Assignment 16.2

1. Limitationof Mapreduce

Ans : It’s based on disk based computing.

● Suitable for single pass computations - not iterative computations.

● Needs a sequence of MR jobs to run iterative tasks.

● Needs integration with several other frameworks/tools to solve bigdata usecases. ○ Apache Storm for stream data processing ○ Apache Mahout for machine learning

**Issue with Small Files**

**Hadoop** is not suited for small data. [**(HDFS)** **Hadoop distributed file system**](http://data-flair.training/blogs/comprehensive-hdfs-guide-introduction-architecture-data-read-write-tutorial/) lacks the ability to efficiently support the random reading of small files because of its high capacity design.

Small files are the major problem in HDFS. A small file is significantly smaller than the[**HDFS block**](http://data-flair.training/blogs/data-blocks-hdfs-hadoop-distributed-file-system/)size (default 128MB). If we are storing these huge numbers of small files, HDFS can’t handle these lots of files, as HDFS was designed to work properly with a small number of large files for storing large data sets rather than a large number of small files. If there are too many small files, then the **NameNode** will be overloaded since it stores the namespace of HDFS.

**Solution-**

* Solution to deal with small file issue is simple merge the small files to create bigger files and then copy bigger files to HDFS.
* **HAR files** (Hadoop Archives) were introduced to reduce the problem of lots files putting pressure on the namenode’s memory. By building a layered filesystem on the top of HDFS, HAR files works. Using Hadoop archive command, HAR files are created, which runs a **[MapReduce](http://data-flair.training/blogs/hadoop-mapreduce-introduction-tutorial-comprehensive-guide/)** job to pack the files being archived into a small number of HDFS files. Reading through files in a HAR is not more efficient than reading through files in HDFS. Since each HAR file access requires two index files read as well the data file to read, this makes it slower.
* **Sequence files**work very well in practice to overcome the ‘small file problem’, in which we use the filename as the key and the file contents as the value. By writing a program for files (100 KB), we can put them into a single Sequence file and then we can process them in a streaming fashion operating on the Sequence file. MapReduce can break Sequence file into chunks and operate on each chunk independently because Sequence file is splittable.
* Storing files in **[HBase](http://data-flair.training/blogs/hbase-tutorial-beginners-guide/)**is a very common design pattern to overcome small file problem with HDFS. We are not actually storing millions of small files into HBase, rather adding the binary content of the file to a cell.

**Slow Processing Speed**

In Hadoop, with a parallel and distributed algorithm, MapReduce process large data sets. There are tasks that need to be performed: [**Map**](http://data-flair.training/blogs/mapper-in-hadoop-mapreduce/) and [**Reduce**](http://data-flair.training/blogs/reducer-in-hadoop-mapreduce/)and, MapReduce requires a lot of time to perform these tasks thereby increasing latency. Data is distributed and processed over the cluster in MapReduce which increases the time and reduces processing speed.

**Solution-**

Spark has overcome this issue, by in-memory processing of data. [**In-memory processing**](http://data-flair.training/blogs/apache-spark-in-memory-computing/) is faster as no time is spent in moving the data/processes in and out of the disk. Spark is 100 times faster than MapReduce as it processes everything in memory. Flink is also used, as it processes faster than spark because of its streaming architecture and Flink may be instructed to process only the parts of the data that have actually changed, thus significantly increases the performance of the job.

Refer this guide to [learn MapReduce job optimization and performance tuning techniques.](http://data-flair.training/blogs/mapreduce-job-optimization-performance-tuning-techniques/)

**Support for Batch Processing only**

Hadoop supports batch processing only, it does not process streamed data, and hence overall performance is slower. MapReduce framework of Hadoop does not leverage the memory of the **[Hadoop cluster](http://data-flair.training/blogs/install-hadoop-2-x-ubuntu-hadoop-multi-node-cluster/)** to the maximum.

**Solution-**

Spark improves the performance, but [**Spark stream processing**](http://data-flair.training/blogs/apache-spark-streaming-comprehensive-guide/)is not as much efficient as Flink as it uses micro-batch processing. Flink improves the overall performance as it provides single run-time for the streaming as well as batch processing. Flink uses native closed loop iteration operators which make [**machine learning**](http://data-flair.training/blogs/machine-learning-tutorial/)and graph processing faster.

**No Real-time Data Processing**

Apache Hadoop is designed for batch processing, that means it take a huge amount of data in input, process it and produce the result. Although batch processing is very efficient for processing a high volume of data, but depending on the size of the data being processed and computational power of the system, an output can be delayed significantly. Hadoop is not suitable for Real-time data processing.

**Solution-**

* **Apache Spark** supports stream processing. Stream processing involves continuous input and output of data. It emphasizes on the velocity of the data, and data is processed within a small period of time. Learn more about [Spark Streaming APIs](http://data-flair.training/blogs/apache-spark-streaming-transformation-operations/).
* **Apache Flink** provides single run-time for the streaming as well as batch processing, so one common run-time is utilized for data streaming application and batch processing application. Flink is a stream processing system that is able to process row after row in real time.

