



DEV 361: Build and Monitor Apache Spark Applications Slide Guide

Version 5.0 – November 2015

This Guide is protected under U.S. and international copyright laws, and is the exclusive property of MapR Technologies, Inc.

© 2015, MapR Technologies, Inc. All rights reserved. All other trademarks cited here are the property of their respective owners.



Lesson 4: Work with Pair RDD



DEV 361: Build and Monitor Apache Spark Applications

Lesson 4: Work with Pair RDD

© 2015 MapR Technologies 1

Notes on Slide 1:

Welcome to Apache Spark: Build and Monitor Apache Spark Applications – Lesson 4: Work with Pair RDD.

In this lesson we will describe key-value pairs as data sets and how and where they are used. We create pair RDD and apply operations to them. This lesson also describes how to control partitioning across nodes and use the partitioner property.

 Learning Goals

- ▶ 1. Describe & create pair RDD
- 2. Apply transformations & actions to pair RDD
- 3. Control partitioning across nodes

© 2015 MapR Technologies 

2

Notes on Slide 2:

At the end of this lesson, you will be able to:

- 1. Describe and create pair RDD
- 2. Apply transformations and actions to pair RDD
- 3. Control partitioning across nodes



 Review

- Create RDD
- Apply transformations to RDD
- Apply actions to RDD

© 2015 MapR Technologies 

3

Notes on Slide 3:

Before we start with pair RDDs, let us review creating RDD and applying actions and transformations to RDD. You should complete DEV 360 before continuing.





Scenario: SFPD Incidents Data

Note:
This information is from the file sfpd.csv

IncidentNum → Incident number	150599321
Category	OTHER_OFFENSES
Description	POSSESSION_OF_BURGLARY_TOOLS
DayOfWeek	Thursday
Date	7/9/15
Time	23:45
PdDistrict → PD District	CENTRAL
Resolution	ARREST/BOOKED
Address	JACKSON_ST/POWELL_ST
X → Longitudinal Coordinate	-122.4099006
Y → Latitudinal Coordinate	37.79561712
PdID → Police Department ID	15059900000000

© 2015 MapR Technologies 

4

Notes on Slide 4:

We will use this dataset in examples and Lab activities. sfpd.csv contains the fields shown here with example data points. This dataset consists of Incidents derived from the SFPD Crime Incident Reporting system between Jan 2013 - July 2015. We are going to explore this data to answer questions such as which districts have the maximum incidents or which five categories have the maximum number of incidents.

First, we will do a quick review.





Review: Map Input Fields

1. Map input fields

```
val IncidntNum = 0  
val Category = 1  
val Descript = 2  
val DayOfWeek = 3  
val Date = 4  
val Time = 5  
val PdDistrict = 6  
val Resolution = 7  
val Address = 8  
val X = 9  
val Y = 10  
val PdId = 11
```

© 2015 MapR Technologies 

5

Notes on Slide 5:

We will use the Spark Interactive Shell to load the data, create RDD and apply transformations and actions. First we map the input fields.



 Review: Load Data into Spark

2. Load data and split on separator

```
val sfpd = sc.textFile("/path/to/file/  
sfpd.csv").map(line=>line.split(","))
```

© 2015 MapR Technologies 

6

Notes on Slide 6:

We load the csv file using the SparkContext method `textFile`. We are also applying the `map` transformation to split on the comma.





Review: Transformations & Actions

How do you know what the data looks like in the RDD?

```
sfpd.first()
```

What is the total number of incidents?

```
val totincs = sfpd.count()
```

What are the Categories?

```
val cat = sfpd.map(inc=>inc(Category)).distinct.collect()
```

© 2015 MapR Technologies 

7

Notes on Slide 7:

This provides a review of creating RDDs, and applying data set operations to RDD that were covered in the DEV 360 course.





Review: Transformations

Note:
Transformations
are lazily evaluated



bayviewRDD; note this is a **new RDD**

```
val bayviewRDD = sfpd.filter(incident=>incident.contains("BAYVIEW"))  
filter()
```

© 2015 MapR Technologies

8

Notes on Slide 8:

Transformations are lazily evaluated. In this example, we are defining the instructions for creating the bayviewRDD. Nothing is computed until we add an action.

The filter transformation is used to filter all those elements in the sfpd RDD that contain "BAYVIEW".

Other examples of transformations include map, filter and distinct.



 Review: Actions

Note:
Actions evaluated immediately. Every action will compute from start.

```
val numTenderloin=sfpd  
.filter(incident=>incident.contains("TENDERLOIN")).count()
```

Returns a value of the
count() to driver

© 2015 MapR Technologies 

9

Notes on Slide 9:

When we run the command for an action, Spark will load the data, create the inputRDD, and then compute any other defined transformations and actions.

In this example, calling the count action will result in the data being loaded into the sfpd RDD, the filter transformation being applied and then the count.

We will now look at pair RDDs that are a type of RDD.





Lab 4.1 – REVIEW: Load Data & Explore Data

Notes on Slide 10:

In this activity you will load data into Apache Spark and explore the data using dataset operations. This activity is a review of creating RDDs and applying transformations and actions on it.



 Learning Goals

- ▶ 1. Describe & create pair RDD
- 2. Apply transformations & actions to pair RDD
- 3. Control partitioning across nodes

© 2015 MapR Technologies 

11

Notes on Slide 11:

In this section we will describe key-value pairs as data sets and create pair RDDs.



The diagram illustrates a **Pair RDD** as a collection of **2-Tuples**. A central blue box labeled "Pair RDD" contains four entries: (CENTRAL,2), (TENDERLOIN, 1), (RICHMOND, 0), and (BAYVIEW, 6). Red callout boxes point to these elements: one points to the first entry with the label "Key Identifier", another points to the second entry with the label "Value", and a third points to the entire list with the label "2-Tuple".

Note:
Pair RDD is a type
of RDD. Each
element of Pair
RDD is a 2-tuple

© 2015 MapR Technologies **MAPR** 12

Notes on Slide 12:

The key-value pair paradigm can be found in a number of programming languages. It is a data type that consists of a group of key identifiers with a set of associated values.

