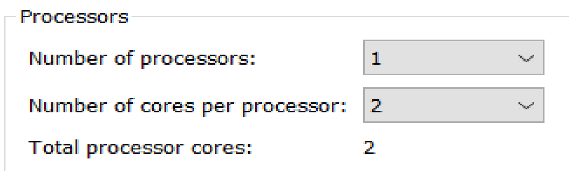
5-1 Introduction:

We will deploy Hadoop and Spark on virtual machines to take advantage of all the advantages of virtualization, which can make cluster management easier, and virtual machines are quick to deploy and can be moved without risk [25], in our research we will use VMware and the cluster in both Hadoop and Spark will contain four Each contract possesses the following specifications:

Operating System: Ubuntu 16.04 - Memory: 3GB – Processors: 2 core -Hard Disk: 60 GB SSD.



Memory for this virtual machine: 3072 MB



Processors

Number of processors: 1

Number of cores per processor: 2

Total processor cores: 2

1. The node master will be the master node (192.168.65.157).
2. The node hdslave1 the first data node (192.168.65.156).
3. 3. The node hdslave2 is the second data node (192.168.65.155).
4. The node hdslave3 is the third data node (192.168.65.161).

5-2 Specifications of the computer used:

1. Processor: Intel® Core™ i7 7500U Processor, 2.7 GHz (4 M Cache, up to 3.5 GHz).
2. Operating System: Windows 10 Pro.
3. Memory: 16GB DDR4, 2133MHz.
4. Graphic: AMD + RadeonTM R5 M420, with 2GB, VRAM.
5. Storage: SATA HDD 1TB 5400RPM + SSD Hard Drive 240 GB Internal.

5-3 Implementation steps:

1. Install Hadoop.
2. Install HiBench and connect it with Hadoop.
3. Choose a set of programs to test for failure as they are executed.
4. Determine the criteria used to evaluate performance.
5. Implementation and evaluation of Hadoop's performance when implementing selected programs.
6. Choosing a set of possible failures on both platforms.
7. Implementation and evaluation of Hadoop's performance when implementing programs with injection of failures into the components of the environment.
8. Install Spark and connect it with HiBench.
9. Implementation and evaluation of the performance of Spark at the selected programs.
10. Implementation and evaluation of the performance of Spark when implementing programs with the injection of failures into the components of the environment.

11. Results and recommendations.

- The previous steps are divided into two parts, illustrated in the following figures:

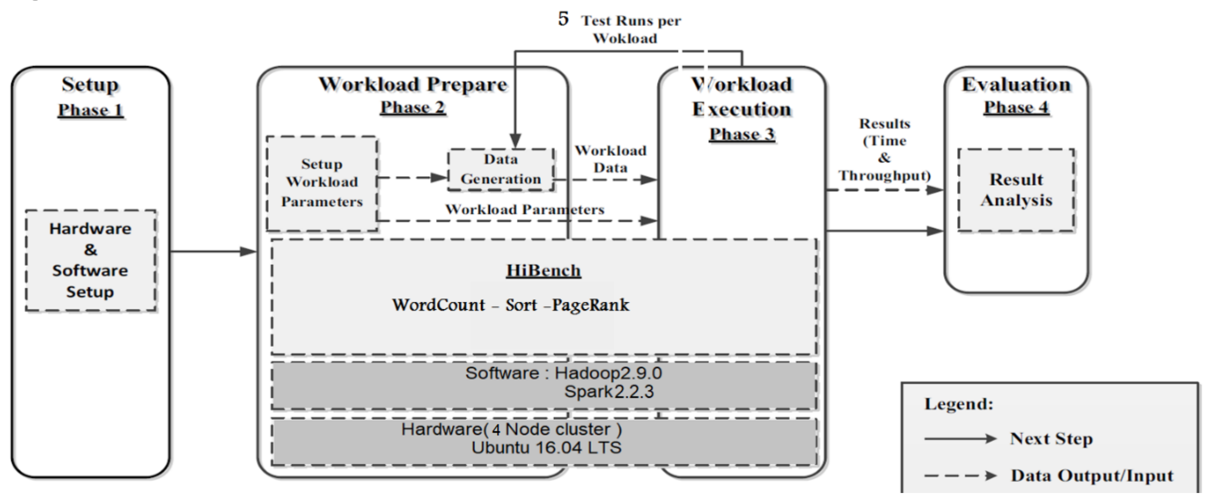


Figure (5-1) is a diagram of the first section of the comparison process using HiBench. [66]

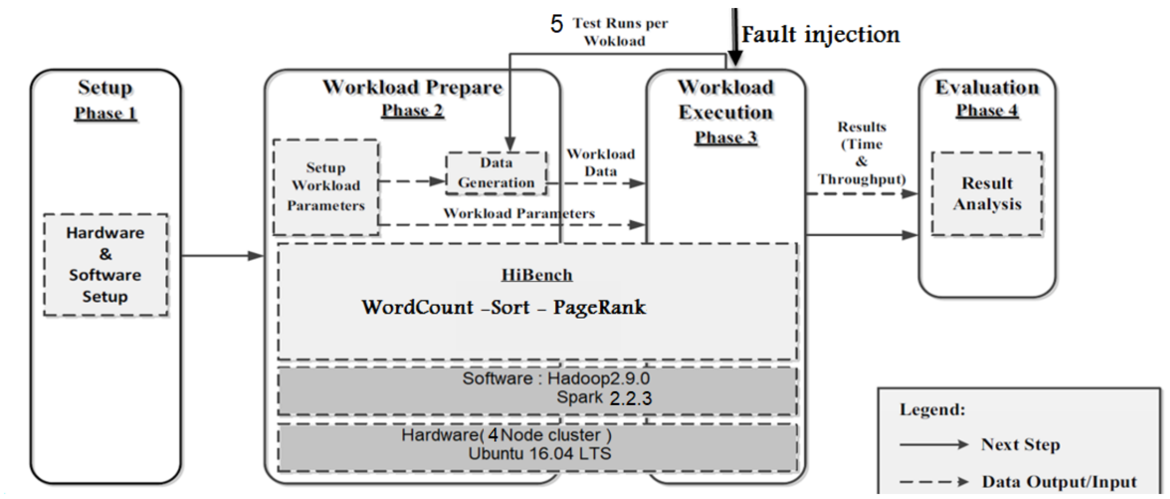


Figure (5-2) is a diagram of the second section of the comparison process using HiBench.

5-3-1 Hadoop cluster installation [64][1]:

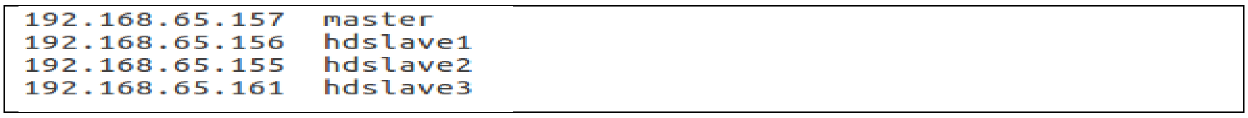
1- Install Java on all nodes with the following instruction:

```
sudo apt-get install openjdk-8-jdk
```

2- Install ssh on all nodes with the following instruction:

```
sudo apt-get install openssh-server
```

3- Edit the hosts file on all nodes with the $sudo gedit /etc/hosts command and insert the following lines:



```
192.168.65.157   master
192.168.65.156   hdslave1
192.168.65.155   hdslave2
192.168.65.161   hdslave3
```

4- Edit hostname file in all nodes and put new node name sudo gedit /etc/hostname $

5- Generate an ssh key on the master node: ssh-keygen.

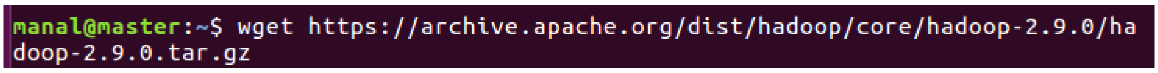
6- Copy the content of the .ssh/id_rsa.pub file in the master node to ssh/authorized_keys in the master node:

```
$ cat $HOME/.ssh/id_rsa.pub >> $HOME/.ssh/authorized_keys
```

7- As well as to all other nodes through the following instructions:

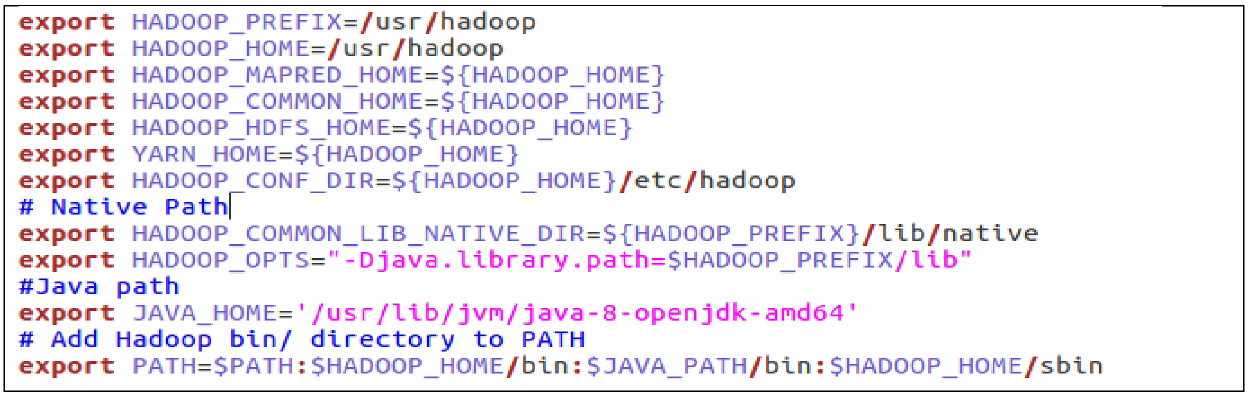
```
$ sudo ssh-copy-id -i /home/manal/.ssh/id_rsa.pub manal@hdslave1
$ sudo ssh-copy-id -i /home/manal/.ssh/id_rsa.pub manal@hdslave2
$ sudo ssh-copy-id -i /home/manal/.ssh/id_rsa.pub manal@hdslave3
```

8- Download the Hadoop platform from the official website link in all nodes in the cluster:



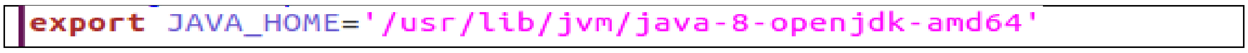
```
manal@master:~$ wget https://archive.apache.org/dist/hadoop/core/hadoop-2.9.0/ha
doop-2.9.0.tar.gz
```

9- Modify the .bashrc file and add an environment variable containing the java path and also add the Hadoop path as an environment variable in all nodes $ sudo gedit .bashrc



```
export HADOOP_PREFIX=/usr/hadoop
export HADOOP_HOME=/usr/hadoop
export HADOOP_MAPRED_HOME=${HADOOP_HOME}
export HADOOP_COMMON_HOME=${HADOOP_HOME}
export HADOOP_HDFS_HOME=${HADOOP_HOME}
export YARN_HOME=${HADOOP_HOME}
export HADOOP_CONF_DIR=${HADOOP_HOME}/etc/hadoop
# Native Path
export HADOOP_COMMON_LIB_NATIVE_DIR=${HADOOP_PREFIX}/lib/native
export HADOOP_OPTS="-Djava.library.path=$HADOOP_PREFIX/lib"
#Java path
export JAVA_HOME='/usr/lib/jvm/java-8-openjdk-amd64'
# Add Hadoop bin/ directory to PATH
export PATH=$PATH:$HADOOP_HOME/bin:$JAVA_PATH/bin:$HADOOP_HOME/sbin
```

10-Add java path to hadoop-env.sh file in all nodes:



```
export JAVA_HOME='/usr/lib/jvm/java-8-openjdk-amd64'
```

11-Modify the core-site.xml file to be as follows in all nodes:

```
<property>
<name>fs.defaultFS</name>
<value>hdfs://master:9000</value>
</property>
```

12-Modify the hdfs-site.xml file to look like this in the master node:

```
<property>
<name>dfs.replication</name>
<value>3</value>
</property>
<property>
<name>dfs.namenode.name.dir</name>
<value>file:/usr/hadoop/hadoop_data/hdfs/namenode</value>
</property>
```

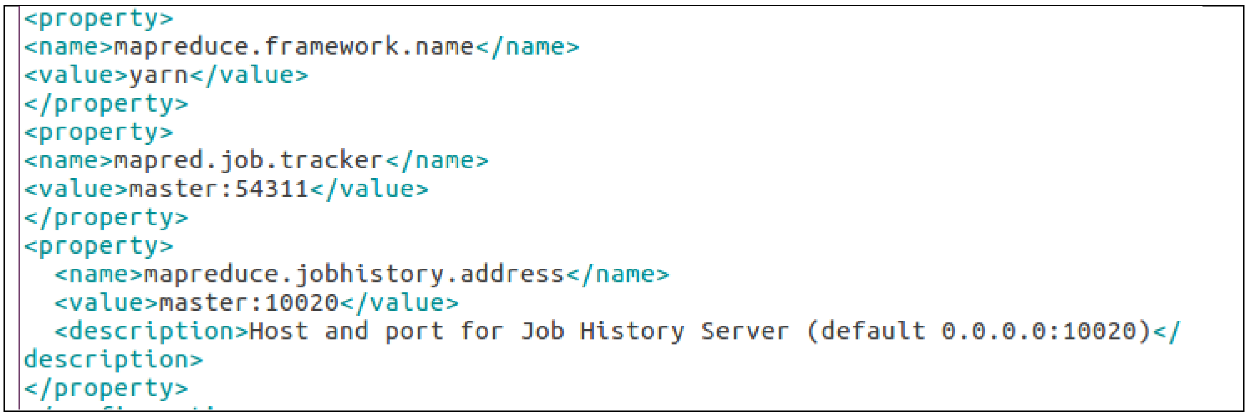
13-Modify the hdfs-site.xml file to look like this in the master node:

```
<property>
<name>dfs.datanode.data.dir</name>
<value>file:/usr/hadoop/hadoop_data/hdfs/datanode</value>
</property>
```

14-Modify the yarn-site.xml file to look like this in the master node and the other nodes:

```
<property>
 <name>yarn.nodemanager.aux-services</name>
 <value>mapreduce_shuffle</value>
 </property>
<property>
<name>yarn.nodemanager.auxservices.mapreduce.shuffle.class</name>
<value>org.apache.hadoop.mapred.ShuffleHandler</value>
</property>
 <property>
 <name>yarn.resourcemanager.resource-tracker.address</name>
 <value>master:8025</value>
 </property>
 <property>
 <name>yarn.resourcemanager.scheduler.address</name>
 <value>master:8030</value>
 </property>
 <property>
 <name>yarn.resourcemanager.address</name>
 <value>master:8050</value>
 </property>
<property>
<name>yarn.nodemanager.vmem-check-enabled</name>
<value>false</value>
</property>
```

15-Modify the mapred-site.xml file in the master node and the other nodes to look like this:



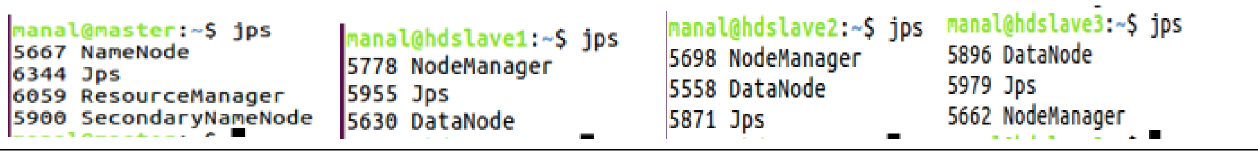
```
<property>
<name>mapreduce.framework.name</name>
<value>yarn</value>
</property>
<property>
<name>mapred.job.tracker</name>
<value>master:54311</value>
</property>
<property>
  <name>mapreduce.jobhistory.address</name>
  <value>master:10020</value>
  <description>Host and port for Job History Server (default 0.0.0.0:10020)</
description>
</property>
```

16-Modify the file master and put the name of the master node inside it, and modify the file salves and put the name of the other nodes inside it.
17-We create the following path in the distributed Hadoop filesystem on the master node:

```
$ sudo mkdir -p /usr/hadoop/hadoop_data/hdfs/namenode
```

18-Also on each data node create the following path /usr/hadoop/hadoop_data/hdfs/datanode
19-Run a hadoop cluster where we initialize the master node on first startup and then perform:

```
$ /usr/hadoop/bin/hadoop namenode –format
$ /usr/hadoop/sbin/start-all.sh
```



```
manal@master:~$ jps
5667 NameNode
6344 Jps
6059 ResourceManager
5900 SecondaryNameNode
```

```
manal@hdslave1:~$ jps
5778 NodeManager
5955 Jps
5630 DataNode
```

```
manal@hdslave2:~$ jps
5698 NodeManager
5558 DataNode
5871 Jps
```

```
manal@hdslave3:~$ jps
5896 DataNode
5979 Jps
5662 NodeManager
```

Figure (3-5) Hadoop proxies operating on all nodes.