**No Delta Iteration**

Hadoop is not so efficient for iterative processing, as Hadoop does not support cyclic data flow(i.e. a chain of stages in which each output of the previous stage is the input to the next stage).

**Solution-**

Apache Spark can be used to overcome this issue, as it accesses data from RAM instead of disk, which dramatically improves the performance of iterative algorithms that access the same dataset repeatedly. Spark iterates its data in batches. For iterative processing in Spark, each iteration has to be scheduled and executed separately.

**Latency**

In Hadoop, MapReduce framework is comparatively slower, since it is designed to support different format, structure and huge volume of data. In **MapReduce**, Map takes a set of data and converts it into another set of data, where individual element are broken down into [**key value pair**](http://data-flair.training/blogs/key-value-pairs-hadoop-mapreduce/) and Reduce takes the output from the map as input and process further and MapReduce requires a lot of time to perform these tasks thereby increasing latency.

**Solution-**

Spark is used to reduce this issue, Apache spark is yet another batch system but it is relatively faster since it caches much of the input data on memory by [**RDD(Resilient Distributed Dataset)**](http://data-flair.training/blogs/rdd-in-apache-spark/)and keeps intermediate data in memory itself. Flink’s data streaming achieves low latency and high throughput.

Refer this guide to [learn how to create RDD in Apache Spark](http://data-flair.training/blogs/how-to-create-rdds-in-apache-spark/).

**Not Easy to Use**

In Hadoop, MapReduce developers need to hand code for each and every operation which makes it very difficult to work. MapReduce has no interactive mode, but adding one such as[**hive**](http://data-flair.training/blogs/apache-hive-tutorial-introductory-guide/)and[**pig**](http://data-flair.training/blogs/apache-pig-tutorial-introduction-guide/)makes working with MapReduce a little easier for adopters.

**Solution-**

While Spark can be used for such issue, Spark has interactive mode so that developers and users alike can have intermediate feedback for queries and other action. Spark is easy to program as it has tons of high-level operators. Flink can also be easily used as it also has high-level operators.

Refer this guide to[learn Apache Spark RDD Transformations and Actions API](http://data-flair.training/blogs/rdd-transformations-actions-apis-apache-spark/).

**Security**

Hadoop can be challenging in managing the complex application. If the user doesn’t know how to enable platform who is managing the platform, your data could be at huge risk. At storage and network levels, Hadoop is missing encryption, which is a major point of concern. Hadoop supports **Kerberos authentication**, which is hard to manage.

HDFS **supports access control lists** (ACLs) and a traditional file permissions model. However, third party vendors have enabled an organization to leverage**Active Directory Kerberos** and**LDAP** for authentication.

**Solution-**

Spark provides security bonus. If we run spark in HDFS, it can use HDFS ACLs and file-level permissions. Additionally, Spark can run on [**YARN**](http://data-flair.training/blogs/hadoop-yarn-tutorial/) giving it the capability of using Kerberos authentication.

**No Abstraction**

Hadoop does not have any type of abstraction so MapReduce developers need to hand code for each and every operation which makes it very difficult to work.

**Solution-**

To overcome this, Spark is used in which for batch we have RDD abstraction. Flink has Dataset abstraction.

**Vulnerable by Nature**

Hadoop is entirely written in **java**, a language most widely used, hence java been most heavily exploited by cyber criminals and as a result, implicated in numerous security breaches.

**No Caching**

Hadoop is not efficient for caching. In Hadoop, MapReduce cannot cache the intermediate data in memory for a further requirement which diminishes the performance of Hadoop.

**Solution-**

Spark and Flink can overcome this, as Spark and Flink cache data in memory for further iterations which enhance the overall performance.

**Lengthy Line of Code**

Hadoop has 1,20,000 line of code, the number of lines produces the number of bugs and it will take more time to execute the program.

**Solution-**

Although Spark and Flink are written in[**scala**](http://data-flair.training/blogs/why-you-should-learn-scala-introductory-tutorial/)and java but they are implemented in Scala, so the number of line of code is lesser than Hadoop. So it will also take less time to execute the program.

**Uncertainty**

Hadoop only ensures that data job is complete, but it’s unable to guarantee when the job will be complete

1. What is RDD? Explain few features of RDD?

**RDD (Resilient Distributed Dataset)** is the fundamental data structure of [**Apache Spark**](http://data-flair.training/blogs/introduction-spark-tutorial-quickstart/) which are an immutable collection of objects which computes on the different node of the cluster. Each and every dataset in Spark RDD is logically partitioned across many servers so that they can be computed on different nodes of the cluster.

**RDD** stands for “**Resilient Distributed Dataset”**. It is the fundamental data structure of Apache Spark. RDD in Apache Spark is an immutable collection of objects which computes on the different node of the cluster.

Decomposing the name RDD:

* **Resilient**, i.e. fault-tolerant with the help of RDD lineage graph(**[DAG](http://data-flair.training/blogs/directed-acyclic-graph-dag-in-apache-spark/)**) and so able to recompute missing or damaged partitions due to node failures.
* **Distributed**,since Data resides on multiple nodes.
* **Dataset**represents records of the data you work with. The user can load the data set externally which can be either JSON file, CSV file, text file or database via JDBC with no specific data structure.

Hence, each and every dataset in RDD is logically partitioned across many servers so that they can be computed on different nodes of the cluster. RDDs are fault tolerant i.e. It posses self-recovery in the case of failure.

There are three [**ways to create RDDs in Spark**](http://data-flair.training/blogs/how-to-create-rdds-in-apache-spark/) such as – *Data in stable storage, other RDDs, and parallelizing already existing collection in driver program*. One can also operate Spark RDDs in parallel with a low-level API that offers *transformations* and *actions*. We will study these Spark RDD Operations later in this section.

Spark RDD can also be **cached** and **manually partitioned**. Caching is beneficial when we use RDD several times. And manual partitioning is important to correctly balance partitions. Generally, smaller partitions allow distributing RDD data more equally, among more executors. Hence, fewer partitions make the work easy.

Programmers can also call a **persist** method to indicate which RDDs they want to reuse in future operations. Spark keeps persistent RDDs [**in memory**](http://data-flair.training/blogs/apache-spark-in-memory-computing/) by default, but it can spill them to disk if there is not enough RAM. Users can also request other persistence strategies, such as storing the RDD only on disk or replicating it across machines, through flags to persist

## Sparkling Features of Spark RDD

There are several advantages of using RDD. Some of them are-

### In-memory computation

The data inside RDD are stored in memory for as long as you want to store. Keeping the data in-memory improves the performance by an order of magnitudes.

### Lazy Evaluation

The data inside RDDs are not evaluated on the go. The changes or the computation is performed only after an action is triggered. Thus, it limits how much work it has to do.

### Fault Tolerance

Upon the failure of worker node, using lineage of operations we can re-compute the lost partition of RDD from the original one. Thus, we can easily recover the lost data.

### Immutability

RDDS are immutable in nature meaning once we create an RDD we can not manipulate it. And if we perform any transformation, it creates new RDD. We achieve consistency through immutability.

### Persistence

We can store the frequently used RDD in in-memory and we can also retrieve them directly from memory without going to disk, this speedup the execution. We can perform Multiple operations on the same data, this happens by storing the data explicitly in memory by calling persist() or cache() function.

### Partitioning

RDD partition the records logically and distributes the data across various nodes in the cluster. The logical divisions are only for processing and internally it has no division. Thus, it provides parallelism.

### Parallel

Rdd, process the data parallelly over the cluster.

### Location-Stickiness

RDDs are capable of defining placement preference to compute partitions. Placement preference refers to information about the location of RDD. The **DAGScheduler**places the partitions in such a way that task is close to data as much as possible. Thus speed up computation.

### Coarse-grained Operation

We apply coarse-grained transformations to RDD**.** Coarse-grained meaning the operation applies to the whole dataset not on an individual element in the data set of RDD.

### Typed

We can have RDD of various types like: RDD [int], RDD [long], RDD [string].

### No limitation

we can have any number of RDD. there is no limit to its number. the limit depends on the size of disk and memory.

1. List down few Spark RDD operations and explain each of them.

s we know Spark RDD is distributed collection of data and it supports two kind of operations on it Transformations and Actions.

###### Apache Spark RDD Operations

* Transformations
* Actions

## Transformation Operations

Transformations are kind of operations which will transform your RDD data from one form to another. And when you apply this operation on any RDD, you will get a new RDD with transformed data (RDDs in Spark are immutable, Remember????). Operations like map, filter, flatMap are transformations.

Now there is a point to be noted here and that is when you apply the transformation on any RDD it will not perform the operation immediately. It will create a DAG(Directed Acyclic Graph) using the applied operation, source RDD and function used for transformation. And it will keep on building this graph using the references till you apply any action operation on the last lined up RDD. That is why the transformation in Spark are lazy.

## Action Operations

This kind of operation will also give you another RDD but this operation will trigger all the lined up transformation on the base RDD (or in the DAG) and than execute the action operation on the last RDD. Operations like collect, count, first, saveAsTextFile are actions.

val logFile = "hdfs://master.backtobazics.com:9000/user/root/sample.txt"

val lineRDD = sc.textFile(logFile)

*//Transformation 1 -> DAG created*

*//{DAG: Start -> [sc.textFile(logFile)]}*

val wordRDD = lineRDD.flatMap(\_.split(" "))

*//Transformation 2 -> wordRDD DAG updated*

*//{DAG: Start -> [sc.textFile(logFile)]*

*// -> [lineRDD.flatMap(\_.split(" "))]}*

val filteredWordRDD = wordRDD.filter(\_.equalsIgnoreCase("the"))

*//Transformation 3 -> filteredWordRDD DAG updated*

*//{DAG: Start -> [sc.textFile(logFile)]*

*// -> [lineRDD.flatMap(\_.split(" "))]*

*// -> [wordRDD.filter(\_.equalsIgnoreCase("the"))]}*

filteredWordRDD.collect

*//Action: collect*

*//Execute DAG & collect result to driver node*