When working with distributed data, it is useful to organize data into key-value pairs as it allows us to aggregate data or regroup data across the network.

Similar to MapReduce, Spark supports Key-Value pairs in the form of Pair RDD.

Spark's representation of a key-value pair is a Scala 2-tuple





Motivation for Pair RDD

Useful for:

- Aggregating data across network
- Regrouping data across network

Example:

Want to view all SFPD incident categories together by district

Key-Value pairs used in Map - Reduce algorithms

© 2015 MapR Technologies MAPR

13

Notes on Slide 13:

Pair RDDs are used in many Spark programs. They are useful when you want to aggregate and/or regroup data across a network.

For example, you can extract the District and use it to view all SFPD incidents together for a district.

Another example, is extracting the customerID and using it as an identifier to view all the customer's orders together.

A very common use case is in map reduce algorithms.

Let us take a brief look at the differences between Hadoop MapReduce and Apache Spark.





Common Example: Pair RDD in Word Count

```
val textFile=spark.textFile("hdfs://...")  
val count=textFile.flatMap(line=>line.split(","))  
    .map(word=>(word,1))  
    .reduceByKey(_+_)  
count.saveAsTextFile("hdfs://...")
```

© 2015 MapR Technologies

14

Notes on Slide 14:

Word count is a typical example used to demonstrate Hadoop MapReduce.

In the code block shown here, the first line defines the location of the input file. The second line loads the data into an RDD; splits the file on the separator into dataset whose elements are the words in the input file; the map transformation is applied to the out put of the flatMap to create a pair RDD consisting of tuples for each occurrence of the word – (word,1); finally the reduceByKey operation then reduces the pair RDD to give the count.





Word Count: Apache Spark vs Apache Hadoop MapReduce

Spark Word Count

```
val textFile=spark.textFile("hdfs://...")
val counts=textFile.flatMap(line=>line.split(","))
    .map(word=>(word,1))
    .reduceByKey{+_}
counts.saveAsTextFile("hdfs://...")
```

MapReduce Word Count

```
import java.io.IOException
import org.apache.hadoop.conf.Configuration
import org.apache.hadoop.fs.Path
import org.apache.hadoop.io.IntWritable
import org.apache.hadoop.io.Text
import org.apache.hadoop.mapreduce.Mapper
import org.apache.hadoop.mapreduce.Reducer
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat
public class WordCountMapper extends Mapper<Text, IntWritable>
{
    private Text word = new Text();
    private IntWritable value = new IntWritable();
    public void map(Text key, IntWritable value, Context context
    ) throws IOException, InterruptedException {
        StringTokenizer it = new StringTokenizer(key.toString());
        while(it.hasMoreTokens()){
            word.set(it.nextToken());
            value.set(1);
            context.write(word, value);
        }
    }
    public static class IntSumReducer
    extends Reducer<Text, IntWritable, Text, IntWritable>
    {
        private IntWritable result = new IntWritable();
        public void reduce(Text key, Iterable<IntWritable> values,
                           Context context
                           ) throws IOException, InterruptedException {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            result.set(sum);
            context.write(key, result);
        }
    }
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "word count");
        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(IntSumReducer.class);
        job.setCombinerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true));
    }
}
```

© 2015 MapR Technologies

15

Notes on Slide 15:

On the left is the code we just saw that gives us the count of words in Scala using Spark. On the right is the Hadoop MapReduce program for word count.

We will discuss some differences between Apache Spark and Hadoop MapReduce next.





Apache Spark vs Apache Hadoop MapReduce

Apache Spark

- Spark tries to keep everything in memory
- Chaining multiple jobs in faster
- A combination of any number of map & reduce operations
- map() & reduce() already implemented in Spark
- Support for the Interactive Shell

Apache Hadoop MapReduce

- Read & write from/to disk after every job
- One map & one reduce per job
- Have to implement these methods in Hadoop MapReduce
- Jobs are all batch (non-interactive)

© 2015 MapR Technologies



16

Notes on Slide 16:

This table lists differences between Apache Spark and Apache Hadoop MapReduce.

- The key differentiator between Apache Spark and MapReduce is that Spark keeps everything in memory while MapReduce reads from and writes to disk for every job.
- You can have only one map and one reduce per MapReduce job. In Spark, you can have a combination of any number of map and reduce operations.
- Furthermore, Spark provides high level operators such as map, reduce and joins whereas in MapReduce, you have to implement these methods.
- Apache Spark provides the Interactive Shell fro Scala and Python that allows you to write your code interactively. Apache Hadoop MapRedude jobs on the other hand are all batch.

We will now take a look at creating pair RDDs.





Create Pair RDD

1. Create from existing non-pair RDD
2. Create pair RDD by loading certain formats
3. Create pair RDD from in-memory collection of pairs

© 2015 MapR Technologies 

17

Notes on Slide 17:

Pair RDD can be created in the following ways:

1. Pair RDDs are most often created from already-existing non-pair RDDs.
2. You can also get pair RDD when loading data. There are many data formats that when loaded, will create pair RDD.
3. Another way is to create a pair RDD from an in-memory collection of pairs.





1. Create From Existing Non-Pair RDD

- Different ways to create from existing non-pair RDD
- Most common way is use `map()` on existing RDD
- Building key-value RDDs differs by language

Example:

```
val incByCat = sfpd.map(incident=>(incident(Category),1))
```

Results:

```
Array((LARCENY/THEFT, 1), ...)
```

© 2015 MapR Technologies



18

Notes on Slide 18:

There are many ways to create a pair RDD from existing non-pair RDDs. The most common way is using a map transformation. The way to create pair RDDs is different in different languages. In Python and Scala, we need to return an RDD consisting of tuples.

In this example, we are creating a pair RDD called incByCat from the sfpd RDD by applying a map transformation. The resulting RDD contains tuples as shown.





2. Create Pair RDD By Loading Certain Formats

- Many formats will directly return pair RDD
- Examples
 - SequenceFiles create pair RDD
 - sc.wholeTextFile on small files creates pair RDD

rdd from SequenceFile

```
Array((OTHER_OFFENSES,1), (LARCENY/THEFT,1),  
(OTHER_OFFENSES,1), ... )
```

© 2015 MapR Technologies  19

Notes on Slide 19:

Many formats will directly return pair RDDs for their key-value data.

