ASSIGNMENT 6.2

1. Limitations of MapReduce:

- **Issue with Small Files**

  Hadoop is not suited for small data. (HDFS) Hadoop distributed file system 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 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 this Drawback of Hadoop to deal with small file issue is simple. Just 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 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 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 and Reduce 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
  As a Solution to this Limitation of Hadoop spark has overcome this issue, by in-memory processing of data. In-memory processing 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.

- **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 to the maximum.

  Solution

  To solve these limitations of Hadoop spark is used that improves the performance, but Spark stream processing 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 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.

  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 type of Limitations of Hadoop, 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 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 limitation of Hadoop, 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) and keeps intermediate data in memory itself. Flink's data streaming achieves low latency and high throughput.

- **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 and pig makes working with MapReduce a little easier for adopters.

  Solution

  To solve this Drawback of Hadoop, we can use spark. 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. This way spark can solve many limitations of Hadoop.

- **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 to overcome these limitations of Hadoop. If we run spark in HDFS, it can use HDFS ACLs and file-level permissions. Additionally, Spark can run on YARN 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 these Drawback of Hadoop, Spark is used in which we have RDD abstraction for batch. 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 limitation of hadoop, 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 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 and solve the lenthy line of code limitations of Hadoop.

- **Uncertainty**

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

2. Resiliant Distributed Datasets(RDD):

- **<u>Explanation:</u>**

  - Resilient Distributed Datasets (RDD) is a fundamental data structure of Spark. It is an immutable distributed collection of objects. Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster. RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes.
  - Formally, an RDD is a read-only, partitioned collection of records. RDDs can be created through deterministic operations on either data on stable storage or other RDDs. RDD is a fault-tolerant collection of elements that can be operated on in parallel.
  - There are two ways to create RDDs − parallelizing an existing collection in your driver program or referencing a dataset in an external storage system, such as a shared file system, HDFS, HBase, or any data source offering a Hadoop Input Format.
  - Spark makes use of the concept of RDD to achieve faster and efficient MapReduce operations

- **<u>Features:</u>**

  - **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.

- **Parallelism**
  Rdd, processes 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.

- **Types**
  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.

3. RDD Operations:

Spark supports two types of operations viz., Actions & Transformations.

<u>RDD Transformations:</u>

Spark Transformation is a function that produces new RDD from the existing RDDs. It takes RDD as input and produces one or more RDD as output. Each time it creates new RDD when we apply any transformation. Thus, the so input RDDs, cannot be changed since RDD are immutable in nature. Applying transformation built an RDD lineage, with the entire parent RDDs of the final RDD(s). RDD lineage, also known as RDD operator graph or RDD dependency graph. It is a logical execution plan i.e., it is Directed Acyclic Graph (DAG) of the entire parent RDDs of RDD.
Transformations are lazy in nature i.e., they get execute when we call an action. They are not executed immediately. Two most basic type of transformations is a map(), filter().
After the transformation, the resultant RDD is always different from its parent RDD. It can be smaller (e.g. filter, count, distinct, sample), bigger (e.g. flatMap(), union(), Cartesian()) or the same size (e.g. map).

There are two types of transformations:
Narrow transformation – In Narrow transformation, all the elements that are required to compute the records in single partition live in the single partition of parent RDD. A limited subset of partition is used to calculate the result. Narrow transformations are the result of map(), filter().
Wide transformation – In wide transformation, all the elements that are required to compute the records in the single partition may live in many partitions of parent RDD. The partition may live in many partitions of parent RDD. Wide transformations are the result of groupbyKey() and reducebyKey().

- **map(func)**
  The map function iterates over every line in RDD and split into new RDD. Using map() transformation we take in any function, and that function is applied to every element of RDD. In the map, we have the flexibility that the input and the return type of RDD may differ from each other. For example, we can have input RDD type as String, after applying the map() function the return RDD can be Boolean.
  For example, in RDD {1, 2, 3, 4, 5} if we apply "rdd.map(x=>x+2)" we will get the result as (3, 4, 5, 6, 7).

- **<u>flatMap()</u>**
  With the help of flatMap() function, to each input element, we have many elements in an output RDD. The most simple use of flatMap() is to split each input string into words.
  Map and flatMap are similar in the way that they take a line from input RDD and apply a function on that line. The key difference between map() and flatMap() is map() returns only one element, while flatMap() can return a list of elements.

- **<u>filter(func)</u>**
  Spark RDD filter() function returns a new RDD, containing only the elements that meet a predicate. It is a narrow operation because it does not shuffle data from one partition to many partitions.

For example, Suppose RDD contains first five natural numbers (1, 2, 3, 4, and 5) and the predicate is check for an even number. The resulting RDD after the filter will contain only the even numbers i.e., 2 and 4.

- **mapPartitions(func)**
  The MapPartition converts each partition of the source RDD into many elements of the result (possibly none). In mapPartition(), the map() function is applied on each partitions simultaneously. MapPartition is like a map, but the difference is it runs separately on each partition(block) of the RDD.

- **mapPartitionWithIndex()**
  It is like mapPartition; Besides mapPartition it provides func with an integer value representing the index of the partition, and the map() is applied on partition index wise one after the other.

RDD Action

Transformations create RDDs from each other, but when we want to work with the actual dataset, at that point action is performed. When the action is triggered after the result, new RDD is not formed like transformation. Thus, Actions are Spark RDD operations that give non-RDD values. The values of action are stored to drivers or to the external storage system. It brings laziness of RDD into motion.
An action is one of the ways of sending data from Executer to the driver. Executors are agents that are responsible for executing a task. While the driver is a JVM process that coordinates workers and execution of the task. Some of the actions of Spark are:

- **count()**
  Action count() returns the number of elements in RDD.
  For example, RDD has values {1, 2, 2, 3, 4, 5, 5, 6} in this RDD “rdd.count()” will give the result 8.

- **collect()**
  The action collect() is the common and simplest operation that returns our entire RDDs content to driver program. The application of collect() is unit testing where the entire RDD is expected to fit in memory. As a result, it makes easy to compare the result of RDD with the expected result. Action Collect() had a constraint that all the data should fit in the machine, and copies to the driver.

- **take(n)**
  The action take(n) returns n number of elements from RDD. It tries to cut the number of partition it accesses, so it represents a biased collection. We cannot presume the order of the elements.
  For example, consider RDD {1, 2, 2, 3, 4, 5, 5, 6} in this RDD “take (4)” will give result { 2, 2, 3, 4}

- **top()**
  If ordering is present in our RDD, then we can extract top elements from our RDD using top(). Action top() use default ordering of data.

- **countByValue()**
  The countByValue() returns, the number of times each element occur in RDD.