**Project3 – Report**

**Team Members**

Megha Modi 1001750561

Nishank Gujar 1001861756

1.What  are  the  key  differences  between  Hadoop  and  Spark, and  their  respective advantages?

Hadoop and Spark are both approaches on distributed computing.  Despite, Hadoop & Spark have some key differences. Here we discuss some of the key differences between Hadoop and Spark.

The difference in storage between Hadoop & Spark leads to another differentiator between the **two frameworks**; the nature of computations they are suitable for. Because Hadoop is entirely built on the MapReduce programming model, it is suitable for tasks that follow the *map* and *reduce* programming model. While many problems can be modeled in this functional approach, not all of them can. Furthermore, the data flow for Hadoop is generally acyclic. In other words, Hadoop generally transforms data in one direction – from the input data to intermediate results from the map workers and finally the output from the reducers.

One other major difference between Hadoop & Spark is in the persisting of **intermediate data**. For intermediate storage, Hadoop relies on buffering intermediate results to the workers' local disks while Spark shares intermediate results in-memory. More specifically, Hadoop, which is based on the MapReduce model, relies on a distributed file system called the Hadoop Distributed File System(HDFS). Intermediate results from the map workers is buffered to map worker's local disk. The reduce workers then read the buffered data from the local disks of the map workers. In contrast, Spark's implementation relies on the use of resilient distributed datasets(RDDs). RDDs are parallel data structures that allows Spark to persist intermediate results in memory. For this reason, Spark has an advantage over Hadoop in that communication between nodes in the cluster is much faster as it is done in memory. Conversely, Hadoop relies on inter-process communication which is more expensive.

On the contrary, Spark is suitable for applications that reuse a working set of data across multiple parallel operations such as **iterative machine learning algorithms** and interactive data analysis tools. This is one of Spark's major advantages over Hadoop. The reason Spark fairs better than Hadoop is due to the fact that intermediate data is stored in memory using the RDDs. As a result, it is possible to build up on the intermediate results without having to fetch entirely from disk.Spark can outperform Hadoop by 10x in iterative machine learning jobs, and can be used to interactively query a 39 GB dataset with sub-second response time.

Another distinction between the two is the roles played by individual nodes in the cluster. Hadoop nodes can either be a map worker, a reduce worker or master (coordinator). Map workers read the corresponding data in their input split and perform operations specified in the map task before finally buffering the intermediate results to memory. Reduce workers read the buffered data from the local disks of the map workers and perform transformations based on the reduce task specified by the user. As the name suggests, the master worker is tasked with the job of scheduling the map and reduce tasks to respective mappers and reducers. In contrast, Spark makes no distinction between nodes in the clusters. All clusters have access to the RDDs and add their transformations accordingly. Spark uses threads to implement the map

Furthermore, there is no single node that is in charge of scheduling. Based on this abstraction and the use of RDDs, Spark tends to have the advantage of being easier to use compared to Hadoop which requires some knowledge of the inner workings of mappers and reducers to get the best results.

2. Discuss how to recover a failed task in Hadoop and Spark, respectively.

Hadoop relies on a heartbeat from worker nodes to detect failing of machines. If the master does not hear from a worker for a while,  it avoids scheduling this worker for future jobs. Since data is replicated in two other nodes, the job scheduler can schedule the jobs for a given input split to another worker which has the same data as the failed node. Even if a task was completed before the node failed, the map tasks will still be executed since the intermediate results are stored in the failed node and therefore inaccessible. Completed reduce tasks need not be computed since the results are available globally.

Since the MapReduce library is designed to help process very large amounts of data using hundreds or thousands of machines, the library must tolerate machine failures gracefully.

Hanging tasks are dealt with differently. The application master notices that it hasn’t received a progress update for a while and proceeds to mark the task as failed. The task will be killed automatically after this period. The timeout period after which tasks are considered failed is normally 10 minutes and can be configured on a per-job basis (or a cluster basis) by setting the mapreduce.task.timeout property to a value in milliseconds.

Setting the timeout to a value of zero disables the timeout, so long-running tasks are never marked as failed. In this case, a hanging task will never free up its container, and over time there may be cluster slowdown as a result. This approach should therefore be avoided, and making sure that a task is reporting progress periodically should suffice.

When the application master is notified of a task attempt that has failed, it will reschedule execution of the task. The application master will try to avoid rescheduling the task on a node manager where it has previously failed. Furthermore, if a task fails four times, it will not be retried again. This value is configurable. The maximum number of attempts to run a task is controlled by the mapreduce.map.maxattempts property for map tasks and mapreduce.reduce.maxattempts for reduce tasks. By default, if any task fails four times ,the whole job fails.

For Spark, all nodes have the same role

Apache spark fault tolerance property means RDD, has a capability of handling if any loss occurs. It can recover the failure itself, here fault refers to **failure**. If any bug or loss found, RDD has the capability to recover the loss.

Spark => Mappers and reducers, are separate threads on the same machine belonging to the same process. Communication is not IPC => they communicate through shared memory => much much faster. There's a shared data structure : resilient distributed dataset. Data is written into the RTD by the mapper, reducer reads from the same file. Downside: Reliability is not guaranteed => if the RTD is corrupted, or the reducer fails, RTD may be lost. There's no way to recover from the RTD. It must be recomputed from the beginning.

Unlike Hadoop's HDFS which relies on writing data to local disks within the distributed workers, RDDs keeps data in cluster nodes' memory.

**References:-**

<https://www.tutorialscampus.com/map-reduce/fault-tolerance.htm>

<https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf>

<https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf>