Assignment16.2

1 Limitations of MapReduce

Ans

Spark processes data in memory, while MR persists back to disk, after a map-reduce job.

MapReduce kills its job, as soon as it’s done. Data is mutable if iterative operations performed on the data.

Hadoop MapReduce in java, is difficult to program.

MapReduce does not has an interactive mode.

Hadoop MapReduce is great for batch processing. But if we want real-time options on top of it, you will have to use platforms like Storm and Impala, Giraph - forgraph processing.

MapReduce relies on hard-drives. So if a process crashes in the middle ofexecution, it can carry on from where it left off.

2 What is RDD? Explain features

Ans

Resilient Distributed Datasets are Immutable and partitioned collection of records, which can only be created by reading data from a stable storage like HDFS or by transformations on existing RDD’s.

As RDD’s are created over a set of transformations, it logs these transformations rather than actual data.

In case of we lose some partition of RDD , we can replay the transformation on that partition in lineage to achieve the same computation. This is the biggest benefit of RDD , because it saves a lot of efforts in data management and replication and thus achieves faster computations.

Features of RDD

Resilient, i.e. fault-tolerant with the help of RDD lineage graph and so,able to recompute missing or damaged partitions due to node failures

Distributed with data residing on multiple nodes in a cluster.

Dataset is a collection of partitioned data with primitive values or values of values, e.g. tuples or other objects

Additional Features:-

In-Memory, i.e. data inside RDD is stored in memory as much (size) and long (time) as possible.

Immutable or Read-Only, i.e. it does not change once created and can only be transformed using transformations to new RDDs.

Lazy evaluated, i.e. the data inside RDD is not available or transformed until an action is executed that triggers the execution.

Cacheable, i.e. you can hold all the data in a persistent "storage" like memory (default and the most preferred) or disk (the least preferred due to access speed).

IParallel, i.e. process data in parallel.

Typed — RDD records have types, e.g. Long in RDD[Long] or (Int, String) in RDD[(Int, String)].

Partitioned — records are partitioned (split into logical partitions) and distributed across nodes in a cluster.

Location-Stickiness — RDD can define placement preferences to compute partitions (as close to the records as possible).

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

TRANSFORMATIONS

map(func)

Returns a new distributed dataset, formed by passing each element of the source through a function func.

filter(func)

Returns 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 funcmust be of type Iterator<T> ⇒ Iterator<U> when running on an RDD of type T.

mapPartitionsWithIndex(func)

Similar to map Partitions, 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 of the data, with or without replacement, using a given random number generator seed.

union(otherDataset)

Returns a new dataset that contains the union of the elements in the source dataset and the argument.

intersection(otherDataset)

Returns a new RDD that contains the intersection of elements in the source dataset and the argument.

distinct([numTasks])

Returns 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.

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 from 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 group With.

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

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()

Returns 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()

Returns the number of elements in the dataset.

first()

Returns the first element of the dataset (similar to take (1)).

take(n)

Returns an array with the first n elements of the dataset.

takeSample (withReplacement,num, [seed])

Returns 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])

Returns the first n elements of the RDD using either their natural order or a custom comparator.

saveAsTextFile(path)

Writes 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 calls toString on each element to convert it to a line of text in the file.

saveAsSequenceFile(path) (Java and Scala)

Writes 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)

Writes the elements of the dataset in a simple format using Java serialization, which can then be loaded using SparkContext.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)

Runs 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.

Note − modifying variables other than Accumulators outside of the foreach() may result in undefined behavior. See Understanding closures for more details.