FREE eBook

LEARNING apache-spark

Unaffiliated free eBook created from **Stack Overflow contributors.**

#apache-spark

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: apache-spark

It is an unofficial and free apache-spark ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official apache-spark.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with apache-spark

Remarks

Apache Spark is an open source big data processing framework built around speed, ease of use, and sophisticated analytics. A developer should use it when (s)he handles large amount of data, which usually imply memory limitations and/or prohibitive processing time.

It should also mention any large subjects within apache-spark, and link out to the related topics. Since the Documentation for apache-spark is new, you may need to create initial versions of those related topics.

Versions

Version	Release Date
2.2.0	2017-07-11
2.1.1	2017-05-02
2.1.0	2016-12-28
2.0.1	2016-10-03
2.0.0	2016-07-26
1.6.0	2016-01-04
1.5.0	2015-09-09
1.4.0	2015-06-11
1.3.0	2015-03-13
1.2.0	2014-12-18
1.1.0	2014-09-11
1.0.0	2014-05-30
0.9.0	2014-02-02
0.8.0	2013-09-25
0.7.0	2013-02-27
0.6.0	2012-10-15

Examples

Introduction

Prototype:

> aggregate(zeroValue, seqOp, combOp)

Description:

`aggregate()` lets you take an RDD and generate a single value that is of a different type than what was stored in the original RDD.

Parameters:

1. `zeroValue`: The initialization value, for your result, in the desired format.
2. `seqOp`: The operation you want to apply to RDD records. Runs once for every record in a partition.
3. `combOp`: Defines how the resulted objects (one for every partition), gets combined.

Example:

> Compute the sum of a list and the length of that list. Return the result in a pair of `(sum, length)`.

In a Spark shell, create a list with 4 elements, with 2 *partitions*:

```
listRDD = sc.parallelize([1,2,3,4], 2)
```

Then define *seqOp*:

```
seqOp = (lambda local_result, list_element: (local_result[0] + list_element, local_result[1] +
1) )
```

Then define *combOp*:

```
combOp = (lambda some_local_result, another_local_result: (some_local_result[0] +
another_local_result[0], some_local_result[1] + another_local_result[1]) )
```

Then aggregated:

```
listRDD.aggregate( (0, 0), seqOp, combOp)
Out[8]: (10, 4)
```

The first partition has the sublist [1, 2]. This applies the seqOp to each element of that list, which produces a local result - A pair of `(sum, length)` that will reflect the result locally, only in that first partition.

`local_result` gets initialized to the `zeroValue` parameter `aggregate()` was provided with. For

example, (0, 0) and `list_element` is the first element of the list:

```
0 + 1 = 1
0 + 1 = 1
```

The local result is (1, 1), which means the sum is 1 and the length 1 for the 1st partition after processing *only* the first element. `local_result` gets updated from (0, 0), to (1, 1).

```
1 + 2 = 3
1 + 1 = 2
```

The local result is now (3, 2), which will be the final result from the 1st partition, since they are no other elements in the sublist of the 1st partition. Doing the same for 2nd partition returns (7, 2).

Apply combOp to each local result to form the final, global result:

```
(3,2) + (7,2) = (10, 4)
```

Example described in 'figure':

```
             (0, 0) <-- zeroValue

[1, 2]                   [3, 4]

0 + 1 = 1                0 + 3 = 3
0 + 1 = 1                0 + 1 = 1

1 + 2 = 3                3 + 4 = 7
1 + 1 = 2                1 + 1 = 2
    |                        |
    v                        v
  (3, 2)                   (7, 2)
     \                       /
      \                     /
       \                   /
        \                 /
         \               /
          \             /
           \           /
            ------------
            |  combOp  |
            ------------
                 |
                 v
              (10, 4)
```

Transformation vs Action

Spark uses **lazy evaluation**; that means it will not do any work, unless it really has to. That approach allows us to avoid unnecessary memory usage, thus making us able to work with big data.

A *transformation* is lazy evaluated and the actual work happens, when an *action* occurs.

Example:

```
In [1]: lines = sc.textFile(file)        // will run instantly, regardless file's size
In [2]: errors = lines.filter(lambda line: line.startsWith("error")) // run instantly
In [3]: errorCount = errors.count()    // an action occurred, let the party start!
Out[3]: 0                              // no line with 'error', in this example
```

So, in `[1]` we told Spark to read a file into an RDD, named `lines`. Spark heard us and told us: "Yes I *will* do it", but in fact it didn't *yet* read the file.

In [2], we are filtering the lines of the file, assuming that its contents contain lines with errors that are marked with an `error` in their start. So we tell Spark to create a new RDD, called `errors`, which will have the elements of the RDD `lines`, that had the word `error` at their start.

Now in `[3]`, we ask Spark to *count* the *errors*, i.e. count the number of elements the RDD called `errors` has. `count()` is an **action**, which leave no choice to Spark, but to actually make the operation, so that it can find the result of `count()`, which will be an integer.

As a result, when `[3]` is reached, `[1]` and `[2]` will actually being performed, i.e. that when we reach `[3]`, then and only then:

1. the file is going to be read in `textFile()` (because of `[1]`)
2. `lines` will be `filter()`'ed (because of `[2]`)
3. `count()` will execute, because of `[3]`

Debug tip: Since Spark won't do any real work until `[3]` is reached, it is important to understand that if an error exist in `[1]` and/or `[2]`, it won't appear, until the action in `[3]` triggers Spark to do actual work. For example if your data in the file do not support the `startsWith()` I used, then `[2]` is going to be properly accepted by Spark and it won't raise any error, but when `[3]` is submitted, and Spark actually evaluates both `[1]` and `[2]`, then and only then it will understand that something is not correct with `[2]` and produce a descriptive error.

As a result, an error may be triggered when `[3]` is executed, but that doesn't mean that the error must lie in the statement of `[3]`!

Note, neither `lines` nor `errors` will be stored in memory after `[3]`. They will continue to exist only as a set of processing instructions. If there will be multiple actions performed on either of these RDDs, spark will read and filter the data multiple times. To avoid duplicating operations when performing multiple actions on a single RDD, it is often useful to store data into memory using `cache`.

You can see more transformations/actions in Spark docs.

Check Spark version

In `spark-shell`:

```
sc.version
```

Generally in a program:

```
SparkContext.version
```

Using `spark-submit`:

```
 spark-submit --version
```

Read Getting started with apache-spark online: http://www.riptutorial.com/apache-spark/topic/833/getting-started-with-apache-spark

Chapter 2: Calling scala jobs from pyspark

Introduction

This document will show you how to call Scala jobs from a pyspark application.

This approach can be useful when the Python API is missing some existing features from the Scala API or even to cope with performance issues using python.

In some use cases, using Python is inevitable e.g you are building models with `scikit-learn`.

Examples

Creating a Scala functions that receives a python RDD

Creating a Scala function that receives an python RDD is easy. What you need to build is a function that get a JavaRDD[Any]

```
import org.apache.spark.api.java.JavaRDD

def doSomethingByPythonRDD(rdd :JavaRDD[Any]) = {
    //do something
    rdd.map { x => ??? }
}
```

Serialize and Send python RDD to scala code

This part of development you should serialize the python RDD to the JVM. This process uses the main development of Spark to call the jar function.

```
from pyspark.serializers import PickleSerializer, AutoBatchedSerializer


rdd = sc.parallelize(range(10000))
reserialized_rdd = rdd._reserialize(AutoBatchedSerializer(PickleSerializer()))
rdd_java = rdd.ctx._jvm.SerDe.pythonToJava(rdd._jrdd, True)

_jvm = sc._jvm #This will call the py4j gateway to the JVM.
_jvm.myclass.apps.etc.doSomethingByPythonRDD(rdd_java)
```

How to call spark-submit

To call this code you should create the jar of your scala code. Than you have to call your spark submit like this:

```
spark-submit --master yarn-client --jars ./my-scala-code.jar --driver-class-path ./my-scala-code.jar main.py
```

This will allow you to call any kind of scala code that you need in your pySpark jobs

Read Calling scala jobs from pyspark online: http://www.riptutorial.com/apache-spark/topic/9180/calling-scala-jobs-from-pyspark

Chapter 3: Client mode and Cluster Mode

Examples

Spark Client and Cluster mode explained

Let's try to look at the differences between client and cluster mode of Spark.