For examples: SequenceFiles will create pair RDD. In this example, the SequenceFile consists of key-value pairs (Category,1) . When loaded into Spark it produces a pair RDD as shown. sc.wholeTextFiles on a group of small text files will create pair RDD where the key is the name of the file.





3. Create Pair RDD From In-Memory Collection of Pairs

Call `SparkContext.parallelize()` on collection of pairs

```
//creating a collection of pairs in memory
val dist1 = Array(("INGLESIDE",1), ("SOUTHERN",1), ("PARK",
1), ("NORTHERN",1))
// Use parallelize() to create a Pair RDD
val distRDD = SparkContext.parallelize(dist1)

Array[(String, Int)] = Array((INGLESIDE,1), (SOUTHERN,1), (PARK,1)
(NORTHERN,1))
```

© 2015 MapR Technologies 

20

Notes on Slide 20:

To create a pair RDD from an in-memory collection in Scala and Python, use `SparkContext.parallelize()` method on a collection of pairs.

In this example, we have a collection of pairs, `dist1`, in memory. We create a pair RDD, `distRDD`, by applying the method `SparkContext.parallelize` to `dist1`.





Which three districts have the highest number of incidents?

1. Create a pair RDD
 - Use existing sfpd RDD (non-pair)
 - Apply `map()`
2. "Combine/collect" by district
3. Sort & show top 3

```
val top3Dists = sfpd.map(inc=>(inc(PdDistrict),1))
```

Result:

```
Array((CENTRAL,1), (CENTRAL,1), (CENTRAL,1), (PARK,1),...)
```

Notes on Slide 21:

We want to answer this question of the sfpd dataset – which three districts (or Categories or Addresses) have the highest number of incidents?

Each row in the dataset represents an incident. We want to count the number of incidents in each district - i.e. how many times does the district occur in the dataset?

The first step is to create a pairRDD from the sfpd RDD.

We can do this by applying the map transformation on sfpd RDD. The map transformation results in pair tuples consisting of the PdDistrict and a count of 1 for each element (incident) in the sfpd RDD that occurs in that PdDistrict.

In the next section we take a look at some of the transformations and actions that we can use on pair RDD to get the final answer to the question.



 Knowledge Check

Given the sfpd RDD, to create a Pair RDD consisting of tuples of the form (Category, 1), in Scala, use:

1. val pairs = sfpd.parallelize()
2. val pairs = sfpd.map(x=>(x(Category),1))
3. val pairs=sfpd.map(x=>x.parallelize())

Notes on Slide 22:

Answer: 2



 Learning Goals

1. Describe & create pair RDD
- ▶ **2. Apply transformations & actions to pair RDD**
3. Control partitioning across nodes

© 2015 MapR Technologies  23

Notes on Slide 23:

In this section we see how to apply transformations and actions to pair RDDs.





Transformation on Pair RDDs

- **Transformations specific to pair RDD**
 - `reduceByKey()`, `groupByKey()`, `aggregateByKey()`,
`combineByKey()`
- **Transformations that work on two pair RDD**
 - `join()`, `cogroup()`, `subtractByKey()`
- **Transformations that apply to RDD**
 - `filter()`, `map()`

© 2015 MapR Technologies 

24

Notes on Slide 24:

When you create a pair RDD, Spark automatically adds a number of additional methods that are specific to pair RDDs such as `reduceByKey`, `groupByKey`, `combineByKey`.

The most common ones are distributed “shuffle” operations, such as grouping or aggregating the elements by a key.

There are also transformations that work on two pair RDDs such as `join`, `cogroup`, `subtractByKey`.

Pair RDDs are a type of RDDs and therefore also support the same operations as other RDDs such as `filter()` and `map()`.





Apply Transformation to Pair RDD

Which three districts have the highest number of incidents?

1. Create a pair RDD
2. “Combine” or “add” by key
3. Sort
4. Show top 3

```
val top3Dists=sfpd.map(incident=>(incident(PdDistrict),1))  
.reduceByKey (x,y)=>x+y)
```

© 2015 MapR Technologies

MAPR

25

Notes on Slide 25:

We usually want to find statistics across elements with the same key. `reduceByKey` is an example of such a transformation. It runs several parallel reduce operations, one for each key in the dataset, where each operation combines values that have the same key. It returns a new RDD consisting of each key and the reduced value for that key.

In this example, use the `reduceByKey` to apply a simple sum to the pair RDD consisting of pair tuples (district, 1). It returns a new RDD with the reduced value (in this case, the sum) for each district.





Apply Transformation to Pair RDD

Which three districts have the highest number of incidents?

1. Create a pair RDD
2. “Combine” by key
3. Sort
4. Show top 3

```
val top3Dists=sfpd.map(incident=>(incident(PdDistrict),1))
    .reduceByKey((x,y)=>x+y)
    .map(x=>(x._2,x._1))
    .sortByKey(false).take(3)
```

© 2015 MapR Technologies 

26

Notes on Slide 26:

We can sort an RDD with key/value pairs, provided that there is an ordering defined on the key. Once we have sorted our data, any future calls on the sorted data to collect() or save() will result in ordered data. The sortByKey() function takes a parameter called ascending, that indicates that the results are in ascending order (by default set to true).

In our example, we apply another map operation to the results of the reduceByKey. This map operation is switching the order of the tuple – we swap district and the sum to give us a dataset of tuples(sum, district). We then apply sortByKey to this dataset. The sorting is then done on the sum which is the number of incidents for each district.

Since we want the top 3, we need the results to be in descending order. We pass in the value “false” to the “ascending” parameter to sortByKey. To return three elements, we use take(3) – this will give us the top 3 elements.

