Session 16

Assignment 2 Questions

ACADGILD

Problem Statement

1) ***Pen down the limitations of MapReduce.***

- Following are limitation of MapReduce: -
- Batch processing not interactive
- Design for specific problem domain
- MapReduce programming paradigm not commonly understood(functional)
- API/Security model are moving targets
- CReal-time processing.
- It's not always very easy to implement each and everything as a MR program.
- When your intermediate processes need to talk to each other (jobs run in isolation).

- When our processing requires lot of data to be shuffled over the network.
- When we need to handle streaming data. MR is best suited to batch process huge amounts of data which you already have with you.
- When we can get the desired result with a standalone system. It's obviously less painful to configure and manage a standalone system as compared to a distributed system.
- When we have OLTP needs. MR is not suitable for a large number of short on-line transactions.

2) What is RDD? Explain few features of RDD?

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**) 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** 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** 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.

Several features of Apache Spark RDD are:

In-memory Computation

Spark RDDs have a provision of **in-memory computation**. It stores intermediate results in distributed memory(RAM) instead of stable storage(disk).

Lazy Evaluations

All transformations in Apache Spark are lazy, in that they do not compute their results right away. Instead, they just remember the transformations applied to some base data set.

Spark computes transformations when an action requires a result for the driver program. Follow this guide for the deep study of **Spark Lazy Evaluation**.

Fault Tolerance

Spark RDDs are fault tolerant as they track data lineage information to rebuild lost data automatically on failure. They rebuild lost data on

failure using lineage, each RDD remembers how it was created from other datasets (by transformations like a map, join or groupBy) to recreate itself. Follow this guide for the deep study of **RDD Fault Tolerance**.

Immutability

Data is safe to share across processes. It can also be created or retrieved anytime which makes caching, sharing & replication easy. Thus, it is a way to reach consistency in computations.

Partitioning

Partitioning is the fundamental unit of parallelism in Spark RDD. Each partition is one logical division of data which is mutable. One can create a partition through some transformations on existing partitions.

Persistence

Users can state which RDDs they will reuse and choose a storage strategy for them (e.g., in-memory storage or on Disk).

Coarse-grained Operations

It applies to all elements in datasets through maps or filter or group by operation.

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.

RDD in Apache Spark supports two types of operations:

- Transformation
- Actions

Transformations

Spark **RDD Transformations** are *functions* that take an RDD as the input and produce one or many RDDs as the output. They do not change the input RDD (since RDDs are immutable and hence one cannot change it), but always produce one or more new RDDs by applying the computations they represent e.g. Map(), filter(), reduceByKey () etc.

Transformations are **lazy** operations on an RDD in Apache Spark. It creates one or many new RDDs, which executes when an Action occurs. Hence, Transformation creates a new dataset from an existing one.

Certain transformations can be pipelined which is an optimization method, that Spark uses to improve the performance of computations. There are two kinds of transformations: narrow transformation, wide transformation.

Narrow Transformations

It is the result of map, filter and such that the data is from a single partition only, i.e. it is self-sufficient. An output RDD has partitions with records that originate from a single partition in the parent RDD. Only a limited subset of partitions used to calculate the result.

Spark groups narrow transformations as a stage known as **pipelining**.

Wide Transformations

It is the result of groupByKey() and reduceByKey() like functions. The data required to compute the records in a single partition may live in many partitions of the parent RDD. Wide transformations are also known as *shuffle transformations* because they may or may not depend on a shuffle.

Actions

An **Action** in Spark returns final result of RDD computations. It triggers execution using lineage graph to load the data into original RDD, carry out all intermediate transformations and return final results to Driver program or write it out to file system. Lineage graph is dependency graph of all parallel RDDs of RDD.

Actions are RDD operations that produce non-RDD values. They materialize a value in a Spark program. An Action is one of the ways to send result from executors to the driver. First (), take (), reduce (), collect (), the count() is some of the Actions in spark.

Using transformations, one can create RDD from the existing one. But when we want to work with the actual dataset, at that point we use Action. When the Action occurs, it does not create the new RDD, unlike transformation. Thus, actions are RDD operations that give no RDD values. Action stores its value either to drivers or to the external storage system. It brings laziness of RDD into motion.

3) ***List down few Spark RDD operations and explain each of them.***

Transformation

- **map**(*func*)- Return a new distributed dataset formed by passing each element of the source through a function *func*.

- **filter**(*func*) -Return a new dataset formed by selecting those elements of the source on which *func*returns true.

- **flatMap**(*func*)- Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item).

- **mapPartitions**(*func*)- Similar to map, but runs separately on each partition (block) of the RDD, so *func* must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.

- **mapPartitionsWithIndex**(*func*)- Similar to mapPartitions, but also provides *func* with an integer value representing the index of the partition, so *func* must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.

- **sample**(*withReplacement*, *fraction*, *seed*)- Sample a fraction *fraction* of the data, with or without replacement, using a given random number generator seed

- **union**(*otherDataset*)- Return a new dataset that contains the union of the elements in the source dataset and the argument.

- **intersection**(*otherDataset*)- Return a new RDD that contains the intersection of elements in the source dataset and the argument.

- **distinct**([*numTasks*]))- Return a new dataset that contains the distinct elements of the source dataset.

- **groupByKey**([*numTasks*])- When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.
  Note: If you are grouping in order to perform an aggregation (such as a sum or average) over each key,
  using reduceByKey or aggregateByKey will yield much better performance.

Note: By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional numTasks argument to set a different number of tasks.

- **reduceByKey**(*func*, [*numTasks*])- When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) => V. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.

- **aggregateByKey**(*zeroValue*)(*seqOp*, *combOp*, [*numTasks*])- When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type,

while avoiding unnecessary allocations. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.

- **sortByKey**([*ascending*], [*numTasks*])- When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument.

- **join**(*otherDataset*, [*numTasks*])- When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin.

- **cogroup**(*otherDataset*, [*numTasks*])- When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable<V>, Iterable<W>)) tuples. This operation is also called groupWith.

- **cartesian**(*otherDataset*)- When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).

- **pipe**(*command*, *[envVars]*)- Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings.

- **coalesce**(*numPartitions*)-
  Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset.

- **repartition**(*numPartitions*)- Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.

- **repartitionAndSortWithinPartitions**(*partitioner*)- Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. This is more efficient than calling repartition and then sorting within each partition because it can push the sorting down into the shuffle machinery.

Actions

The following table lists some of the common actions supported by Spark.

- **reduce**(*func*)- Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.

- **collect**()-Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.

- **count**()-Return the number of elements in the dataset.

- **first**()-Return the first element of the dataset (similar to take(1)).

- **take**(*n*)- Return an array with the first *n* elements of the dataset.

- **takeSample**(*withReplacement*, *num*, [*seed*])- Return an array with a random sample of *num* elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.

- **takeOrdered**(*n*, *[ordering]*- Return the first *n* elements of the RDD using either their natural order or a custom comparator.

- **saveAsTextFile**(*path*)- Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.

- **saveAsSequenceFile**(*path*)
  (Java and Scala)- Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).

- **saveAsObjectFile**(*path*)
  (Java and Scala)- Write the elements of the dataset in a simple

format using Java serialization, which can then be loaded usingSparkContext.objectFile().

- **countByKey**()-Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.

- **foreach**(*func*)- Run a function *func* on each element of the dataset. This is usually done for side effects such as updating an Accumulator or interacting with external storage systems.