Show Submission Credentials

P4. Introduction to Apache Spark A short introduction to Apache Spark

41 days 23 hours left

Introduction

Introduction

Information

Learning Objectives

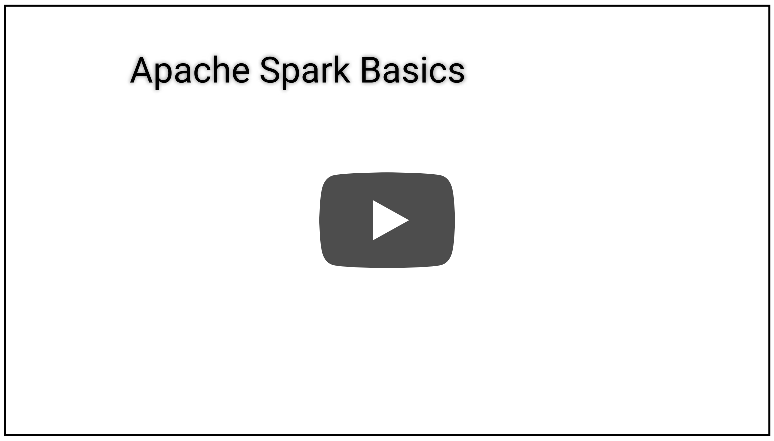
- Discuss the differences between a transformation and an action in the context of a Spark RDD.
- Identify the differences between map() and flatMap() operations in a Spark.
- Recall the differences between the different reduce operations in Spark.
- Explain the usage of the join() operation to merge two RDDs or DataFrames.

- Discuss the benefits of caching an RDD, and identify when it might improve performance.
- Discuss the differences between RDD, DataFrame, and Dataset. Identify the application scenarios for each of the APIs.
- Utilize the DataFrame API to select and aggregate data.

So far, you have used Apache Hadoop's MapReduce programming model in various ways to process large amounts of data. You may have noticed that we have used MapReduce for jobs that are fairly data-parallel, and our analysis would typically require a single Map and Reduce function to finish the job at hand. As MapReduce got more popular over the years, communities working on different types of large-scale data analytics have attempted to use MapReduce for applications that are more complex (such as machine learning tasks) that require multiple iterations of a MapReduce job to be applied in a chained fashion to get the required result. These kinds of computations are known as iterative computations, and implementers started to experience the inherent performance limitations of MapReduce. Recall that MapReduce stores the result to the distributed file system (HDFS) in disk at the end of every job. At the start of the next job, the same data needs to be read back from the distributed file system. Back in 2006 when it was first released, keeping the data in HDFS was a good approach to address fault-tolerance since replicas of the data were available in case of node or even rack failures. However, with the years, as hardware became more reliable and with the growing popularity of Infrastructure as a Service, fault-tolerance could be achieved by keeping data in memory (and taking other precautions for fault-tolerance), and thus accelerate such iterative computation by orders of magnitude. This is what Apache Spark attempts to achieve.

Apache Spark

Spark is an open source cluster computing framework, basically in-memory MapReduce, developed at the UC Berkeley AMPLab. It uses in-memory primitives that allow it to perform over 100x faster than traditional MapReduce for certain applications. The following video covers the basics of Spark:



Video 1: Apache Spark Basics

At the heart of the Spark framework lies a data abstraction called **Resilient Distributed Datasets** (RDDs), which allows for a distributed dataset to remain in-memory in the nodes of a cluster during various stages of computation. As to be fault-tolerant to node failures, Spark also stores the lineage information about the RDD - it keeps a record of all the operations that were performed to bring an RDD to its present state. This way, if a node fails on a Spark cluster, the lost data that was in-memory can be reloaded from the source (often a distributed file system), and the operations that were recorded in the lineage information can be reapplied to bring the data to its present state. Thus, data can remain in memory through multiple stages of transformation without spilling to disk, and applications can potentially run many times faster than traditional frameworks that rely on disk accesses between stages.

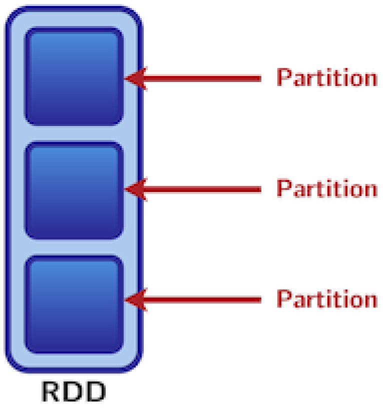


Figure 1: RDD

RDDs are split as partitions. Partitions are atomic pieces of the dataset. There can be one or many per compute node.

Let's take a look at how Spark works with an example. Consider an iterative application that runs a machine learning algorithm on a large graph. Just like distributed Hadoop, in Spark, the master node stores the details of the program to be executed, keeps track of the data in the workers, and has a cluster manager that converts these operations into tasks and executes them on worker nodes. The worker nodes, on the other hand, store the actual data of the graph as a Resilient Distributed Dataset (RDD)(Figure 1). Any cluster requires applications to be scheduled well to maximize the utilization and improve performance.

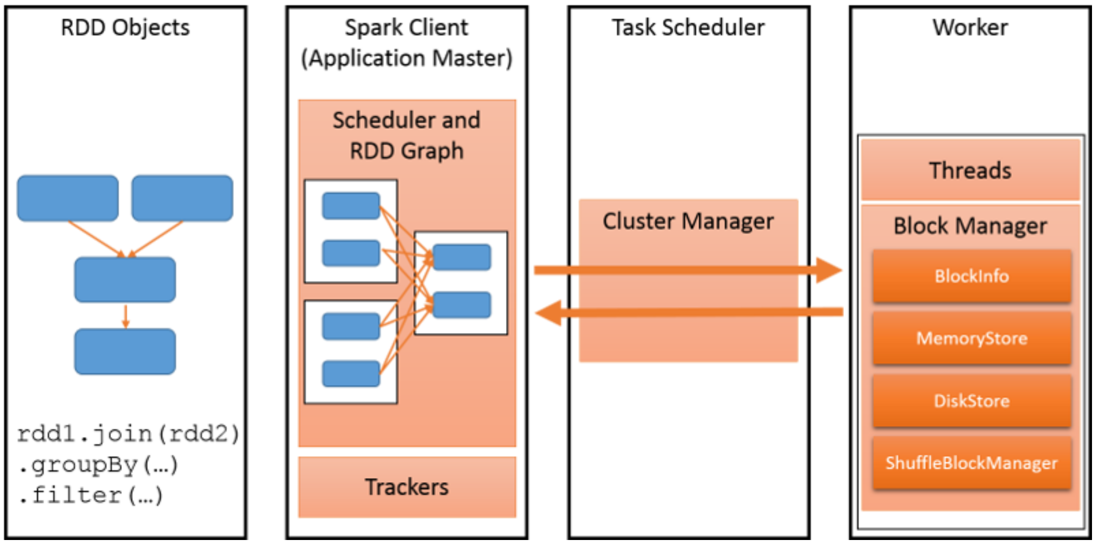


Figure 2: Spark Overview

To use Spark, you need to write a driver program to connect to a Spark cluster of workers. The driver defines one or more RDDs and invokes actions on them. The driver also tracks the RDDs' lineage, which records the history of how this RDD is generated as a Directed Acyclic Graph (DAG). The workers are long-lived processes (running for the entire lifetime of an application) that can store RDD partitions in RAM across operations.

The most important object in Spark is called the `SparkSession`. It can connect to several types of cluster managers that handle the scheduling of applications and tasks (Figure 2). The cluster manager isolates multiple Spark programs from each other - each application has its own driver and runs on isolated executors coordinated by the cluster manager.