5-3-2 Install HiBench on the master node and link it with Hadoop: [65]

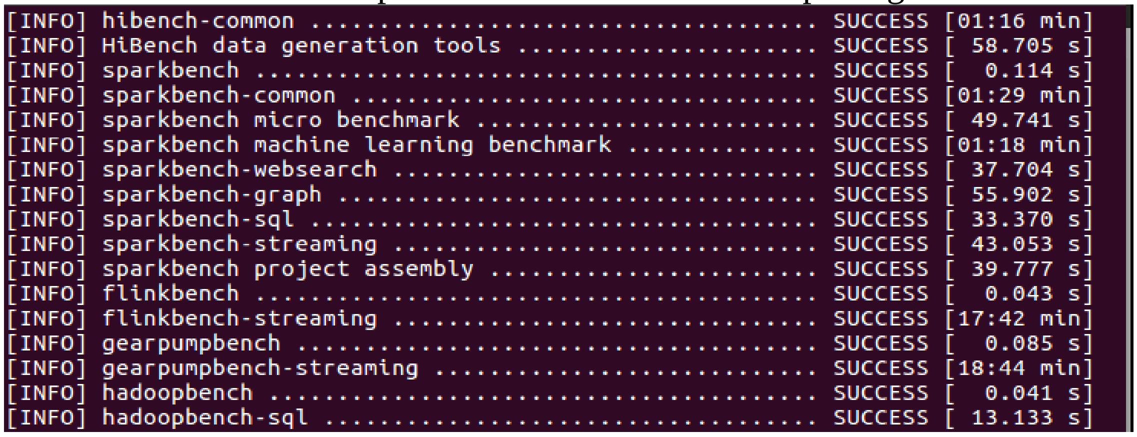
In the experiments, we will use the open source HiBench reference set, which is used to evaluate the performance of Hadoop and Spark big data frameworks and consists of workloads of various big data systems and a data generator that generates data of different sizes for those workloads. The workloads are comparatively abstract and can be easily configured to generate and run HiBench standards and is compatible with all major distributions of Hadoop and Spark frameworks such as: Apache, Cloudera and Hortonworks.

HiBench uses the Hadoop Ecosystem which includes stacks and frameworks for software such as MapReduce, Storm, Flink, Nutch and Hive as well as shapes most Spark workloads.
The metrics used in HiBench are: execution time per workload, throughput, and the amount of system resource usage. At the end of the workload run, all these metrics are appended to an output file for analysis. HiBench also provides a web-based output that shows system resource usage for each workload executed.
The current version of HiBench is 7.0 that contains 19 workloads categorized into six main areas: Micro, Machine Learning, SQL, Graph, Web Search, and Streaming.
HiBench for Micro Benchmarks uses a configurable data generator which mostly uses Hadoop's RandomTextWriter to write Binary Text directly to HDFS, For Web Search and Bayesian Classification HiBench uses the Wikipedia page-to-page link database and Wikipedia dump file respectively, For KMeans they developed a random data generator using the statistical distribution to generate the data. HiBench mostly creates unstructured data sets targeting the Hadoop file system.

1- Download HiBench from the following link and unzip it in your desired path on the master node:

https://github.com/intel-hadoop/HiBench | GitHub, Inc. (US)

2- HiBench build: $mvn -Dspark=2.1 -Dscala=2.11 clean package



```
[INFO] hibench-common ................................... SUCCESS [01:16 min]
[INFO] HiBench data generation tools .................... SUCCESS [ 58.705 s]
[INFO] sparkbench ....................................... SUCCESS [  0.114 s]
[INFO] sparkbench-common ................................ SUCCESS [01:29 min]
[INFO] sparkbench micro benchmark ....................... SUCCESS [ 49.741 s]
[INFO] sparkbench machine learning benchmark ............ SUCCESS [01:18 min]
[INFO] sparkbench-websearch ............................. SUCCESS [ 37.704 s]
[INFO] sparkbench-graph ................................. SUCCESS [ 55.902 s]
[INFO] sparkbench-sql ................................... SUCCESS [ 33.370 s]
[INFO] sparkbench-streaming ............................. SUCCESS [ 43.053 s]
[INFO] sparkbench project assembly ...................... SUCCESS [ 39.777 s]
[INFO] flinkbench ....................................... SUCCESS [  0.043 s]
[INFO] flinkbench-streaming ............................. SUCCESS [17:42 min]
[INFO] gearpumpbench .................................... SUCCESS [  0.085 s]
[INFO] gearpumpbench-streaming .......................... SUCCESS [18:44 min]
[INFO] hadoopbench ...................................... SUCCESS [  0.041 s]
[INFO] hadoopbench-sql .................................. SUCCESS [ 13.133 s]
```

Figure (5-4) HiBench construction.

- To build the Hadoop and Spark standards together:

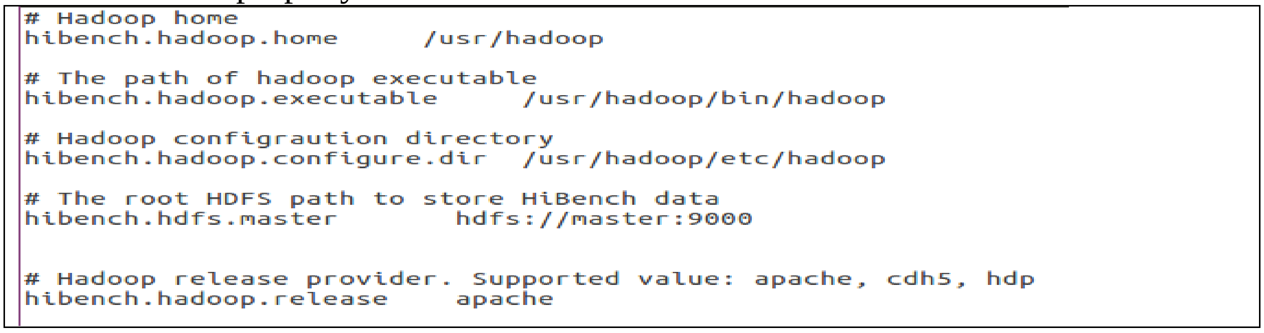
```
$ mvn -Phadoopbench -Psparkbench -Dspark=2.1 -Dscala=2.11 clean package
```

3- Determine the Scala version: To specify the version of Scala use -Dscala=xxx By default it is version (2.10) or (2.11):

```
$ mvn -Dscala=2.10 clean package
```

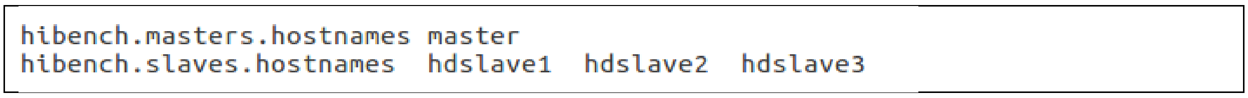
4- Connecting Hadoop with Hibench:

- Requirements: Python >=2.6, bc (for report generation), HDFS and Yarn start in the cluster.
- Create and modify the conf/hadoop.conf file and correctly add the following property values inside the file:



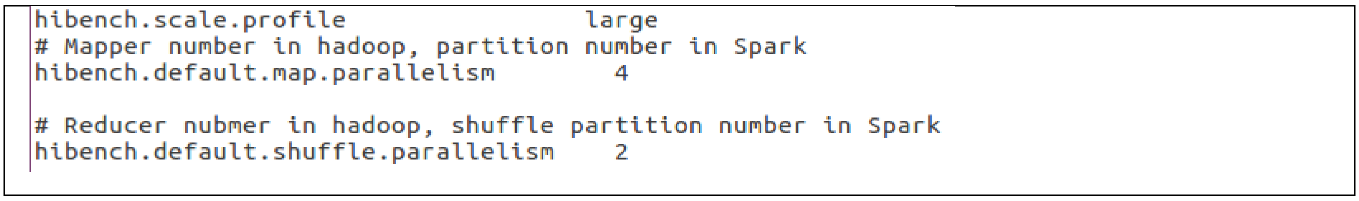
```
# Hadoop home
hibench.hadoop.home      /usr/hadoop

# The path of hadoop executable
hibench.hadoop.executable      /usr/hadoop/bin/hadoop

# Hadoop configraution directory
hibench.hadoop.configure.dir   /usr/hadoop/etc/hadoop

# The root HDFS path to store HiBench data
hibench.hdfs.master        hdfs://master:9000


# Hadoop release provider. Supported value: apache, cdh5, hdp
hibench.hadoop.release     apache
```

- Prepare the file conf/ hibench.conf and insert the following lines:



```
hibench.masters.hostnames master
hibench.slaves.hostnames  hdslave1  hdslave2  hdslave3
```

- The size of the input files can also be controlled by giving one of the values (tiny, small, large, huge, gigantic, big data) to the hibench.scale.profile property. It is also possible to control the number of abbreviated interviews in Hadoop to control the degree of branching as shown in the figure:



```
hibench.scale.profile                large
# Mapper number in hadoop, partition number in Spark
hibench.default.map.parallelism        4

# Reducer nubmer in hadoop, shuffle partition number in Spark
hibench.default.shuffle.parallelism    2
```

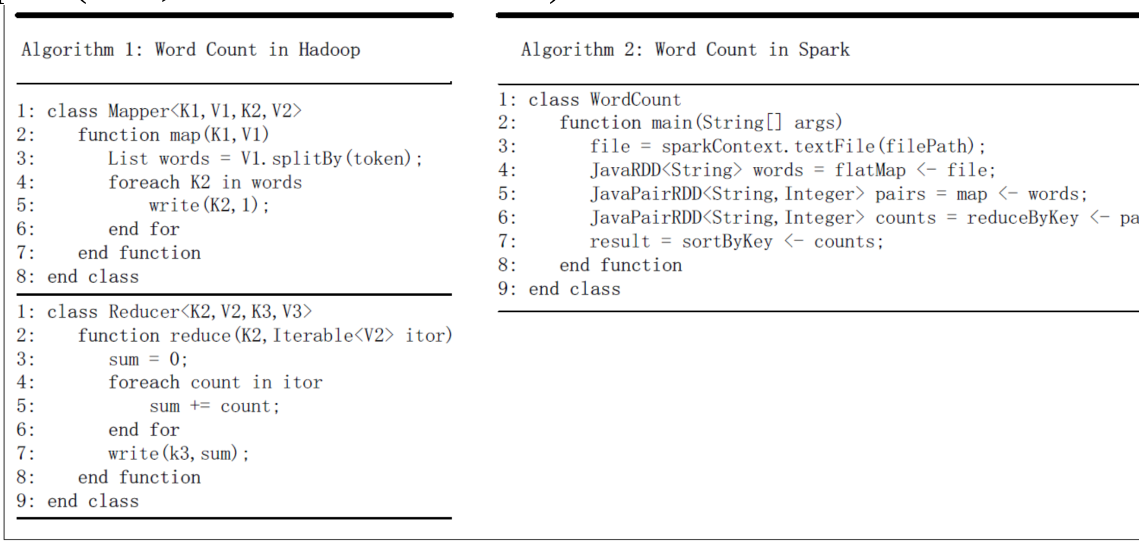
5-3-3 Choosing a set of load burdens:

Since we mainly focus on the performance comparison between Hadoop and Spark, a commonly used algorithm [1] will be used in both of them:

1. **WordCount:** It is considered to be the classic example supported by both Hadoop and Spark platforms that use MapReduce to do the job, this program counts the number of times each word appears in the input file generated by RandomTextWriter and this program depends on the intensive CPU that counts the number of words in the input file. [67].
   This program is implemented in Hadoop as follows:
   The corresponding function takes as its input one line from the input file and divides it into words and issues a pair of (key/value) for the word in the form of (word, 1), then the reduction function collects the number of times that each word appears and issues for each word a pair of (key/ value) as (word, sum) and the shorthand function puts the final result on [1] HDFS.

In Spark, an RDD is first created by loading data from HDFS using the textFile() method. Each line of the RDD consists of one line from the raw file, then it applies the flatMap() , map() and reduceByKey() transformations in order which are registered Its metadata is not executed until the saveAsText() procedure is called.

- **flatMap:** It takes an RDD made up of lines and converts it to an RDD made up of words.
- **Map:** RDD converts words into RDD (word, 1) pairs, also called RDD (key/value).
- **reduceByKey:** For each key (word) all values corresponding to it are reduced by adding all the values together, now we have an RDD of pairs (word, <number of occurrences>).



Algorithm 1: Word Count in Hadoop

```
class Mapper<K1,V1,K2,V2>
    function map(K1,V1)
        List words = V1.splitBy(token);
        foreach K2 in words
            write(K2,1);
        end for
    end function
end class
```

```
class Reducer<K2,V2,K3,V3>
    function reduce(K2,Iterable<V2> itor)
        sum = 0;
        foreach count in itor
            sum += count;
        end for
        write(k3,sum);
    end function
end class
```

Algorithm 2: Word Count in Spark

```
class WordCount
    function main(String[] args)
        file = sparkContext.textFile(filePath);
        JavaRDD<String> words = flatMap <- file;
        JavaPairRDD<String,Integer> pairs = map <- words;
        JavaPairRDD<String,Integer> counts = reduceByKey <- pa
        result = sortByKey <- counts;
    end function
end class
```

Figure (5-5) The WordCount algorithm in both Hadoop and Spark. [1]

5-3-4 Benchmarks used and supported by HiBench:

1. **Execution time:** It is the time between the beginning of the execution of the work (program) and its end, measured in seconds.
2. **Productivity:** It is a measure of the amount of data that the system can process during one time, and it is also the rate of input data during the execution time and is expressed in (bytes/second).
3. **Recovery time from failure:** It is the increase in the program execution time when a failure occurs, when choosing the superior platform in terms of fault tolerance will be based on the resulting increase in the execution time and not the total execution time after completing the work during a failure.

5-3-4-1 Research methodology used in carrying out the experiments:

1. In both systems, the settings were tested such as the number of interviews and redactors in Hadoop and the number of implementers in

Spark, and the experiments were carried out according to the settings for which both systems give the best possible performance.