Client: When running Spark in the client mode, the SparkContext and Driver program run external to the cluster; for example, from your laptop. Local mode is only for the case when you do not want to use a cluster and instead want to run everything on a single machine. So Driver Application and Spark Application are both on the same machine as the user. Driver runs on a dedicated server (Master node) inside a dedicated process. This means it has all available resources at it's disposal to execute work. Because the Master node has dedicated resources of it's own, you don't need to "spend" worker resources for the Driver program. If the driver process dies, you need an external monitoring system to reset it's execution.

Cluster: Driver runs on one of the cluster's Worker nodes.It runs as a dedicated, standalone process inside the Worker. When working in Cluster mode, all JARs related to the execution of your application need to be publicly available to all the workers. This means you can either manually place them in a shared place or in a folder for each of the workers. Each application gets its own executor processes, which stay up for the duration of the whole application and run tasks in multiple threads. This has the benefit of isolating applications from each other, on both the scheduling side (each driver schedules its own tasks) and executor side (tasks from different applications run in different JVMs

Cluster Manager Types

Apache Mesos – a general cluster manager that can also run Hadoop MapReduce and service applications. Hadoop YARN – the resource manager in Hadoop.
Kubernetes- container-centric infrastructure.it is experimental yet.

Read Client mode and Cluster Mode online: http://www.riptutorial.com/apache-spark/topic/10808/client-mode--and-cluster-mode

Chapter 4: Configuration: Apache Spark SQL

Introduction

In this topic Spark Users can find different configurations of Spark SQL, which is the most used component of Apache Spark framework.

Examples

Controlling Spark SQL Shuffle Partitions

In Apache Spark while doing shuffle operations like `join` and `cogroup` a lot of data gets transferred across network. Now, to control the number of partitions over which shuffle happens can be controlled by configurations given in Spark SQL. That configuration is as follows:

```
spark.sql.shuffle.partitions
```

Using this configuration we can control the number of partitions of shuffle operations. By default, its value is `200`. But, 200 partitions does not make any sense if we have files of few GB(s). So, we should change them according to the amount of data we need to process via Spark SQL. Like as follows:

In this scenario we have two tables to be joined `employee` and `department`. Both tables contains only few records only, but we need to join them to get to know the department of each employee. So, we join them using Spark DataFrames like this:

```
val conf = new SparkConf().setAppName("sample").setMaster("local")
val sc = new SparkContext(conf)

val employee = sc.parallelize(List("Bob", "Alice")).toDF("name")
val department = sc.parallelize(List(("Bob", "Accounts"), ("Alice", "Sales"))).toDF("name",
"department")

employeeDF.join(departmentDF, "employeeName").show()
```

Now, the number of partitions that gets created while doing join are 200 by default which is of course too much for this much amount of data.

So, lets change this value so that we can reduce the number of shuffle operations.

```
val conf = new
SparkConf().setAppName("sample").setMaster("local").set("spark.sql.shuffle.partitions", 2)
val sc = new SparkContext(conf)

val employee = sc.parallelize(List("Bob", "Alice")).toDF("name")
val department = sc.parallelize(List(("Bob", "Accounts"), ("Alice", "Sales"))).toDF("name",
"department")
```

```
employeeDF.join(departmentDF, "employeeName").show()
```

Now, the number of shuffle partitions are reduced to only 2, which will not only reduce the number of shuffling operations but also reduce the time taken to join the DataFrames from `0.878505 s` to `0.077847 s`.

So, always configure the number of partitions for shuffle operations according to the data being processed.

Read Configuration: Apache Spark SQL online: http://www.riptutorial.com/apache-spark/topic/8169/configuration--apache-spark-sql

Chapter 5: Error message 'sparkR' is not recognized as an internal or external command or '.binsparkR' is not recognized as an internal or external command

Introduction

This post is for those who were having trouble installing Spark in their windows machine. Mostly using sparkR function for R session.

Remarks

Used reference from r-bloggers

Examples

details for set up Spark for R

Use below URL to get steps for download and install- https://www.r-bloggers.com/installing-and-starting-sparkr-locally-on-windows-os-and-rstudio-2/ Add the environment variable path for your 'Spark/bin', 'spark/bin' , R and Rstudio path. I have added below path (initials will vary based on where you have downloaded files) C:\spark-2.0.1 C:\spark-2.0.1\bin C:\spark-2.0.1\sbin C:\Program Files\R\R-3.3.1\bin\x64 C:\Program Files\RStudio\bin\x64

To set the environment variable please follow below steps: Windows 10 and Windows 8 In Search, search for and then select: System (Control Panel) Click the Advanced system settings link. Click on Advanced tab under Sytem Properties Click Environment Variables. In the section System Variables, find the PATH environment variable and select it. Click Edit. If the PATH environment variable does not exist, click New. In the Edit System Variable (or New System Variable) window, specify the value of the PATH environment variable. Click OK. Close all remaining windows by clicking OK. Reopen Command prompt window, and run sparkR (no need to change directory).

Windows 7 From the desktop, right click the Computer icon. Choose Properties from the context menu. Click the Advanced system settings link. Click Environment Variables. In the section System Variables, find the PATH environment variable and select it. Click Edit. If the PATH environment variable does not exist, click New. In the Edit System Variable (or New System Variable) window, specify the value of the PATH environment variable. Click OK. Close all remaining windows by clicking OK. Reopen Command prompt window, and run sparkR (no need to change directory).

Read Error message 'sparkR' is not recognized as an internal or external command or '.binsparkR'

is not recognized as an internal or external command online: http://www.riptutorial.com/apache-spark/topic/9649/error-message--sparkr--is-not-recognized-as-an-internal-or-external-command-or---binsparkr--is-not-recognized-as-an-internal-or-external-command

Chapter 6: Handling JSON in Spark

Examples

Mapping JSON to a Custom Class with Gson

With `Gson`, you can read JSON dataset and map them to a custom class `MyClass`.

Since `Gson` is not serializable, each executor needs its own `Gson` object. Also, `MyClass` must be serializable in order to pass it between executors.

Note that the file(s) that is offered as a json file is not a typical JSON file. Each line must contain a separate, self-contained valid JSON object. As a consequence, a regular multi-line JSON file will most often fail.

```
val sc: org.apache.spark.SparkContext // An existing SparkContext

// A JSON dataset is pointed to by path.
// The path can be either a single text file or a directory storing text files.
val path = "path/to/my_class.json"
val linesRdd: RDD[String] = sc.textFile(path)

// Mapping json to MyClass
val myClassRdd: RDD[MyClass] = linesRdd.map{ l =>
    val gson = new com.google.gson.Gson()
    gson.fromJson(l, classOf[MyClass])
}
```

If creation of `Gson` object becomes too costly, `mapPartitions` method can be used to optimize it. With it, there will be one `Gson` per partition instead of per line:

```
val myClassRdd: RDD[MyClass] = linesRdd.mapPartitions{p =>
    val gson = new com.google.gson.Gson()
    p.map(l => gson.fromJson(l, classOf[MyClass]))
}
```

Read Handling JSON in Spark online: http://www.riptutorial.com/apache-spark/topic/2799/handling-json-in-spark

Chapter 7: How to ask Apache Spark related question?

Introduction

The goal of this topic is to document best practices when asking Apache Spark related questions.

Examples

Environment details:

When asking Apache Spark related questions please include following information

- Apache Spark version used by the client and Spark deployment if applicable. For API related questions major (1.6, 2.0, 2.1 etc.) is typically sufficient, for questions concerning possible bugs always use full version information.
- Scala version used to build Spark binaries.
- JDK version (`java -version`).
- If you use guest language (Python, R) please provide information about the language version. In Python use tags: python-2.x, python-3.x or more specific ones to distinguish between language variants.
- Build definition (`build.sbt`, `pom.xml`) if applicable or external dependency versions (Python, R) when applicable.
- Cluster manager (`local[n]`, Spark standalone, Yarn, Mesos), mode (`client`, `cluster`) and other submit options if applicable.

