Literature Study and Seminar

Battle of the big data

Author: Tom Peerdeman
Student number 2559584

Date: February 11, 2016

Abstract

Since the introduction of MapReduce and Hadoop in 2004 the popularity of big data processing has skyrocketed. Since that introduction 12 years ago many new projects have been created to compete with Hadoop. In this paper we will perform a literature study to compare three of the most used systems, Hadoop, Spark and Storm. We will look at the design, reliability, application development and performance of these systems.

1 Introduction

Since the introduction of the MapReduce abstraction by Google in 2004 the usage of big data processing has skyrocketed. In this time the developments of this kind of frameworks of course have continued. In this paper we will conduct a literature study to compare three of the most popular big data frameworks. We will look at four aspects of these frameworks. The first aspect is the design of the framework. What are the major components of the frameworks, and how do they work together. In what way do they differ from the existing systems.
Of course when using multiple machines, which is often required to process large amounts of data, failure becomes a bigger problem. The second aspect we will be focusing on is therefore reliability of the frameworks.
The third aspect is application development. How hard is it for users to create the implementations that can process their data. This is important because you can create the perfect system in terms of design, but if it is hard to program it, likely nobody will use it. The final aspect is performance. At the time of writing the original MapReduce paradigm is about 12 years old! Can we expect that the new designs perform better?

The first framework we will be looking at is the Hadoop framework. Hadoop is

an implementation of the MapReduce abstraction. This abstraction was initially created by Google to perform calculations on their massive amounts of data. Examples of the applications it was used for were inverted indices, various representations of the graph structure of web documents, summaries of the number of pages crawled per host, the set of most frequent queries in a given day, etc [1]. Since there was interest for this kind of framework from the outside, an open source implementation called Hadoop was created. The first release of this framework was in 2007.
On 23 May 2013 the first release of a reworked Hadoop framework was released [2]. This second version of the framework has the same programming model, as it still uses MapReduce. The design of this new version is however different from the original design.

The second framework is called Spark. Spark was created to be an improvement of the Hadoop system. It was initially created by the University of California at Berkeley's AMPLab. In 2014 it was handed over to the Apache Software Foundation [3]. It still resides there, and is still being actively developed. The latest release at the time of writing is 1.6.0 dating from 4 January 2016.

The final framework is the Storm framework. Storm was created around 2011 by Nathan Marz. In May 2011 the company Marz worked for was acquired by Twitter, allowing it to use the technology. In September 2013 it was proposed for incubation at the Apache Software Foundation [4]. Today it still resides there as an top project, and is being actively developed. The latest release is from 5 November 2015.
The main difference between Storm, Spark and Hadoop is that Storm is aimed primarily at real-time data processing.

These three systems are of course not the only ones available. These three were chosen as they are used the most, and are actively developed. A short list of other system can be found in appendix A.

2 Design

2.1 Hadoop

To understand the design of Hadoop we first have to understand the MapReduce abstraction. This is best explained by the creators of MapReduce in their paper *MapReduce: Simplified data processing on large clusters* [1]. The MapReduce model is based on taking an key/value pair as input, doing the computation on this pair and outputting key/value pairs. It does so in two steps. Two first step is the Map operation. The goal of this step is to split up the input. For example one file splits up into many words. The output of the map step is then used as input for the reduce step. The goal of this step is to merge the input values to generate a smaller amount of values. This reducer gets called exactly one time for each key with the set of values associated with this key.

Since the reducers need all the values from one key, all the input data has to be processed by the mapping function before the reducer can be run. This means that this model is data parallel.

Hadoop implements the MapReduce model. It's design is based on a master-slave structure. The master is called the JobTracker. All the other machines are called TaskTrackers. Note that a single machine can also be the JobTracker and TaskTracker, this allows for single machine Hadoop setups. The goal of the JobTracker is to instruct and monitor the TaskTracker instances. When a job is submitted to the JobTracker he instructs the TaskTrackers to start the map and reduce tasks [5].

The data itself is provided by a distributed file system called HDFS, Hadoop Distributed File System. HDFS is loosely based on the Google file system, GFS, but differs in terms of replication management and data balancing [6].
A job starts of with some data stored in HDFS. The Hadoop system is aware of the distribution of the data over the nodes, so it can schedule the map tasks on the nodes where the data is available. The output of these map operations is then again stored in HDFS. The data is distributed so that the data with the same keys is stored on the node that will run the reducer for this key. Finally the output of the reducers is stored in HDFS. The job has now finished.

2.2 Hadoop 2.0

Each Hadoop setup has only one JobTracker and many TaskTrackers. The Hadoop system can also run multiple jobs at the same time. One can imagine that this approach does not scale very well. Actually the system only could handle up to 4000 nodes. This and other resource management issues were the reason why Hadoop 2.0 was introduced [7].
Where in Hadoop 1.0 the resource management and data processing were coupled, they are split in Hadoop 2.0. This was done by introducing a new project called YARN. YARN, Yet Another Resource Negotiator, takes over the job of resource management. It is designed using the same master-slave model. The master is called the Resource Manager, and the other nodes Node Manager. The Resource Manager node takes over the task of scheduling and monitoring the resources supplied by the Node Managers.
When a job is submitted to the Resource Manager it allocates resources on a Node Manager. Using these resources it then starts an Application Manager. This application manager is dedicated for that job, and takes over the monitoring of the job status. The application manager then decides how much additional resources are required, and requests those from the Resource Manager [8, 9]. Using this model each job is somewhat isolated from each other. Running multiple jobs won't overburden the Resource Manager, as it does not have to track the calculation itself. Therefore the scalability is much improved over the JobTracker/TaskTracker structure.
An overview of a system running YARN can be seen in figure 1. The application managers communicate with the resource manager to allocate resources for workers.