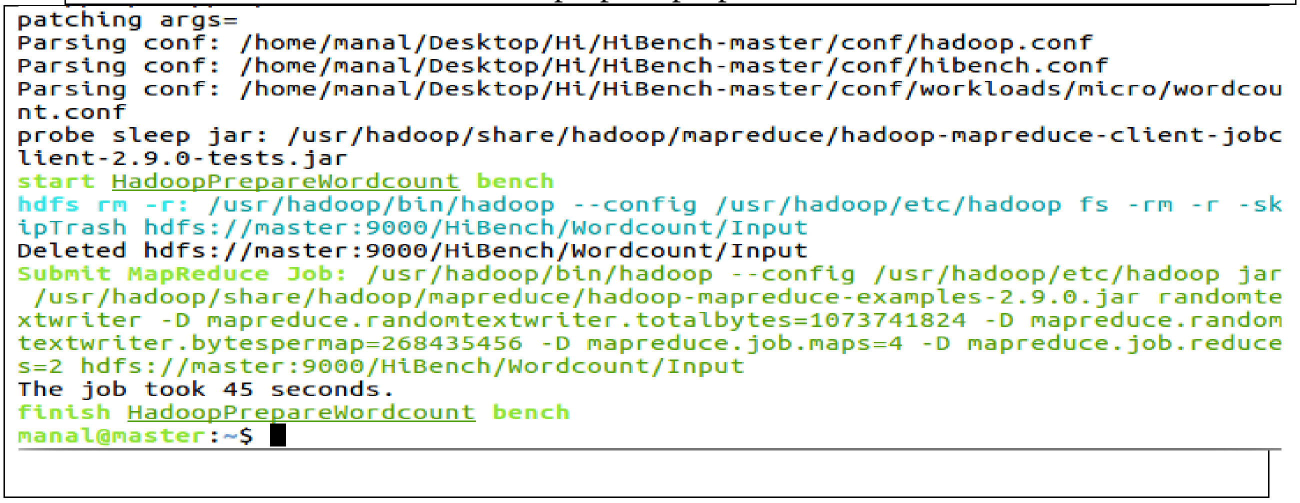
2. The program was selected according to the load it affects the system (WordCount is a processor consuming).
3. The cluster manager for Spark Yarn has been selected.
4. Each program will be executed five times and the average values of the parameters are taken to obtain the most accurate results.
5. The failure injection moment will be determined on the basis of a percentage of the average program execution time without failure (for example): if the execution time of a program is ten minutes in Hadoop and eight minutes in Spark and the failure injection moment is chosen after 25% of the time required to complete this program execution It means failure will be injected after 2.5 minutes into Hadoop and 2 minutes into Spark.
6. When choosing a platform that is superior in terms of fault tolerance, it will be based on the resulting increase in execution time and not the total execution time after the work is completed during a failure.

5-3-5 Running the Hadoop WordCount workload:

This is done in two stages:

1- Run the prepare.sh file that launches the hadoop task responsible for generating income data on HDFS:

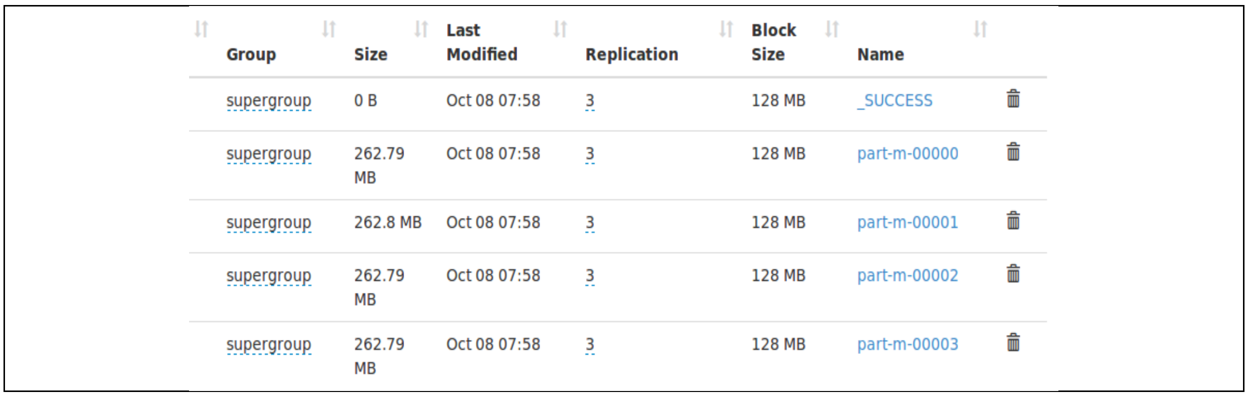
```
bin/workloads/micro/wordcount/prepare/prepare.sh  -2
```



```
patching args=
Parsing conf: /home/manal/Desktop/Hi/HiBench-master/conf/hadoop.conf
Parsing conf: /home/manal/Desktop/Hi/HiBench-master/conf/hibench.conf
Parsing conf: /home/manal/Desktop/Hi/HiBench-master/conf/workloads/micro/wordcou
nt.conf
probe sleep jar: /usr/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-client-jobc
lient-2.9.0-tests.jar
start HadoopPrepareWordcount bench
hdfs rm -r: /usr/hadoop/bin/hadoop --config /usr/hadoop/etc/hadoop fs -rm -r -sk
ipTrash hdfs://master:9000/HiBench/Wordcount/Input
Deleted hdfs://master:9000/HiBench/Wordcount/Input
Submit MapReduce Job: /usr/hadoop/bin/hadoop --config /usr/hadoop/etc/hadoop jar
 /usr/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.9.0.jar randomte
xtwriter -D mapreduce.randomtextwriter.totalbytes=1073741824 -D mapreduce.random
textwriter.bytespermap=268435456 -D mapreduce.job.maps=4 -D mapreduce.job.reduce
s=2 hdfs://master:9000/HiBench/Wordcount/Input
The job took 45 seconds.
finish HadoopPrepareWordcount bench
manal@master:~$ 
```

➢ The files generated from the previous instruction can be seen by placing the following link in the browser.

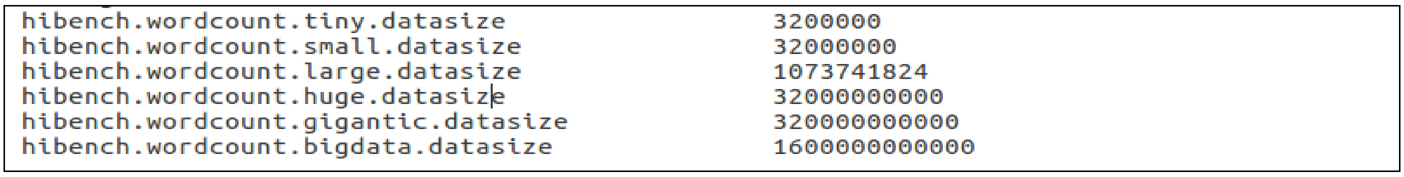
http://master:50070/explorer.html#/HiBench/Wordcount/Input



Group	Size	Last Modified	Replication	Block Size	Name
supergroup	0 B	Oct 08 07:58	3	128 MB	_SUCCESS
supergroup	262.79 MB	Oct 08 07:58	3	128 MB	part-m-00000
supergroup	262.8 MB	Oct 08 07:58	3	128 MB	part-m-00001
supergroup	262.79 MB	Oct 08 07:58	3	128 MB	part-m-00002
supergroup	262.79 MB	Oct 08 07:58	3	128 MB	part-m-00003

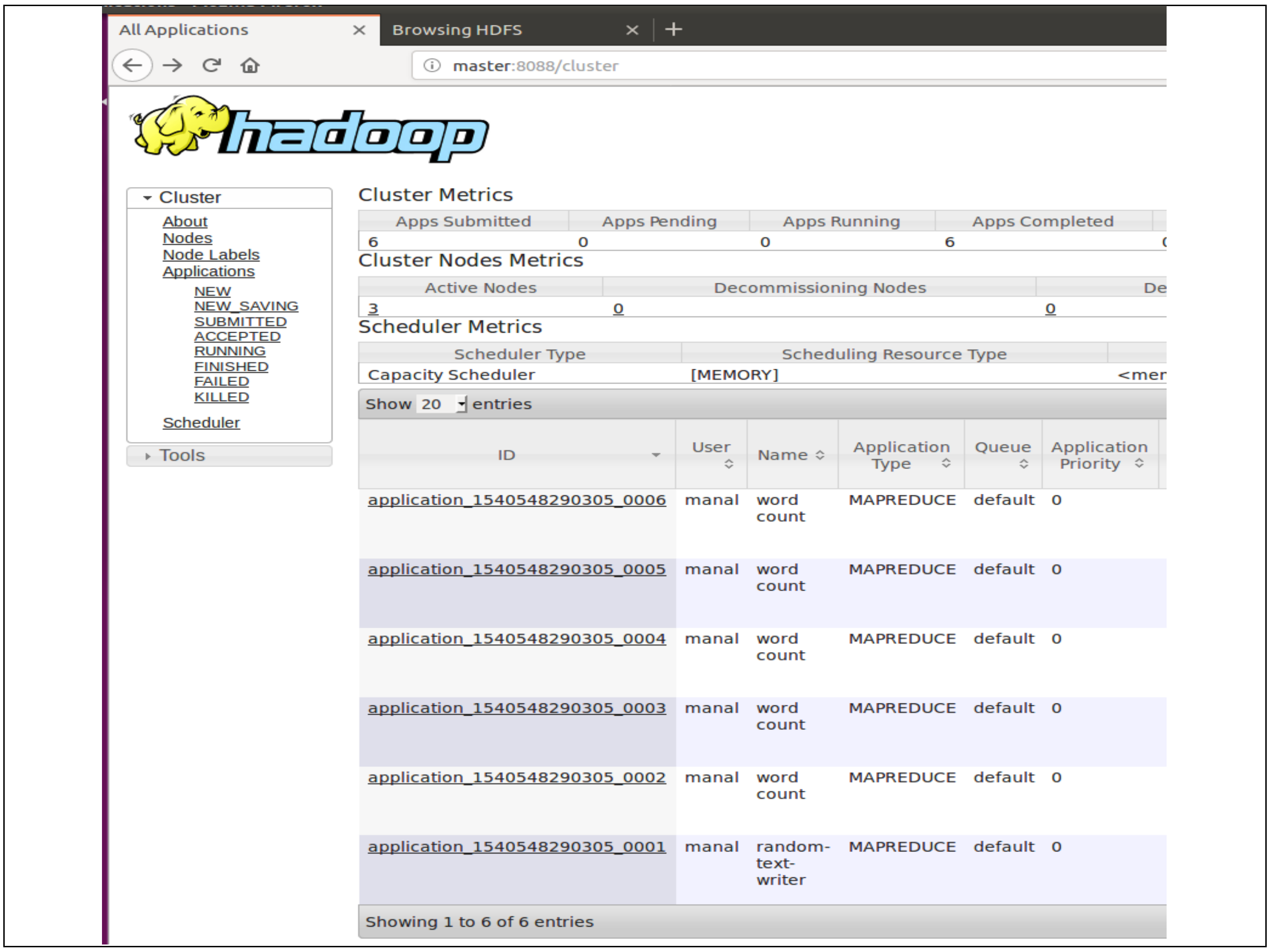
Figure (5-6) Income files for WordCount and Sort.

- or through the following command: /usr/hadoop/bin/hadoop dfs -ls /HiBench/Wordcount/Input
- Where the meaning of the input file sizes specified in the previous file is controlled as numbers included in the setup file for each workload separately. For example, for the wordcount program, the wordcount.conf setup file is modified in the following path: /HiBench-master/conf/workloads/micro/ wordcount .conf
- The folder in which the program's input files and its output files will be placed is also specified:



```
hibench.wordcount.tiny.datasize                 3200000
hibench.wordcount.small.datasize                32000000
hibench.wordcount.large.datasize                1073741824
hibench.wordcount.huge.datasize                 32000000000
hibench.wordcount.gigantic.datasize             320000000000
hibench.wordcount.bigdata.datasize              1600000000000
```

- Through the link master:8088/cluster, the webUI interface is displayed containing details of all the programs that are launched and implemented within the cluster:



Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed
6	0	0	6

Cluster Nodes Metrics

Active Nodes	Decommissioning Nodes	De
3	0	0

Scheduler Metrics

Scheduler Type	Scheduling Resource Type	
Capacity Scheduler	[MEMORY]	<mer

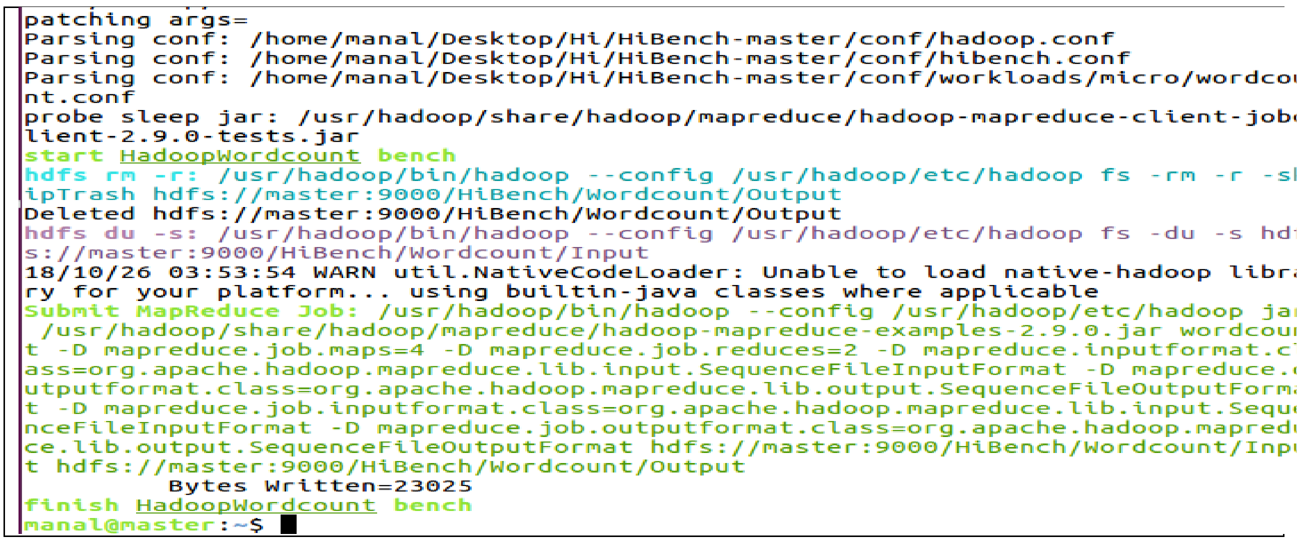
Show 20 entries

ID	User	Name	Application Type	Queue	Application Priority
application_1540548290305_0006	manal	word count	MAPREDUCE	default	0
application_1540548290305_0005	manal	word count	MAPREDUCE	default	0
application_1540548290305_0004	manal	word count	MAPREDUCE	default	0
application_1540548290305_0003	manal	word count	MAPREDUCE	default	0
application_1540548290305_0002	manal	word count	MAPREDUCE	default	0
application_1540548290305_0001	manal	random-text-writer	MAPREDUCE	default	0

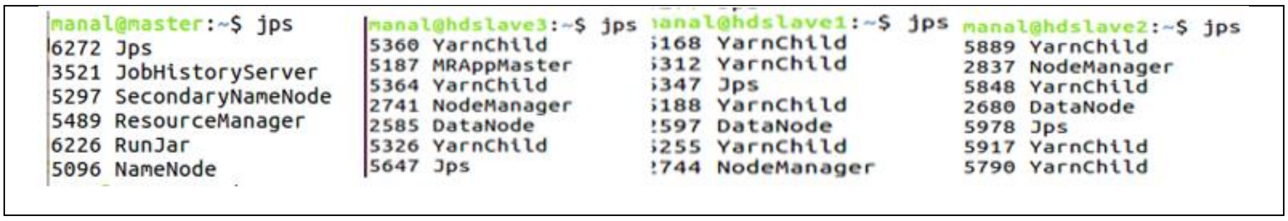
Showing 1 to 6 of 6 entries

Figure (5-7) the web interface of the Hadoop implementation of WordCount five times.

