Module-2 Part-2

COMPILED BY,

DR. PRAKASH KALINGRAO AITHAL

Resilient Distributed Dataset

Resilient: RDDs are resilient, meaning that if a node performing an operation in Spark is lost, the dataset can be reconstructed. This is because Spark knows the lineage of each RDD, which is the sequence of steps to create the RDD.

Distributed: RDDs are distributed, meaning the data in RDDs is divided into one or many partitions and distributed as in-memory collections of objects across Worker nodes in the cluster. As mentioned earlier in this chapter, RDDs provide an effective form of shared memory to exchange data between processes (Executors) on different nodes (Workers).

Dataset: RDDs are datasets that consist of records. Records are uniquely identifiable data collections within a dataset. A record can be a collection of fields similar to a row in a table in a relational database, a line of text in a file, or multiple other formats. RDDs are created such that each partition contains a unique set of records and can be operated on independently.

RDD

Another key property of RDDs is their immutability, which means that after they are instantiated and populated with data, they cannot be updated. Instead, new RDDs are created by performing transformations such as map or filter functions, on existing RDDs.

Actions are the other operations performed on RDDs. Actions produce output that can be in the form of data from an RDD returned to a Driver program, or they can save the contents of an RDD to a filesystem (local, HDFS, S3, or other). There are many other actions as well, including returning a count of the number of records in an RDD.

Sample PySpark Program to Search for Errors in Log Files

```
# load log files from local filesystem

logfilesrdd = sc.textFile("file:///var/log/hadoop/hdfs/hadoop-hdfs-*")

# filter log records for errors only

onlyerrorsrdd = logfilesrdd.filter(lambda line: "ERROR" in line)

# save onlyerrorsrdd as a file

onlyerrorsrdd.saveAsTextFile("file:///tmp/onlyerrorsrdd")
```

Loading data to RDD

To start any Spark routine, you need to initialize at least one RDD with data from an external source. This initial RDD is then used to create further intermediate RDDs or the final RDD through a series of transformations and actions. The initial RDD can be created in several ways, including the following:

- Loading data from a file or files
- Loading data from a data source, such as a SQL or NoSQL datastore
- Loading data programmatically
- Loading data from a stream

Creating an RDD from a File or Files

Spark provides API methods to create RDDs from a file, files, or the contents of a directory. Files can be of various formats, from unstructured text files, to semistructured files such as JSON files, to structured data sources such as CSV files. Spark also supports several common serialized binary encoded formats, such as SequenceFiles and protocol buffers (protobufs), as well as columnar file formats such as Parquet and ORC