The resource manager does not communicate with the workers itself.

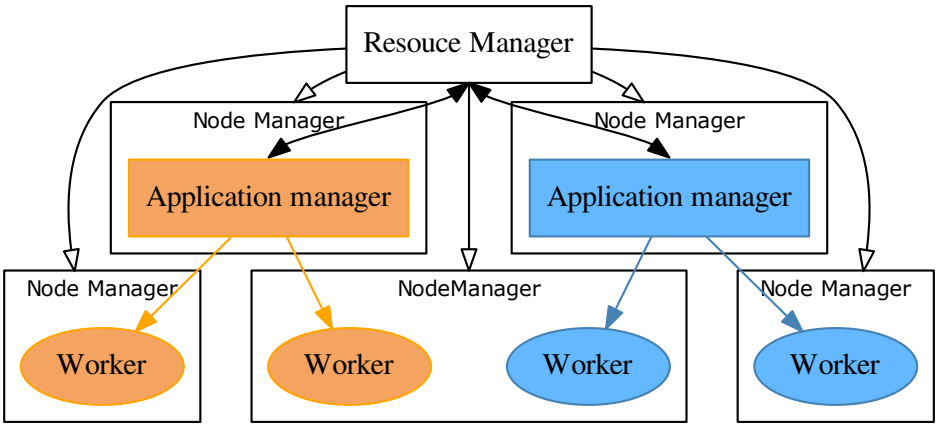

Figure 1: Example of YARN running on 5 machines (+1 resource manager machine) with 2 independent applications

Another improvement of the original system is that a chain of jobs can be started. This way it becomes easier to process big data, where the computation does not fit a to single map reduce operation.

The Hadoop system actually contains many more projects which interact with HDFS and MapReduce. For simplicity, if we speak of Hadoop 2 in this paper we mean the combination of YARN, HDFS and the MapReduce.

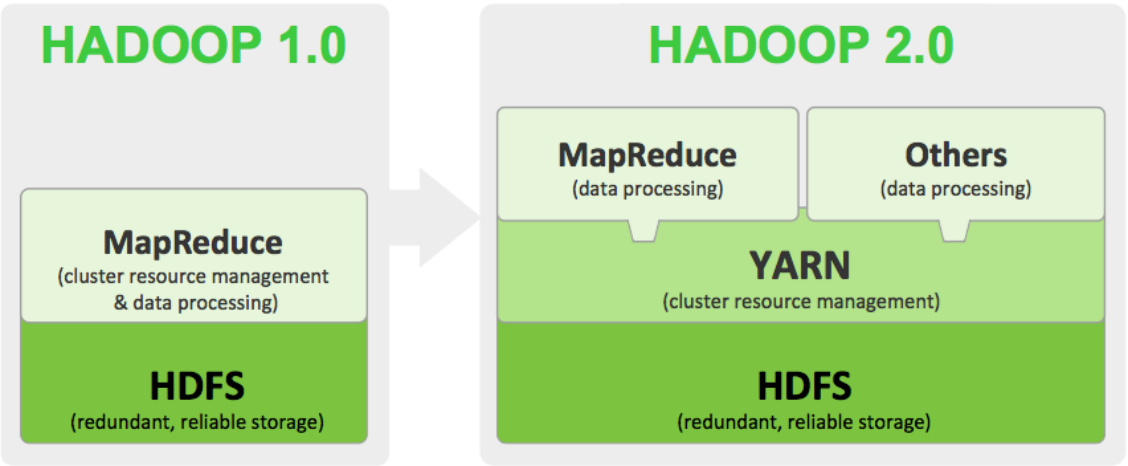

Figure 2: Hadoop components

2.3 Spark

It is a known fact that one of the slowest parts of computers is still the disk. Hadoop however relies on storing all the intermediate results in HDFS, which ultimately stores it on disk. This is where Spark jumps in. Spark makes use of RDD's, Resilient Distributed Data set. These RDD's are data set that are stored in memory, distributed over the nodes. A new RDD can be created by transforming another RDD [10, 11].
If we would look back at Hadoop, which Spark is trying to improve, the available transformations would be instances of Map and Reduce. Spark improves this by allowing many more transformations and actions. For example filter data, group the data, or count the amount of data elements [12, 13]. This also means that the jobs running on Spark are not limited to two operations per job. Spark makes use of a DAG, Directed Acyclic Graph, engine.
When a job is submitted first the individual transformations on RDD's are split into the individual tasks. These tasks require input and have an output, therefore they can be placed in a graph. The engine then splits up these tasks into stages using this graph. The tasks in one stage either have no dependencies on each other, or have narrow dependencies. A narrow dependency is where task B, which depends on task A, can be run on the same machine as task A. Between the stages the tasks have wide dependencies. This means the data has to be redistributed over the machines [10]. In Hadoop terms this is the output of a Map task being placed on the correct machine running the reducer for that key.

While this does explain the general design of the Spark system, it does not explain how resources are allocated and monitored. If we look back at YARN in Hadoop 2.0 one can notice that the Application Managers doe not have to be MapReduce instances. This was one of the goals when designing YARN. Multiple applications should be able to run on the same cluster, with little additional effort. This can be seen in figure 2. Spark has built in support for YARN, and can thus be ran on a YARN cluster.
Besides YARN it also has support for Amazon EC2, Apache Mesos and standalone mode [14]. In standalone mode the user has to start the master and workers by itself on the machines of the cluster.