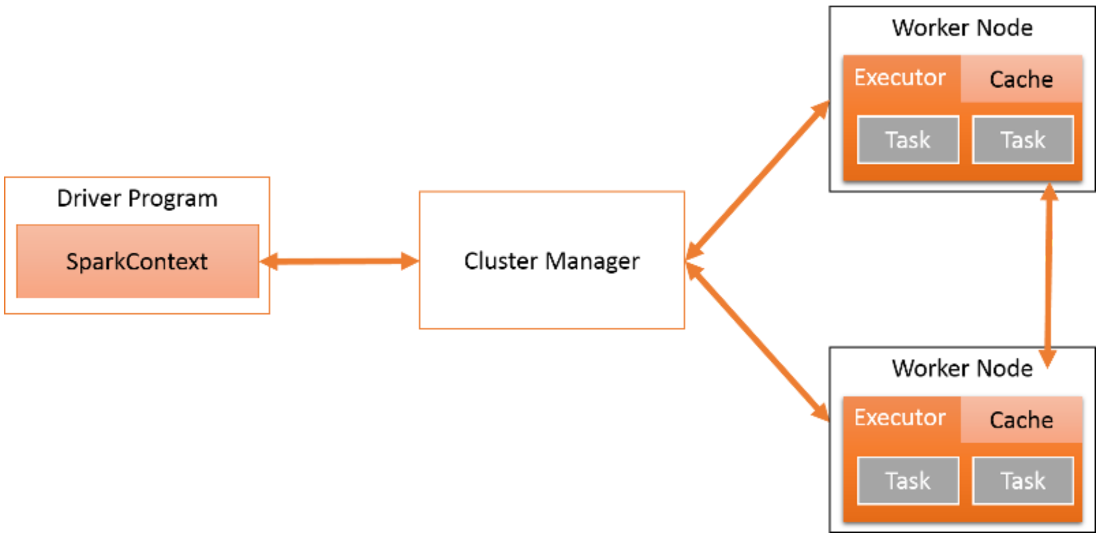


Figure 3: Spark Cluster

Once the `SparkSession` connects to the Cluster Manager, Spark acquires executors on the worker nodes, which are the actual processes that run computation and store data. After an executor is acquired, the program is sent to the executor and run as tasks. Note that each application has its own executor processes, which run tasks in multiple threads. The executor exists for the entire application lifecycle.

An advantage of this approach is that applications are isolated from each other. Scheduling decisions are made by individual drivers independent of other applications. Also, executors for different applications are isolated and each one runs in a separate JVM. The disadvantage is that it is more difficult to share data between applications.

Each Spark application runs as an independent set of processes on a distributed cluster. The driver is the process that runs the `main()` function of the application and creates a `SparkSession` object. Spark applications are coordinated by the `SparkSession` object. The `SparkSession` in turn connects to a Cluster Manager, which allocates resources across all applications on the cluster.

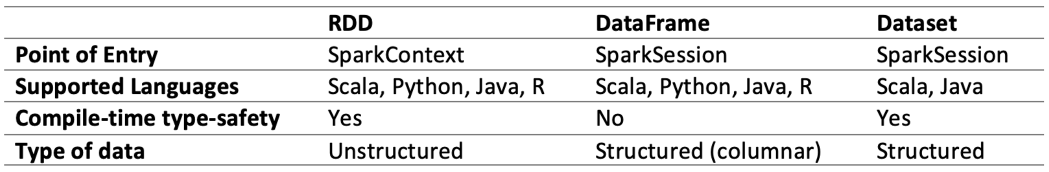
Spark APIs

Apache Spark APIs

You can work with Spark on Scala, Java, Python and R. Given that Spark's source code is written in Scala, it is the fastest of the four.

What is more, Spark has three APIs for interacting with data: RDDs, DataFrames and Datasets.

In the following sections we will go into more detail on how these APIs can be used, and when should you use one or the other. However, here is a general overview:



	RDD	**DataFrame**	**Dataset**
Point of Entry	SparkContext	SparkSession	SparkSession
Supported Languages	Scala, Python, Java, R	Scala, Python, Java, R	Scala, Java
Compile-time type-safety	Yes	No	Yes
Type of data	Unstructured	Structured (columnar)	Structured

Figure 4: Spark API Comparison

Launching a Spark Cluster

Launching Spark

Danger

When launching resources for testing, make sure to tag them with the appropriate tags specified in the ongoing project as to avoid tagging penalties.

Make sure to also delete all resources once you are done testing to not incur in budget penalties.

Azure HDInsight

Launch a Spark cluster on Azure HDInsight; you can refer to Create an HDInsight Spark cluster (https://docs.microsoft.com/en-us/azure/hdinsight/spark/apache-spark-jupyter-spark-sql-use-portal#create-an-hdinsight-spark-cluster) for detailed instructions. For your convenience, we have listed the following steps:

1. Make sure you have an active subscription by checking the Azure portal (https://portal.azure.com/#blade/Microsoft_Azure_Billing/SubscriptionsBlade).
2. In the Azure portal, select **Create a resource** > **Analytics** > **Azure HDInsight**.
3. Under **Basics**, expand **Cluster Type**, and then select **Spark** as the cluster type, and specify the Spark cluster version **Spark 2.4 (HDI 4.0)**. Set other options such as **Cluster name**, **Subscription**, **Resource group**, **Location** and **Login Password**. Click **Next** to continue to the **Storage** page.
4. Under **Storage**, **Select a Storage account**: select **Create new**, and then give a name to the new storage account. The **Default container** has a default name. You can change the name if you want. Under Tags, make sure to tag your cluster accordingly (tags associated with the current ongoing project). Select **Next** to continue to the **Summary** page.
5. Under **Configuration + pricing** choose the number of worker nodes to use. For this primer 1 worker will do.
6. Under **Tags** make sure to tag your resources with the appropriate tags.
7. On **Summary**, select **Create**. It takes about 20 minutes to create the cluster.

There are different ways of running a Spark application:

1. After the cluster is created, you can SSH into the master node and initialize the `spark-shell` for Java or Scala users:

   ```
   $ spark-shell
   ```

   or `pyspark` for Python users

   ```
   $ pyspark
   ```

2. From within the cluster you can use `spark-submit` to submit your standalone Spark job.

   Java or Scala applications

   ```
   $ spark-submit --class <class name> <path_to_jar> <optional arguments>
   ```

   Python applications

   ```
   $ spark-submit <path_to_python_file>
   ```

   See Submitting Applications (https://spark.apache.org/docs/latest/submitting-applications) for more details or use the `--help` option to see all options. You can refer to the bottom of this section for more information on building standalone Spark programs.

3. On the resources page of your **HDInsight** cluster, click on Zeppelin notebook link. You can create a Scala or Python notebook and run Spark on both of them. Refer to the Zeppelin for Apache Spark Primer.

Spark Standalone Mode

In addition to running on clusters, Spark can also be run on a standalone machine. You can do this on the cloud platform of your choice (or locally), and are highly encouraged to test and develop your code on a standalone machine before spinning up a cluster to run it.

Information

If you prefer AWS, you may use a t3.micro to save your budget. You need to properly tag your resources. The required tags are specified in the ongoing project.

The following instructions are for setting up Spark on an Ubuntu 16.04 LTS VM.

1. SSH to the VM and update the package index

```
$ sudo apt-get update
```

2. Install Java 8

```
$ sudo apt install openjdk-8-jre-headless
```

3. Install Scala 2.11

```
$ wget http://www.scala-lang.org/files/archive/scala-2.11.7.tgz
$ tar xvf scala-2.11.7.tgz
$ sudo mv scala-2.11.7 /usr/lib
$ sudo ln -s /usr/lib/scala-2.11.7 /usr/lib/scala
$ echo "export PATH=$PATH:/usr/lib/scala/bin" >> ~/.bashrc
$ source ~/.bashrc
$ rm scala-2.11.7.tgz
```

4. Install Python 3

```
$ sudo apt install python3
```

5. Install Apache Spark. You can find a list of Spark releases here (https://spark.apache.org/downloads.html) and download archived versions here (https://archive.apache.org/dist/spark/).

```
$ wget https://archive.apache.org/dist/spark/spark-2.4.5/spark-2.4.5-bin-ha
doop2.7.tgz
$ tar -xvf spark-2.4.5-bin-hadoop2.7.tgz
$ echo "export PATH=$PATH:/home/ubuntu/spark-2.4.5-bin-hadoop2.7/bin" >>
~/.bashrc
$ source ~/.bashrc
```

6. Run a sample program

```
$ run-example SparkPi
...
Pi is roughly 3.1400557002785012
...
```

Warning

Spark runs on Scala 2.11 and Java 1.8. Make sure to use those versions if you are installing it in a different environment.

Spark Standalone Applications

As we mentioned earlier, `spark-submit` can be used to launch custom standalone programs. This section intends to serve as an overview for how to create such programs in Scala, Java and Python.

The main difference between writing your own Spark programs and using the Spark shell directly is that you have to explicitly initialize your own `SparkSession` in the former case. You need to create a SparkSession (http://spark.apache.org/docs/latest/sql-programming-guide.html#starting-point-sparksession) when you are writing your own program.

On `spark-shell` , `pypark` and the notebooks the `SparkSession` is initialized as `spark` (there is no need to manually initialize it).

Scala

You should probably be using a package manager such as `maven` .