- Run run.sh to send the Hadoop program to the cluster and launch its execution and bin/run_all.sh can also be used to run all workloads listed in conf/benchmarks.lst and conf/frameworks.lst.



```
patching args=
Parsing conf: /home/manal/Desktop/Hi/HiBench-master/conf/hadoop.conf
Parsing conf: /home/manal/Desktop/Hi/HiBench-master/conf/hibench.conf
Parsing conf: /home/manal/Desktop/Hi/HiBench-master/conf/workloads/micro/wordcou
nt.conf
probe sleep jar: /usr/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-client-jobc
lient-2.9.0-tests.jar
start HadoopWordcount bench
hdfs rm -r: /usr/hadoop/bin/hadoop --config /usr/hadoop/etc/hadoop fs -rm -r -sl
ipTrash hdfs://master:9000/HiBench/Wordcount/Output
Deleted hdfs://master:9000/HiBench/Wordcount/Output
hdfs du -s: /usr/hadoop/bin/hadoop --config /usr/hadoop/etc/hadoop fs -du -s hd
s://master:9000/HiBench/Wordcount/Input
18/10/26 03:53:54 WARN util.NativeCodeLoader: Unable to load native-hadoop libr
ry for your platform... using builtin-java classes where applicable
Submit MapReduce Job: /usr/hadoop/bin/hadoop --config /usr/hadoop/etc/hadoop ja
 /usr/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.9.0.jar wordcou
t -D mapreduce.job.maps=4 -D mapreduce.job.reduces=2 -D mapreduce.inputformat.c
ass=org.apache.hadoop.mapreduce.lib.input.SequenceFileInputFormat -D mapreduce.
utputformat.class=org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputForm
t -D mapreduce.job.inputformat.class=org.apache.hadoop.mapreduce.lib.input.Sequ
nceFileInputFormat -D mapreduce.job.outputformat.class=org.apache.hadoop.mapred
ce.lib.output.SequenceFileOutputFormat hdfs://master:9000/HiBench/Wordcount/Inp
t hdfs://master:9000/HiBench/Wordcount/Output
         Bytes Written=23025
finish HadoopWordcount bench
manal@master:~$ 
```



```
manal@master:~$ jps
6272 Jps
3521 JobHistoryServer
5297 SecondaryNameNode
5489 ResourceManager
6226 RunJar
5096 NameNode
```

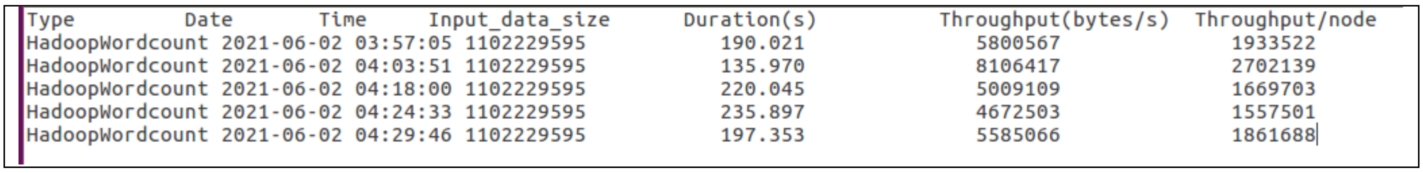
```
manal@hdslave3:~$ jps
5360 YarnChild
5187 MRAppMaster
5364 YarnChild
2741 NodeManager
2585 DataNode
5326 YarnChild
5647 Jps
```

```
anal@hdslave1:~$ jps
168 YarnChild
312 YarnChild
347 Jps
188 YarnChild
597 DataNode
255 YarnChild
744 NodeManager
```

```
manal@hdslave2:~$ jps
5889 YarnChild
2837 NodeManager
5848 YarnChild
2680 DataNode
5978 Jps
5917 YarnChild
5790 YarnChild
```

Figure (5-8) Hadoop proxies on all nodes during WordCount execution.

- ➢ In order to write down the correct results, we re-implement WordCount and all of the following programs in both Hadoop and Spark five times and then take the average of the measures based on the results of the five times they were executed.
- ➢ The metrics that will be adopted are the metrics provided by HiBench, which is stored in the file /report/Hibench.report/ which contains the following data: Program type – Launch date – Launch time – Income data size (bytes) – Execution time in seconds – Cluster-level throughput (bytes/second).



```
Type            Date        Time      Input_data_size       Duration(s)           Throughput(bytes/s)  Throughput/node
HadoopWordcount 2021-06-02 03:57:05 1102229595              190.021                5800567              1933522
HadoopWordcount 2021-06-02 04:03:51 1102229595              135.970                8106417              2702139
HadoopWordcount 2021-06-02 04:18:00 1102229595              220.045                5009109              1669703
HadoopWordcount 2021-06-02 04:24:33 1102229595              235.897                4672503              1557501
HadoopWordcount 2021-06-02 04:29:46 1102229595              197.353                5585066              1861688
```

Table (5-1) values of Hadoop's word count performance parameters without fail.

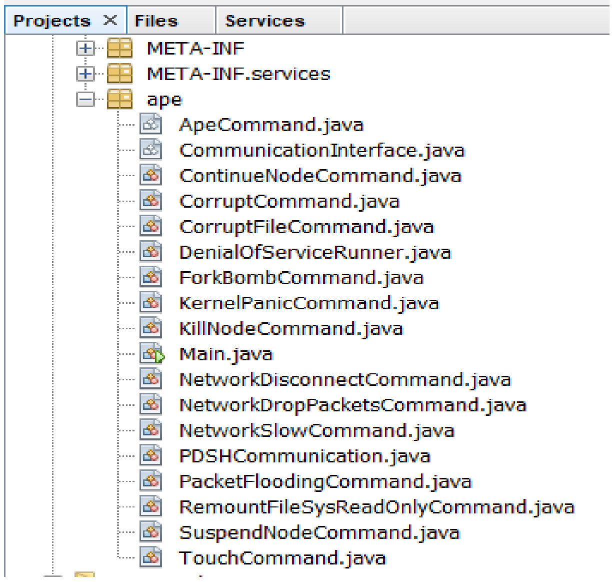
- ➢ Program name HadoopWordcount: Data size (bytes) = 1 gigabyte = 1073741824 bytes, this size is divided into equal parts to be entered into the interviews Number of Mapper = 4, Number of Reducer = 2.

1- **Average execution time** 190,021 + 135,970 + 220,045 + 235,897 + 197,353 = 980,186 / 5 = 196,0372 bytes/second (3 minutes and 16 seconds).
2- **The throughput in the cluster** = 5800567 + 8106417 + 5009109 + 4672503 + 5585066 = 24968409 / 5 = 4993681,8 bytes / sec (5.22 MB / sec).

5-3-6 Choosing a set of possible failures on both platforms:

The failure tolerance test aims to find faults in the implementation or specification of the failure tolerance mechanisms, for this purpose the system is implemented in a test environment with injection of known faults [6], The performance of the MapReduce function is usually measured by execution time, and can be significantly affected by many factors such as network or communication protocol, number of corresponding tasks, number of reduction tasks, type of staging data being shuffled, and volume of scrambled data [19].
We will use the tool AnarchyApe [70] which is an open source project created by Yahoo and developed for failure injection in Hadoop 1.0

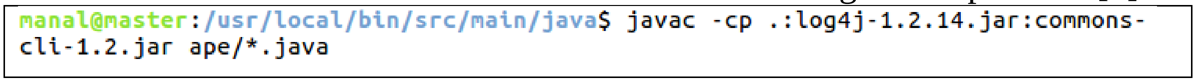
clusters in 2012 so we developed the tool for failure injection in Hadoop 2.0 YARN environment and Apache Spark, AnarchyApe supports 16 common failures in distributed systems every fail It is represented by its instruction or command and each failure is represented by a class in Java:



```
Projects × | Files | Services
  ⊞ META-INF
  ⊞ META-INF.services
  ⊟ ape
      ApeCommand.java
      CommunicationInterface.java
      ContinueNodeCommand.java
      CorruptCommand.java
      CorruptFileCommand.java
      DenialOfServiceRunner.java
      ForkBombCommand.java
      KernelPanicCommand.java
      KillNodeCommand.java
      Main.java
      NetworkDisconnectCommand.java
      NetworkDropPacketsCommand.java
      NetworkSlowCommand.java
      PDSHCommunication.java
      PacketFloodingCommand.java
      RemountFileSysReadOnlyCommand.java
      SuspendNodeCommand.java
      TouchCommand.java
```

Figure (5-9) of the failure types supported by the AnarchyApe tool.

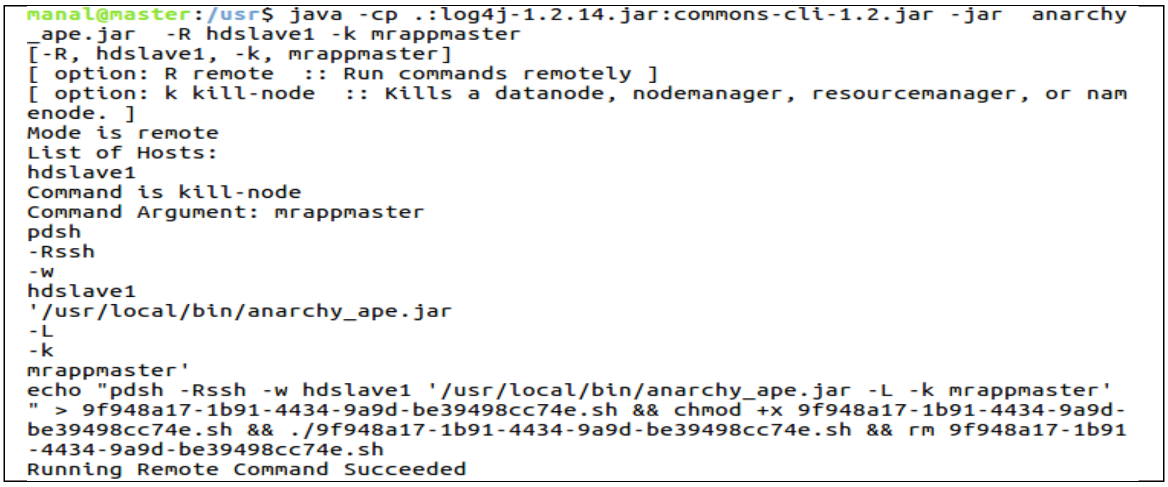
- **Compilation:** The tool is placed on the master node where it will compile and failover on the master node locally or on the rest of the nodes in the remote cluster using PDSH protocol:[7]



```
manal@master:/usr/local/bin/src/main/java$ javac -cp .:log4j-1.2.14.jar:commons-
cli-1.2.jar ape/*.java
```

- **Running Failures:** Each failure is executed by a command at the command prompt.

Example: anarchyape instruction to terminate the application manager running on node hdslave1:

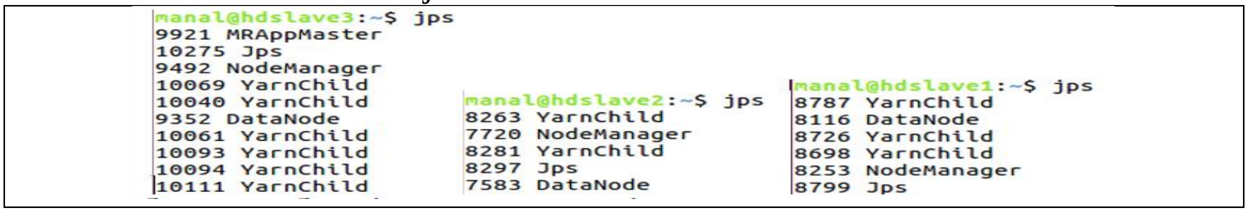


```
manal@master:/usr$ java -cp .:log4j-1.2.14.jar:commons-cli-1.2.jar -jar  anarchy
_ape.jar  -R hdslave1 -k mrappmaster
[-R, hdslave1, -k, mrappmaster]
[ option: R remote  :: Run commands remotely ]
[ option: k kill-node  :: Kills a datanode, nodemanager, resourcemanager, or nam
enode. ]
Mode is remote
List of Hosts:
hdslave1
Command is kill-node
Command Argument: mrappmaster
pdsh
-Rssh
-w
hdslave1
'/usr/local/bin/anarchy_ape.jar
-L
-k
mrappmaster'
echo "pdsh -Rssh -w hdslave1 '/usr/local/bin/anarchy_ape.jar -L -k mrappmaster'
" > 9f948a17-1b91-4434-9a9d-be39498cc74e.sh && chmod +x 9f948a17-1b91-4434-9a9d-
be39498cc74e.sh && ./9f948a17-1b91-4434-9a9d-be39498cc74e.sh && rm 9f948a17-1b91
-4434-9a9d-be39498cc74e.sh
Running Remote Command Succeeded
```

5-3-7 Implementation and evaluation of the performance of Hadoop at different loads with injection of failures in the components of the environment:

5-3-7-1 Running WordCount workload with Node Manager failing:

- We will not kill a random node manager in the cluster, but rather we will work to kill the node manager that the application manager resorted to and launched help containers on it, and study the ability of Hadoop to bypass the failure of the node manager, which will be injected permanently after about a quarter of the time required to complete the application without failure.
- Before injection failure:

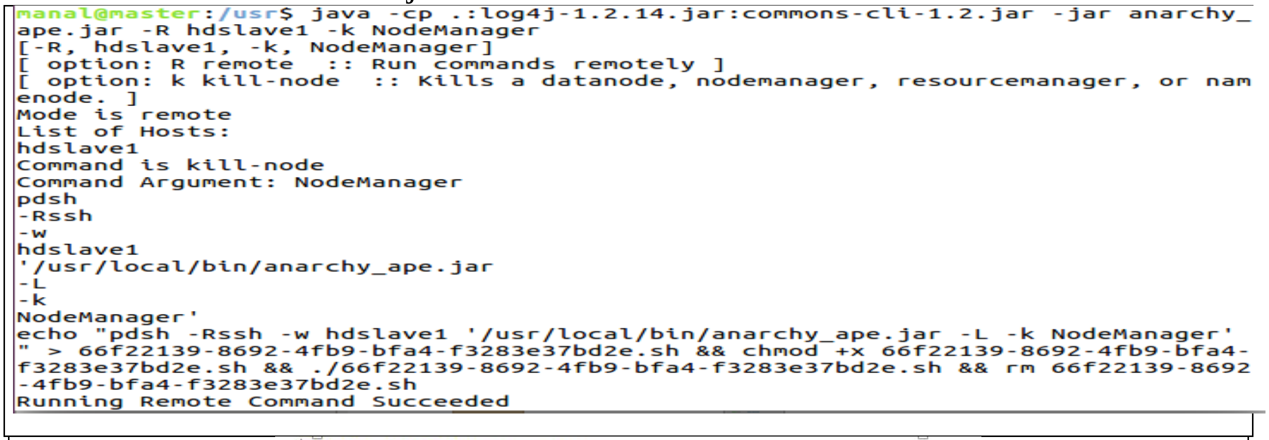


```
manal@hdslave3:~$ jps
9921 MRAppMaster
10275 Jps
9492 NodeManager
10069 YarnChild
10040 YarnChild
9352 DataNode
10061 YarnChild
10093 YarnChild
10094 YarnChild
10111 YarnChild
```

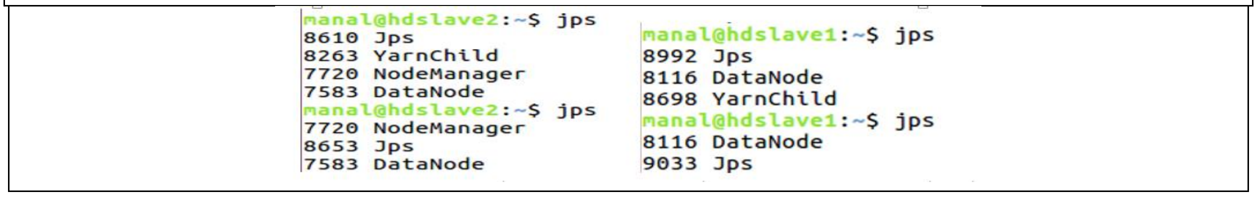
```
manal@hdslave2:~$ jps
8263 YarnChild
7720 NodeManager
8281 YarnChild
8297 Jps
7583 DataNode
```

```
manal@hdslave1:~$ jps
8787 YarnChild
8116 DataNode
8726 YarnChild
8698 YarnChild
8253 NodeManager
8799 Jps
```