In section 1 it is mentioned that Storm differs from Hadoop and Spark in that it uses real time processing. This is however not entirely true as Spark also has a mechanism to process live data. Spark's design is however oriented around tasks, and is thus data parallel. This is solved by introducing DStreams. A DStream, discretized stream, is basically a sequence of RDD's. The DStream represents the stream of input data. The individual RDD's of this DStream are created by collecting data from the input stream for a configurable amount of time. While the next RDD is being collected, the previous RDD is put trough the system, just as if it were a batch job [15].

2.4 Storm

While Spark does real time data processing it does so by processing micro batches. The design of Storm is conceptually different as it achieves real time processing by streaming the tuples one pair at a time trough all the computational transformations. The difference is that batch jobs are finite, they have a state where all the data has been processed. Storm however is a complex event processing system [16, 17]. This means that all the tasks run at the same time. Storm is therefore task parallel unlike Hadoop and Spark, which are data parallel. This also means that the tasks will never end normally. There is simply no end of the data, or events, where the system is done.

The Storm system is designed around the notion of streams, spouts and bolts. Streams are unbounded pairs of tuples. A spout is a source of a stream. An example of a spout is a live twitter feed [16, 18]. A bolt takes one or more input streams, does some computation on them, and possibly outputs a new stream. The whole graph, or network, of spouts and bolts is called the topology. The output of a topology is also implemented by a bolt. A bolt can for example write to a database, or write to a file [18]. An example of such a topology is shown in figure 3.

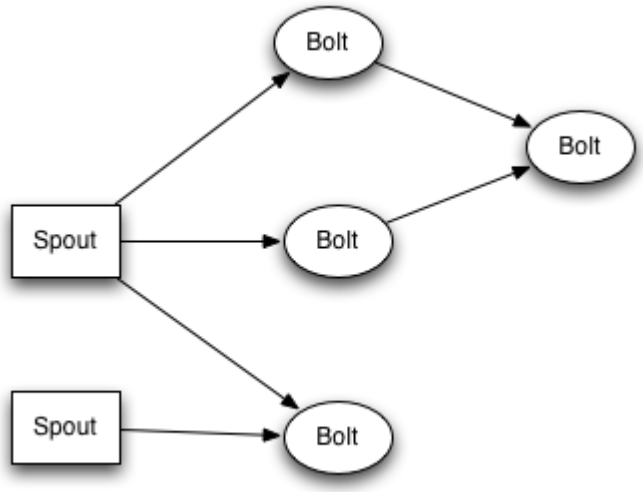

Figure 3: Example storm topology

Storm is not built on top of a resource manager like YARN, instead it has it's own process management system. This system is based on the master-worker paradigm and uses Zookeeper. Zookeeper is a project which enables highly reliable distributed coordination. The master is called Nimbus. All the other nodes are called supervisors. When a new topology is submitted to the Nimbus it instructs the supervisor nodes via zookeeper to start processes [18]. These worker processes will then start tasks. Each spout or bolt corresponds to one or more tasks. The actual amount is specified by the implementation of the spout or bolt. The tasks are then distributed

over the workers. This number of tasks is fixed for the topology.
Each task inside the worker process may have one or more executors running as threads inside the worker process. These executors are the actual instances of the spouts and bolts running inside the system.
The system uses streams, this means that those streams can be redirected. In Storm this is used to be able to adjust the amount of executors [19]. This comes in handy when you notice your system is not working perfectly because a ratio between a spout/bolt and another bolt is slightly off. In such a situation you can modify the amount of executors, while the system s running.

Of course when using a real time processing system a fast system for communication between the tasks is required. The requirements for inter task communication is slightly different for Storm and on the other hand Hadoop and Spark. This is because in Hadoop and Spark the tasks that depend on each other are not expected to be running concurrently. In Storm it is a given that all the tasks are running concurrently. The communication was initially handled by ZeroMQ. ZeroMQ is an project created to provide low latency distributed messaging. Later on this was replaced by the Netty project due to licensing issues [4, 20].
Some tasks run withing the same worker process as threads. This means that the communication between these tasks can be done in the memory of that process. Storm uses Disruptor for this communication. Disruptor is a bounded queue for exchanging data between threads. Since the tasks, and thus the threads are running concurrently a precaution has to be in place so that the inter thread messages arrive correctly. It could be that a tasks tries to read a message which has not been written completely yet. To overcome this problem usually locks are used. Lock however are quite costly. Disruptor circumvents this problem by using compare and swap operations [21].

3 Reliability

Of course when processing large amounts of data it is important that the system used can provide some reliability. This reliability can be split up in two parts. The first part is fault tolerance. In a large data processing situation often a cluster is used. When using multiple machines the chance that some part will fail increases. A big data system should be able to continue working, or recover from such failures. The second part of the reliability is the ability to guarantee that all data is processed. This means that in a environment with components that can fail the outcome is always the same as in the hypothetical failure free environment.

3.1 Hadoop

The fault tolerance system of Hadoop is described in *MapReduce: Simplified data processing on large clusters* [1]. It is based on the master continuously tracking the workers, TaskTrackers, progress. If the master, JobTracker, fails to receive the sta-

tus of a worker it assumes it has gone offline, and is no longer part of the computing cluster. The master then redistributes the tasks assigned to this machine to the machines that are online. He can do this since the data is still available in HDFS, since HDFS has it's own fault tolerance mechanism based on replications.
There is one special case, which is when the master dies. In this case all the information is lost of the jobs, and the status of the workers. The status can't be retrieved from the workers as a worker that has been offline may come back online and report it is doing a job, which has been rescheduled. In such a case no other possibility exists than stopping all the tasks. In later versions this was improved by periodically writing the status to disk. The new JobTracker could then read this status and start off from that point.