Maven

Include the following dependency in `pom.xml`

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-core_2.11</artifactId>
    <version>2.4.5</version>
</dependency>
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-sql_2.11</artifactId>
    <version>2.4.5</version>
</dependency>
```

Make sure the version matches that of your cluster.

Initialize the SparkSession

Scala

```
import org.apache.spark.sql.SparkSession

val spark = SparkSession
   .builder()
   .appName("SparkSessionExample")
   .getOrCreate()
```

In Standalone local mode you need to set the Spark master URL to connect to local:

```
import org.apache.spark.sql.SparkSession

val spark = SparkSession
   .builder()
   .master("local[*]")
   .appName("SparkSessionExample")
   .getOrCreate()
```

`[*]` specifies that it should use all cores in the machine

Java

```
import org.apache.spark.sql.SparkSession;

SparkSession spark = SparkSession
            .builder()
            .master("local[*]")
            .appName("SparkSessionExample")
            .getOrCreate();
```

Python

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \
                    .master("local[*]") \
                    .appName("Word Count") \
                    .config("spark.some.config.option", "SparkSessionExample") \
                    .getOrCreate()
```

RDD Operations

RDD Operations

As described in the previous section, computation in Apache Spark involves the manipulation of in-memory data stored in the form of RDDs. RDDs support two types of operations: **transformations** and **actions**.

Information

RDD Transformations vs Actions

RDDs support two types of operations:

1. Transformations, which create a new dataset from an existing one
2. Actions, which return a value to the driver program after running a computation on the dataset.

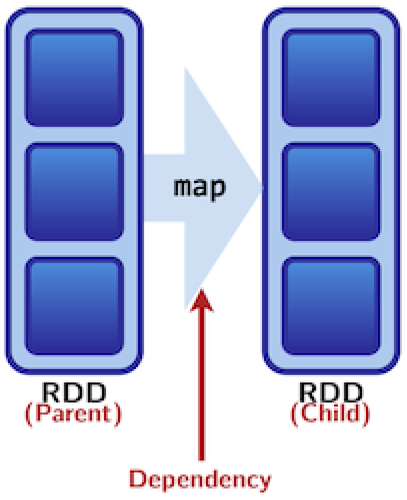
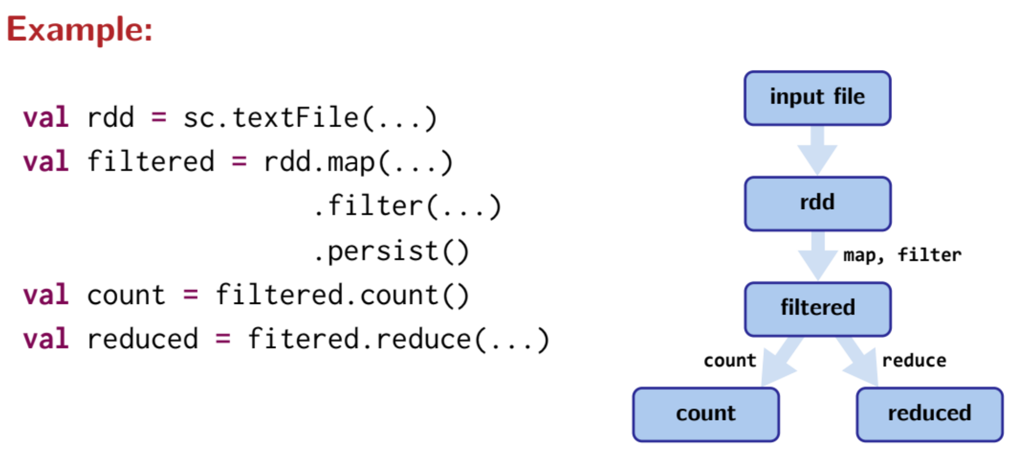


Figure 5: A transformation produces a new RDD.

All transformations in Spark are lazy: they do not compute their results straight-away. Rather, they remember the transformations applied to some base dataset, which are computed when an action requires a result to be returned to the driver program. By lazy evaluation, Spark can make many optimization decisions after it had a chance to look at the DAG in entirety, and is able to be fault tolerant - if a node fails, it can recompute all steps to produce the same data.


Example:

```
val rdd = sc.textFile(...)
val filtered = rdd.map(...)
                  .filter(...)
                  .persist()
val count = filtered.count()
val reduced = fitered.reduce(...)
```

Figure 6: DAG of Spark code.

By default, each transformed RDD may be recomputed each time you run an action on it. However, you may persist an RDD in memory using `persist` (or `cache`), in which case Spark will keep the elements on the cluster for faster subsequent accesses.

Here are some common RDD transformations (http://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations) and actions (http://spark.apache.org/docs/latest/rdd-programming-guide.html#actions).

Note: The default place to read a file in a Hadoop cluster is HDFS (or WASB on Azure); you should upload files to HDFS before you read the file, or you can use `file:///absolute/path/to/file` to let Spark read a file from the local file system. On AWS you can read/write data directly from s3 by providing a path as `s3://...` (as long as the cluster has the necessary read permissions).

You can go through the following examples in a small Spark cluster or VM, on your local machine, or on a notebook. The **Launching Spark** section above contains the steps for setting up a Spark environment.

RDD Operations

The RDD API can be accessed from the SparkContext. You can get the SparkContext from the SparkSession by doing

Scala:

```
val sc = spark.sparkContext // Note that spark is a SparkSession object
```

Java:

```
import org.apache.spark.SparkContext;

SparkContext sc = spark.sparkContext(); // Note that spark is a SparkSession obj
ect
```

Python:

```
sc = spark.sparkContext # Note that spark is a SparkSession object
```

Common RDD Operations

As you may have noticed, the RDD API makes use of higher order functions. One common RDD operation is `map`. The `map` operation on an RDD takes in a function and applies it to every element in the RDD.

The following `map` example takes in a value of type `RDD[Int]` and squares every element in the RDD.

Information

In the examples we use the function parallelize which distributes a local collection into an RDD.

Scala:

```
val data = Array(1, 2, 3, 4, 5)
val distData = sc.parallelize(data)
val squared = distData.map(x => x * x)
```

Java:

```
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.Function;
import java.util.Arrays;
import java.util.List;

JavaSparkContext jsc = JavaSparkContext.fromSparkContext(sc);

List<Integer> data = Arrays.asList(1, 2, 3, 4);
JavaRDD<Integer> distData = jsc.parallelize(data);
JavaRDD<Integer> squared = rdd.map(new Function<Integer, Integer>() {
  public Integer call(Integer x) { return x * x; }
});
```

Python:

```
data = [1, 2, 3, 4]
distData = sc.parallelize(data)
squared = distData.map(lambda x: x * x)
```

PairRDD operations on Key-Value Pairs

Spark has a number of operations that are designed for key-value pairs. These are called PairRDD (just an `RDD[(K, V)]`) operations are applied on each key/element in parallel. Some of the common key-value pair transformations include `reduceByKey` , `groupByKey` , `aggregateByKey` and `combineByKey` .

These transformations take in arguments of (K, V) pairs, but evaluate to values of different types. While many operations can be performed using any of the three transformations, they are liable to have a large difference in performance. In situations where you are grouping to aggregate data (e.g. summing over keys), using `reduceByKey` , `aggregateByKey` or `combineByKey` will be more performant than `groupByKey` , but can only be used if the operation is both commutative and associative.

In the following examples, we will go through how to find the number of occurrences of words in a text. Suppose we are applying each of the transformations to `String` . To get a PairRDD you first need to convert it to a `(String, Int)` pair. If you are following along in `spark-shell` , `pyspark` or a notebook, you can start by initializing an RDD as follows:

Scala:

```
val data = sc.parallelize(Array("hello", "world", "hello", "15619", "is", "aweso
me")).map { word => (word, 1)}
```

Java:

```
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.Function;
import scala.Tuple2;

import java.util.Arrays;

JavaSparkContext jsc = JavaSparkContext.fromSparkContext(spark.sparkContext());
JavaRDD<Tuple2<String, Integer>> data = jsc.parallelize(
        Arrays.asList("hello", "world", "hello", "15619", "is", "awesome")).map(
        new Function<String, Tuple2<String, Integer>>() {
          public Tuple2<String, Integer> call(String word) {
            return Tuple2.apply(word, 1);
          }
        });
```

Python:

```
data = sc.parallelize(["hello", "world", "hello", "15619", "is", "awesome"]).map
(lambda x: (x, 1))
```

groupByKey

`groupByKey` works very much like MapReduce. It aggregates results into an Iterable of the value. The downside is that data is sent over the network, without reducing it first on each worker, which might cause out of memory issues. If the task is not commutative and associative, results cannot be aggregated before shuffling, and they need to be sent to the reducers for aggregation (similar to the Combiner in MapReduce). Therefore, groupByKey should be used.