Example data and code

Example Data

Please try to provide a minimal example input data in a format that can be directly used by the answers without tedious and time consuming parsing for example input file or local collection with all code required to create distributed data structures.

When applicable always include type information:

- In RDD based API use type annotations when necessary.
- In DataFrame based API provide schema information as a `StrucType` or output from `Dataset.printSchema`.

Output from `Dataset.show` or `print` can look good but doesn't tell us anything about underlying types.

If particular problem occurs only at scale use random data generators (Spark provides some

useful utilities in `org.apache.spark.mllib.random.RandomRDDs` and `org.apache.spark.graphx.util.GraphGenerators`

Code

Please use type annotations when possible. While your compiler can easily keep track of the types it is not so easy for mere mortals. For example:

```
val lines: RDD[String] = rdd.map(someFunction)
```

or

```
def f(x: String): Int = ???
```

are better than:

```
val lines = rdd.map(someFunction)
```

and

```
def f(x: String) = ???
```

respectively.

Diagnostic information

Debugging questions.

When question is related to debugging specific exception always provide relevant traceback. While it is advisable to remove duplicated outputs (from different executors or attempts) don't cut tracebacks to a single line or exception class only.

Performance questions.

Depending on the context try to provide details like:

- `RDD.debugString` / `Dataset.explain`.
- Output from Spark UI with DAG diagram if applicable in particular case.
- Relevant log messages.
- Diagnostic information collected by external tools (Ganglia, VisualVM).

Before you ask

- Search Stack Overflow for duplicate questions. There common class of problems which have been already extensively documented.
- Read How do I ask a good question?.

- Read What topics can I ask about here?
- Apache Spark Community resources

Read How to ask Apache Spark related question? online: http://www.riptutorial.com/apache-spark/topic/8815/how-to-ask-apache-spark-related-question-

Chapter 8: Introduction to Apache Spark DataFrames

Examples

Spark DataFrames with JAVA

A DataFrame is a distributed collection of data organized into named columns. It is conceptually equivalent to a table in a relational database. DataFrames can be constructed from a wide array of sources such as: structured data files, tables in Hive, external databases, or existing RDDs.

Reading a Oracle RDBMS table into spark data frame::

```
SparkConf sparkConf = new SparkConf().setAppName("SparkConsumer");

sparkConf.registerKryoClasses(new Class<?>[]{
        Class.forName("org.apache.hadoop.io.Text"),
        Class.forName("packageName.className")
});

JavaSparkContext sparkContext=new JavaSparkContext(sparkConf);
SQLContext sqlcontext= new SQLContext(sparkContext);

Map<String, String> options = new HashMap();
options.put("driver", "oracle.jdbc.driver.OracleDriver");
options.put("url", "jdbc:oracle:thin:username/password@host:port:orcl"); //oracle url to
connect
options.put("dbtable", "DbName.tableName");
DataFrame df=sqlcontext.load("jdbc", options);
df.show(); //this will print content into tablular format
```

We can also convert this data frame back to rdd if need be :

```
JavaRDD<Row> rdd=df.javaRDD();
```

Create a dataframe from a file:

```
public class LoadSaveTextFile {

    //static schema class
    public static class Schema implements Serializable {

        public String getTimestamp() {
            return timestamp;
        }
        public void setTimestamp(String timestamp) {
            this.timestamp = timestamp;
        }
        public String getMachId() {
            return machId;
        }
```

```
        public void setMachId(String machId) {
            this.machId = machId;
        }
        public String getSensorType() {
            return sensorType;
        }
        public void setSensorType(String sensorType) {
            this.sensorType = sensorType;
        }

        //instance variables
        private String timestamp;
        private String machId;
        private String sensorType;
    }

    public static void main(String[] args) throws ClassNotFoundException {

        SparkConf sparkConf = new SparkConf().setAppName("SparkConsumer");

        sparkConf.registerKryoClasses(new Class<?>[]{
                Class.forName("org.apache.hadoop.io.Text"),
                Class.forName("oracle.table.join.LoadSaveTextFile")
        });

        JavaSparkContext sparkContext=new JavaSparkContext(sparkConf);
        SQLContext sqlcontext= new SQLContext(sparkContext);

        //we have a file which ";" separated
        String filePath=args[0];

        JavaRDD<Schema> schemaRdd = sparkContext.textFile(filePath).map(
                new Function<String, Schema>() {
                    public Schema call(String line) throws Exception {
                        String[] tokens=line.split(";");
                        Schema schema = new Schema();
                        schema.setMachId(tokens[0]);
                        schema.setSensorType(tokens[1]);
                        schema.setTimestamp(tokens[2]);
                        return schema;
                    }
                });

        DataFrame df = sqlcontext.createDataFrame(schemaRdd, Schema.class);
        df.show();
    }
}
```

Now we have data frame from oracle as well from a file. Similarly we can read a table from hive as well. On data frame we can fetch any column as we do in rdbms. Like get a min value for a column or max value. Can calculate a mean/avg for a column. Some other functions like select,filter,agg, groupBy are also available.

Spark Dataframe explained

In Spark, a DataFrame is a distributed collection of data organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations under the hood. DataFrames can be constructed from a wide array of sources

such as structured data files, tables in Hive, external databases, or existing RDDs.

Ways of creating Dataframe

```
val data= spark.read.json("path to json")
```

`val df = spark.read.format("com.databricks.spark.csv").load("test.txt")` in the options field, you can provide header, delimiter, charset and much more

you can also create Dataframe from an RDD

```
val rdd = sc.parallelize(
  Seq(
    ("first", Array(2.0, 1.0, 2.1, 5.4)),
    ("test", Array(1.5, 0.5, 0.9, 3.7)),
    ("choose", Array(8.0, 2.9, 9.1, 2.5))
  )
)

val dfWithoutSchema = spark.createDataFrame(rdd)
```

If you want to create df with schema

```
def createDataFrame(rowRDD: RDD[Row], schema: StructType): DataFrame
```

Why we need Dataframe if Spark has provided RDD

An RDD is merely a Resilient Distributed Dataset that is more of a blackbox of data that cannot be optimized as the operations that can be performed against it, are not as constrained.

No inbuilt optimization engine: When working with structured data, RDDs cannot take advantages of Spark's advanced optimizers including catalyst optimizer and Tungsten execution engine. Developers need to optimize each RDD based on its attributes. Handling structured data: Unlike Dataframe and datasets, RDDs don't infer the schema of the ingested data and requires the user to specify it.

DataFrames in Spark have their execution automatically optimized by a query optimizer. Before any computation on a DataFrame starts, the Catalyst optimizer compiles the operations that were used to build the DataFrame into a physical plan for execution. Because the optimizer understands the semantics of operations and structure of the data, it can make intelligent decisions to speed up computation.

Limitation of DataFrame

Compile-time type safety: Dataframe API does not support compile time safety which limits you from manipulating data when the structure is not known.

Read Introduction to Apache Spark DataFrames online: http://www.riptutorial.com/apache-spark/topic/6514/introduction-to-apache-spark-dataframes

Chapter 9: Joins

Remarks

One thing to note is your resources versus the size of data you are joining. This is where your Spark Join code might fail giving you memory errors. For this reason make sure you configure your Spark jobs really well depending on the size of data. Following is an example of a configuration for a join of 1.5 million to 200 million.