The correctness of the output in such a environment where components can fail is also tied to this system. When a worker fails the tasks running on that node are rescheduled to other nodes. All the reducer tasks previously executed by this node are safe, as their output is written to HDFS. The output of the mapper tasks is however stored locally on the node, when all the map tasks are finished these are moved to the correct reducer nodes. This however means that the output of the mapper tasks on the failing node are now inaccessible. This thus means that all the map tasks executed have to be rescheduled. This mechanism ensures that all the data is being processed at least once.
It is however possible that a previous offline node becomes online again. Such a node may continue the work it was doing, and thus completing tasks that already may have been rescheduled. This can cause the same output to appear twice, which may lead to incorrect results. The JobTracker can prevent this since it knows the status of each task. The Hadoop system is implemented so that when a worker finishes a task he first has to inform the JobTracker. If the JobTracker notices that that task has already been completed it informs the worker, which then discards the result.
These two principles ensure that all the data is processed exactly once from the point of the final output. This ensures that the outcome is the same in an environment with and without failures.

In Hadoop 2 the job of monitoring the status of the machines is handled by YARN.
The fault tolerance and availability of YARN is described in *Apache Hadoop yarn: Yet another resource negotiator* [22]. As described before the YARN system has two components, the ResourceManager and many NodeManagers. Inside the machines monitored by the NodeManagers run containers which can contain ApplicationManagers and workers, which are linked to application managers.
When a machine goes offline the resource manager notices that the node manager of that machine no longer responds. It can't simply restart the containers running on that node on another node, as it has no idea what was actually running inside those containers. It does know which program was running, but not if it can recover from being killed. The only thing it can do is inform all the application managers that the node has gone down, and with it possibly some containers. If the node happens to come back from its offline state, and contacts the resource manager, it is told to

kill all containers in that node, since they have been marked as dead already.
If the containers by chance contained a application manager the resource manager notices it has no more connection to that application manager. It then restarts the application manager, which can then recontact its workers.

A special case is when the machine running the resource manager fails. In earlier versions of YARN when the resource manager died and restarted, it had to kill all the running containers. In version 2.4 an high availability resource manager was introduced [23]. This resource manager relies on having an standby copy of itself. It synchronizes its state with that copy. When the primary resource manager fails the standby can either be automatically or manually activated. The application managers and node managers then reconnect, and continue with what they were doing.

This fault tolerance model can easily be applied to the MapReduce system running on top of YARN. The actual handling of the signals provided by the resource manager is similar to the fault tolerance system of Hadoop 1. The situation where the application manager goes down, and is being restarted, is equivalent to the JobTracker going down. The signal from the resource manager that one of the containers has gone down is equivalent to the JobTracker noticing one of the workers has gone down. The only difference is that the application manager for MapReduce can ask for new worker containers when a worker does go down. In Hadoop 1 this was not possible.

3.2 Spark

Since Spark can run on YARN, its fault recovery system is very similar to the MapReduce 2 system. The master will be restarted if it died, and recovers its state via checkpoints made by the original master. Any work done by the workers that died will be rescheduled. The difference between the two systems is that Spark uses the DAG engine. The DAG engine allows more than two operations per job whereas MapReduce only allows two operations per job, map and reduce. Combining this with the fact that all data is stored in memory, it could be possible that a task on a failing machine has to be redone on input data that is no longer in memory. Spark solves this by using it's RDD mechanics.
The fault recovery mechanism of the RDD's is described in *Resilient distributed data sets: A fault-tolerant abstraction for in-memory cluster computing* [10]. The basic concept is that an RDD stores the transformations that are required to compute it from input data. This way, assuming a reliable input like HDFS, if a RDD or a section of a RDD gets lost it can be recomputed from the input data.
This of course can be quite expensive, as a small subset of a RDD can depend on the whole RDD it was transformed from. To solve this Spark introduces persistence. With persistence the programmer can specify that machines keep a RDD. Note that with big data storing these intermediate results can be quite big in memory. Therefore an action can be specified what to do when the RDD does not fit into memory.

The options are discard overflow and recompute as needed, store the overflow on disk, and finally store the RDD only on disk [12].

Of course the Spark system not only is suitable for batch jobs, but can also handle streaming jobs. The requirements regarding reliability for streaming jobs are a bit different. First of all the input is not always reliable. If the input is for example a stream of tweets, there is no way of reprocessing a tweet that has passed in the stream, as the data simply is not available any more. This is solved by replicating the received data over multiple workers [15]. This way the chance is decreased that streaming data is lost.

The second requirement reliability in a streaming setting is that it should minimize delays. With streaming purposes it often is critical that the data is processed as quickly as possible. It is thus unwanted that a failing machine causes a big holdup since some data has to be recomputed. It is of course inevitable that the computation will take more time when parts of the system fail. Spark at least makes sure that parts of the system are not idle when waiting for the recomputation. It does so by parallel recovery. The RDD that has to be recomputed is split up in sections. These sections are then scheduled to be recalculated [24]. The machines that are idle due to dependencies on the lost data can now be used to recalculate the missing data as fast as possible. The slowdown due to failures is thus minimized.