The function definition is as follow:

`groupByKey(): RDD[(K, Iterable[V])]`

Scala:

```
$ val count = data.groupByKey().map(x => (x._1, x._2.sum))
$ count.collect().foreach(println)
(is,1)
(15619,1)
(hello,2)
...
```

Python:

```
$ data.groupByKey().mapValues(sum).collect()
[('world', 1), ('hello', 2), ('is', 1), ('awesome', 1), ('15619', 1)]
```

reduceByKey

If the task is commutative and associative, then `reduceByKey` can be used. It reduces the results on each worker before sending them to the reducers avoiding potential out of memory issues.

The function definition is as follow:

`reduceByKey(func: (V, V) => V, [numTasks]): RDD[(K, V)]`

Here, `func` has type `(V, V) => V`. `func` aggregates the values for each key separately. And `numTasks` represents the number of reduce tasks which is configurable.

The function transforms an `RDD[(K, V)] => RDD[(K, V)]`.

Scala:

```
$ def add(left: Int, right: Int): Int = left + right
$ val count = data.reduceByKey(add)
$ count.collect().foreach(println)
(is,1)
(15619,1)
(hello,2)
...

// Note that you could have used Scala's Placeholder syntax:
$ val count = data.reduceByKey(_ + _)
```

Python:

```
$ data = data.reduceByKey(lambda (x, y): x + y)
$ data.collect()
[('world', 1), ('hello', 2), ('is', 1), ('awesome', 1), ('15619', 1)]
```

aggregateByKey

`aggregateByKey` works similarly to `reduceByKey` but one can specify an arbitrary initial value (it can be a different type from the original).

The function definition is as follow:

`aggregateByKey(zeroValue: U)(seqOp: (U, V) => U, combOp: (U, U) => U, [numTasks]): RDD[(K, U)])`

Using the analogy of MapReduce: 1. `seqOp` is called in the mappers to add a new value 2. `combOp` is used to combine different combined values

Scala:

```
$ val count = data.aggregateByKey(0)((x, y) => x + 1, (a, b) => a + b)
$ count.collect().foreach(println)
(is,1)
(15619,1)
(hello,2)
...
```

Python:

```
$ data.aggregateByKey(0, lambda x, y: x + 1, lambda a, b: a + b).collect()

[('world', 1), ('hello', 2), ('is', 1), ('awesome', 1), ('15619', 1)]
```

combineByKey

`combineByKey` works similarly to `aggregateByKey`, but instead of having a fixed value for all keys one can specify a function that returns the initial value.

`combineByKey(createCombiner: (V) ? U, mergeValue: (U, V) ? U, mergeCombiners: (U, U) ? U): RDD[(K, U)]`

Scala:

```
$ val count = data.combineByKey((z: Int) => z, (x: Int, y: Int) => x + 1, (a: In
t, b: Int) => a + b)
$ count.collect().foreach(println)
(is,1)
(15619,1)
(hello,2)
...
```

Python:

```
$ data.combineByKey(lambda z: z, lambda x, y: x + 1, lambda a, b: a + b).collect
()
[('world', 1), ('hello', 2), ('is', 1), ('awesome', 1), ('15619', 1)]
```

Joining RDDs

Spark offers numerous types of joins: `join` is an inner join between two PairRDDs. Outer joins are supported through `leftOuterJoin` , `rightOuterJoin` and `fullOuterJoin` . The definitions of these transformations are similar to those in SQL commands.

The function definition is as follow:

`join[W](other: RDD[(K, W)]): RDD[(K, (V, W))]`

Therefore, given two RRDs of type `RDD[(K, V)]` and `RDD[(K, W)]` , it returns of type `RDD[(K, (V, W))]` that is the INNER JOIN of both RDDs.

Using pattern matching, the resulting RDD can be transformed as follows:

```
rdd1.join(rdd2).map { case (key, (left, right)) =>
    ...
}
```

Spark also has a number of set-like transformations, such as `subtract` , `union` and `intersection` . Check out the official Scala (http://spark.apache.org/docs/latest/api/scala/scala/index.html) and Python (http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD) documentation for the full list of available RDD API transformations.

Wordcount with Spark

No big data framework primer would be complete without a walkthrough of WordCount. Here's what it looks like with Spark. If you are working through the examples on `spark-shell` , here is a file that you can use to get started.

Firstly, download the sample data:

```
$ wget https://www.gutenberg.org/files/1112/1112.txt
```

If you are working with a cluster, you would need to load the file into HDFS:

```
$ hdfs dfs -mkdir /input
$ hdfs dfs -put 1112.txt /input
```

Scala:

```
val file = sc.textFile("input/1112.txt") // change the path if working locally
val words = file.flatMap(line => line.split(" "))
val pairs = words.map(word => (word, 1))
val counts = pairs.reduceByKey(_+_)
counts.saveAsTextFile("output")
```

Java:

```
JavaRDD<String> file = jsc.textFile("hdfs:///input/1112.txt");
JavaRDD<String> words = file.flatMap(new FlatMapFunction<String, String>() {
  public Iterator<String> call(String s) throws Exception {
    return Arrays.asList(s.split(" ")).iterator();
  }
});
JavaPairRDD<String, Integer> pairs =
    words.mapToPair(new PairFunction<String, String, Integer>() {
  public Tuple2<String, Integer> call(String s) {
    return new Tuple2<String, Integer>(s, 1);
  }
});
JavaPairRDD<String, Integer> counts =
    pairs.reduceByKey(new Function2<Integer,
        Integer,
        Integer>() {
  public Integer call(Integer a, Integer b) {
    return a + b;
  }
});

counts.saveAsTextFile("hdfs:///output");
```

Python:

```
file = sc.textFile("hdfs:///input/1112.txt")
words = file.flatMap(lambda line: line.split(" "))
pairs = words.map(lambda word: (word, 1))
counts = pairs.reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs:///output")
```

The walk-through of the code is as follows and is also illustrated in Figure 7:

1. `SparkContext` will be initialized as `sc` in `spark-shell` or `pyspark`. You need to initialize Spark (https://spark.apache.org/docs/latest/programming-guide#initializing-spark) by creating a `SparkSession` object when you are writing your own program, and accessing the `SparkContext` as `spark.sparkContext`.
2. First, we load the text files at the HDFS path (say `/input`) by `textFile()` as an RDD. This RDD would represent the entire string contents of the file.
3. Next, we split each line using space to get all the words in the line. We use the `flatMap()` function here, because we expect to get an RDD of words from the RDD of

lines and each line can have multiple words.

4. We now need to emit the count 1 along with each word. So we use `map()` to transform the RDD of words into an RDD of (word, count) pairs. Steps 2 and 3 are equivalent to the mapper in MapReduce.
5. We can now sum the count for each word by using `reduceByKey()` . This transforms the RDD of (word, count) pair where count = 1, into the final RDD of (word, count) pair where count is the number of times the word appears. Step 4 is equivalent to the reducer in MapReduce.
6. Finally, we can persist the RDD as plain text files to the HDFS path (say `/output`).

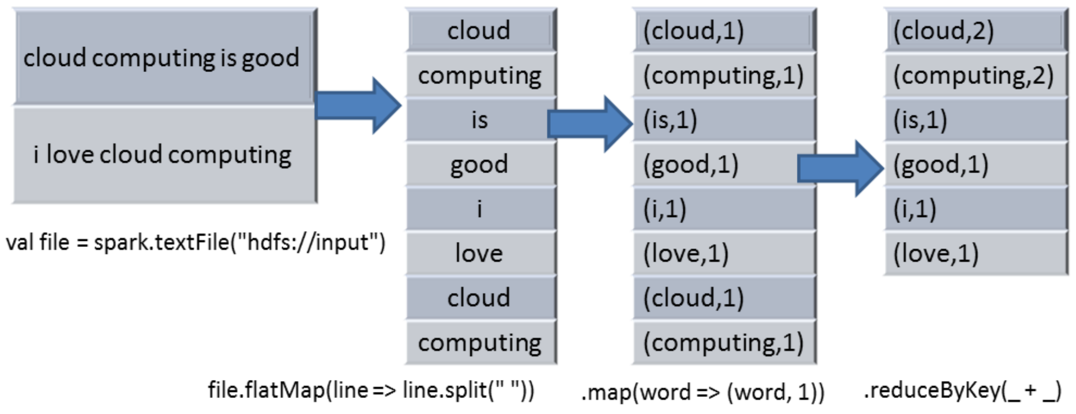


Figure 7: Wordcount Example in Spark

For more Spark code examples, please visit Apache Spark Examples (https://spark.apache.org/examples.html) and Spark Examples (https://github.com/apache/spark/tree/master/examples/src/main/scala/org/apache/spark/examples). The latter can also be found in your downloaded Spark distribution.