Using Spark-Shell

```
spark-shell   --executor-memory 32G   --num-executors 80  --driver-memory 10g --executor-cores
10
```

Using Spark Submit

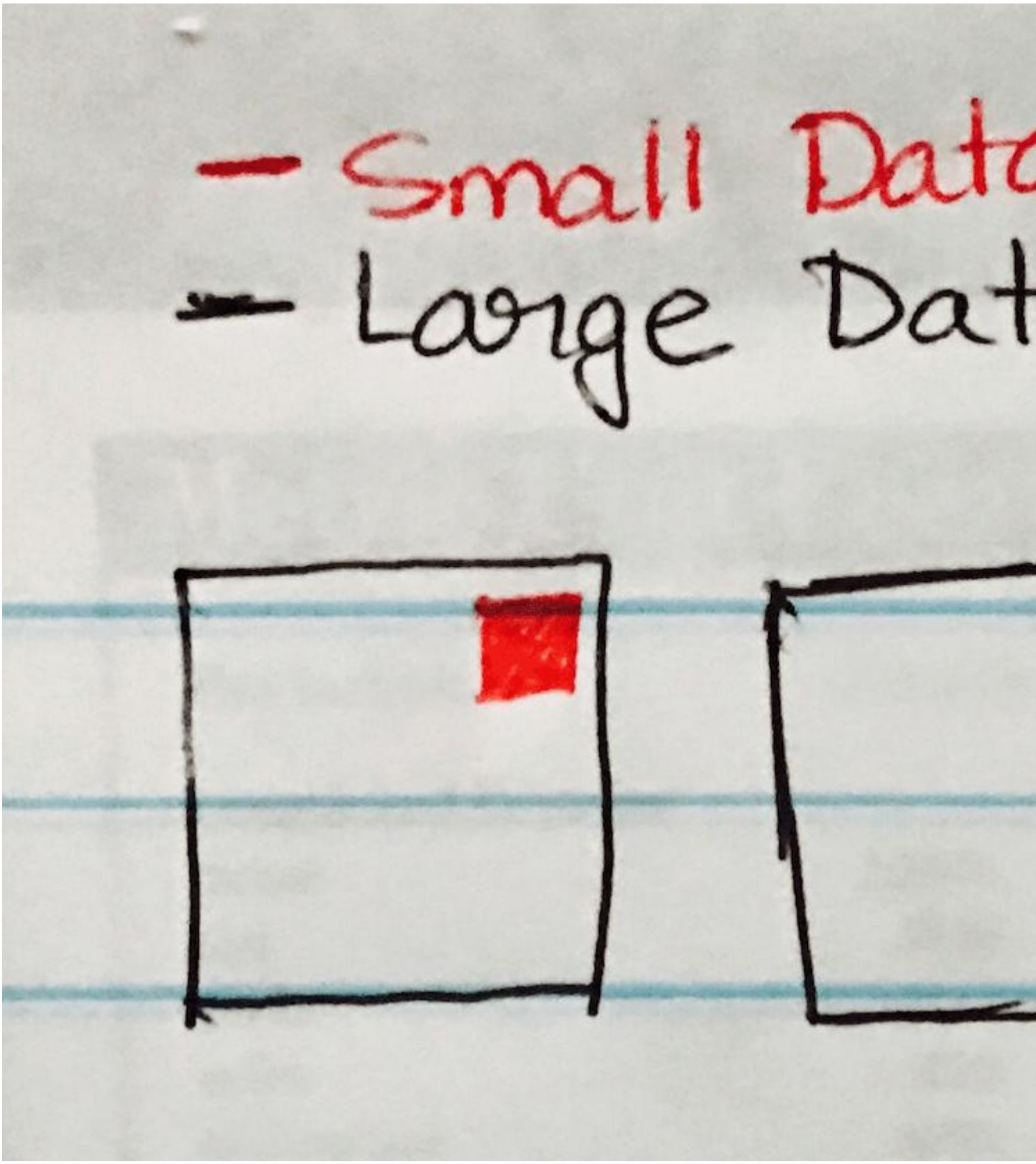
```
spark-submit   --executor-memory 32G   --num-executors 80  --driver-memory 10g --executor-
cores 10 code.jar
```

Examples

Broadcast Hash Join in Spark

A broadcast join copies the small data to the worker nodes which leads to a highly efficient and super-fast join. When we are joining two datasets and one of the datasets is much smaller than the other (e.g when the small dataset can fit into memory), then we should use a Broadcast Hash Join.

The following image visualizes a Broadcast Hash Join whre the the small dataset is broadcasted to each partition of the Large Dataset.



Following is code sample which you can easily implement if you have a similar scenario of a large and small dataset join.

```
case class SmallData(col1: String, col2:String, col3:String, col4:Int, col5:Int)

val small = sc.textFile("/datasource")
```

```
val df1 = sm_data.map(_.split("\\|")).map(attr => SmallData(attr(0).toString,
attr(1).toString, attr(2).toString, attr(3).toInt, attr(4).toInt)).toDF()

val lg_data = sc.textFile("/datasource")

case class LargeData(col1: Int, col2: String, col3: Int)

val LargeDataFrame = lg_data.map(_.split("\\|")).map(attr => LargeData(attr(0).toInt,
attr(2).toString, attr(3).toInt)).toDF()


val joinDF = LargeDataFrame.join(broadcast(smallDataFrame), "key")
```

Read Joins online: http://www.riptutorial.com/apache-spark/topic/7828/joins

Chapter 10: Migrating from Spark 1.6 to Spark 2.0

Introduction

Spark 2.0 has been released and contains many enhancements and new features. If you are using Spark 1.6 and now you want to upgrade your application to use Spark 2.0, you have to take into account some changes in the API. Below are some of the changes to the code that need to be made.

Examples

Update build.sbt file

Update build.sbt with :

```
scalaVersion := "2.11.8" // Make sure to have installed Scala 11
sparkVersion := "2.0.0"  // Make sure to have installed Spark 2.0
```

Note that when compiling with `sbt package`, the `.jar` will now be created in `target/scala-2.11/`, and the `.jar` name will also be changed, so the `spark-submit` command need to be updated as well.

Update ML Vector libraries

ML `Transformers` now generates `org.apache.spark.ml.linalg.VectorUDT` instead of `org.apache.spark.mllib.linalg.VectorUDT`.

They are also mapped locally to subclasses of `org.apache.spark.ml.linalg.Vector`. These are not compatible with old MLLib API which is moving towards deprecation in Spark 2.0.0.

```
//import org.apache.spark.mllib.linalg.{Vector, Vectors} // Depreciated in Spark 2.0
import org.apache.spark.ml.linalg.Vector // Use instead
```

Read Migrating from Spark 1.6 to Spark 2.0 online: http://www.riptutorial.com/apache-spark/topic/6506/migrating-from-spark-1-6-to-spark-2-0

Chapter 11: Partitions

Remarks

The number of partitions is critical for an application's performance and/or successful termination.

A Resilient Distributed Dataset (RDD) is Spark's main abstraction. An RDD is split into partitions, that means that a partition is a part of the dataset, a slice of it, or in other words, a chunk of it.

The greater the number of partitions is, the smaller the size of each partition is.

However, notice that a large number of partitions puts a lot of pressure on Hadoop Distributed File System (HDFS), which has to keep a significant amount of metadata.

The number of partitions is related to the memory usage, and a memoryOverhead issue can be related to this number (personal experience).

A **common pitfall** for new users is to transform their RDD into an RDD with only one partition, which usually looks like that:

```
data = sc.textFile(file)
data = data.coalesce(1)
```

That's usually a very bad idea, since you are telling Spark to put **all the data** is just one partition! Remember that:

> A stage in Spark will operate on one partition at a time (and load the data in that partition into memory).

As a result, you tell Spark to handle all the data at once, which usually results in memory related errors (Out of Memory for example), or even a null pointer exception.

So, unless you know what you are doing, avoid repartitioning your RDD in just one partition!

Examples

Partitions of an RDD

As mentioned in "Remarks", a partition is a part/slice/chunk of an RDD. Below is a minimal example on how to request a minimum number of partitions for your RDD:

```
In [1]: mylistRDD = sc.parallelize([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], 2)

In [2]: mylistRDD.getNumPartitions()
Out[2]: 2
```

Notice in [1] how we passed 2 as a second parameter of `parallelize()`. That parameter says that we want our RDD to has at least 2 partitions.