3.3 Storm

Storm has its own fault recovery system which is based on a fail fast mechanism. Both the master, Nimbus, and slaves, Supervisors, are stateless by itself. Their state is saved by Zookeeper, which means it is distributed, and thus allows for failures. The nimbus and supervisors itself are run using an utility program which restarts them if it notices they exited. In this way if something goes wrong the components can just kill themselves, and rebuild upon a correct state when restarted. When a supervisor continuously fails, or never recovers the nimbus may choose to reassign the tasks of that supervisor. The supervisor on its own monitors the workers running on that node. Those workers are also fail fast, so if they exit for some reason the supervisor can restart them.
The final situation is where the machine running the nimbus fails. In such a case the system can't recover the nimbus on its own. The system however can continue on its own. The jobs currently running will continue just fine without the nimbus. The only difference is that tasks cannot be reassigned when other nodes fail [25]. The Nimbus however can be restarted manually on a different machine, allowing the system to continue as normal.

While guaranteeing message processing is a built in function of Hadoop and Storm, it is optional in Storm. The message guaranteeing in Storm is based on acknowledgements. Storm has a special task called acker. The goal of this task is to track all the tuples generated by the spouts and bolts. When a bolt has fully processed

a tuple it may ack it. When it does so it sends an message to the acker it is done with that tuple. When a bolt generates a new tuple it also anchors it to the input tuple. This way a tree can be generated with root the tuple generated by a spout. When the acker task notices that a tree has been fully processed, that is all tuples are acked and no new anchored ones are generated, it informs the spout of the root tuple. When a root tuple is generated a timer is also started. When this timer exceeds 30 seconds, without the tuples tree being fully processed the spout is informed that the tuple failed. This indicates that a task processing one of the tuples in the tree died. The spout can then retry the tuple [26].
An example can be seen in figure 4. In this simple topology a spout generates a tuple *t1*, this tuple is processed by a bolt to generate a tuple *t2*. Finally another bolt consumes *t2*. When a tuple is generated the (in this example single) acker task is informed (1 and 3). When a bolt processes a tuple it also informs the acker task it is finished with a tuple (2 and 4). Finally when the final bolt is done with *t2* the spout can be informed that tuple *t1* processed correctly. The acker task knows that *t2* is part of the tree of *t1* since *t2* is anchored to *t1* when it was created by the first bolt.

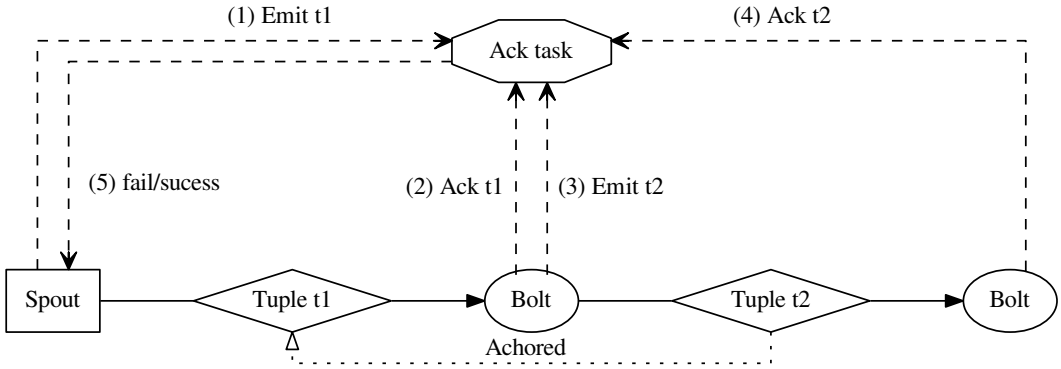

Figure 4: Example Storm topology with guaranteed message processing

Note that the tree does not contain the actual data of the tuples, it just contains unique id's for the tuples. So when a spout is informed that a tuple failed, it has to fetch the data from it's data source again. This means that, unlike Spark streaming, Storm cannot handle input streams that are unreliable. However unreliable streams can be easily converted to reliable using for example Apache Kafka. Kafka is a distributed messaging system that allows to replay already consumed messages. In such a way unreliable data can be streamed into Kafka, which streams it into Storm. If something goes bad Storm notifies the Kafka spout, which replays the message for that tuple.

4 Application development

4.1 Hadoop

Both versions of MapReduce are implemented using Java. The API used to create processing implementations is therefore also Java based. This API is very well described in the MapReduce tutorial [9], and is almost identical for the MapReduce component of Hadoop 1 and 2. To create a working implementation the user has to do two things. First the user has to implement or extend (depending on the version of the API) a mapper, depending on what computation the user wants. Secondary the user has to create a main class. When this main class is executed it starts a new job containing the just implemented mapper. While a MapReduce job can work without a reducer it is common that a reducer implementation is created as well.

If the user requires more control over the input and output format he can specify which classes to use for these features. The MapReduce framework has many built in classes that can fulfill these functions. Most of those classes are based on file, and thus HDFS, based operations. That means that classes providing input read this from files. Classes that provide an output function do so by writing the output to files. The default input format, which is used when no format is specified, is TextInputFormat. The input format reads specified files line by line. The key/value pairs produced are the position in the file as the key, and the line as text as the value.
The default output class is TextOutputFormat. This class writes the key/value pairs produced by reducers to a file which is specified by the actual reducer that produced the key/value pair. The pairs are written using a default tab separator, one pair per line.
The user may also write their own classes that can extend existing input and output formats, or create whole new ones. This does mean that the MapReduce system is not tied to HDFS as input source and output sink.

If you do not like writing your application in Java Hadoop has a partial solution called Streaming. This Streaming has nothing to do with real time data processing as provided by Spark and Storm. Instead it allows you to specify programs that replace the mapper and reducer. The MapReduce system will start these programs instead of creating a mapper instance. The system feeds this program the key/value pairs as lines to the programs stdin stream where the key and value are separated by the tab character. The output key/value pairs are parsed from the stdout stream from this program, which should be emitted in the format which is the same as the input format [27].
While this does allow you to program the mapper and reducer in your favourite language, the user still has to provide a main class that starts the job. If required the user also has to provide a custom input and output format, written in Java.