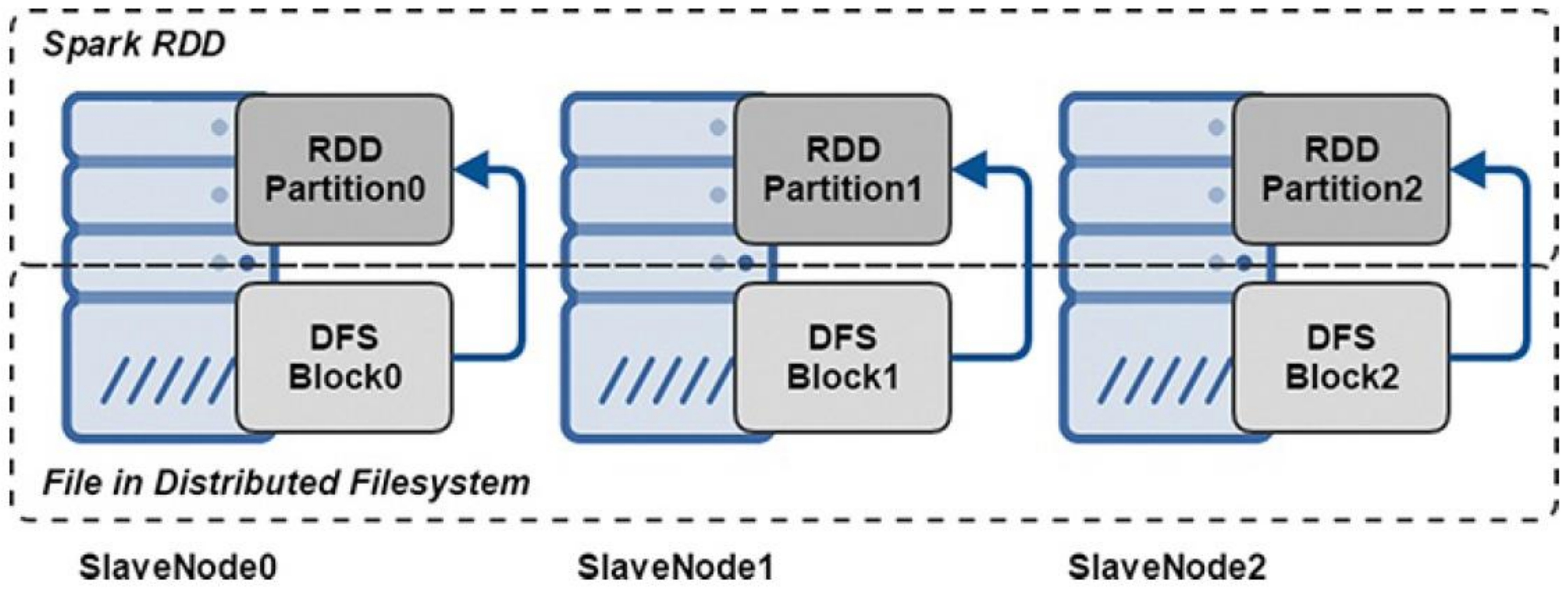
Data Locality with RDD

By default, Spark tries to read data into an RDD from the nodes close to it. Because Spark usually accesses distributed partitioned data, such as data from HDFS or S3, to optimize transformation operations, it creates partitions to hold the underlying blocks from the distributed filesystem.

$\downarrow$

contain the data in rdd

Data Locality with RDD



Methods of creating RDD from text file or files

```
sc.textFile(name, minPartitions=None, use_unicode=True)

# load the contents of the entire directory

logs = sc.textFile("hdfs:///demo/data/Website/Website-Logs/")

# load an individual file

logs = sc.textFile("hdfs:///demo/data/Website/Website-

Logs/IB_WebsiteLog_1001.txt")

# load a file or files using a glob pattern

logs = sc.textFile("hdfs:///demo/data/Website/Website-Logs/*_1001.txt")
```

Methods of creating RDD from text file or files

sc.wholeTextFiles(path, minPartitions=None, use_unicode=True)

The wholeTextFiles() method lets you read a directory containing multiple files. Each file is represented as a record consisting of a key containing the filename and a value containing the contents of the file. In contrast, when reading all files in a directory with the textFile() method, each line of each file represents a separate record with no context of the line’s file origin. Typically with event processing, the originating filename is not required because the record contains a timestamp field.

Comparing textfile()and wholeTextfile() methods

```
# load the contents of the entire directory into an RDD named
licensefiles
licensefiles = sc.textFile("file:///opt/spark/licenses/")
# inspect the object created
licensefiles
# returns:
# file:///opt/spark/licenses/ MapPartitionsRDD[1] at textFile at
```

Comparing textfile()and wholeTextfile() methods

```
# NativeMethodAccessorImpl.java:0
licensefiles.take(1)
# returns a list containing the first line of the first file read in the
directory:
# [u'The MIT License (MIT)']
licensefiles.getNumPartitions()
```

Comparing textfile()and wholeTextfile() methods

# there is a partition created for each file in the directory, in this case the

# return value is 36

licensefiles.count()