In this example, `flatMap()` , `map()` , and `reduceByKey()` are **transformations**, which create a new dataset from an existing one (Figure 5). `saveAsTextFile()` is an **action**, which returns a value to the driver program after running a computation on the dataset.

Map and Reduce functions in Spark (Scala)

Just like MapReduce, Spark code often uses `map()` and `reduce()` primitives, allowing you to apply a single function to an entire RDD (Notice that we also learned about `map()` method in the Scala primer, you need distinguish between Scala Collection and RDD). You may have noticed the two variants: `flatMap()` and `reduceByKey()` .

Similar to Scala's `flatMap()` , Spark's `flatMap()` has the same logic. While `map` is a one-to-one mapping, `flatMap` allows for a one-to-many mapping. Think of it as two steps: 1) do the mapping with the mapper, which generates an RDD collection of list; 2) flatten the RDD collection of list to an RDD collection of the object in the list.

Try the following example using the spark shell. To see the result, use `collect()` to aggregate all the RDD data on the master node as an Array.

```
val data = Array(1, 2, 3, 4, 5)
val rdd = sc.parallelize(data)
rdd.map(x => List(x, x + 1)).collect()
// Array(List(1, 2), List(2, 3), List(3, 4), List(4, 5), List(5, 6))
rdd.flatMap(x => List(x, x + 1)).collect()
// Array(1, 2, 2, 3, 3, 4, 4, 5, 5, 6)
```

Caching and Persisting

RDDs are lazily evaluated and are NOT data per se. Spark defers evaluation of RDDs to when an action is called. The evaluation may consist of a series of transformations. To create a new RDD from an existing RDD, the new RDD carries a pointer to the parent RDD. The dependency graph of all the parent RDDs of an RDD is called an RDD lineage. When an RDD action is called, the action triggers the Spark scheduler to build a directed acyclic graph (DAG) based on RDD transformations and Spark evaluates the RDD based on the DAG. When another action is called on the RDD, Spark has to re-evaluate the RDD from scratch based on the DAG unless the RDD has been materialized via cache() or persist().

You probably should not call cache() if there is only one action of your Spark job. Blindly overusing cache() or persist() increases memory pressure and slows down the job. You should NOT cache the following job, where the RDD lineage is linear with no branches.

```
val textFile = sc.textFile("...")
val numberRDD = textFile.flatMap(line => line.split("\t"))
numberRDD.cache() # this is a bad example, you should NOT cache the RDD
val numberCount = rdd.count()
```

The following job will benefit from caching, where the RDD lineage branches out.

```
val textFile = sc.textFile("...")
val numberRDD = textFile.flatMap(line => line.split("\t"))
numberRDD.cache() # this is a good example
val positiveCount = numberRDD.filter(number => number > 0).count()
val negativeCount = numberRDD.filter(number => number < 0).count()
```

Information

Shared Variables

Spark transformations can be viewed as pure functional mappings from one RDD to another. During such processes, there should be no communication among nodes. This mechanism makes Spark more efficient and fault tolerant. Supporting general, read-write shared variables across tasks would be inefficient. Instead, Spark provides an abstraction for sharing variables: Broadcast Variables (https://spark.apache.org/docs/2.2.0/rdd-programming-guide.html#broadcast-variables).

To understand Broadcast variables, you have to remember that we are working in a distributed setting. Say you had a `Map[String, String]` that is used in a transformation. As the `Map` is a non-distributed Scala object, it exists on the driver node, and it is sent to the worker nodes with each transformation of the dataset. Therefore, it gets potentially sent once per number of records in the dataset.

Given that it is wasteful having to send it each time, Broadcasts provide an abstraction that wraps around structures. It sends a copy of the structure to each worker node once, and the copy is kept in each worker's memory as long as the broadcast variable is not destroyed. Therefore, if the worker wants to access a value, it can access it from within its local copy rather than having to request a copy from the master node.

Broadcast variables have some limitations though. These structures have to remain immutable, as a copy is sent to each worker and changes do not get propagated to the other workers. There's also a size limitation, given that it sends the data to each worker, you cannot broadcast huge collections (the same way python doesn't work well if you were working with a 10GB list for example).

There are also Accumulators (https://spark.apache.org/docs/2.2.0/rdd-programming-guide.html#accumulators) that aggregate information across the cluster, but they can be tricky to use. If you want to learn more please refer to the documentation.

Warning

Common Pitfalls

Closures

One tricky aspect of programming in Spark is the notion of closures. RDD operations that modify variables outside their scope can be the source of undefined behavior. Take the snippet below as an example.

```
var counter = 0
rdd.foreach(x => counter += x)
println("Counter: " + counter)
```

Because Spark does not guarantee the behavior of mutations to objects referenced from outside closures, the value of `counter` is not guaranteed to be the sum of the number of elements in the RDD. Even if `counter` gives the correct value in `local` mode, this is by accident and not by design.

In this case, you can use the `count()` action or an accumulator to achieve the desired outcome.

Printing RDDs

One way to print the contents of an RDD would be to do `rdd.foreach(println)`. While this will work as expected on a single machine, it could lead to some unexpected results when you are working in a cluster. Since the output to stdout is being called by the executors, they are writing to the executor's stdout and they do not appear on the driver node's stdout. You should use `collect()` to bring the entire RDD to the master node for printing.

One last thing to note is that we will be working with some pretty sizable data in the Spark project. If you were to call `collect()` on RDDs of that size, you will most likely cause the driver node to run out of memory because the collective memory of the cluster is greater than the memory of the individual master where data is collected to. If you need to print the contents of an RDD, one way around this would be to use `take()` to print a subset of the RDD contents `take(100).foreach(println)` .

Laziness

Because of Spark's lazy execution, it can be hard to trace the source of an exception. For example, the execution of this line does not cause the application to crash immediately:

```
val failureRDD = rdd.map {_ => throw new RuntimeException() }
```

The exception would only be thrown once an action is executed:

```
val failureRDD = rdd.map {_ => throw new RuntimeException() }
                        ... (more transformations) ...

failureRDD.count() // <---- Exception is thrown when an action is executed
```

Spark SQL

Spark SQL

RDD was the primary user-facing API in Spark since its inception. On top of RDDs, Spark introduced the Spark SQL module providing an interface which lets Spark know more information about the structure of both the data and the computation being performed. Internally, **Spark SQL uses this extra information to perform extra optimizations.** What is more, it bridges the gap between software/data engineers and data scientists, as the latter are more used to running SQL code rather than writing their own map-reduce jobs.

You can then use the `SparkSession` object to create DataFrames/Datasets from RDDs, or you can read JSON, CSV and text files files directly with the same object.

Datasets and DataFrames

A `Dataset` is an abstraction on top of `RDDs` . Unlike the latter, `Dataset` are structured. Thus, it is organized into named columns (think of it as a table in a relational database). Therefore, `Dataset[T]` is a collection of **strongly-typed** JVM objects, dictated by a case class `T` you define in Scala or a class in Java.

`DataFrame` , on the other hand, is an *alias* for a collection of generic objects `Dataset[Row]` , where a `Row` is a generic **untyped** JVM object.

Therefore, `Dataset` takes on two distinct APIs: a strongly-typed API (`Dataset[T]`) and an untyped API (`DataFrame`).