4.2 Spark

Spark is written in Java and Scala. Despite this it has API bindings for Java, Scala, Python and R. This API is described in the Spark documentation [12, 28]. To create a job the user has to create a single main function or class. This main function describes how to generate the output RDD, and what to do with it. The generation of the output RDD is specified by the input RDD, and the transformations upon that RDD to finally create the output RDD.

The first step is to create the input RDD or RDD's. This can be done in three ways. The first option is to create a RDD from an existing collection. This collection can be available in the main function, the Spark system then converts it to a RDD by distributing the data of the collection over the workers. The second option is to read the RDD from a text file. Spark has a method called textFile which reads the file specified line by line. The data elements of this RDD are the individual lines that were read. The final option is a consequence from the goal of improving Hadoop. That is namely that an RDD can be generated from a input format from the MapReduce system described in the previous section. When using this method the input RDD will consist of key/value pairs. Note how all three of the options can provide different types of data in the RDD. The textFile creates text, the MapReduce system produces key/value pairs. The data type that the collection method is unknown, as the collection can also contain any kind of data type.

The second step is to define the transformations required to generate the output RDD from the input RDD. Each transformation has one or more inputs and generates a new RDD. This new RDD does not have to be of the exact same data type. For example the cartesian transformation takes two RDD's of type T and U as input and generates a RDD of all the T, U pairs. Some of the transformations also require a function. For example the map function transforms one data element into another one given a function. This function is often expressed as a lambda function, due to its simplicity. For Lambda expressions in Java are introduced in version 8. Version 7 of Java is however still used very frequently. If a user wants to use those kind of transformations in Java 7 (and below) he has create a class that implements a interface containing a single function. The actual interface, and thus the arguments and return values of the function depend on the transformation used.
Spark also has a method that is equivalent to Streaming in MapReduce. The pipe transformation starts a given program. The input is again passed to stdin, and the output parsed from the stdout stream of the program.

Finally the output of the RDD is described. The output can be written as a text file, one line per value. If Java or Scala is used the output can also be serialized using the built in serialization of the JVM. Finally if the output RDD contains key/value pairs the Hadoop output formats can also be used.

4.3 Storm

Storm is a bit different than Spark and Hadoop in that it is written in Clojure. Clojure is a function programming language. It can however run on the Java virtual machine. This allows it to expose a Java API [18]. To create a working topology the user has to do a few things. First the user has to provide an implementation for the spouts and bolts. Some implementations are already available. For example spouts for various queues like Kafka are available.
Second the user has to declare the actual topology. This can be done by defining a Topology object, and adding instances of the spouts and bolts to it using a name. Each time a bolt is added, the input can be declared. This way dependencies can be declared. This is done by calling a method with the name of the spout or bolt that outputs the tuples that node depends on. Multiple of these methods are available. The difference between the methods is the manner how the tuples are distributed over the bolt's tasks. For example a random distribution is available, which ensures that each tasks receives the same workload. Also a distribution is available that makes sure that all tuples with the same key in the tuple are distributed to the same task.

Storm also has support for other languages. A special bolt implementation called ShellBolt is available. When using this bolt the user can specify the language and a script. If a special adapter for this language is available it can be used to define the internal workings of the bolt. Storm itself comes with support for Python, Ruby and Fancy. When using this the system communicates with the program via stdin/stdout just like with MapReduce and Spark. The difference is that Storm uses JSON instead of raw lines to communicate. This however does mean that the user still has to create a bolt in Java, as simple as it is, to access the bolts created in other languages.

The downside of having to implement the bolts by yourself is that the API is low level. The user has to program the transformations by hand, even if it is a very often used transformation. Storm tries to solve this by introducing the Trident API [29]. The Trident API is a high level abstraction for computation on top of Storm. This means that Trident defines operations like map, filter and count, much like the Spark API. These high level operations are automatically mapped to bolts. Spouts are however still required as no other mean of getting input data is available in the system.

There is however one downside of the system. Each spout and bolt requires the user to specify how many of the executors running that spout or bolt should be started. Even with the Trident API a hint of the parallelism should be given. If it is not specified how many executors should be started the system assumes 1 thread. This is a downside as the amount of executors you want to run of course depends on your cluster set up. This thus requires either recompilation for each cluster, or manually adjusting the amount of executors after start up.

5 Performance

Of course comparing the performance of the three systems is very difficult. Hadoop MapReduce was created for batch jobs, while Storm was created for streaming purposes. There is no good way to compare these systems. Spark however can do batch jobs, and is also suitable for streaming jobs. This means we can compare Hadoop to Spark for batch jobs. We can also compare Spark and Storm for streaming purposes.

5.1 Batch performance comparison

If we just focus on Hadoop and running batch jobs on Spark we would expect Spark to perform better than Hadoop. This simply due to the fact that Spark tries to improve. Spark uses in memory computation of the data, while Hadoop uses file storage. It is a known fact that memory is in most cases faster than disk access, certainly when the file system is distributed.