- After injection failure:



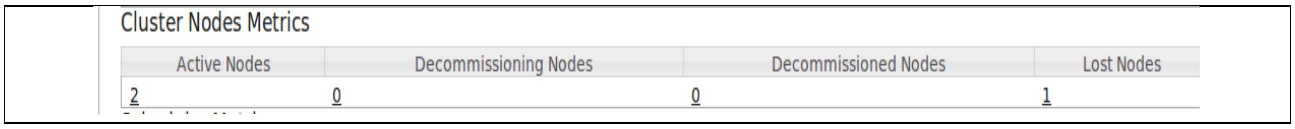
```
manal@master:/usr$ java -cp .:log4j-1.2.14.jar:commons-cli-1.2.jar -jar anarchy_
ape.jar -R hdslave1 -k NodeManager
[-R, hdslave1, -k, NodeManager]
[ option: R remote  :: Run commands remotely ]
[ option: k kill-node  :: Kills a datanode, nodemanager, resourcemanager, or nam
enode. ]
Mode is remote
List of Hosts:
hdslave1
Command is kill-node
Command Argument: NodeManager
pdsh
-Rssh
-w
hdslave1
'/usr/local/bin/anarchy_ape.jar
-L
-k
NodeManager'
echo "pdsh -Rssh -w hdslave1 '/usr/local/bin/anarchy_ape.jar -L -k NodeManager'
" > 66f22139-8692-4fb9-bfa4-f3283e37bd2e.sh && chmod +x 66f22139-8692-4fb9-bfa4-
f3283e37bd2e.sh && ./66f22139-8692-4fb9-bfa4-f3283e37bd2e.sh && rm 66f22139-8692
-4fb9-bfa4-f3283e37bd2e.sh
Running Remote Command Succeeded
```



```
manal@hdslave2:~$ jps
8610 Jps
8263 YarnChild
7720 NodeManager
7583 DataNode
manal@hdslave2:~$ jps
7720 NodeManager
8653 Jps
7583 DataNode
```

```
manal@hdslave1:~$ jps
8992 Jps
8116 DataNode
8698 YarnChild
manal@hdslave1:~$ jps
8116 DataNode
9033 Jps
```

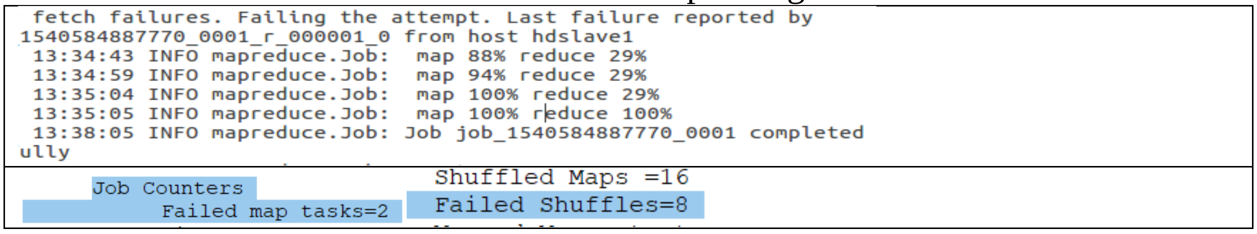
- In the previous figure, the application manager was launched on the hdslave3 node, which started containers on the nodes hdslave2 and hdslave1 whose node manager was killed after a quarter of the time required to finish the work in the absence of failure. If the node manager fails it will stop sending the heartbeat to the resource manager and the resource manager will notice that the node manager has stopped sending heartbeats and delete it from the list of nodes on which containers will be scheduled.



Cluster Nodes Metrics

Active Nodes	Decommissioning Nodes	Decommissioned Nodes	Lost Nodes
2	0	0	1

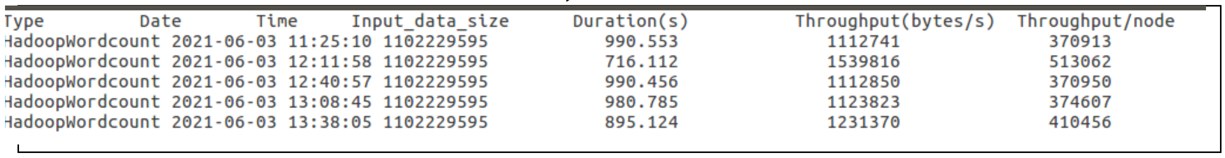
- In addition the application manager restarts the corresponding tasks that have been run and successfully completed on the failed node manager on another node manager which is here hdslave1 to be restarted if it belongs to uncompleted works because their staging output on the local file system of the failed node manager may not It can be accessed by reduction tasks.
- After discovering the failure, we notice that despite the completion of the interview tasks on the node whose manager was killed, Hadoop could not bring its interim results to the reduction tasks, which leads to a failure in the mixing stage, so the application manager re-booked containers on other nodes and re-executed the corresponding tasks on the new node.



```
 fetch failures. Failing the attempt. Last failure reported by
1540584887770_0001_r_000001_0 from host hdslave1
 13:34:43 INFO mapreduce.Job:  map 88% reduce 29%
 13:34:59 INFO mapreduce.Job:  map 94% reduce 29%
 13:35:04 INFO mapreduce.Job:  map 100% reduce 29%
 13:35:05 INFO mapreduce.Job:  map 100% reduce 100%
 13:38:05 INFO mapreduce.Job: Job job_1540584887770_0001 completed
ully
```

```
    Job Counters              Shuffled Maps =16
         Failed map tasks=2   Failed Shuffles=8
```

- The application was implemented five times to obtain more accurate results, and the results were as follows:



```
Type            Date        Time       Input_data_size     Duration(s)          Throughput(bytes/s)  Throughput/node
HadoopWordcount 2021-06-03 11:25:10 1102229595              990.553               1112741              370913
HadoopWordcount 2021-06-03 12:11:58 1102229595              716.112               1539816              513062
HadoopWordcount 2021-06-03 12:40:57 1102229595              990.456               1112850              370950
HadoopWordcount 2021-06-03 13:08:45 1102229595              980.785               1123823              374607
HadoopWordcount 2021-06-03 13:38:05 1102229595              895.124               1231370              410456
```

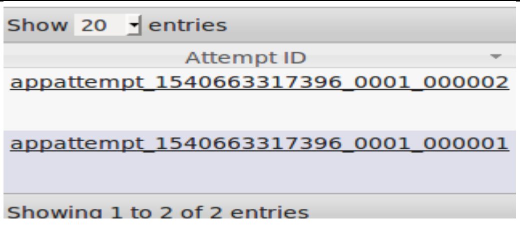
Table (5-2) Performance parameter values for Hadoop WordCount with node manager failure.

1- **Average execution time =** 990,553 + 761,112 + 990,456 + 980,785 + 895,124 = 4573,03/5 = 914,606 seconds (15 minutes and 14 seconds).
2- **The throughput in the cluster =** 1112741+1539816+1112850+1123823+1231370 = 6120600/5 = 1224120 bytes/sec (1.16MB/s).
3- **Failover Time: The time interval between the moment the failure is injected and the failure begins to pass, which is equal to the resulting increment in the execution time from the absence of failure:** 914,606 – 196,0372 = 718,5688 (11d and 58s).

5-3-7-2 Running WordCount workload with MRAppMaster application manager failing:

- The application manager sends periodic pulses to the resource manager and in the event of a failure in the application manager, the resource manager will detect the failure and start a new instance of the application manager under a new container (managed by the node manager).

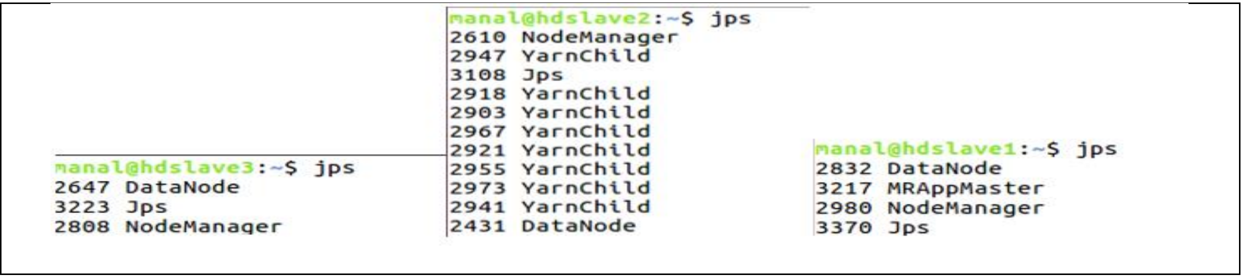
➢ In the practical implementation of the experiment, we notice that when the failure is injected, it is immediately detected, unlike the case of the failure of the node manager, which takes a somewhat longer time, where the resource manager chooses another data node to be called the application manager again:



```
Show 20 entries
Attempt ID
appattempt_1540663317396_0001_000002
appattempt_1540663317396_0001_000001
Showing 1 to 2 of 2 entries
```

➢ Upon execution, we note that if the program has completed a set of reduction tasks when the failure is injected, the corresponding tasks are not re-executed, if he has not started with the reduction tasks, then all corresponding tasks will be re-executed until the completed ones.

➢ Before injecting the failure the application manager was launched on node hdslave1:

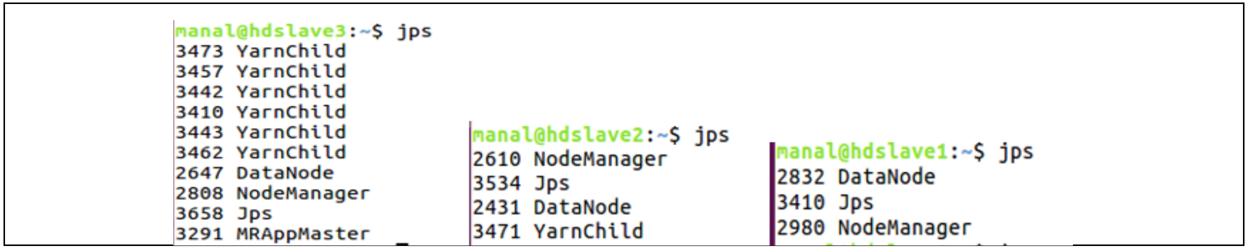


```
manal@hdslave3:~$ jps
2647 DataNode
3223 Jps
2808 NodeManager
```

```
manal@hdslave2:~$ jps
2610 NodeManager
2947 YarnChild
3108 Jps
2918 YarnChild
2903 YarnChild
2967 YarnChild
2921 YarnChild
2955 YarnChild
2973 YarnChild
2941 YarnChild
2431 DataNode
```

```
manal@hdslave1:~$ jps
2832 DataNode
3217 MRAppMaster
2980 NodeManager
3370 Jps
```

➢ After the failure injection was attempted to restart the application manager on node hdslave3:

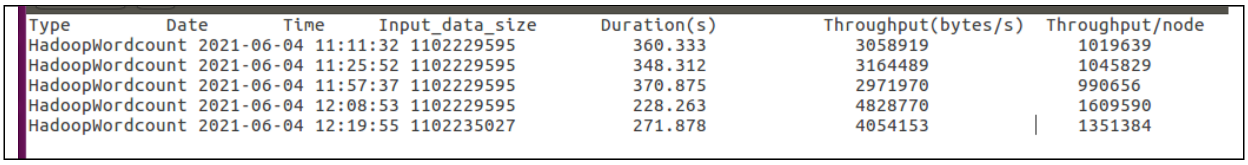


```
manal@hdslave3:~$ jps
3473 YarnChild
3457 YarnChild
3442 YarnChild
3410 YarnChild
3443 YarnChild
3462 YarnChild
2647 DataNode
2808 NodeManager
3658 Jps
3291 MRAppMaster
```

```
manal@hdslave2:~$ jps
2610 NodeManager
3534 Jps
2431 DataNode
3471 YarnChild
```

```
manal@hdslave1:~$ jps
2832 DataNode
3410 Jps
2980 NodeManager
```

➢ Experiment results:



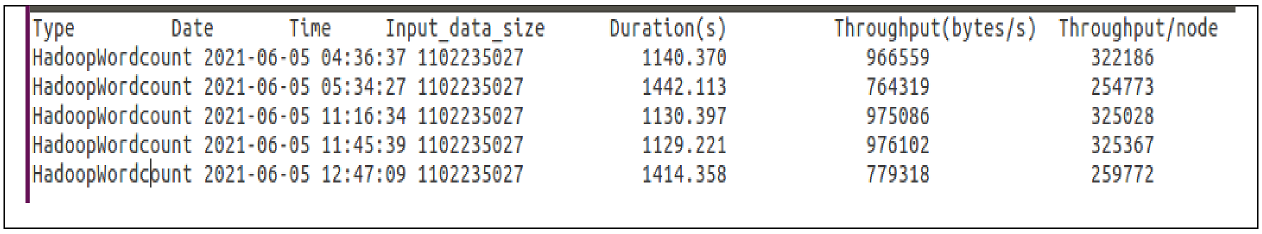
```
Type            Date        Time      Input_data_size       Duration(s)          Throughput(bytes/s)  Throughput/node
HadoopWordcount 2021-06-04 11:11:32 1102229595             360.333               3058919              1019639
HadoopWordcount 2021-06-04 11:25:52 1102229595             348.312               3164489              1045829
HadoopWordcount 2021-06-04 11:57:37 1102229595             370.875               2971970              990656
HadoopWordcount 2021-06-04 12:08:53 1102229595             228.263               4828770              1609590
HadoopWordcount 2021-06-04 12:19:55 1102235027             271.878               4054153              1351384
```

Table 3-5 performance parameter values for Hadoop wordCount with application manager failure.

1- **Average execution time =** 360,333 + 348,312 + 370,875 + 228,263 + 271,878 = 1579,661/5 = 315.9322 seconds (5 minutes and 15 seconds).
2- **Cluster throughput =** 3058919 + 3164489 + 2971970 + 4828770 + 4054153 = 18078301 /5 = 3615660.2 bytes/sec (3.47 MB/s).
3- **Failover Time:** The time interval between the moment the failure is injected and the failure begins to pass, which is equal to the resulting increment in the execution time from the absence of failures = 315.9322-196.0372 = 119,895 (1 min and 58 sec).

5-3-7-3 Running WordCount Workload with Node Crash Failure:

- We simulated Hadoop's performance when a data node is completely crashed by shutting down the machine on which the node is running after a quarter of the time it takes to complete the work without failure, Since there are two computation nodes and the number of containers reserved on each of them varies, and since the selection of the node is random, the execution time varies and increases as the number of containers reserved on the crashed node is greater.
- A node crash failure is different from a node manager or data node failure on the computation node. The difference between a full crash and crashing agents such as the node manager, data node, or application manager is that the operating system sends a TCP reset (RST) packet only when the agent is killed, which is an early indication of a failure.
- Experiment results:



```
Type           Date       Time     Input_data_size     Duration(s)         Throughput(bytes/s)  Throughput/node
HadoopWordcount 2021-06-05 04:36:37 1102235027          1140.370            966559               322186
HadoopWordcount 2021-06-05 05:34:27 1102235027          1442.113            764319               254773
HadoopWordcount 2021-06-05 11:16:34 1102235027          1130.397            975086               325028
HadoopWordcount 2021-06-05 11:45:39 1102235027          1129.221            976102               325367
HadoopWordcount 2021-06-05 12:47:09 1102235027          1414.358            779318               259772
```

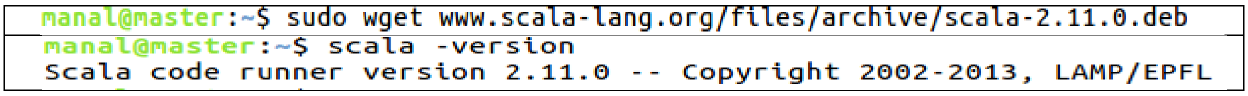
Table (5-4) values of the Hadoop wordcount performance parameter with crash failure.