Repartition an RDD

Sometimes we want to repartition an RDD, for example because it comes from a file that wasn't created by us, and the number of partitions defined from the creator is not the one we want.

The two most known functions to achieve this are:

```
repartition(numPartitions)
```

and:

```
coalesce(numPartitions, shuffle=False)
```

As a rule of thumb, use the first when you want to repartition your RDD in a greater number of partitions and the second to reduce your RDD, in a smaller number of partitions. Spark - repartition() vs coalesce().

For example:

```
data = sc.textFile(file)
data = data.coalesce(100) // requested number of #partitions
```

will decrease the number of partitions of the RDD called 'data' to 100, given that this RDD has more than 100 partitions when it got read by `textFile()`.

And in a similar way, if you want to have more than the current number of partitions for your RDD, you could do (given that your RDD is distributed in 200 partitions for example):

```
data = sc.textFile(file)
data = data.repartition(300) // requested number of #partitions
```

Rule of Thumb about number of partitions

As rule of thumb, one would want his RDD to have as many partitions as the product of the number of executors by the number of used cores by 3 (or maybe 4). Of course, that's a heuristic and it really depends on your application, dataset and cluster configuration.

Example:

```
In [1]: data  = sc.textFile(file)

In [2]: total_cores = int(sc._conf.get('spark.executor.instances')) *
int(sc._conf.get('spark.executor.cores'))

In [3]: data = data.coalesce(total_cores * 3)
```

Partitions Intro

> How does an RDD gets partitioned?

By default a partition is created for each HDFS partition, which by default is 64MB. Read more here.

> How to balance my data across partitions?

First, take a look at the three ways one can *repartition* his data:

1. Pass a second parameter, the desired *minimum* number of partitions for your RDD, into textFile(), but be careful:

   In [14]: lines = sc.textFile("data")

   In [15]: lines.getNumPartitions() Out[15]: 1000

   In [16]: lines = sc.textFile("data", 500)

   In [17]: lines.getNumPartitions() Out[17]: 1434

   In [18]: lines = sc.textFile("data", 5000)

   In [19]: lines.getNumPartitions() Out[19]: 5926

As you can see, `[16]` doesn't do what one would expect, since the number of partitions the RDD has, is already greater than the minimum number of partitions we request.

2. Use repartition(), like this:

   In [22]: lines = lines.repartition(10)

   In [23]: lines.getNumPartitions() Out[23]: 10

Warning: This will invoke a shuffle and should be used when you want to **increase** the number of partitions your RDD has.

From the docs:

> The shuffle is Spark's mechanism for re-distributing data so that it's grouped differently across partitions. This typically involves copying data across executors and machines, making the shuffle a complex and costly operation.

3. Use coalesce(), like this:

   In [25]: lines = lines.coalesce(2)

   In [26]: lines.getNumPartitions() Out[26]: 2

Here, Spark knows that you will shrink the RDD and gets advantage of it. Read more about

repartition() vs coalesce().

But will all this **guarantee** that your data will be perfectly balanced across your partitions? Not really, as I experienced in How to balance my data across the partitions?

Show RDD contents

To show contents of an RDD, it have to be printed:

```
myRDD.foreach(println)
```

To limit number of rows printed:

```
myRDD.take(num_of_rows).foreach(println)
```

Read Partitions online: http://www.riptutorial.com/apache-spark/topic/5822/partitions

Chapter 12: Shared Variables

Examples

User Defined Accumulator in Scala

Define `AccumulatorParam`

```
import org.apache.spark.AccumulatorParam

object StringAccumulator extends AccumulatorParam[String] {
  def zero(s: String): String = s
  def addInPlace(s1: String, s2: String)=  s1 + s2
}
```

Use:

```
val accumulator = sc.accumulator("")(StringAccumulator)
sc.parallelize(Array("a", "b", "c")).foreach(accumulator += _)
```

User Defined Accumulator in Python

Define `AccumulatorParam`:

```
from pyspark import AccumulatorParam

class StringAccumulator(AccumulatorParam):
    def zero(self, s):
        return s
    def addInPlace(self, s1, s2):
        return s1 + s2

accumulator = sc.accumulator("", StringAccumulator())

def add(x):
    global accumulator
    accumulator += x

sc.parallelize(["a", "b", "c"]).foreach(add)
```

Broadcast variables

Broadcast variables are read only shared objects which can be created with `SparkContext.broadcast` method:

```
val broadcastVariable = sc.broadcast(Array(1, 2, 3))
```

and read using `value` method:

```
val someRDD = sc.parallelize(Array(1, 2, 3, 4))

someRDD.map(
    i => broadcastVariable.value.apply(i % broadcastVariable.value.size)
)
```

Accumulators

Accumulators are write-only variables which can be created with `SparkContext.accumulator`:

```
val accumulator = sc.accumulator(0, name = "My accumulator") // name is optional
```

modified with `+=`:

```
val someRDD = sc.parallelize(Array(1, 2, 3, 4))
someRDD.foreach(element => accumulator += element)
```

and accessed with `value` method:

```
accumulator.value // 'value' is now equal to 10
```

Using accumulators is complicated by Spark's run-at-least-once guarantee for transformations. If a transformation needs to be recomputed for any reason, the accumulator updates during that transformation will be repeated. This means that accumulator values may be very different than they would be if tasks had run only once.

Note:

1. Executors *cannot* read accumulator's value. Only the driver program can read the accumulator's value, using its value method.
2. It is almost similar to counter in Java/MapReduce. So you can relate accumulators to counters to understanding it easily

Read Shared Variables online: http://www.riptutorial.com/apache-spark/topic/1736/shared-variables

Chapter 13: Spark DataFrame

Introduction

A DataFrame is an abstraction of data organized in rows and typed columns. It is similar to the data found in relational SQL-based databases. Although it has been transformed into just a type alias for Dataset[Row] in Spark 2.0, it is still widely used and useful for complex processing pipelines making use of its schema flexibility and SQL-based operations.

Examples

Creating DataFrames in Scala

There are many ways of creating DataFrames. They can be created from local lists, distributed RDDs or reading from datasources.

Using toDF

By importing spark sql implicits, one can create a DataFrame from a local Seq, Array or RDD, as long as the contents are of a Product sub-type (tuples and case classes are well-known examples of Product sub-types). For example:

```
import sqlContext.implicits._
val df = Seq(
  (1, "First Value", java.sql.Date.valueOf("2010-01-01")),
  (2, "Second Value", java.sql.Date.valueOf("2010-02-01"))
).toDF("int_column", "string_column", "date_column")
```

Using createDataFrame

Another option is using the `createDataFrame` method present in SQLcontext. This option also allows the creation from local lists or RDDs of Product sub-types as with `toDF`, but the names of the columns are not set in the same step. For example:

```
val df1 = sqlContext.createDataFrame(Seq(
  (1, "First Value", java.sql.Date.valueOf("2010-01-01")),
  (2, "Second Value", java.sql.Date.valueOf("2010-02-01"))
))
```

Additionally, this approach allows creation from RDDs of `Row` instances, as long as a `schema` parameter is passed along for the definition of the resulting DataFrame's schema. Example:

```
import org.apache.spark.sql.types._
val schema = StructType(List(
```

```
    StructField("integer_column", IntegerType, nullable = false),
    StructField("string_column", StringType, nullable = true),
    StructField("date_column", DateType, nullable = true)
))

val rdd = sc.parallelize(Seq(
  Row(1, "First Value", java.sql.Date.valueOf("2010-01-01")),
  Row(2, "Second Value", java.sql.Date.valueOf("2010-02-01"))
))

val df = sqlContext.createDataFrame(rdd, schema)
```

Reading from sources

Maybe the most common way to create DataFrame is from datasources. One can create it from a parquet file in hdfs, for example:

```
val df = sqlContext.read.parquet("hdfs:/path/to/file")
```