A relative fair comparison between the two systems is made with the Graysort Daytona benchmark. The goal of this benchmark is to measure the performance of sorting very large data. The performance is given in Terabytes per minute, and has a minimum of 100 TB of data to sort [30].
In 2013 the contest who can achieve the highest TB/min was won by Hadoop. The attempt is described in *GraySort and MinuteSort at Yahoo on Hadoop 0.23* [31]. The attempt was made by using 2100 machines, sorting an impressive amount of 1.42 TB per minute.
This record did not last long, as in 2014 a attempt was made by using Spark. The attempt is described in *GraySort on Apache Spark by Databricks* [32]. It used 206 worker machines, and achieved a sort rate of 4.27 TB/min. This is a speedup of about 3 times over the Hadoop version. However we do have to keep in mind that the implementations for this benchmark are highly optimized. It may be that the performance is very different when using different applications. Also a note to keep in mind is that Hadoop and Spark are still under continuous development. The paper describing the Hadoop implementation dates from May 2013, the Spark paper dates from November 2014. This is already a 1.5 year gap, where the Hadoop system may have improved a lot, and thus changing the speedup of Spark over Hadoop.

Table 1: Graysort Daytona benchmark results for Hadoop and Spark

Platform	**Date**	**# Machines**	**Data size**	**Sort rate**
Hadoop 0.23	May 2013	2100	102.5 TB	1.42 TB/min
Spark 1.2.0	November 2014	206	100 TB	4.27 TB/min

5.2 Streaming performance comparison

A comparison between Spark streaming and Storm is made in the paper *Discretized streams: Fault-tolerant streaming computation at scale* [24]. In this paper three

applications are tested, grep, a word count and finding the k most frequent words. The results are shown in figure 5. The results for 100 byte records indicate that Spark has a throughput of about 5 to 6 times higher than Storm. For 1000 byte records this has gone down to 2 to 3 times more throughput. These results are however a bit questionable, as they are published by the creators of Spark. No implementation details are available except them taking several precautions to improve performance. For example sending batched updates for grep. Also it is not mentioned which versions are used.

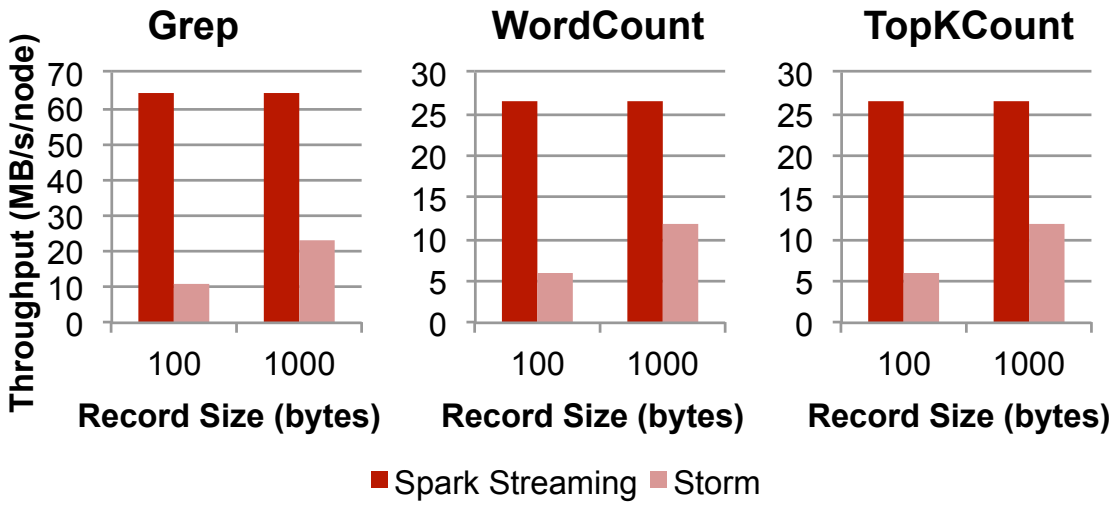

Figure 5: Comparing the throughput of Storm and Spark streaming [24]

6 Conclusion

From the design we can see that the three systems are inherently different. Both Hadoop and Spark focus on batch jobs which can use data parallelism. Hadoop stores all data on a distributed file system, while Spark tries to store it in memory. Storm on the other hand can be used for streaming data, which can utilize task parallelism. Spark also has a feature for Streaming data. This feature is however very different from Storm's design, as it uses micro batching.

All three systems provide a mean to allow the system to continue and recover in case of faults. The system of Hadoop and Spark is a bit stronger as it can ensure that each data element is processed exactly once. The Storm system can only guarantee that each element is processed at least once. It does however have to ability to not use the reliability system at all, whereas it is built in for Hadoop and Spark. Of course the requirements for the reliability depend on the application used.

This brings us on the topic of application development. First of all Spark has by default the biggest choice of language of API. It has an API in four languages,

where Hadoop and Storm stick with only Java. Spark defines a couple of transformations that can be applied on the data. If these transformations require a function, this can often be given in form of a simple lambda expression. Storm also defines transformations in the Trident API. These transformations however do not allow lambda functions. Storm also exposes its core API in the form of spouts and bolts. This allows the user to have full control over the system, and not be tied to fixed transformations. Finally Hadoop exposes the bare mapper and reducer API, which the user has to implement. The most important fact of the API's, and the design, is that Hadoop uses key/value pairs as data elements. Storm uses similar tuples. Spark however has no constraints on the data. This allows for much more flexible programs.

The final aspect of the comparison is the performance of the systems. We have seen some benchmarks of the systems. These indicate that Spark has the best performance for batch jobs and streaming purposes. We however cannot mark Spark as the best performing big data platform. The benchmarks where either outdated, biased or focusing on a single aspect of the system. To create a fair comparison between the system's performance more research in the form of an independent experiment running multiple applications is required.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, Jan. 2008.

[2] Apache Software Foundation, "Apache Hadoop Releases." `http://hadoop.apache.org/releases.html`, December 2015. Accessed: 20-01-2016.