1- **Average execution time =** 1140,370 + 1442,113 +1130,397 +1129,221 + 1414,358 = 6256,459/5 = 1251,2918 seconds (20 minutes 51 seconds).
2- **The throughput in the cluster =** 966559 + 764319 + 975086 + 976102 + 779318 = 4461384/5 = 892276,8 bytes / sec (0.846 MB / sec).
3- **Failover Time:** The time interval between the moment the failure is injected and the failure begins to pass, which is equal to the resulting increment in the execution time for the

absence of failure = 1251,2918 -196,0372 = 1055,2546 (17 min and 35 sec).

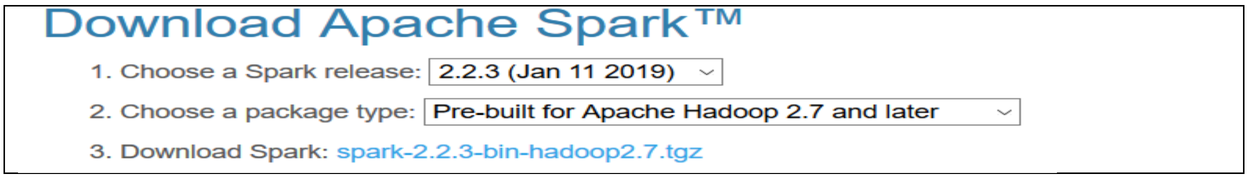
5-3-8 Install Spark and connect it with HiBench:

1- Install scala on all the nodes:



```
manal@master:~$ sudo wget www.scala-lang.org/files/archive/scala-2.11.0.deb
manal@master:~$ scala -version
Scala code runner version 2.11.0 -- Copyright 2002-2013, LAMP/EPFL
```

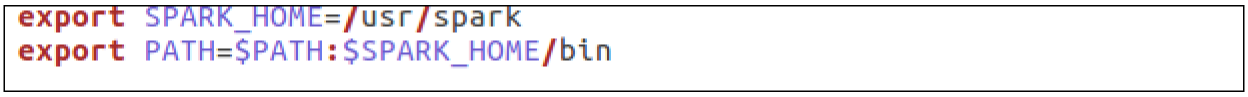
2- Download Spark on all nodes and install it at /usr/spark:



Download Apache Spark™

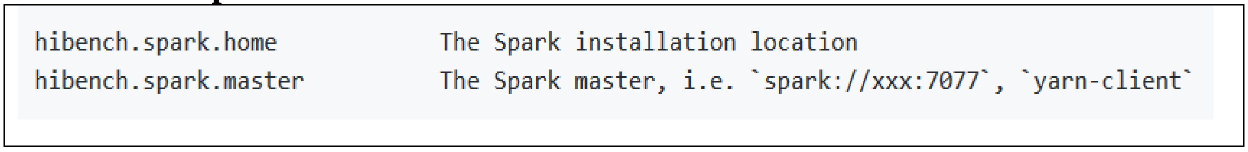
1. Choose a Spark release: 2.2.3 (Jan 11 2019)
2. Choose a package type: Pre-built for Apache Hadoop 2.7 and later
3. Download Spark: spark-2.2.3-bin-hadoop2.7.tgz

3- Add the spark path to the bash file:



```
export SPARK_HOME=/usr/spark
export PATH=$PATH:$SPARK_HOME/bin
```

4- Connect spark with HiBench:



```
hibench.spark.home          The Spark installation location
hibench.spark.master        The Spark master, i.e. `spark://xxx:7077`, `yarn-client`
```

5-3-9 Implementation and evaluation of the performance of Spark at various loads (without failure):

➢ **To configure the Spark cluster, there are two perspectives for determining the number of ports, the number of cores, and the memory allocated to each port [76]:**

1- First case (one port per core on each node):

- Number of executors = 3 * 2 = 6.
- executor-cores = 1 per port.
- executor-memory = node memory / number of executors on it = 3 /2 = 1.5 gigabytes.

In this case we won't be able to benefit from running multiple tasks in the same JVM either, the shared variables will be duplicated in each kernel and we don't leave enough memory space for Hadoop / Yarn daemon processes.

2- Second case (one port on each node):

- Number of executors = 3.

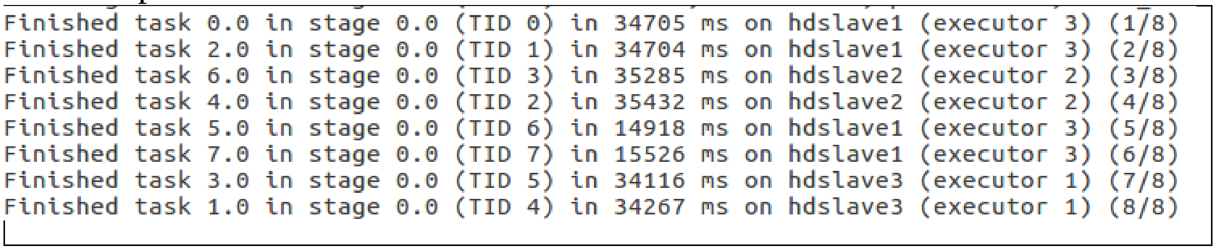
- Executor-cores = 2.
- executor-memory = node memory / number of executors on it = 3 GB.

In this case we did not leave enough memory space for Hadoop / Yarn daemon processes as it was noted that the system consumes a large load to write on HDFS with 5 tasks per port, so it is good to keep the number of cores per port below this number and this is true in the case of our cluster.

➢ **As in Hadoop, the two cases were tested and we chose in the experiments the second case that gave the shortest possible execution time. The experiments were also done by relying on the Hadoop Cluster Manager Yarn and the Client Mode.**

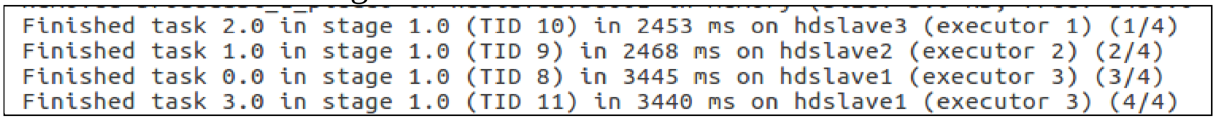
5-3-9-1 Spark runs WordCount workload without fail:

➢ The WordCount program is implemented in the form of two stages. The corresponding stage contains 8 corresponding tasks distributed to the three implementers:



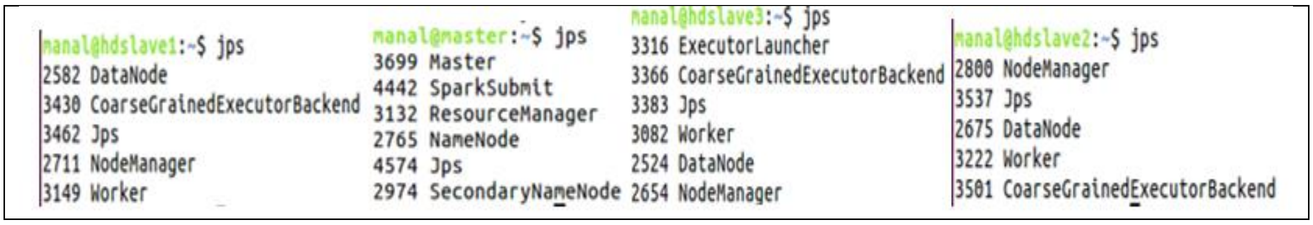
```
Finished task 0.0 in stage 0.0 (TID 0) in 34705 ms on hdslave1 (executor 3) (1/8)
Finished task 2.0 in stage 0.0 (TID 1) in 34704 ms on hdslave1 (executor 3) (2/8)
Finished task 6.0 in stage 0.0 (TID 3) in 35285 ms on hdslave2 (executor 2) (3/8)
Finished task 4.0 in stage 0.0 (TID 2) in 35432 ms on hdslave2 (executor 2) (4/8)
Finished task 5.0 in stage 0.0 (TID 6) in 14918 ms on hdslave1 (executor 3) (5/8)
Finished task 7.0 in stage 0.0 (TID 7) in 15526 ms on hdslave1 (executor 3) (6/8)
Finished task 3.0 in stage 0.0 (TID 5) in 34116 ms on hdslave3 (executor 1) (7/8)
Finished task 1.0 in stage 0.0 (TID 4) in 34267 ms on hdslave3 (executor 1) (8/8)
```

➢ The reduction stage contains four reduction tasks:



```
Finished task 2.0 in stage 1.0 (TID 10) in 2453 ms on hdslave3 (executor 1) (1/4)
Finished task 1.0 in stage 1.0 (TID 9) in 2468 ms on hdslave2 (executor 2) (2/4)
Finished task 0.0 in stage 1.0 (TID 8) in 3445 ms on hdslave1 (executor 3) (3/4)
Finished task 3.0 in stage 1.0 (TID 11) in 3440 ms on hdslave1 (executor 3) (4/4)
```

➢ Program Name (ScalaSparkWordcount), Data Size (bytes) = 1GB (byte) = 1073741824 bytes, this size is divided into four equal parts, Number of Ports = 3, Deployment Mode: Client Mode.



```
manal@hdslave1:~$ jps
2582 DataNode
3430 CoarseGrainedExecutorBackend
3462 Jps
2711 NodeManager
3149 Worker
```

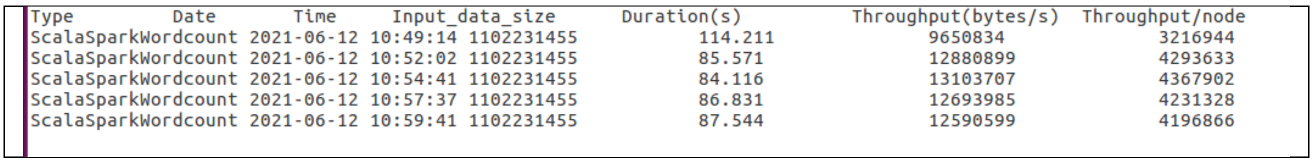
```
manal@master:~$ jps
3699 Master
4442 SparkSubmit
3132 ResourceManager
2765 NameNode
4574 Jps
2974 SecondaryNameNode
```

```
manal@hdslave3:~$ jps
3316 ExecutorLauncher
3366 CoarseGrainedExecutorBackend
3383 Jps
3082 Worker
2524 DataNode
2654 NodeManager
```

```
manal@hdslave2:~$ jps
2800 NodeManager
3537 Jps
2675 DataNode
3222 Worker
3501 CoarseGrainedExecutorBackend
```

Figure (5-10) Spark Agents on Nodes after Wordcount is run.

➢ Experiment results:



```
Type                Date       Time     Input_data_size      Duration(s)         Throughput(bytes/s)  Throughput/node
ScalaSparkWordcount 2021-06-12 10:49:14 1102231455                114.211              9650834              3216944
ScalaSparkWordcount 2021-06-12 10:52:02 1102231455                85.571               12880899             4293633
ScalaSparkWordcount 2021-06-12 10:54:41 1102231455                84.116               13103707             4367902
ScalaSparkWordcount 2021-06-12 10:57:37 1102231455                86.831               12693985             4231328
ScalaSparkWordcount 2021-06-12 10:59:41 1102231455                87.544               12590599             4196866
```

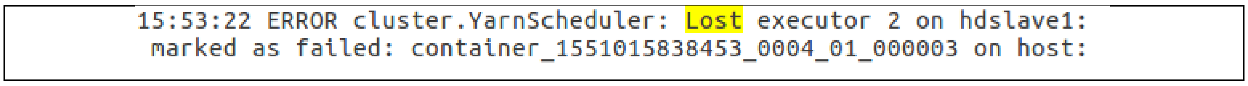
Table (5-5) values of performance parameters of Spark WordCount program without fail.

1- **Average execution time** = 114,211 +85,571 + 84,116 + 86,831 + 87,544 = 458,273/5 = 91,6546 seconds (1 minute and 32 seconds).
2- **Throughput in the cluster =** 9650834 + 12880899 + 13103707 + 12693985 + 12590599 = 60920024/5 = 12184004,8 bytes/sec (11.75MB/s).

5-3-10 Implementation and evaluation of the performance of Spark at different load loads (with failed injection):

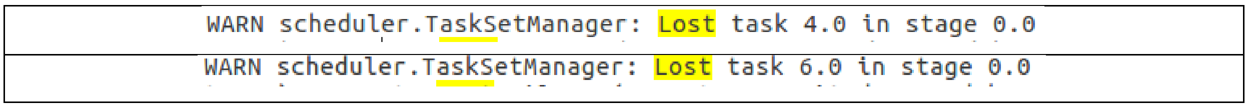
5-3-10-1 Running WordCount with Coarse Grained Executor Backend Failed Injection:

➢ A random executor is killed after a quarter of the time required to execute the program without failure:



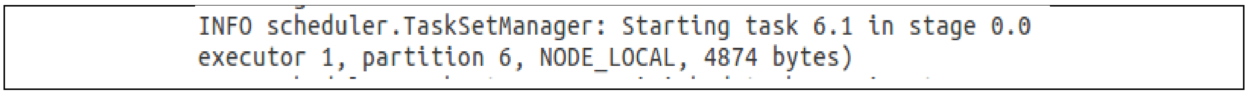
```
15:53:22 ERROR cluster.YarnScheduler: Lost executor 2 on hdslave1:
 marked as failed: container_1551015838453_0004_01_000003 on host:
```

➢ Resulting in the failure of the scheduled tasks to execute on it:



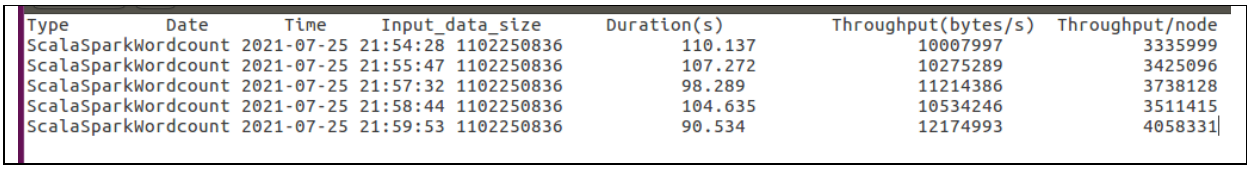
```
WARN scheduler.TaskSetManager: Lost task 4.0 in stage 0.0
WARN scheduler.TaskSetManager: Lost task 6.0 in stage 0.0
```

➢ This port is removed and its tasks are scheduled on an existing port or on another new port called the same node or on another node if the need arises:



```
INFO scheduler.TaskSetManager: Starting task 6.1 in stage 0.0
executor 1, partition 6, NODE_LOCAL, 4874 bytes)
```

➢ After recording the results, we note Spark's ability to detect and overcome failures in the two ports without affecting performance:



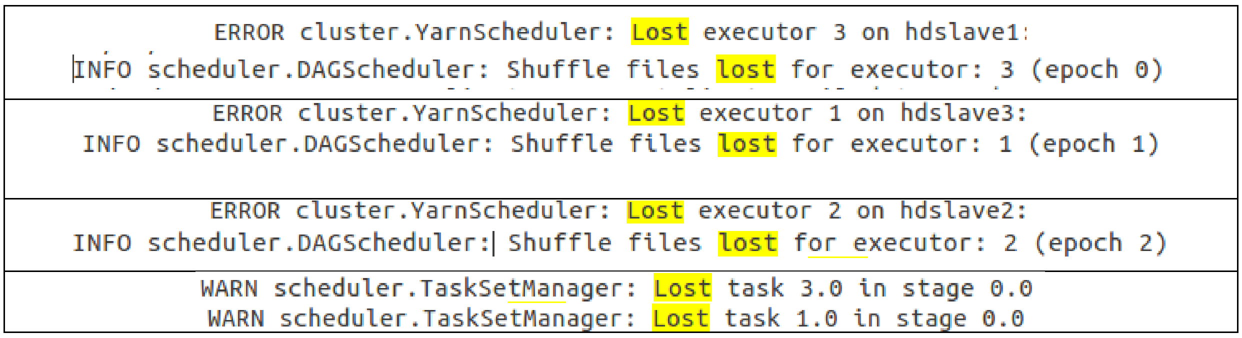
```
Type               Date       Time     Input_data_size      Duration(s)         Throughput(bytes/s)  Throughput/node
ScalaSparkWordcount 2021-07-25 21:54:28 1102250836              110.137                 10007997          3335999
ScalaSparkWordcount 2021-07-25 21:55:47 1102250836              107.272                 10275289          3425096
ScalaSparkWordcount 2021-07-25 21:57:32 1102250836              98.289                  11214386          3738128
ScalaSparkWordcount 2021-07-25 21:58:44 1102250836              104.635                 10534246          3511415
ScalaSparkWordcount 2021-07-25 21:59:53 1102250836              90.534                  12174993          4058331
```

Table (5-6) values of performance parameters of Spark WordCount program with port failure.