Read Spark DataFrame online: http://www.riptutorial.com/apache-spark/topic/8358/spark-dataframe

Chapter 14: Spark Launcher

Remarks

Spark Launcher can help developer to poll status of spark job submitted. There are basically eight statuses that can be polled.They are listed below with there meaning::

```
/** The application has not reported back yet. */
UNKNOWN(false),
/** The application has connected to the handle. */
CONNECTED(false),
/** The application has been submitted to the cluster. */
SUBMITTED(false),
/** The application is running. */
RUNNING(false),
/** The application finished with a successful status. */
FINISHED(true),
/** The application finished with a failed status. */
FAILED(true),
/** The application was killed. */
KILLED(true),
/** The Spark Submit JVM exited with a unknown status. */
LOST(true);
```

Examples

SparkLauncher

Below code is basic example of spark launcher.This can be used if spark job has to be launched through some application.

```
val sparkLauncher = new SparkLauncher
//Set Spark properties.only Basic ones are shown here.It will be overridden if properties are
set in Main class.
sparkLauncher.setSparkHome("/path/to/SPARK_HOME")
  .setAppResource("/path/to/jar/to/be/executed")
  .setMainClass("MainClassName")
  .setMaster("MasterType like yarn or local[*]")
  .setDeployMode("set deploy mode like cluster")
  .setConf("spark.executor.cores","2")

// Lauch spark application
val sparkLauncher1 = sparkLauncher.startApplication()

//get jobId
val jobAppId = sparkLauncher1.getAppId

//Get status of job launched.THis loop will continuely show statuses like RUNNING,SUBMITED
etc.
while (true) {
    println(sparkLauncher1.getState().toString)
}
```

Read Spark Launcher online: http://www.riptutorial.com/apache-spark/topic/8026/spark-launcher

Chapter 15: Stateful operations in Spark Streaming

Examples

PairDStreamFunctions.updateStateByKey

`updateState` by key can be used to create a stateful `DStream` based on upcoming data. It requires a function:

```
object UpdateStateFunctions {
  def updateState(current: Seq[Double], previous: Option[StatCounter]) = {
    previous.map(s => s.merge(current)).orElse(Some(StatCounter(current)))
  }
}
```

which takes a sequence of the `current` values, an `Option` of previous state and returns an `Option` of the updated state. Putting this all together:

```
import org.apache.spark._
import org.apache.spark.streaming.dstream.DStream
import scala.collection.mutable.Queue
import org.apache.spark.util.StatCounter
import org.apache.spark.streaming._

object UpdateStateByKeyApp {
  def main(args: Array[String]) {

    val sc = new SparkContext("local", "updateStateByKey", new SparkConf())
    val ssc = new StreamingContext(sc, Seconds(10))
    ssc.checkpoint("/tmp/chk")

    val queue = Queue(
      sc.parallelize(Seq(("foo", 5.0), ("bar", 1.0))),
      sc.parallelize(Seq(("foo", 1.0), ("foo", 99.0))),
      sc.parallelize(Seq(("bar", 22.0), ("foo", 1.0))),
      sc.emptyRDD[(String, Double)],
      sc.emptyRDD[(String, Double)],
      sc.emptyRDD[(String, Double)],
      sc.parallelize(Seq(("foo", 1.0), ("bar", 1.0)))
    )

    val inputStream: DStream[(String, Double)] = ssc.queueStream(queue)

    inputStream.updateStateByKey(UpdateStateFunctions.updateState _).print()

    ssc.start()
    ssc.awaitTermination()
    ssc.stop()
  }
}
```

PairDStreamFunctions.mapWithState

`mapWithState`, similarly to `updateState`, can be used to create a stateful DStream based on upcoming data. It requires `StateSpec`:

```
import org.apache.spark.streaming._

object StatefulStats {
  val state = StateSpec.function(
    (key: String, current: Option[Double], state: State[StatCounter]) => {
      (current, state.getOption) match {
        case (Some(x), Some(cnt)) => state.update(cnt.merge(x))
        case (Some(x), None) => state.update(StatCounter(x))
        case (None, None) => state.update(StatCounter())
        case _ =>
      }

      (key, state.get)
    }
  )
}
```

which takes key `key`, current `value` and accumulated `State` and returns new state. Putting this all together:

```
import org.apache.spark._
import org.apache.spark.streaming.dstream.DStream
import scala.collection.mutable.Queue
import org.apache.spark.util.StatCounter

object MapStateByKeyApp {
  def main(args: Array[String]) {
    val sc = new SparkContext("local", "mapWithState", new SparkConf())

    val ssc = new StreamingContext(sc, Seconds(10))
    ssc.checkpoint("/tmp/chk")

    val queue = Queue(
      sc.parallelize(Seq(("foo", 5.0), ("bar", 1.0))),
      sc.parallelize(Seq(("foo", 1.0), ("foo", 99.0))),
      sc.parallelize(Seq(("bar", 22.0), ("foo", 1.0))),
      sc.emptyRDD[(String, Double)],
      sc.parallelize(Seq(("foo", 1.0), ("bar", 1.0)))
    )

    val inputStream: DStream[(String, Double)] = ssc.queueStream(queue)

    inputStream.mapWithState(StatefulStats.state).print()

    ssc.start()
    ssc.awaitTermination()
    ssc.stop()
  }
}
```

Finally expected output:

```
-------------------------------------------
Time: 1469923280000 ms
-------------------------------------------
(foo,(count: 1, mean: 5.000000, stdev: 0.000000, max: 5.000000, min: 5.000000))
(bar,(count: 1, mean: 1.000000, stdev: 0.000000, max: 1.000000, min: 1.000000))

-------------------------------------------
Time: 1469923290000 ms
-------------------------------------------
(foo,(count: 3, mean: 35.000000, stdev: 45.284287, max: 99.000000, min: 1.000000))
(foo,(count: 3, mean: 35.000000, stdev: 45.284287, max: 99.000000, min: 1.000000))

-------------------------------------------
Time: 1469923300000 ms
-------------------------------------------
(bar,(count: 2, mean: 11.500000, stdev: 10.500000, max: 22.000000, min: 1.000000))
(foo,(count: 4, mean: 26.500000, stdev: 41.889736, max: 99.000000, min: 1.000000))

-------------------------------------------
Time: 1469923310000 ms
-------------------------------------------

-------------------------------------------
Time: 1469923320000 ms
-------------------------------------------
(foo,(count: 5, mean: 21.400000, stdev: 38.830916, max: 99.000000, min: 1.000000))
(bar,(count: 3, mean: 8.000000, stdev: 9.899495, max: 22.000000, min: 1.000000))
```

Read Stateful operations in Spark Streaming online: http://www.riptutorial.com/apache-spark/topic/1924/stateful-operations-in-spark-streaming

Chapter 16: Text files and operations in Scala

Introduction

Reading Text files and performing operations on them.

Examples

Join two files read with textFile()

Joins in Spark:

- Read textFile 1

```
val txt1=sc.textFile(path="/path/to/input/file1")
```

Eg:

```
  A B
  1 2
  3 4
```

- Read textFile 2

```
val txt2=sc.textFile(path="/path/to/input/file2")
```

Eg:

```
  A C
  1 5
  3 6
```

- Join and print the result.

```
txt1.join(txt2).foreach(println)
```

Eg:

```
  A B C
  1 2 5
  3 4 6
```

The join above is based on the first column.