[3] Apache Software Foundation, "The Apache Software Foundation Announces Apache Spark as a Top-Level Project." `https://blogs.apache.org/foundation/entry/the_apache_software_foundation_announces50`, February 2014. Accessed: 20-01-2016.

[4] N. Marz, "History of Apache Storm and lessons learned." `http://nathanmarz.com/blog/history-of-apache-storm-and-lessons-learned.html`, October 2014. Accessed: 20-01-2016.

[5] Apache Software Foundation, "MapReduce tutorial." `http://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html`, April 2013. Accessed: 19-01-2016.

[6] Y. Carmel, "Gfs vs hdfs." `http://www.slideshare.net/YuvalCarmel/gfs-vs-hdfs`, June 2013. Accessed: 20-01-2016.

[7] I. Cloudera, "Introduction to YARN and MapReduce 2." `http://www.slideshare.net/cloudera/introduction-to-yarn-and-mapreduce-2`, December 2013. Accessed: 19-01-2016.

[8] Apache Software Foundation, "Map-Reduce 2.0." https://issues.apache.org/jira/browse/MAPREDUCE-279, January 2008. Accessed: 20-01-2016.

[9] Apache Software Foundation, "MapReduce tutorial." https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html, June 2015. Accessed: 19-01-2016.

[10] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," 2012.

[11] M. Armbrust, T. Das, A. Davidson, A. Ghodsi, A. Or, J. Rosen, I. Stoica, P. Wendell, R. Xin, and M. Zaharia, "Scaling spark in the real world: Performance and usability," *Proc. VLDB Endow.*, vol. 8, pp. 1840–1843, Aug. 2015.

[12] Apache Software Foundation, "Spark programming guide - Spark 1.6.0 documentation." http://spark.apache.org/docs/latest/programming-guide.html. Accessed: 20-01-2016.

[13] J. Richards, "Apache Spark vs. Hadoop MapReduce." http://intersog.com/blog/apache-spark-vs-hadoop-mapreduce/, September 2015. Accessed: 19-01-2016.

[14] Apache Software Foundation, "Cluster mode overview - Spark 1.6.0 documentation." http://spark.apache.org/docs/latest/cluster-overview.html. Accessed: 22-01-2016.

[15] "Spark streaming programming guide." http://spark.apache.org/docs/latest/streaming-programming-guide.html. Accessed: 19-01-2016.

[16] M. T. Jones, "Process real-time big data with Twitter Storm," *IBM Technical Library*, 2013.

[17] K. Ballou, "Apache Storm vs. Apache Spark." http://zdatainc.com/2014/09/apache-storm-apache-spark/. Accessed: 19-01-2016.

[18] Apache Software Foundation, "Apache Storm - Tutorial." http://storm.apache.org/tutorial.html, 2015. Accessed: 19-01-2016.

[19] Apache Software Foundation, "Understanding the Parallelism of a Storm Topology." http://storm.apache.org/documentation/Understanding-the-parallelism-of-a-Storm-topology.html, 2015. Accessed: 19-01-2016.

[20] M. G. Noll, "Understanding the Internal Message Buffers of Storm." http://www.michael-noll.com/blog/2013/06/21/understanding-storm-internal-message-buffers/, June 2013. Accessed: 19-01-2016.

[21] M. Thompson, D. Farley, M. Barker, P. Gee, and A. Stewart, "High performance alternative to bounded queues for exchanging data between concurrent threads," *technical paper, LMAX Exchange*, May 2011.

[22] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*, p. 5, ACM, 2013.

[23] Apache Software Foundation, "Resourcemanager high availability." `https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/ResourceManagerHA.html`, June 2015. Accessed: 25-01-2016.

[24] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 423–438, ACM, 2013.

[25] Apache Software Foundation, "Fault Tolerance." `http://storm.apache.org/documentation/Fault-tolerance.html`, 2015. Accessed: 25-01-2016.

[26] Apache Software Foundation, "Guaranteeing Message Processing." `http://storm.apache.org/documentation/Guaranteeing-message-processing.html`, 2015. Accessed: 19-01-2016.

[27] Apache Software Foundation, "Hadoop streaming." `https://hadoop.apache.org/docs/current/hadoop-streaming/HadoopStreaming.html`, June 2015. Accessed: 26-01-2016.

[28] Apache Software Foundation, "Quick start - Spark 1.6.0 documentation." `http://spark.apache.org/docs/latest/quick-start.html`. Accessed: 19-01-2016.

[29] Apache Software Foundation, "Trident Tutorial." `http://storm.apache.org/documentation/Trident-tutorial.html`, 2015. Accessed: 26-01-2016.

[30] "Sort FAQ." `http://sortbenchmark.org/FAQ-2015.html`, March 2015. Accessed: 27-01-2016.

[31] T. Graves, "GraySort and MinuteSort at Yahoo on Hadoop 0.23," May 2013.

[32] R. Xin, P. Deyhim, A. Ghodsi, X. Meng, and M. Zaharia, "GraySort on Apache Spark by Databricks," 2014.

A Other systems

Of course many other big data systems exists. Below is a list of projects that are (still) being actively developed. Of course many more systems exists, but a lot of them are not developed any more.

Name	**Organization**	**Link**
Prajna	Microsoft	http://msrccs.github.io/Prajna/paper/Prajna_v1.pdf
Flink	Apache	http://flink.apache.org/
Disco	independent	http://discoproject.org/
Esper	EsperTech	http://www.espertech.com/esper/
Samza	Apache	http://samza.apache.org/
S4	Yahoo, Apache	http://incubator.apache.org/s4/
Apex	Apache	http://apex.incubator.apache.org/