You can also use:

```
val top3 Dists= sfpd.map(incident=>(incident(PdDistrict),1)).reduceByKey(_ +
_).top(3)(Ordering.by(_.-_2))
```





Note:
You can use
groupByKey instead
of reduceByKey



Which three districts have the highest number of incidents?

```
val pddists = sfpd.map(x=> (x(PdDistrict),1))
  .groupByKey.map(x => (x._1, x._2.size))
  .map(x=>(x._2,x._1))
  .sortByKey(false).take(3)
```

© 2015 MapR Technologies

27

Notes on Slide 27:

This is another way to answer the question:

What are the top 3 districts with the max number of incidents?

You can use groupByKey instead of reduceByKey.

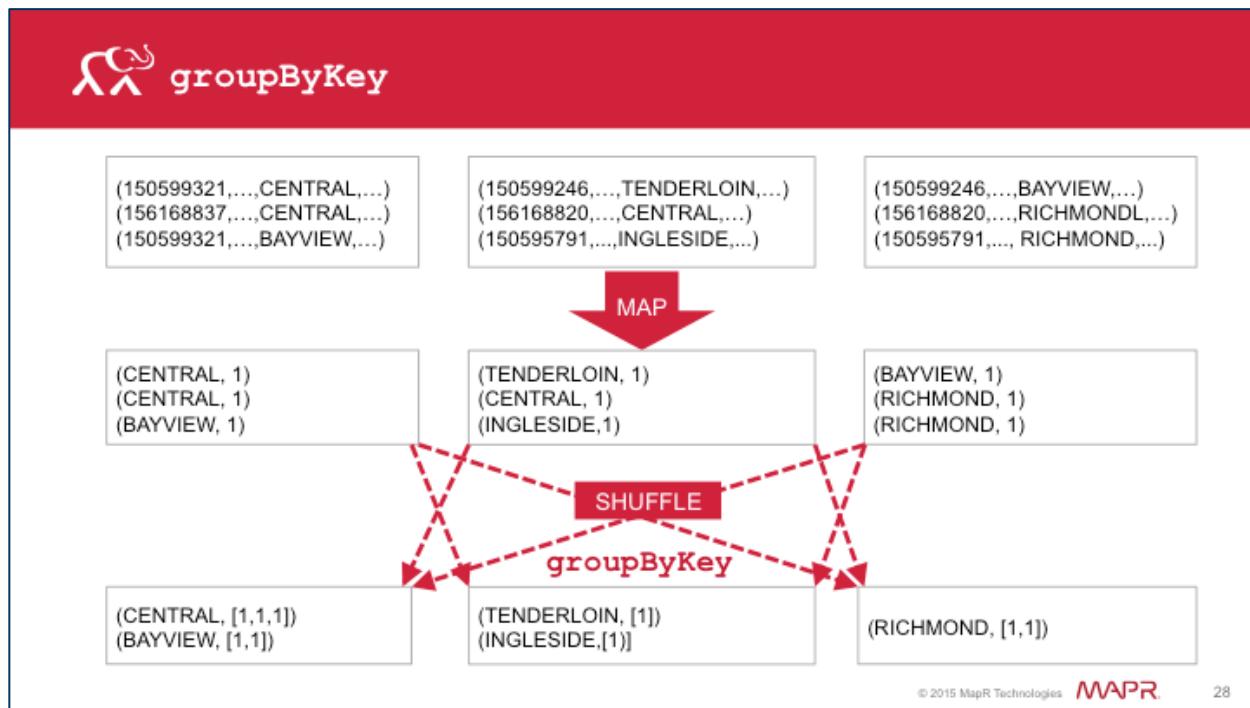
It groups all values that have the same key. As a result, it takes no argument. It results in a dataset of key/value pairs where the value is an iterable list.

Here we apply the groupByKey to the dataset consisting of the tuples (district, 1). The map transformation highlighted here is applied to every element of the result of the groupByKey transformation and it returns the district and the size of the iterable list.

Next we will see the difference between using groupByKey versus using reduceByKey.

You can also use: val pddists = sfpd.map(x=>
 (x(PdDistrict),1)).**groupByKey**.map(x => (x._1,
 x._2.size)).top(3)(Ordering.by(_._2))

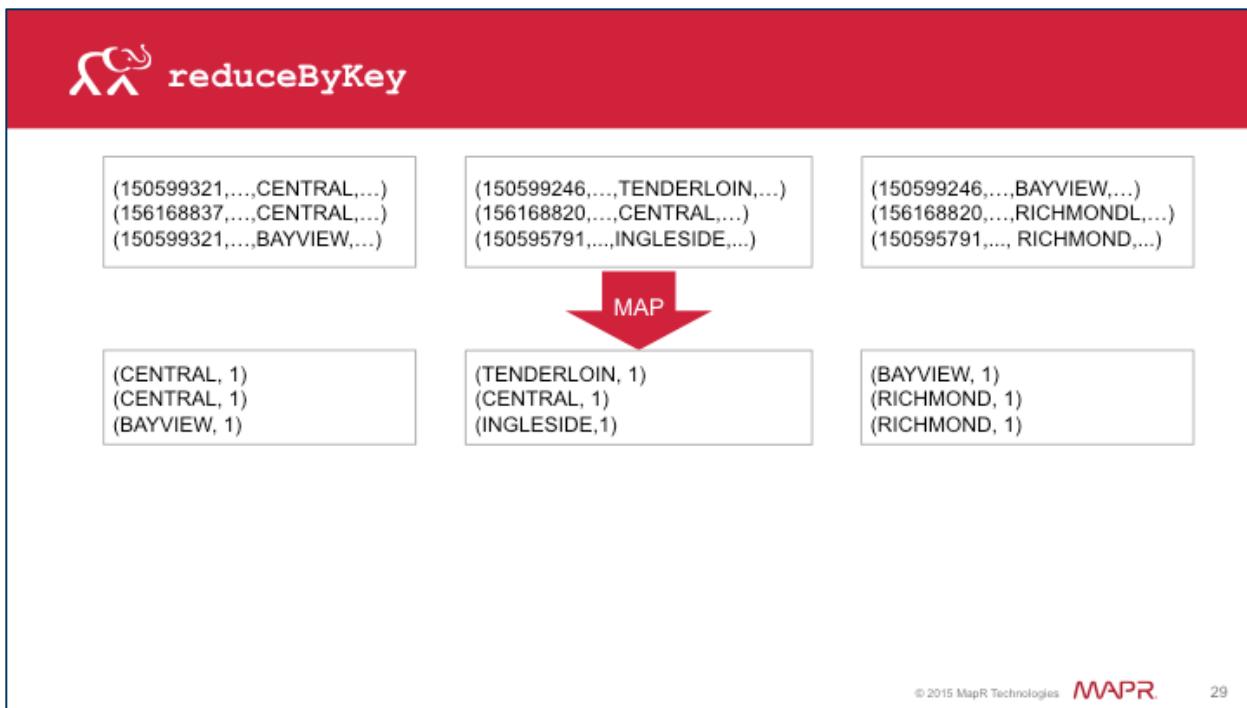




Notes on Slide 28:

The first map transformation results in a pairRDD consisting of pairs of District name and 1. The `groupByKey` groups all values that have the same key resulting in a key-iterable value pair. It groups all values with the same key onto the same machine. `groupByKey` results in one key-value pair per key. This single key-value pair cannot span across multiple worker nodes. When handling large datasets, `groupByKey` causes a lot of unnecessary transfer of data over the network. When there is more data shuffled onto a single executor machine than can fit in memory, Spark spills data to disk. However, it flushes out the data to disk one key at a time. Thus, if a single key has more key-value pairs than can fit in memory, there will be an out of memory exception.



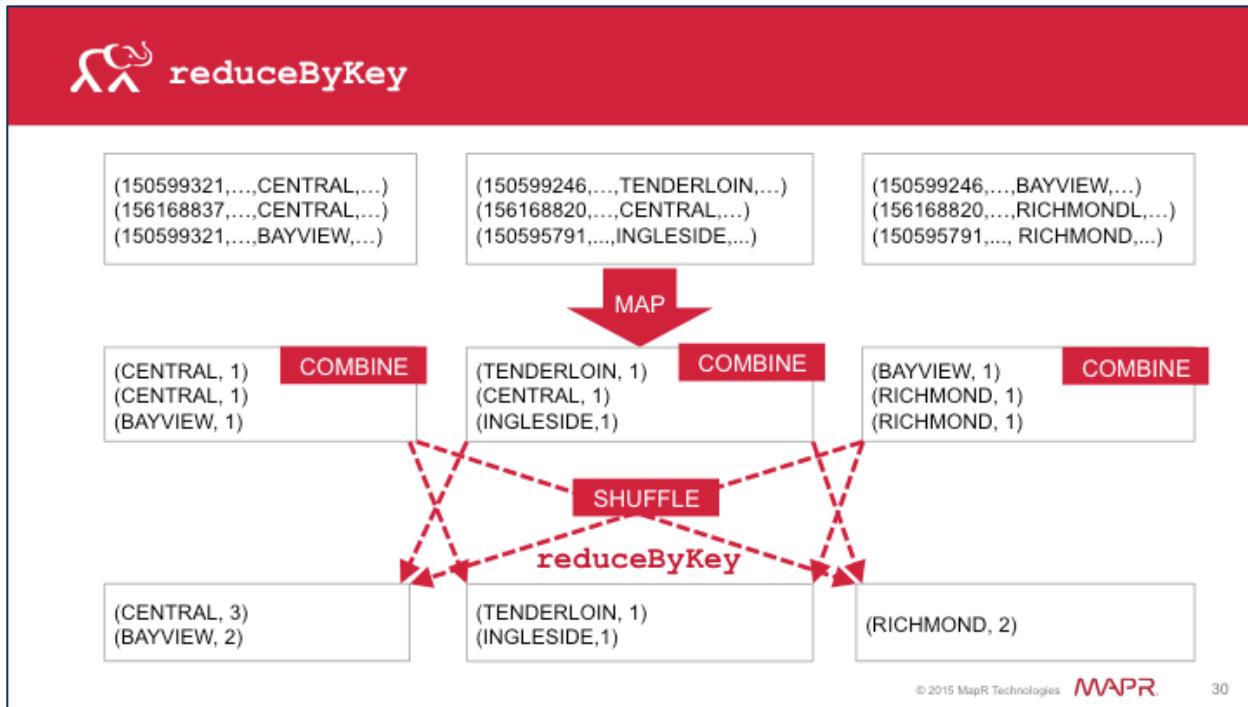


Notes on Slide 29:

`reduceByKey` combines values that have the same key .

The first map transformation results in a pairRDD consisting of pairs of District name and 1.





Notes on Slide 30:

`reduceByKey` will automatically perform a combine locally on each machine. The user does not have to specify a combiner. The data is reduced again after the shuffle.

`reduceByKey` can be thought of as a combination of `groupByKey` and a `reduce` on all the values per key. It is more efficient though, than using each separately. With `reduceByKey`, data is combined so each partition outputs at most one value for each key to send over the network resulting in better performance. In general, `reduceByKey` is more efficient especially for large datasets.

Not all problems that can be solved by `groupByKey` can be computed using `reduceByKey`. This is because `reduceByKey` requires combining all values into another value with the exact same type.

http://databricks.gitbooks.io/databricks-spark-knowledge-base/content/best_practices/prefer_reducebykey_over_groupbykey.html



 Knowledge Check

groupByKey is less efficient than reduceByKey on large datasets because:

- A. groupByKey will group all values with the same key on one machine
- B. With groupByKey, if a single key has more key-value pairs than can fit in memory, an out of memory exception occurs
- C. reduceByKey combines locally first and then reduces after the shuffle
- D. All of the above

© 2015 MapR Technologies 

31

Notes on Slide 31:

Answer: 4



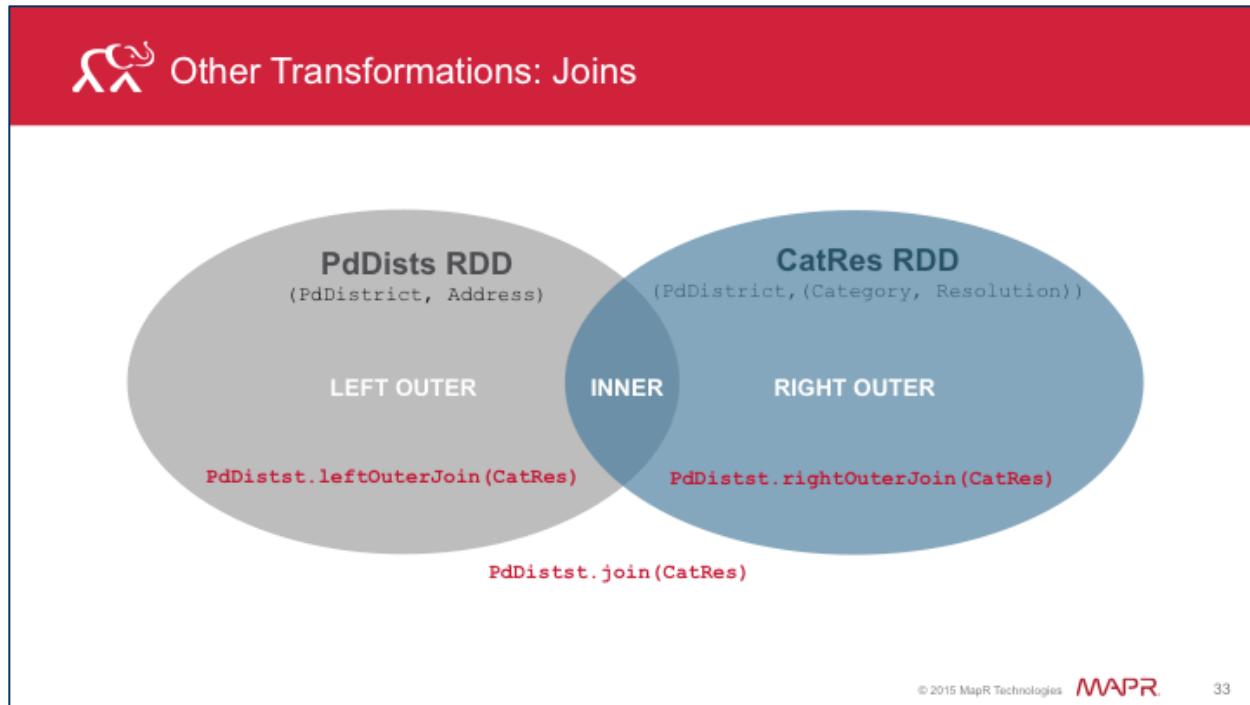


Lab 4.2.1: Create & Explore Pair RDD

Notes on Slide 32:

In this activity you will create Pair RDD and apply actions and transformations on Pair RDD.





Notes on Slide 33:

Some of the most useful operations that we can use with key-value pairs is using it with other similar data. One of the most common operations on a pair RDD is to join two pair RDD. You can do right and left outer joins, cross joins, and inner joins.

You can join two datasets that have been loaded into separate RDDs by joining the resulting RDDs.

- Join is the same as an inner join, where only keys that are present in both RDDs are given in the output.
- leftOuterJoin results in a pair RDD that has entries for each key from the source RDD, joining values from just these keys included in the other RDD.
- rightOuterJoin is similar to the leftOuterJoin but results in a pair RDD that has entries for each key in the other pair RDD, joining values from these keys included in the source RDD.



 Example: Joins

```
PdDists → Array((PdDistrict,Address))
CatRes → Array((PdDistrict,(Category,Resolution))
```

```
PdDists.join(CatRes)
Array((INGLESIDE,(DELTA_ST/RAYMOND_AV,(ASSAULT,ARREST/
BOOKED)))...)
```

```
PdDists.leftOuterJoin(CatRes)
Array((INGLESIDE,(DELTA_ST/RAYMOND_AV,Some ((ASSAULT,ARREST/
BOOKED))),...))
```

```
PdDists.rightOuterJoin(CatRes)
Array((INGLESIDE,(Some(DELTA_ST/RAYMOND_AV),(ASSAULT,ARREST/
BOOKED))),...))
```

© 2015 MapR Technologies 

34

Notes on Slide 34:

In this example, PdDists is a pair RDD consisting of the pairs where the key is PdDistrict and the value is the address.

CatRes is another pair RDD with the same key – PdDistrict and the value is a tuple consisting of the Category and the Resolution.

Since the key – PdDistrict- is present in both RDDs the output of the join contains all the records as shown here. The value consists of the tuple from the source RDD and an option for values from the other pair.

The rightOuterJoin returns pairs with the keys being the key from the other RDD (CatRes) which is the same.

As you can see here, the results in all three cases are the same because both RDD have the same set of keys.



 Example: Joins

```
PdDists → Array((PdDistrict,Address))
IncCatRes → Array((IncidntNum, (Category,Descript,Resolution)))
```

```
PdDists.join(IncCatRes)
empty collection
```

```
PdDists.leftOuterJoin(IncCatRes)
Array((TENDERLOIN, (LEAVENWORTH_ST/TURK_ST,None)),...)
```

```
PdDists.rightOuterJoin(IncCatRes)
Array((130959836, (None, (VANDALISM,MALICIOUS_MISCHIEF/
BREAKING_WINDOWS,NONE))),...)
```

© 2015 MapR Technologies  35

Notes on Slide 35:

In this example, the RDD PdDists is the same as before. We have another pair RDD with IncidntNum as the key. The value is the tuple consisting of Category, Descript and Resolution. The join returns an empty collection since the RDDs do not have any keys in common. The leftOuterJoin returns pairs with the key being the key from the source RDD and the value consisting of the tuple from the source RDD and an option for values from the other pair. The rightOuterJoin returns pairs with the keys being the key from the other RDD (IncCatRes).



 Discussion: Joining Pair RDD

Can you think of examples in your organization data where you would use “joins”?