Example usage

Read text file from path:

```
val sc: org.apache.spark.SparkContext = ???
sc.textFile(path="/path/to/input/file")
```

Read files using wildcards:

```
sc.textFile(path="/path/to/*/*")
```

Read files specifying minimum number of partitions:

```
sc.textFile(path="/path/to/input/file", minPartitions=3)
```

Read Text files and operations in Scala online: http://www.riptutorial.com/apache-spark/topic/1620/text-files-and-operations-in-scala

Chapter 17: Unit tests

Examples

Word count unit test (Scala + JUnit)

For example we have `WordCountService` with `countWords` method:

```
class WordCountService {
    def countWords(url: String): Map[String, Int] = {
        val sparkConf = new
SparkConf().setMaster("spark://somehost:7077").setAppName("WordCount"))
        val sc = new SparkContext(sparkConf)
        val textFile = sc.textFile(url)
        textFile.flatMap(line => line.split(" "))
                .map(word => (word, 1))
                .reduceByKey(_ + _).collect().toMap
    }
}
```

This service seems very ugly and not adapted for unit testing. SparkContext should be injected to this service. It can be reached with your favourite DI framework but for simplicity it will be implemented using constructor:

```
class WordCountService(val sc: SparkContext) {
    def countWords(url: String): Map[String, Int] = {
        val textFile = sc.textFile(url)
        textFile.flatMap(line => line.split(" "))
                .map(word => (word, 1))
                .reduceByKey(_ + _).collect().toMap
    }
}
```

Now we can create simple JUnit test and inject testable sparkContext to WordCountService:

```
class WordCountServiceTest {
    val sparkConf = new SparkConf().setMaster("local[*]").setAppName("WordCountTest")
    val testContext = new SparkContext(sparkConf)
    val wordCountService = new WordCountService(testContext)

    @Test
    def countWordsTest() {
        val testFilePath = "file://my-test-file.txt"

        val counts = wordCountService.countWords(testFilePath)

        Assert.assertEquals(counts("dog"), 121)
        Assert.assertEquals(counts("cat"), 191)
    }
}
```

Read Unit tests online: http://www.riptutorial.com/apache-spark/topic/3333/unit-tests

Chapter 18: Window Functions in Spark SQL

Examples

Introduction

Window functions are used to do operations(generally aggregation) on a set of rows collectively called as window. Window functions work in Spark 1.4 or later. Window functions provides more operations then the built-in functions or UDFs, such as substr or round (extensively used before Spark 1.4). Window functions allow users of Spark SQL to calculate results such as the rank of a given row or a moving average over a range of input rows. They significantly improve the expressiveness of Spark's SQL and DataFrame APIs.

At its core, a window function calculates a return value for every input row of a table based on a group of rows, called the Frame. Every input row can have a unique frame associated with it. This characteristic of window functions makes them more powerful than other functions. The types of window functions are

- Ranking functions
- Analytic functions
- Aggregate functions

To use window functions, users need to mark that a function is used as a window function by either

- Adding an `OVER` clause after a supported function in SQL, e.g. `avg(revenue) OVER (...);` or
- Calling the over method on a supported function in the DataFrame API, e.g. `rank().over(...)` .

This documentation aims to demonstrate some of those functions with example. It is assumed that the reader has some knowledge over basic operations on Spark DataFrame like: adding a new column, renaming a column etc.

Reading a sample dataset:

```
val sampleData = Seq(
("bob","Developer",125000),("mark","Developer",108000),("carl","Tester",70000),("peter","Developer",185
```

List of import statements required:

```
import org.apache.spark.sql.expressions.Window
import org.apache.spark.sql.functions._
```

The first statement imports `Window Specification`. A Window Specification contains conditions/specifications indicating, which rows are to be included in the window.

```
scala> sampleData.show
+------+---------+------+
|  Name|     Role|Salary|
+------+---------+------+
|   bob|Developer|125000|
|  mark|Developer|108000|
|  carl|   Tester| 70000|
| peter|Developer|185000|
|   jon|   Tester| 65000|
| roman|   Tester| 82000|
| simon|Developer| 98000|
|  eric|Developer|144000|
|carlos|   Tester| 75000|
| henry|Developer|110000|
+------+---------+------+
```

Moving Average

To calculate moving average of salary of the employers based on their role:

```
val movAvg = sampleData.withColumn("movingAverage", avg(sampleData("Salary"))
             .over( Window.partitionBy("Role").rowsBetween(-1,1)) )
```

- `withColumn()` creates a new column named `movingAverage`, performing `average` on `Salary` column
- `over()` is used to define window specification.
- `partitionBy()` partitions the data over the column `Role`
- `rowsBetween(start, end)` This function defines the rows that are to be included in the window. The parameters (`start` and `end`) takes numerical inputs,`0` represents the current row, `-1` is the previous row, `1` is the next row and so on. The function includes all rows in between `start` and `end`, thus in this example three rows(-1,0,1) are included in the window.

```
    scala> movAvg.show
+------+---------+------+------------------+
|  Name|     Role|Salary|     movingAverage|
+------+---------+------+------------------+
|   bob|Developer|125000|          116500.0|
|  mark|Developer|108000|139333.33333333334|
| peter|Developer|185000|130333.33333333333|
| simon|Developer| 98000|142333.33333333334|
|  eric|Developer|144000|117333.33333333333|
| henry|Developer|110000|          127000.0|
|  carl|   Tester| 70000|           67500.0|
|   jon|   Tester| 65000| 72333.33333333333|
| roman|   Tester| 82000|           74000.0|
|carlos|   Tester| 75000|           78500.0|
+------+---------+------+------------------+
```

Spark automatically ignores previous and next rows,if the current row is first and last row respectively.

In the above example, movingAverage of first row is average of current & next row only, as previous row doesn't exist. Similarly the last row of the partition (i.e 6th row) is average of current

& previous row, as next row doesn't exist.

Cumulative Sum

To calculate moving average of salary of the employers based on their role:

```
val cumSum = sampleData.withColumn("cumulativeSum", sum(sampleData("Salary"))
             .over( Window.partitionBy("Role").orderBy("Salary")))
```

- `orderBy()` sorts salary column and computes cumulative sum.

```
scala> cumSum.show
+------+---------+------+-------------+
|  Name|     Role|Salary|cumulativeSum|
+------+---------+------+-------------+
| simon|Developer| 98000|        98000|
|  mark|Developer|108000|       206000|
| henry|Developer|110000|       316000|
|   bob|Developer|125000|       441000|
|  eric|Developer|144000|       585000|
| peter|Developer|185000|       770000|
|   jon|   Tester| 65000|        65000|
|  carl|   Tester| 70000|       135000|
|carlos|   Tester| 75000|       210000|
| roman|   Tester| 82000|       292000|
+------+---------+------+-------------+
```

Window functions - Sort, Lead, Lag , Rank , Trend Analysis

This topic demonstrates how to use functions like withColumn, lead, lag, Level etc using Spark. Spark dataframe is an sql abstract layer on spark core functionalities. This enable user to write SQL on distributed data. Spark SQL supports hetrogenous file formats including JSON, XML, CSV , TSV etc.

In this blog we have a quick overview of how to use spark SQL and dataframes for common use cases in SQL world.For the sake of simplicity we will deal with a single file which is CSV format. File has four fields, employeeID, employeeName, salary, salaryDate

```
1,John,1000,01/01/2016
1,John,2000,02/01/2016
1,John,1000,03/01/2016
1,John,2000,04/01/2016
1,John,3000,05/01/2016
1,John,1000,06/01/2016
```

Save this file as emp.dat. In the first step we will create a spark dataframe using , spark CSV package from databricks.

```
val sqlCont = new HiveContext(sc)
//Define a schema for file
val schema = StructType(Array(StructField("EmpId", IntegerType, false),
          StructField("EmpName", StringType, false),
```

```
          StructField("Salary", DoubleType, false),
          StructField("SalaryDate", DateType, false)))
//Apply Shema and read data to a dataframe
val myDF = sqlCont.read.format("com.databricks.spark.csv")
          .option("header", "false")
          .option("dateFormat", "MM/dd/yyyy")
          .schema(schema)
          .load("src/resources/data/employee_salary.dat")
//Show dataframe
myDF.show()
```

myDF is the dataframe used in remaining excercise. Since myDF is used repeatedly it is recommended to persist it so that it does not need to be reevaluated.

```
 myDF.persist()
```