# this action will count the combined total number of lines in all the files, in

# this case the return value is 1075

# now let's perform a similar exercise using the same directory using the

# wholeTextFiles() method instead

licensefile_pairs = sc.wholeTextFiles("file:///opt/spark/licenses/") *# inspect the object created*

licensefile_pairs

Comparing textfile()and wholeTextfile() methods

# returns: # org.apache.spark.api.java.JavaPairRDD@3f714d2d

licensefile_pairs.take(1)

# returns the first key/value pair as a list of tuples, with the key being each file # and the value being the entire #contents of that file:

# [(u'file:/opt/spark/licenses/LICENSE-scopt.txt', u'The MIT License (MIT)\n...)..]

licensefile_pairs.getNumPartitions()

# this method will create a single partition (1) containing key/value pairs for each # file in the directory

licensefile_pairs.count()

# this action will count the number of files or key/value pairs in this case return value is 36

Create RDD from object file and data source

Spark supports several common object file implementations. The term *object files* refers to serialized data structures that are not normally human readable and that are designed to provide structure and context to data, making access to data more efficient for the requesting platform.

It is commonly required to load data from a database into an RDD in a Spark program as a source of historical data, master data, or reference or lookup data. This data can come from a variety of host systems and database platforms, including Oracle, MySQL, Postgres, and SQL Server.

Creating RDD from JSON File

```
spark.read.json(path, schema=None)
```

The path argument specifies the full path to the JSON file you are using as a data source. You can use the optional schema argument to specify the target schema for creating the DataFrame.

Creating RDD programmatically

It is possible to create an RDD programmatically from data in your program, whether the data is in lists, arrays, or collections. The data from your collection is partitioned and distributed in much the same way as it is using the previous methods. However, creating RDDs this way can be limiting because it requires all of the dataset to exist or be created in memory on one system.

Creating RDD programmatically

```
sc.parallelize(c, numSlices=None)
```

The parallelize() method assumes that you have a list created already and that you supply it as the c (for collection) argument. The numSlices argument specifies the desired number of partitions to create.

Creating RDD programmatically

parallelrdd = sc.parallelize([0, 1, 2, 3, 4, 5, 6, 7, 8])

rdd <- list

parallelrdd

# notice the type of RDD created:

# ParallelCollectionRDD[0] at parallelize at PythonRDD.scala:423

parallelrdd.count()

# this action will return 9 as this is the number of elements in our collection

parallelrdd.collect()

# will return the parallel collection as a list as follows:

# [0, 1, 2, 3, 4, 5, 6, 7, 8]

Creating RDD programmatically

sc.range(start, end=None, step=1, numSlices=None)

The range() method generates a list for you and creates and distributes the RDD. The start, end, and step arguments define the sequence of values, and numSlices specifies the desired number of partitions.

Creating RDD programmatically

```
# create an RDD with 1000 integers starting at 0 in increments of 1

# across 2 partitions

range_rdd = sc.range(0, 1000, 1, 2)

range_rdd

# note the PythonRDD type, as range is a native Python function

# PythonRDD[1] at RDD at PythonRDD.scala:43

range_rdd.getNumPartitions()

# should return 2 as we requested
```

Creating RDD programmatically

numSlices=2 range_rdd.min()

# should return 0 as this was out start argument

range_rdd.max()

# should return 999 as this is 1000 increments of 1 starting from 0

range_rdd.take(5)

# should return [0, 1, 2, 3, 4]

RDD transformations and actions

Transformations in Spark are functions that operate on an RDD and return a new RDD, whereas *actions* operate against an RDD and return a value or perform an output operation.

RDD transformations and actions

Transformation

```
originalrdd = sc.parallelize([0, 1, 2, 3, 4, 5, 6, 7, 8])
```

```
newrdd = originalrdd.filter(lambda x: x % 2)
```

Action

```
newrdd.collect()
```

Lazy Evaluation

Spark uses *lazy evaluation*, also called *lazy execution*, in processing Spark programs. Lazy evaluation defers processing until an action is called (that is, when output is required). This is easily demonstrated using an interactive shell, where you can enter one or more transformation methods to RDDs one after the other without any processing starting. Instead, each statement is parsed for syntax and object references only. After requesting an action such as count() or saveAsTextFile(), a DAG is created along with logical and physical execution plans. The Driver then orchestrates and manages these plans across Executors.