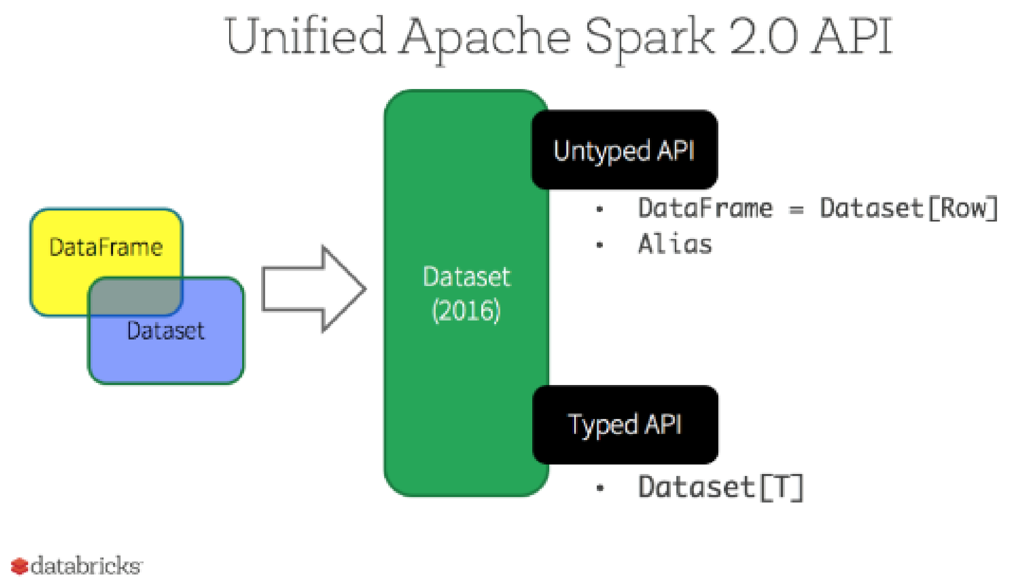
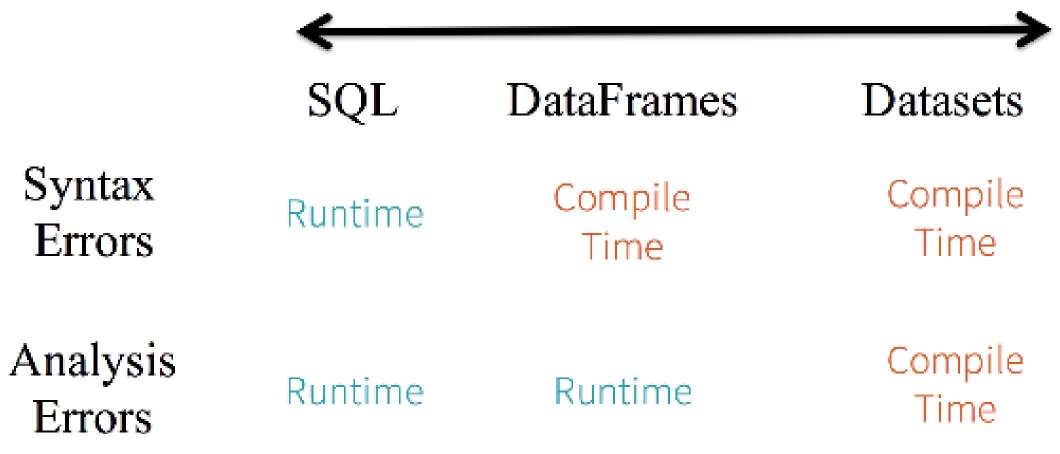


Figure 8 Spark 2.0 API

Note: Python only has DataFrame, since it has no compile-time type-safety.



	SQL	DataFrames	Datasets
Syntax Errors	Runtime	Compile Time	Compile Time
Analysis Errors	Runtime	Runtime	Compile Time

Figure 9 Type safety in Spark APIs

Type-safety of `Dataset` and `DataFrame` are different. For example, if you invoke a function in `DataFrame` that is not part of the API, the compiler will catch it. However, the compiler cannot detect a non-existing column name until the runtime with a `DataFrame`. The compiler will detect both cases with a `Dataset`.

Except the type-safety benefit, which can catch analysis errors at compile time, their performance is almost the same.

Using the DataFrame and Dataset APIs

Suppose we have two files `people.json` and `people.csv` with the following contents respectively, :

```
{"name":"Michael"}
{"name":"Andy", "age":30}
{"name":"Justin", "age":19}

Michael,
Andy,30
Justin,19
```

Loading data

Spark can load a JSON file directly as a DataFrame.

```
val df = spark.read.json("people.json")
//df: org.apache.spark.sql.DataFrame = [age: bigint, name: string]

df.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+

df.select("name").show()
// +-------+
// |   name|
// +-------+
// |Michael|
// |   Andy|
// | Justin|
// +-------+
```

It can also read a file directly as CSV, with and without providing a schema:

```
import org.apache.spark.sql.types.{StructType, IntegerType, StringType}

val schema = new StructType()
  .add("name", StringType)
  .add("age", IntegerType)

val df = spark.read.schema(schema).csv("people.csv")
//df: org.apache.spark.sql.DataFrame = [name: string, age: int]

df.show()

// +-------+----+
// |   name| age|
// +-------+----+
// |Michael|null|
// |   Andy|  30|
// | Justin|  19|
// +-------+----+
```

By providing a **case class**, a DataFrame can be converted to a Dataset. Note that the DataFrame column names and case class field names need to be the same.

```
import spark.implicits._

case class Person(name: String, age: Long)
val ds = df.as[Person]
// ds: org.apache.spark.sql.Dataset[Person] = [age: bigint, name: string]
```

Also, note that an RDD can be converted (http://spark.apache.org/docs/latest/sql-programming-guide.html#interoperating-with-rdds) into a DataFrame or Dataset as well.

Operating on data

Just like SQL you can run SELECT, WHERE, etc.

```
import org.apache.spark.sql.functions.col

// Select only the "name" column
df.select("name").where(col("age") > 20).show()
// +----+
// |name|
// +----+
// |Andy|
// +----+
```

Here is a word count example; its result `counts` is a DataFrame (try to think what each function is doing).

```
import org.apache.spark.sql.functions.{col, count, split, explode}

val df = spark.read.text("hdfs:///input/README.md")
val counts = df
    .select(explode(split(col("value"), " ")).as("words"))
    .groupBy("words")
    .agg(count("*").as("count"))
// org.apache.spark.sql.DataFrame = [value: string, count: bigint]
```

You can save a DataFrame/Dataset as different file formats. One common format is Parquet (http://parquet.apache.org/). It is a columnar storage format that is supported by other data processing systems, and has some optimizations when reading the files.

```
val df = spark.read.json("people.json")
df.write.parquet("people")

val df = spark.read.parquet("people") # to read it
```

Finally, you can register DataFrames and Datasets as temporary view (like a table in RDB), where you can run SQL queries. The result of the queries is a DataFrame.

```
df.createOrReplaceTempView("people")
df.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+

val sqlDF = spark.sql("SELECT name FROM people WHERE age > 20").show()
// +----+
// |name|
// +----+
// |Andy|
// +----+
```

You should refer to the SparkSQL documentation here (https://spark.apache.org/docs/latest/sql-programming-guide.html) for more information about the API.

SparkSQL

`DataFrame` and `Dataset` APIs are built on top of the Spark SQL engine. They use Catalyst (https://databricks.com/glossary/catalyst-optimizer) to generate an optimized logical and physical query plan. Across Java, Scala, and Python DataFrame/Dataset APIs, all relation type queries undergo the same code optimizer, providing space and speed efficiency.

The following is an experiment you can try running to help you understand the optimizations Spark does under the hood. You can run the example in the Spark shell (`spark-shell`), or on a Zeppelin notebook.

The data we will be working with has the following schema:

```
// Grades Schema
+----+--------+------+------+
|  id| semester| course| grade|
+----+--------+------+------+

// Student Schema
+----+----+----+
|  id| name| age|
+----+----+----+
```

What we would like to do is get the maximum age of students in 15619. If we were doing it on SQL, we would do something like this:

```
SELECT MAX(s.age)
FROM grades g INNER JOIN students s
ON s.id = s.id
WHERE g.course = 15619
```

To begin with, download these two datasets:

```
wget https://clouddeveloper.blob.core.windows.net/assets/iterative-processing/pr
imer/spark-primer/datasets/grades.csv
wget https://clouddeveloper.blob.core.windows.net/assets/iterative-processing/pr
imer/spark-primer/datasets/students.csv
```

Warning

Depending where you are running the code you might need to adjust the paths in the examples.

We will first do it with the DataFrame API. The following snippet reads the data as a DataFrame by providing a schema, joins both DataFrames, filters and finds the max:

```
import org.apache.spark.sql.types.{StructType, IntegerType, StringType}
import org.apache.spark.sql.functions.max

val schema = new StructType()
  .add("id", IntegerType)
  .add("name", StringType)
  .add("age", IntegerType)

val students = spark.read.schema(schema).csv("students.csv")

val schemaGrades = new StructType()
  .add("id", IntegerType)
  .add("semester", StringType)
  .add("course", StringType)
  .add("grade", IntegerType)

val grades = spark.read.schema(schemaGrades).csv("grades.csv")

val maxDF = grades.join(students, Seq("id")).where("course == 15619").select(max
("age"))

// Forces the execution - remember transformations are lazy
maxDF.show()
```

Information

Note you can do the same by running the following SQL code:

```
// Creates temporary tables for those DataFrames
students.createOrReplaceTempView("students")
grades.createOrReplaceTempView("grades")

spark.sql("""
SELECT MAX(s.age)
FROM grades g INNER JOIN students s
ON s.id = g.id
WHERE g.course = 15619""").show()
```

Let's now do the same with RDDs:

```
val studentsRDD = spark.sparkContext.textFile("students.csv").map{ line =>
    val Array(id, _, age) = line.split(",")
    (id.toInt, age.toInt)
}

val gradesRDD = spark.sparkContext.textFile("grades.csv").map{ line =>
    val Array(id, _, course, _) = line.split(",")
    (id.toInt, course.toInt)
}

gradesRDD.join(studentsRDD)
         .filter(row => row._2._1 == 15619)
         .map(row => row._2._2)
         .max()
```

Notice the time difference? Why do you think that is?