1- **Average execution time =** 110,137 + 107, 272 + 98,289 + 104,635 + 90,534 = 510,867 /5 = 102,1734 seconds (1 minute and 54 seconds).
2- **Productivity in cluster =** 10007997 + 10275289 + 11214386 + 12174993 + 10534246 =54206911/5=10841382.2 bytes/sec (10.40MB/sec).
3- **Failover time:** 102,1734-91,6546=10.5sec.

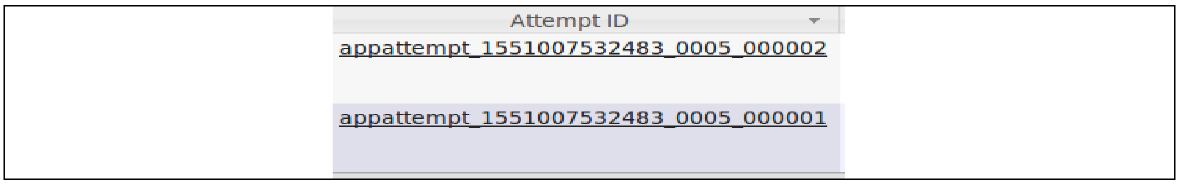
5-3-10-2 Running WordCount with Executor Launcher injection failed:

➢ The launcher is killed after a quarter of the time required to execute the program without failure. We note that killing the launcher leads to the failure of all the implementers it launched and all the tasks scheduled for them:



```
ERROR cluster.YarnScheduler: Lost executor 3 on hdslave1:
INFO scheduler.DAGScheduler: Shuffle files lost for executor: 3 (epoch 0)
ERROR cluster.YarnScheduler: Lost executor 1 on hdslave3:
INFO scheduler.DAGScheduler: Shuffle files lost for executor: 1 (epoch 1)
ERROR cluster.YarnScheduler: Lost executor 2 on hdslave2:
INFO scheduler.DAGScheduler: Shuffle files lost for executor: 2 (epoch 2)
WARN scheduler.TaskSetManager: Lost task 3.0 in stage 0.0
WARN scheduler.TaskSetManager: Lost task 1.0 in stage 0.0
```

- ➢ We note that the launcher in Spark corresponds to the application manager in Hadoop, where the failure is immediately detected and a new instance of it is started on another node:

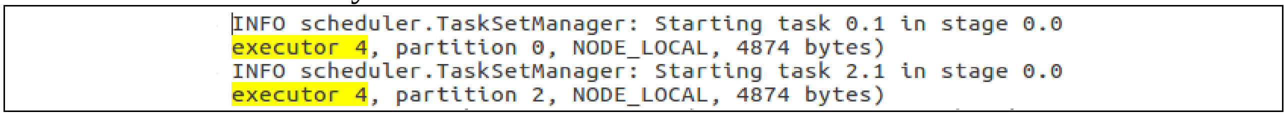


Attempt ID

appattempt_1551007532483_0005_000002

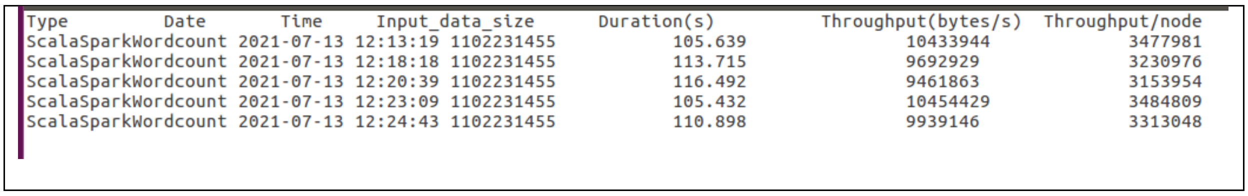
appattempt_1551007532483_0005_000001

- ➢ A new launcher is re-launched on a new node, which in turn re-launches two new executors (Executor4) and schedules the implementation of the unfinished stages with the tasks they contain anew:



```
INFO scheduler.TaskSetManager: Starting task 0.1 in stage 0.0
executor 4, partition 0, NODE_LOCAL, 4874 bytes)
INFO scheduler.TaskSetManager: Starting task 2.1 in stage 0.0
executor 4, partition 2, NODE_LOCAL, 4874 bytes)
```

- ➢ Experiment results:



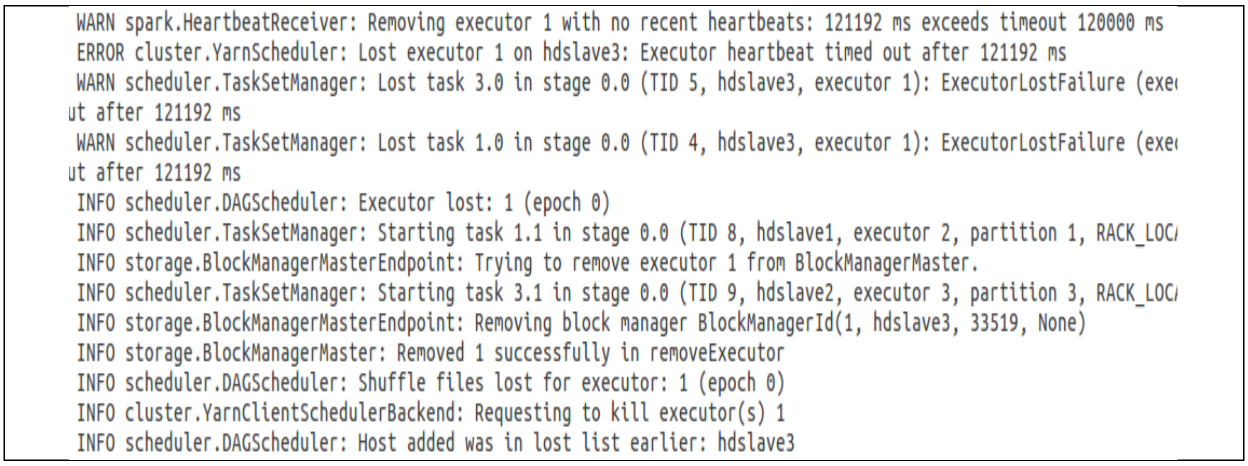
```
Type             Date       Time     Input_data_size     Duration(s)        Throughput(bytes/s)  Throughput/node
ScalaSparkWordcount 2021-07-13 12:13:19 1102231455            105.639              10433944             3477981
ScalaSparkWordcount 2021-07-13 12:18:18 1102231455            113.715              9692929              3230976
ScalaSparkWordcount 2021-07-13 12:20:39 1102231455            116.492              9461863              3153954
ScalaSparkWordcount 2021-07-13 12:23:09 1102231455            105.432              10454429             3484809
ScalaSparkWordcount 2021-07-13 12:24:43 1102231455            110.898              9939146              3313048
```

Table (5-7) values of performance parameters of Spark WordCount program with absolute failure of the two implements.

1- **Average execution time =** 105,639 + 113,715 + 116,492 + 105,432 + 110,898 = 552,176/5 = 110.4352 seconds (1 minute and 50 seconds).
2- **The throughput in the cluster =** 10433944 + 9692929 + 9461863 + 10454429 + 9939146 = 49982311/5 = 9996462.2 bytes / sec (9,435 MB / sec).
3- **Failover time:** 110,4352 - 91,6546 = 18.7 seconds.

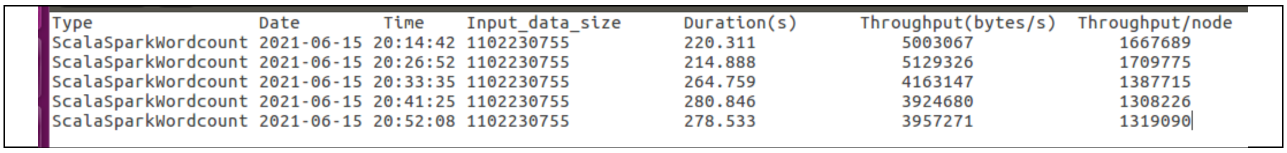
5-3-10-3 Running WordCount with crash failure injection:

- ➢ To simulate a crash failure, the running node that contains one of the simulated crash failure ports has been closed:



```
 WARN spark.HeartbeatReceiver: Removing executor 1 with no recent heartbeats: 121192 ms exceeds timeout 120000 ms
 ERROR cluster.YarnScheduler: Lost executor 1 on hdslave3: Executor heartbeat timed out after 121192 ms
 WARN scheduler.TaskSetManager: Lost task 3.0 in stage 0.0 (TID 5, hdslave3, executor 1): ExecutorLostFailure (exe
ut after 121192 ms
 WARN scheduler.TaskSetManager: Lost task 1.0 in stage 0.0 (TID 4, hdslave3, executor 1): ExecutorLostFailure (exe
ut after 121192 ms
 INFO scheduler.DAGScheduler: Executor lost: 1 (epoch 0)
 INFO scheduler.TaskSetManager: Starting task 1.1 in stage 0.0 (TID 8, hdslave1, executor 2, partition 1, RACK_LOC
 INFO storage.BlockManagerMasterEndpoint: Trying to remove executor 1 from BlockManagerMaster.
 INFO scheduler.TaskSetManager: Starting task 3.1 in stage 0.0 (TID 9, hdslave2, executor 3, partition 3, RACK_LOC
 INFO storage.BlockManagerMasterEndpoint: Removing block manager BlockManagerId(1, hdslave3, 33519, None)
 INFO storage.BlockManagerMaster: Removed 1 successfully in removeExecutor
 INFO scheduler.DAGScheduler: Shuffle files lost for executor: 1 (epoch 0)
 INFO cluster.YarnClientSchedulerBackend: Requesting to kill executor(s) 1
 INFO scheduler.DAGScheduler: Host added was in lost list earlier: hdslave3
```

- The node will not send a heartbeat and therefore the failure will be detected and tasks scheduled on this port will fail and will be reassigned to two other ports:



```
Type               Date       Time     Input_data_size      Duration(s)      Throughput(bytes/s)  Throughput/node
ScalaSparkWordcount 2021-06-15 20:14:42 1102230755          220.311              5003067              1667689
ScalaSparkWordcount 2021-06-15 20:26:52 1102230755          214.888              5129326              1709775
ScalaSparkWordcount 2021-06-15 20:33:35 1102230755          264.759              4163147              1387715
ScalaSparkWordcount 2021-06-15 20:41:25 1102230755          280.846              3924680              1308226
ScalaSparkWordcount 2021-06-15 20:52:08 1102230755          278.533              3957271              1319090
```

Table (7-21) Spark WordCount performance parameter values with crash failure.

1- **Average execution time =** 220,311 + 214, 888 + 264,759 + 280,846 + 278,533 = 1259,337/5 = 251,8674 seconds (4 minutes and 11 seconds).
2- **The throughput in the cluster =** 5003067 + 5129326 + 4163147 + 3924680 + 3957271 = 22177491 /5 = 4435498,2 bytes/sec (4.226 MB/s).
3- **Failover time:** 251,8674-91,6546 = 160,2128 seconds (2 minutes 40 seconds).

Chapter Six

Results & Suggestions

6-1 Execution time and failover time (sec):

HWF	hadoop without failure
NMF	node manager failure
AMF	application master failure
CF	crash failure
spark wf	spark without failure
EXF	executer failure
EXLF	executer lunch failure
CF*	crash failure

Table (6-1) abbreviations used in the results.

CF*	EXLF	EXF	SWF	CF	AMF	NMF	HWF	
2,518,674	1,104,352	1,021,734	916,546	12,512,918	3,159,322	914,606	1,960,372	WORD COUNT

Table (6-2) Execution time (seconds).

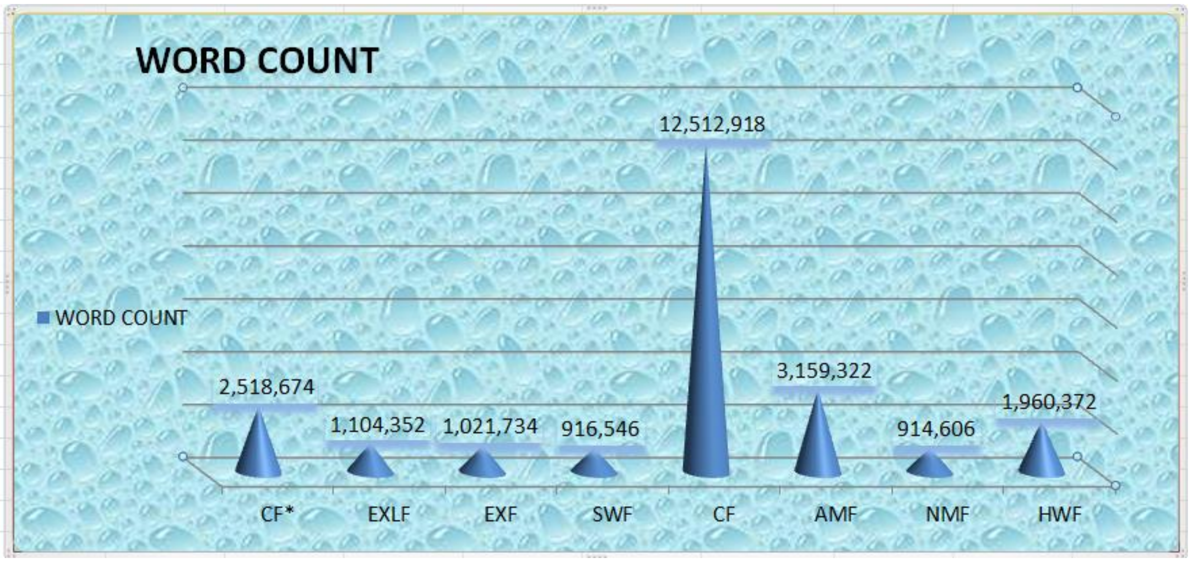


Figure (6-1) diagram of the execution time (seconds).

WR F	Spark CF	EX L F	EX F	YC F	Hadoop CF	NM F	AM F	
3	159.586	18	10	58	1054.444	715.747	117.8716	WordCount
0	393.1254	135.277	127.1736	20.798	828.538	140.003	55	Sort
3	128.4068	37.1192	5	0	881.093	535.223	0	pageRank

Table (6-3) fail-over time (sec) for all experiments.