RDD Persistence and Reuse

RDDs are created and exist predominantly in memory on Executors. By default, RDDs are transient objects that exist only while they are required. After they transform into new RDDs and aren’t needed for any other operations, they are removed permanently. This may be problematic if an RDD is required for more than one action because it must be reevaluated in its entirety each time. An option to address this is to cache or persist the RDD by using the persist() method.

RDD Lineage

Spark keeps track of each RDD’s lineage—that is, the sequence of transformations that resulted in the RDD. As discussed previously, every RDD operation recomputes the entire lineage by default unless RDD persistence is requested.

RDD Lineage

In an RDD's lineage, each RDD has a parent RDD and/or a child RDD. Spark creates a directed acyclic graph (DAG) consisting of dependencies between RDDs. RDDs are processed in stages, which are sets of transformations. RDDs and stages have dependencies that can be narrow or wide.

RDD Lineage Narrow dependencies traits

- Operations can collapse into a single stage; for instance, a map() and filter() operation against elements in the same dataset can be processed in a single pass of each element in the dataset.
- Only one child RDD depends on the parent RDD; for instance, an RDD is created from a text file (the parent RDD), with one child RDD to perform the set of transformations in one stage.
- No shuffling of data between nodes is required.

RDD Lineage Wide dependencies traits

- Operations define new stages and often require shuffling.
- RDDs have multiple dependencies; for instance, a join() operation (covered shortly) requires an RDD to be dependent upon two or more parent RDDs.

Fault Tolerance with RDDs

Spark records the lineage of each RDD, including the lineage of all parent RDDs and parents' parents, and so on. Any RDD with all of its partitions can be reconstructed to the state it was in at the time of the failure, which could have resulted from a node failure, for example. Because RDDs are distributed, they can tolerate and recover from the failure of any single node.

Types of RDDs

PairRDD: An RDD of key/value pairs. You have already seen this type of RDD as it is automatically created by using the wholeTextFiles()

method.

DoubleRDD: An RDD consisting of a collection of double values only. Because the values are of the same numeric type, several additional statistical functions are available, including mean(), sum(), stdev(), variance(), and histogram(), among others.

DataFrame (formerly known as SchemaRDD): A distributed collection of data organized into named and typed columns. A DataFrame is equivalent to a relational table in Spark SQL. DataFrames originated with the read.jdbc() and read.json() functions discussed earlier.

Types of RDDs

SequenceFileRDD: An RDD created from a SequenceFile, either compressed or uncompressed.

HadoopRDD: An RDD that provides core functionality for reading data stored in HDFS using the v1 MapReduce API.

Types of RDDs

NewHadoopRDD: An RDD that provides core functionality for reading data stored in Hadoop—for example, files in HDFS, sources in HBase, or S3—using the new MapReduce API (org.apache.hadoop.mapreduce).

CoGroupedRDD: An RDD that cogroups its parents. For each key in parent RDDs, the resulting RDD contains a tuple with the list of values for that key.

Types of RDDs

JdbcRDD: An RDD resulting from a SQL query to a JDBC connection. It is available in the Scala API only.

PartitionPruningRDD: An RDD used to prune RDD partitions or other partitions to avoid launching tasks on all partitions. For example, if you know the RDD is partitioned by range, and the execution DAG has a filter on the key, you can avoid launching tasks on partitions that don’t have the range covering the key.

Types of RDDs

ShuffledRDD: The resulting RDD from a shuffle, such as repartitioning of data.

UnionRDD: An RDD resulting from a union() operation against two or more RDDs.

There are other RDD variants, including ParallelCollectionRDD and PythonRDD, which are created from the parallelize() and range() functions discussed previously.

References

Jeffery Aven, Data Analytics with Spark using Python, Pearson, 2018