If you pay close attention to what is being done, we first join both RDDs, and then filter by course id. Wouldn't it be smarter to filter first, and then join? Thus joining on a smaller RDD.

On RDDs Spark has no way of knowing what operations are being done to try to optimize them. It is up for the programmer to realize that, and write proper code. What is more, by specifying the schema when reading the CSV, Spark can parse the data as it reads it instead of having to load it as String, and then map it to the appropriate data types.

Let's run the snippet above again, but changing the order of the operations:

```
val studentsRDD = spark.sparkContext.textFile("students.csv").map{ line =>
    val Array(id, _, age) = line.split(",")
    (id.toInt, age.toInt)
}

val gradesRDD = spark.sparkContext.textFile("grades.csv").map{ line =>
    val Array(id, _, course, _) = line.split(",")
    (id.toInt, course.toInt)
}

gradesRDD.filter(row => row._2 == 15619)
         .join(studentsRDD)
         .map(row => row._2._2)
         .max()
```

Now the performance is comparable to that of the DataFrame.

We can actually take a look at the DAG that it builds when executing the DataFrame code above:

```
maxDF.explain

// == Physical Plan ==
// *(6) HashAggregate(keys=[], functions=[max(age#2)])
// +- Exchange SinglePartition
//    +- *(5) HashAggregate(keys=[], functions=[partial_max(age#2)])
//       +- *(5) Project [age#2]
//          +- *(5) SortMergeJoin [id#6], [id#0], Inner
//             :- *(2) Sort [id#6 ASC NULLS FIRST], false, 0
//             :  +- Exchange hashpartitioning(id#6, 200)
//             :     +- *(1) Project [id#6]
//             :        +- *(1) Filter ((isnotnull(course#8) && (course#8 = 1561
9)) && isnotnull(id#6))
//             :           +- *(1) FileScan csv [id#6,course#8] Batched: false,
Format: CSV, Location: InMemoryFileIndex[file:/grades.csv], PartitionFilters:
[], PushedFilters: [IsNotNull(course), EqualTo(course,15619), IsNotNull(id)], Re
adSchema: struct<id:int,course:int>
//             +- *(4) Sort [id#0 ASC NULLS FIRST], false, 0
//                +- Exchange hashpartitioning(id#0, 200)
//                   +- *(3) Project [id#0, age#2]
//                      +- *(3) Filter isnotnull(id#0)
//                         +- *(3) FileScan csv [id#0,age#2] Batched: false, For
mat: CSV, Location: InMemoryFileIndex[file:/students.csv], PartitionFilters: [],
PushedFilters: [IsNotNull(id)], ReadSchema: struct<id:int,age:int>
```

The plan is read following the numbers in brackets. Notice how Spark was smart enough to realize that filtering should happen before trying to JOIN.

Some takeaways from this experiments are:

1. If working with structured data (data with a fixed schema), try using the DataFrame/Dataset APIs.
2. Join is a very expensive operation that usually entails shuffling across the network.
3. If possible, filtering first is always the better option.

Monitoring

Monitoring Spark Programs

Every SparkContext launches a web UI by default on port `4040`.

The UI provides a wealth of information about the application you are running, including scheduler tasks and stages, RDD sizes and memory usage, environment information and executors that are running. It is a good starting point for monitoring your Spark programs and debugging them, should the need arises.

Making Sense of the UI

You can access the UI from `http://driver-node:4040`. On a standalone Spark instance, that will correspond to the IP address of the virtual machine and `localhost` if you are running Spark on your local machine.

While using an HDInsight cluster, since Spark is configured to run on YARN in EMR, instead of viewing the Spark application UI at port 4040, you should instead start from the YARN ResourceManager, then click on the ApplicationMaster link for the Spark application you are interested in. Visit the HDInsight page on Azure portal (https://portal.azure.com/#blade/HubsExtension/Resources/resourceType/Microsoft.HDInsight%2Fclusters), click `cluster name - Cluster dashboards - Yarn`, and sign in as `admin` (not `sshuser`) using your password as you input as clusterLoginPassword when creating the cluster. You can follow the Azure docs (https://docs.microsoft.com/en-us/azure/hdinsight/spark/apache-spark-job-debugging#track-an-application-in-the-yarn-ui) for the definitive guide.

Accessing the UI

History Server

The Spark History Server provides an interface to past applications you have run on the cluster. You can access it at `cluster name - Cluster dashboards - Spark history server` on the HDInsight page or `https://<HDINSIGHT-HEAD-NODE-URL>/sparkhistory/` in your browser.

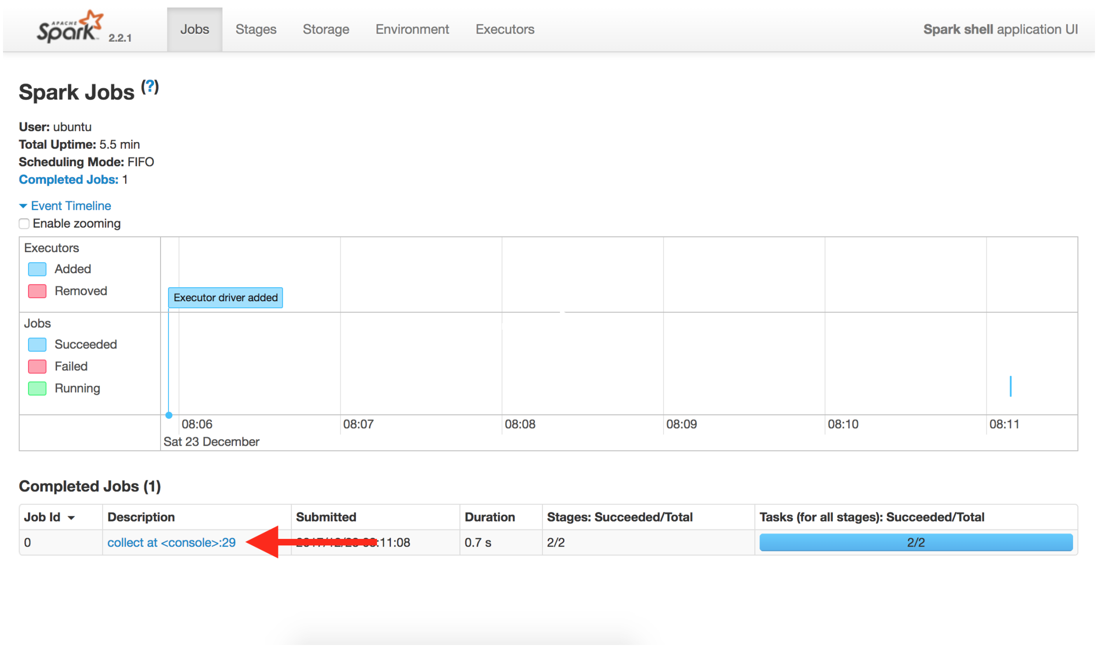
Incomplete applications

You can access the Spark UI for running applications at `http://<HDINSIGHT-HEAD-NODE-URL>/sparkhistory/?showIncomplete=true`.

Interpreting the UI

Jobs

On the UI, each `job` corresponds to an RDD action. You can click into one of them to view the stages or tasks inside of it.



Spark Jobs (?)