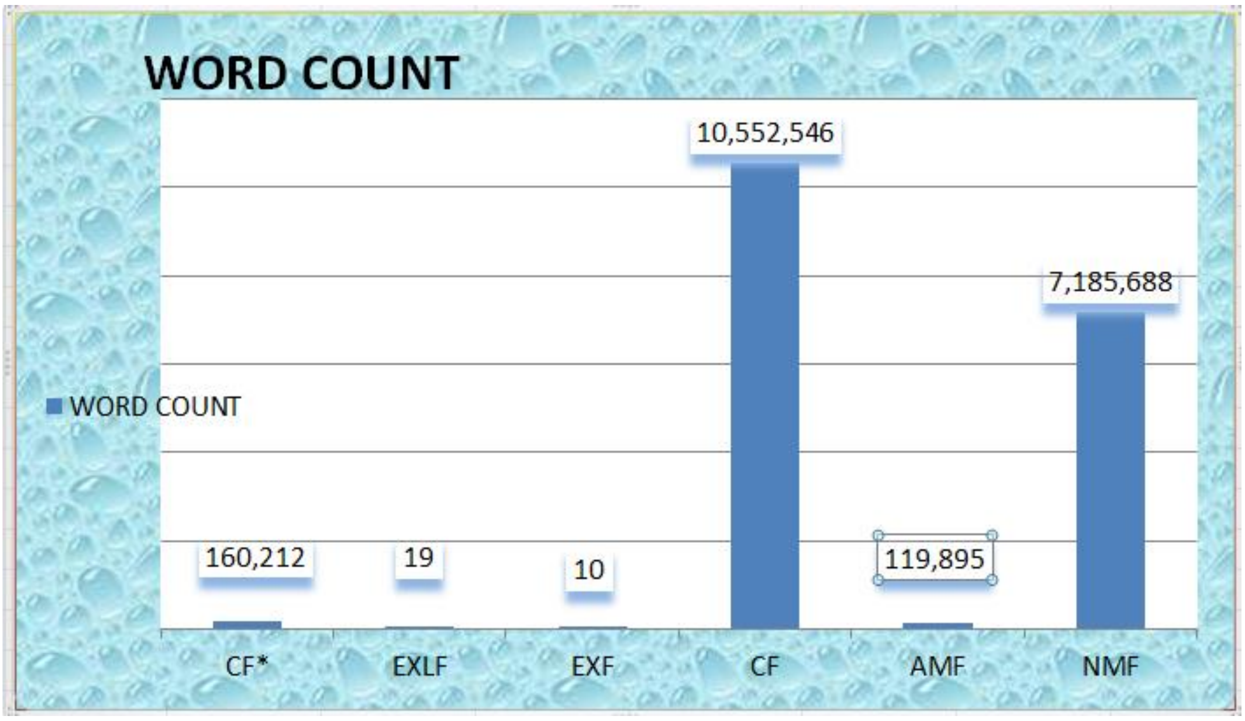


Figure (6-2) Failover time (sec) plot for all experiments.

- **Node manager failed:**

1- When a node manager failure is injected into WordCount, all corresponding tasks are completed before the resource manager can detect a failure in the node manager and after it detects the failure, it takes a lot of time trying to get the results of the corresponding tasks, and a failure to fetch results is thrown, The application manager completes the work by launching new containers on other nodes and initiating corresponding tasks until completed ones that have not sent their output to the reduction tasks the application manager does not return to reserve containers on the node whose manager failed and when the application manager receives a notification that the last task of the work is finished, it tries to release the containers before printing a message telling the user that the work finished successfully and printing the work stats and counters, so it takes time because that is the task of the failed node manager.

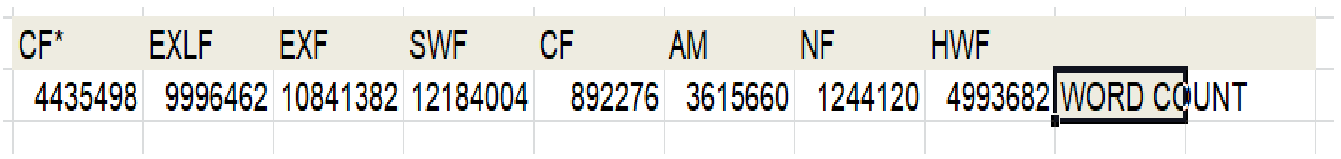
- **Application manager failed:**

  1- When executing the experiment in WordCount, we noticed that when the failure is injected into the application manager, it is immediately detected by the resource manager, who launches a new copy of it on another node.

When executing, we note that if the program has finished a set of shorthand tasks when the failure is injected, the corresponding tasks are not re-executed.
If it has not started with the reduction tasks, then all the corresponding tasks will be re-executed until the completed ones, and the failure is detected and quickly bypassed, unlike the case of the failure of the node manager, which takes somewhat longer.

- **Hadoop data node crash failed:**
  1- The difference between a full crash and crashing agents such as a node manager, data node, or application manager, is that the operating system sends a TCP reset (RST) packet only when the agent is killed, Which is an early indication of the failure and this is what we notice, as it takes Hadoop about ten minutes to detect the failure and start remedial it when the crash fails. The application manager will re-execute all the tasks on another node and it will take a lot of time trying to finish the Reserved Containers on the crashed node, which leads to a very significant deterioration of the execution time of up to seventeen minutes in WordCount.
- **Executor Launcher failed:**
  Killing the launcher leads to the failure of all the implementers it launched and all the tasks scheduled for them, as the failure is detected directly as in Hadoop and a new launcher is started on a new node, which in turn re-launches new implementers and schedules the execution of the incomplete stages with the contents of the tasks again.
  ((The launcher in Spark corresponds to the application manager in Hadoop, and if we come to the comparison of its failover in both, we note that, in WordCount, Spark is superior to Hadoop in its failover.))
- **Spark failed crash:**
  The node will not send a heartbeat so the failure will be detected and the tasks scheduled on this port will fail and will be reassigned to two other ports.
  ((We note that Spark is faster to bypass it because the number of agents running on the node is fixed, but Hadoop, the broken node contains the node manager and an unlimited number of running tasks whose killing affects the execution time)).

6-2 Throughput (MB/sec):



CF*	EXLF	EXF	SWF	CF	AM	NF	HWF	
4435498	9996462	10841382	12184004	892276	3615660	1244120	4993682	WORD COUNT

Table (6-4) throughput (MB/sec) for all experiments.

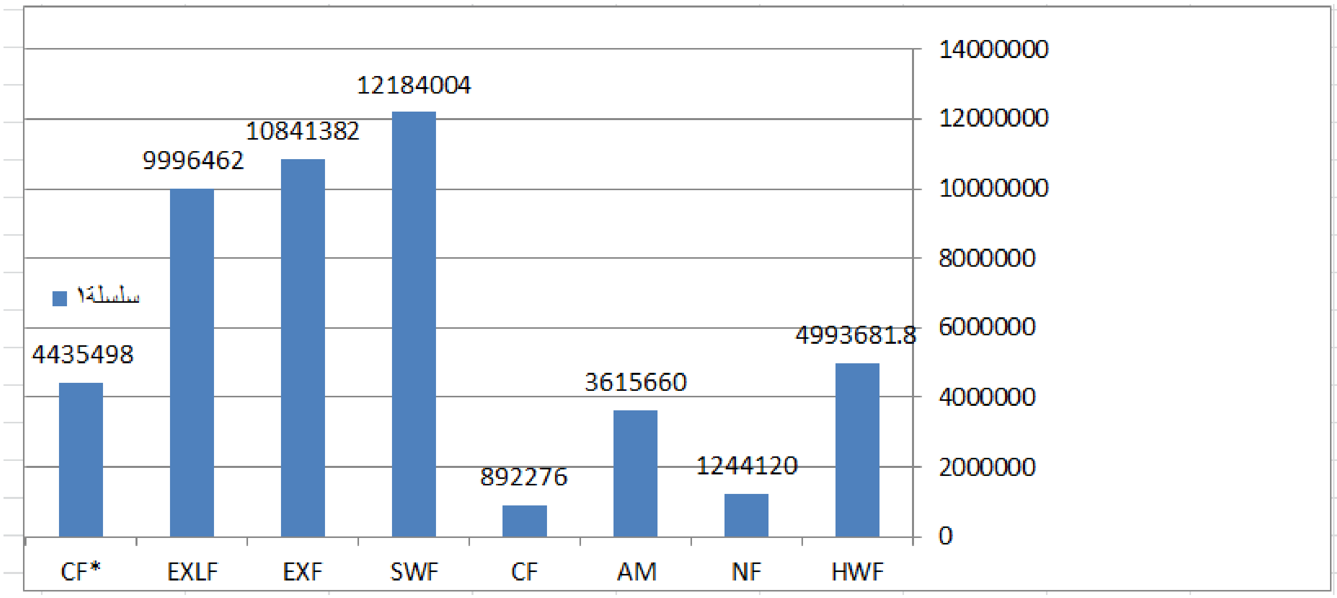


Figure (6-3) throughput plot (MB/sec) for all experiments.

- We note that the longer the execution time, the lower the productivity of the system and therefore, since the increase in execution in the event of a crash failure is greater than in other cases, so we find a significant deterioration in the productivity of the system when it occurs in both systems, while the failure of the application manager in Hadoop and the implementer in Spark is considered the least influential on productivity.

6-3 Results and Recommendations:

We note that in terms of the time both systems exceed the injected types of failure, the failure that affects both systems is the failure of the account node to completely crash, with its agents, but Hadoop is affected much more than Spark by the failure of the crash, as it takes about seventeen minutes in a program WordCount to bypass it, Then comes the failure of the node manager, as its failure time in WordCount reaches twelve minutes, and the increase in the time that occurs mostly in the application manager's attempt to release containers on the node whose manager failed, while the failure in the application manager is considered the least effective.

In WordCount, Spark is very tolerant of all types of failures except for crash failures. Nevertheless, it outperforms Hadoop in overcoming this failure, followed by launcher failures, while failure of the port is considered the least effective.

References:

[1] Pan, Shengti, The Performance Comparison of Hadoop and Spark (2016). Culminating Projects in Computer Science and Information Technology .

[2] MapReduce Tutorial, 2015, http://hadoop.apache.org/.

[3] T. White," Hadoop: The Definitive Guide (Fourth edition)". Sebastopol, CA: O'Reilly Media, 2015.

[4] Bessani, A. N., Cogo, V. V., Correia, M., Costa, P., Pasin, M., Silva, F., … Sopena, J. (2010). Making Hadoop MapReduce Byzantine Fault-Tolerant. Proc. of the DSN - Intl. Conf. on Dependable Systems and Networks, (February 2016), 1–2.

[5] Costa, P., Pasin, M., Bessani, A. N., & Correia, M. P. (2013). On the performance of byzantine fault-tolerant mapreduce. IEEE Transactions on Dependable and Secure Computing, 10(5), 301–313.

[6] Marynowski, J. E., Santin, A. O., & Pimentel, A. R. (2015). Method for testing the fault tolerance of MapReduce frameworks. Computer Networks, 86, 1–13.

[7] Faghri, F., Overholt, M., Campbell, R. H., & Sanders, W. H. (2008)." Failure Scenario as a Service (FSaaS) for Hadoop Clusters".

[8] Troubitsyna, Elena A., "Faults, errors, failures," [Online]. Available: http://users.abo.fi/etroubit/SWS13Lecture2.pdf.

[9] M. Isard, V. Prabhakaran, J. Currey, U. Wieder,K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, SOSP '09, pages 261–276, New York, NY, USA, 2009. ACM.

[10] Wang, H., Chen, H., Du, Z., & Hu, F. (2016). BeTL: MapReduce Checkpoint Tactics Beneath the Task Level. IEEE Transactions on Services Computing, 9(1), 84–95.

[11] Sangroya, A., Serrano, D., Bouchenak, S., Sangroya, A., Serrano, D., Bouchenak, S., & Dependability, B. (2012). Benchmarking Dependability of MapReduce Systems. To cite this version HAL Id: hal-01472165.

[12] Hao, Z., & Alnawasreh, K. (2016). Distributed Systems verification using fault injection approach.

[13] Tinetti Fernando, G. Distributed systems: principles and paradigms (2nd edition): Andrew s. tanenbaum, maarten van steen pearson education, inc.,2007 isbn: 0-13-239227-5. Journal of Computer Science and Technology 11, 2 (2011), 115–116.

[14] Kashkouli, A., & Soleimani, B. (2017). Investigating Hadoop Architecture and Fault Tolerance in Map- Reduce, 17(6), 81–87.

[15] Ch. Lam, Hadoop in action. Printed in the United States of America, 2011.

[16] Bilal, K., Khalid, O., Malik, S. U. R., Khan, M. U. S., Khan, S. U., & Zomaya, A. Y. (2016). Fault Tolerance in the Cloud. Encyclopedia of Cloud Computing, (ITProPortal), 291–300.

[17] Costa, P. A. R. S., Bai, X., Ramos, F. M. V., & Correia, M. (2016). Medusa: An Efficient Cloud Fault-Tolerant MapReduce. Proceedings - 2016 16th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2016, 443–452.

[18] Evans, J. (n.d.). Fault Tolerance in Hadoop for Work Migration. Salsahpc.Indiana.Edu, 3–7.

[19]Lu, X., Wasi-ur-Rahman, M., Islam, N. S., & Panda, D. K. (2014). A Micro-benchmark Suite for Evaluating Hadoop MapReduce on High-Performance Networks. Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 8585, 32–42.

[20] https://github.com/david78k/anarchyape

[21] Samadi, Y., Zbakh, M., & Tadonki, C. (2017). Performance comparison between Hadoop and Spark frameworks using HiBench benchmarks. Concurrency Computation, (October), 1–13.

[22] Mavridis, I., & Karatza, E. (2015). Log File Analysis in Cloud with Apache Hadoop and Apache Spark. Proceedings of the Second International Workshop on Sustainable Ultrascale Computing Systems (NESUS 2015), 51–62.

[23] Liu, L. (2015). Performance comparison by running benchmarks on Hadoop, Spark, and HAMR. Retrieved from https://dspace.udel.edu/handle/19716/17628

[24] Samadi, Y., Zbakh, M., & Tadonki, C. (2016). Comparative study between Hadoop and Spark based on Hibench benchmarks.

[25] Huang, S., Huang, J., Dai, J., Xie, T., & Huang, B. (2011). The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. Lecture Notes in Business Information Processing, 74 LNBIP, 209–228.

[26] Xu, G., Xu, F., & Ma, H. (2012). Deploying and Researching Hadoop in Virtual Machines, 2(August), 395–399.

[27] Reyes-Ortiz, J. L., Oneto, L., & Anguita, D. (2015). Big data analytics in the cloud: Spark on Hadoop vs MPI/OpenMP on Beowulf. Procedia Computer Science, 53(1), 121–130.

[28] Kang S, Yeon S, Myung K. Performance comparison of OpenMP, MPI, and MapReduce in practical problems. Adv Multimedia. 2015;2015:9

[30] Aaron D, Andrew O. Optimizing shuffle performance in Spark. Berkeley-Department of Electrical Engineering and Computer Sciences, California Technical Report.

[31] Gu L, Li H. Memory or time performance evaluation for iterative operation on Hadoop and Spark. In: IEEE 10th International Conference on High Performance Computing and Communications; 2013; Zhangjiajie.721-727.

[32] Veiga, J., Exp, R. R., Pardo, C., Taboada, G. L., & Touri, J. (n.d.). Performance Evaluation of Big Data Frameworks for Large-Scale Data Analytics.

[33] P. Jakovits and S. N. Srirama, “Evaluating MapReduce frameworks for iterative scientific computing applications,” in Proc. of the International Conference on High Performance Computing & Simulation (HPCS’14), Bologna, Italy, 2014,pp. 226–233.

[34] Hadoop vs Apache Spark. (n.d.).

[35] Ahmed, H., Ismail, M. A., Hyder, M. F., Sheraz, S. M., & Fouq, N. (2016). Performance Comparison of Spark Clusters Configured Conventionally and a Cloud Service. Procedia Computer Science, 82(March), 99–106.

[36] Dinu, F., & Ng, T. S. E. (2012). Understanding the effects and implications of compute node related failures in hadoop. Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing - HPDC ’12, 187.