Output of dataframe show

```
+-----+-------+------+----------+
|EmpId|EmpName|Salary|SalaryDate|
+-----+-------+------+----------+
| 1| John|1000.0|2016-01-01|
| 1| John|2000.0|2016-02-01|
| 1| John|1000.0|2016-03-01|
| 1| John|2000.0|2016-04-01|
| 1| John|3000.0|2016-05-01|
| 1| John|1000.0|2016-06-01|
+-----+-------+------+----------+
```

Add a new column to dataframe

Since spark dataframes are immutable, adding a new column will create a new dataframe with added column. To add a column use withColumn(columnName,Transformation). In below example column empName is formatted to uppercase.

```
withColumn(columnName,transformation)
myDF.withColumn("FormatedName", upper(col("EmpName"))).show()


+-----+-------+------+----------+------------+
|EmpId|EmpName|Salary|SalaryDate|FormatedName|
+-----+-------+------+----------+------------+
| 1| John|1000.0|2016-01-01| JOHN|
| 1| John|2000.0|2016-02-01| JOHN|
| 1| John|1000.0|2016-03-01| JOHN|
| 1| John|2000.0|2016-04-01| JOHN|
| 1| John|3000.0|2016-05-01| JOHN|
| 1| John|1000.0|2016-06-01| JOHN|
+-----+-------+------+----------+------------+
```

Sort data based on a column

```
val sortedDf = myDF.sort(myDF.col("Salary"))
sortedDf.show()
```

```
+-----+-------+------+----------+
|EmpId|EmpName|Salary|SalaryDate|
+-----+-------+------+----------+
| 1| John|1000.0|2016-03-01|
| 1| John|1000.0|2016-06-01|
| 1| John|1000.0|2016-01-01|
| 1| John|2000.0|2016-02-01|
| 1| John|2000.0|2016-04-01|
| 1| John|3000.0|2016-05-01|
+-----+-------+------+----------+
```

Sort Descending

desc("Salary")

```
 myDF.sort(desc("Salary")).show()


+-----+-------+------+----------+
|EmpId|EmpName|Salary|SalaryDate|
+-----+-------+------+----------+
| 1| John|3000.0|2016-05-01|
| 1| John|2000.0|2016-02-01|
| 1| John|2000.0|2016-04-01|
| 1| John|1000.0|2016-06-01|
| 1| John|1000.0|2016-01-01|
| 1| John|1000.0|2016-03-01|
+-----+-------+------+----------+
```

Get and use previous row (Lag)

LAG is a function in SQL which is used to access previous row values in current row. This is useful when we have use cases like comparison with previous value. LAG in Spark dataframes is available in Window functions

```
lag(Column e, int offset)
Window function: returns the value that is offset rows before the current row, and null if
there is less than offset rows before the current row.


import org.apache.spark.sql.expressions.Window
//order by Salary Date to get previous salary.
//For first row we will get NULL
val window = Window.orderBy("SalaryDate")
//use lag to get previous row value for salary, 1 is the offset
val lagCol = lag(col("Salary"), 1).over(window)
myDF.withColumn("LagCol", lagCol).show()

+-----+-------+------+----------+------+
|EmpId|EmpName|Salary|SalaryDate|LagCol|
+-----+-------+------+----------+------+
| 1| John|1000.0|2016-01-01| null|
| 1| John|2000.0|2016-02-01|1000.0|
| 1| John|1000.0|2016-03-01|2000.0|
| 1| John|2000.0|2016-04-01|1000.0|
| 1| John|3000.0|2016-05-01|2000.0|
```

```
| 1| John|1000.0|2016-06-01|3000.0|
+-----+-------+------+----------+------+
```

Get and use next row (Lead)

LEAD is a function in SQL which is used to access next row values in current row. This is useful when we have usecases like comparison with next value. LEAD in Spark dataframes is available in Window functions

```
lead(Column e, int offset)
Window function: returns the value that is offset rows after the current row, and null if
there is less than offset rows after the current row.


import org.apache.spark.sql.expressions.Window
//order by Salary Date to get previous salary. F
//or first row we will get NULL
val window = Window.orderBy("SalaryDate")
//use lag to get previous row value for salary, 1 is the offset
val leadCol = lead(col("Salary"), 1).over(window)
myDF.withColumn("LeadCol", leadCol).show()




+-----+-------+------+----------+-------+
|EmpId|EmpName|Salary|SalaryDate|LeadCol|
+-----+-------+------+----------+-------+
| 1| John|1000.0|2016-01-01| 1000.0|
| 1| John|1000.0|2016-03-01| 1000.0|
| 1| John|1000.0|2016-06-01| 2000.0|
| 1| John|2000.0|2016-02-01| 2000.0|
| 1| John|2000.0|2016-04-01| 3000.0|
| 1| John|3000.0|2016-05-01| null|
+-----+-------+------+----------+-------+
```

Trend analysis with window functions Now, let us put window function LAG to use with a simple trend analysis. If salary is less than previous month we will mark it as "DOWN", if salary has increased then "UP". The code use Window function to order by, lag and then do a simple if else with WHEN.

```
   val window = Window.orderBy("SalaryDate")
    //Derive lag column for salary
    val laggingCol = lag(col("Salary"), 1).over(trend_window)
    //Use derived column LastSalary to find difference between current and previous row
    val salaryDifference = col("Salary") - col("LastSalary")
    //Calculate trend based on the difference
    //IF ELSE / CASE can be written using when.otherwise in spark
    val trend = when(col("SalaryDiff").isNull || col("SalaryDiff").===(0), "SAME")
    .when(col("SalaryDiff").>(0), "UP")
    .otherwise("DOWN")
    myDF.withColumn("LastSalary", laggingCol)
    .withColumn("SalaryDiff",salaryDifference)
   .withColumn("Trend", trend).show()

+-----+-------+------+----------+----------+----------+-----+
```

```
|EmpId|EmpName|Salary|SalaryDate|LastSalary|SalaryDiff|Trend|
+-----+-------+------+----------+----------+----------+-----+
| 1| John|1000.0|2016-01-01| null| null| SAME|
| 1| John|2000.0|2016-02-01| 1000.0| 1000.0| UP|
| 1| John|1000.0|2016-03-01| 2000.0| -1000.0| DOWN|
| 1| John|2000.0|2016-04-01| 1000.0| 1000.0| UP|
| 1| John|3000.0|2016-05-01| 2000.0| 1000.0| UP|
| 1| John|1000.0|2016-06-01| 3000.0| -2000.0| DOWN|
+-----+-------+------+----------+----------+----------+-----+
```

Read Window Functions in Spark SQL online: http://www.riptutorial.com/apache-spark/topic/3903/window-functions-in-spark-sql

Credits

S. No	Chapters	Contributors
1	Getting started with apache-spark	4444, Ani Menon, Community, Daniel de Paula, David, gsamaras, himanshuIIITian, Jacek Laskowski, KartikKannapur, Naresh Kumar, user8371915, zero323
2	Calling scala jobs from pyspark	eliasah, Thiago Baldim
3	Client mode and Cluster Mode	Nayan Sharma
4	Configuration: Apache Spark SQL	himanshuIIITian
5	Error message 'sparkR' is not recognized as an internal or external command or '.binsparkR' is not recognized as an internal or external command	Rajesh
6	Handling JSON in Spark	Furkan Varol, zero323
7	How to ask Apache Spark related question?	user7337271
8	Introduction to Apache Spark DataFrames	Mandeep Lohan, Nayan Sharma
9	Joins	Adnan, CGritton
10	Migrating from Spark 1.6 to Spark 2.0	Béatrice Moissinac, eliasah, Shaido
11	Partitions	Ani Menon, Armin Braun, gsamaras
12	Shared Variables	Community, Jonathan Taws, RBanerjee, saranvisa, spiffman,

		whaleberg, zero323
13	Spark DataFrame	Daniel de Paula
14	Spark Launcher	Ankit Agrahari
15	Stateful operations in Spark Streaming	zero323
16	Text files and operations in Scala	Ani Menon, Community, spiffman
17	Unit tests	Cortwave
18	Window Functions in Spark SQL	Daniel Argüelles, Hari, Joshua Weinstein, Tejus Prasad, vdep