User: ubuntu
Total Uptime: 5.5 min
Scheduling Mode: FIFO
Completed Jobs: 1

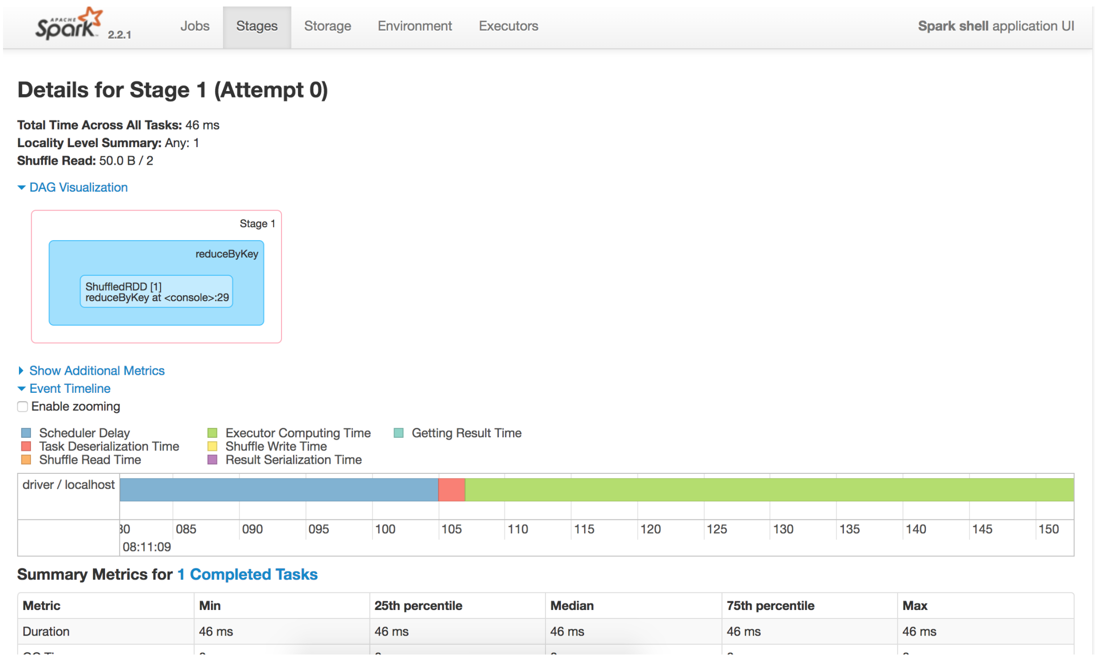
Completed Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	collect at <console>:29	2017/12/23 08:11:08	0.7 s	2/2	2/2

Figure 10 Jobs tab on the Spark UI

Stages

For each job, you can view metrics such as the amount of memory that was used, the time each task took and even what proportion of the time was taken up by the scheduler, executor and shuffling.



Details for Stage 1 (Attempt 0)

Total Time Across All Tasks: 46 ms
Locality Level Summary: Any: 1
Shuffle Read: 50.0 B / 2

DAG Visualization

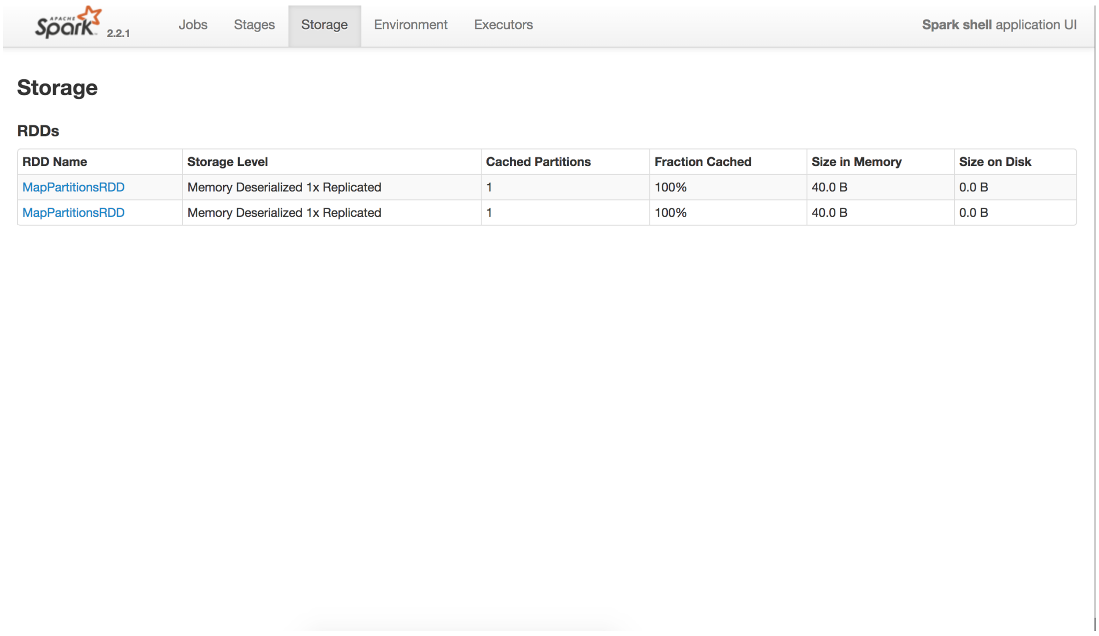
Summary Metrics for 1 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	46 ms	46 ms	46 ms	46 ms	46 ms

Figure 11 Stages tab on the Spark UI

Storage

The storage tab contains information about RDDs that you used `cache()` or `persist()` and later used in some other job. It tells you what fraction and size of the RDD is in memory or on disk. If you cache too many RDDs, it is possible that not all of them are going to fit in memory. This can provide a good starting point for performance tuning your application.



Storage

RDDs

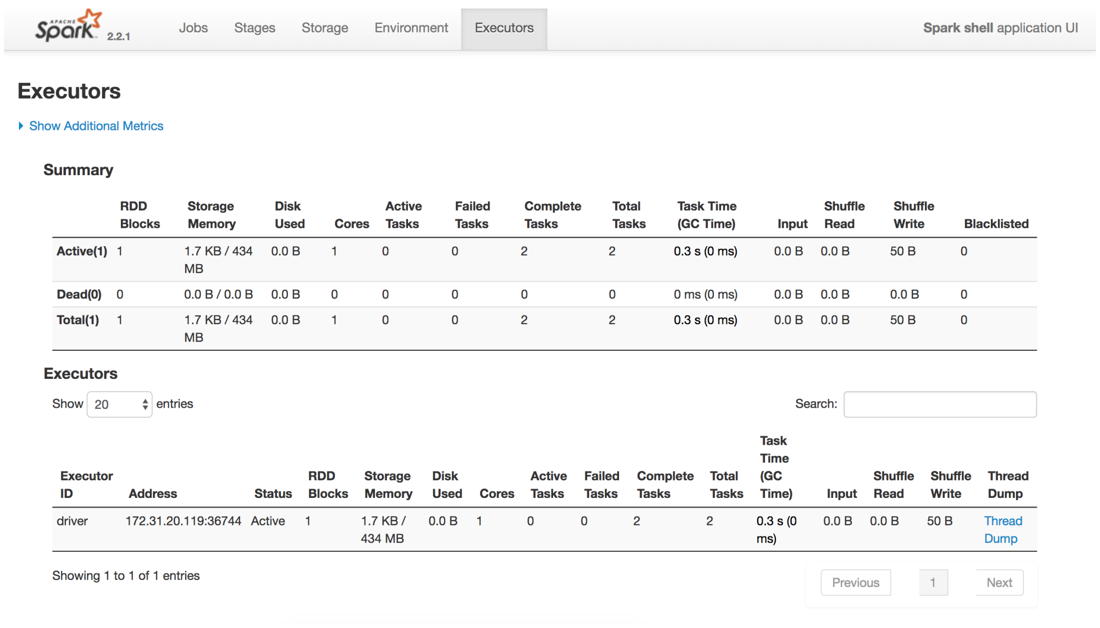
RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
MapPartitionsRDD	Memory Deserialized 1x Replicated	1	100%	40.0 B	0.0 B
MapPartitionsRDD	Memory Deserialized 1x Replicated	1	100%	40.0 B	0.0 B

Figure 12 Storage tab on the Spark UI

Executors

In this course, we will write Spark applications to handle large amounts of data on machines with a limited amount of memory. It is possible that your code may be space inefficient to the point of using more memory than what is available. The executors tab provides information on the memory footprint of your application.

An executor is a worker process that runs tasks in a Spark job and returns the results to the Spark driver. Executors also provide in-memory storage for RDDs that are cached by user programs. Executors are generally launched once at the beginning of a Spark application.



Executors

Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Blacklisted
Active(1)	1	1.7 KB / 434 MB	0.0 B	1	0	0	2	2	0.3 s (0 ms)	0.0 B	0.0 B	50 B	0
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	0
Total(1)	1	1.7 KB / 434 MB	0.0 B	1	0	0	2	2	0.3 s (0 ms)	0.0 B	0.0 B	50 B	0

Executors

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Thread Dump
driver	172.31.20.119:36744	Active	1	1.7 KB / 434 MB	0.0 B	1	0	0	2	2	0.3 s (0 ms)	0.0 B	0.0 B	50 B	Thread Dump

Showing 1 to 1 of 1 entries

Figure 13 Executors tab on the Spark UI

References

Spark Programming Guide (https://spark.apache.org/docs/1.6.2/programming-guide.html) 10-405/10-605: Machine Learning with Large Datasets (https://10605.github.io/)