Actions on Pair RDDs

All traditional actions on RDD available to Pair RDD

- **countByKey ()**
 - Count the number of elements for each key
- **collectAsMap ()**
 - Collect the result as a map to provide easy lookup
- **lookup (key)**
 - Return all values associated with the provided key

© 2015 MapR Technologies

37

Notes on Slide 37:

All traditional actions on RDDs are available to use on Pair RDD. In addition, the following actions are available just for pair RDD:

countByKey() will Count the number of elements for each key

collectAsMap() Collects the result as a map to provide for easy lookup

lookup(key) Return all values associated with the provided key.

There are more actions that can be performed on Pair RDDs. Refer to the API documentation.





Actions on Pair RDDs: Example

Caution:
Use this action
only if dataset is
small enough to
fit in memory



countByKey

```
val num_inc_dist=sfpd  
.map(incident=>(incident(PdDistrict),1))  
.countByKey()
```

```
res10:scala.collection.Map[String,Long] = Map(SOUTHERN -> 73308,  
INGLESIDE -> 33159, TENDERLOIN -> 30174, MISSION -> 50164,  
TARAVAL -> 27470, RICHMOND -> 21221, NORTHERN -> 46877, PARK ->  
23377, CENTRAL -> 41914, BAYVIEW -> 36111)
```

© 2015 MapR Technologies 38

Notes on Slide 38:

Example - if we just want to return number of incidents by district:, we can use countByKey on the pair RDD consisting of pair tuples (district, 1).

Use this operation only if the dataset size is small enough to fit in memory.





Lab 4.2.2: Join Pair RDD



Notes on Slide 39:

In this activity you will join Pair RDD.





Learning Goals

1. Create pair RDD
2. Apply transformations & actions to pair RDD
3. Control partitioning across nodes

© 2015 MapR Technologies **MAPR** 40

Notes on Slide 40:

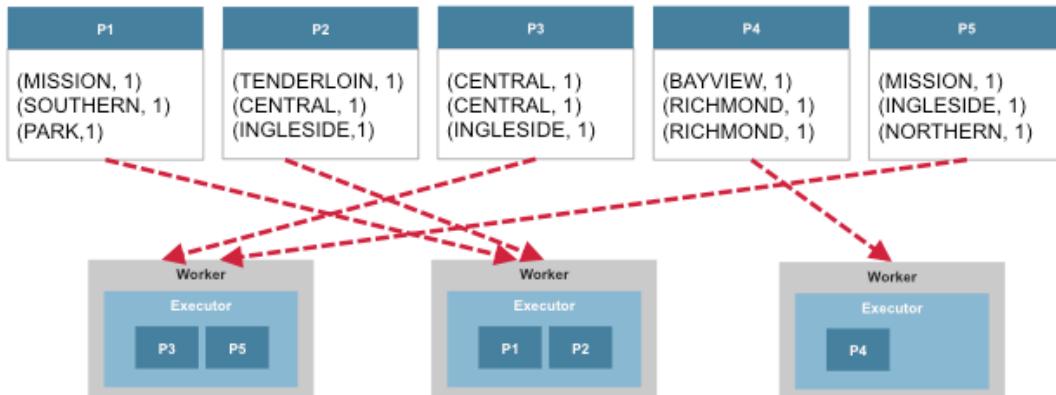
In this section, we will look at the partitioning of RDDs and how to control the partitioning of RDDs across nodes.





Why Partition

More partitions → More parallelism



© 2015 MapR Technologies 

41

Notes on Slide 41:

In a distributed environment, how your data is laid out can affect performance. A data layout that minimizes network traffic can significantly improve performance.

The data in a Spark RDD is split into several partitions. We can control the partitioning of RDD in our Spark program to improve performance.

Note that partitioning is not necessarily helpful in all applications. If you have a dataset that is reused multiple times, it is useful to partition. However, if the RDD is scanned only once, then there is no need to partition the dataset ahead of time.



 RDD Partitions**The data within an RDD is split into several partitions.**

- System groups elements based on a function of each key
 - Set of keys appear together on the same node
- Each machine in cluster contains one or more partitions
 - Number of partitions is at least as large as number of cores in the cluster

© 2015 MapR Technologies 42Notes on Slide 42:

The data within an RDD is split into several partitions.

Spark's partitioning is available on all RDDs of key-value pairs.

The system groups elements based on a function of each key. Functions include hash or range partitions. Set of keys will appear together on a node. Spark will group all keys from the same set on the same node.

Each machine in the cluster contains one or more partitions. The number of partitions is at least as large as the number of cores in the cluster





Types of Partitioning

Two kinds of partitioning:

- Hash partitioning
- Range partitioning

© 2015 MapR Technologies **MAPR** 43

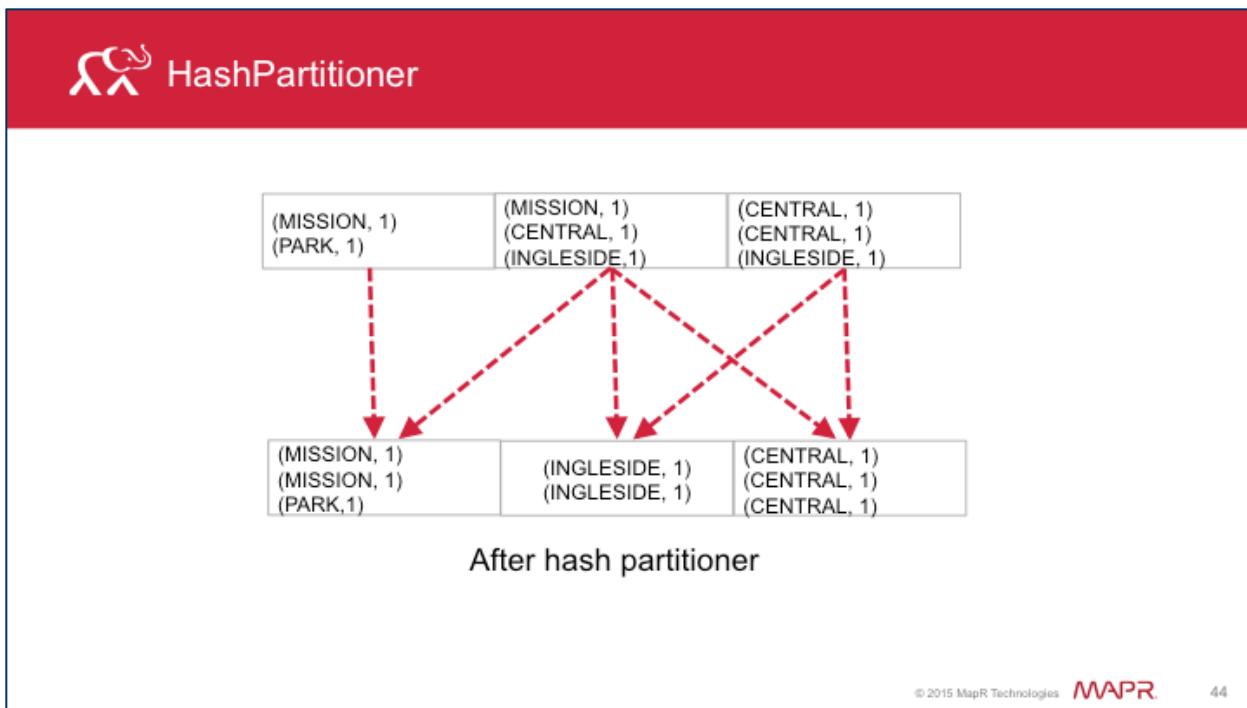
Notes on Slide 43:

There are two kinds of partitioning available in Spark:

1. Hash partitioning and
2. Range partitioning

Customizing partitioning is only possible on Pair RDDs.





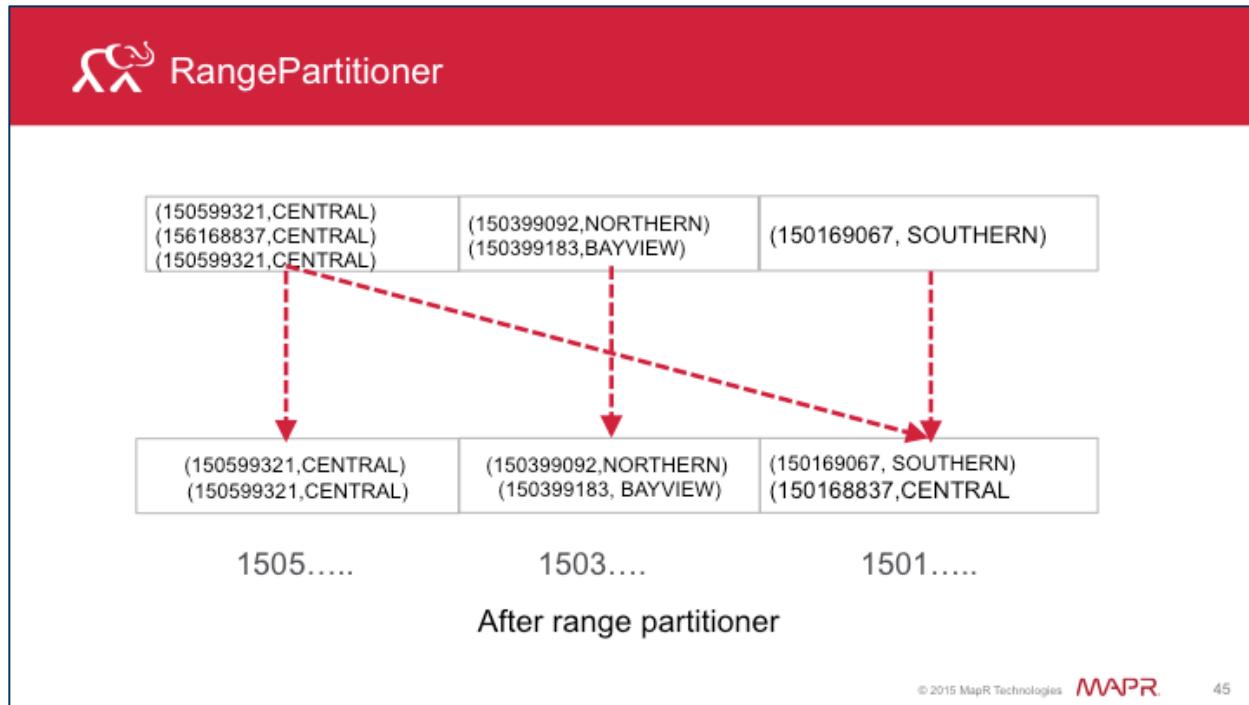
© 2015 MapR Technologies

44

Notes on Slide 44:

You can apply the hash partitioner to an RDD using the `partitionBy` transformation at the start of a program. The hash partitioner will shuffle all the data with the same key to the same worker. In this example, the data for the same keys have been shuffled to the same workers.



Notes on Slide 45:

The range partitioner will shuffle the data for keys in a range to the same workers as shown in this example.

If you have a pair RDD that contains keys with an ordering defined, use a range partitioner. The range partitioner will partition the keys based on the order of the keys and the range specified. It results in tuples with keys in the same range on the same machine.

 Specifying Partitions**There are two ways to specify partitioning**

- `partitionBy` with partitioner explicitly specified
- Specify partition in transformation

© 2015 MapR Technologies  46

Notes on Slide 46:

There are two ways to create RDDs with specific partitioning:

1. Call `partitionBy` on an RDD, providing an explicit Partitioner.
2. Specify partitions in transformations –this will return RDDs partitioned





Using partitionBy: RangePartitioner

partitionBy is transformation → returns RDD with specific partitioning

To create a RangePartitioner:

1. Specify the desired number of partitions.
2. Provide a Pair RDD with ordered keys.

```
val pair1 = sfpd.map(x=>(x(PdDistrict),  
  (x(Resolution),x(Category))))  
val rpart1 = new RangePartitioner(4, pair1)  
val partrdd1 = pair1.partitionBy(rpart1).persist()
```

© 2015 MapR Technologies

47

Notes on Slide 47:

partitionBy is a transformation and creates an RDD with a specified partitioner.

To create a RangePartitioner:

1. Specify the desired number of partitions.
2. Provide a Pair RDD with ordered keys.





Using partitionBy: HashPartitioner

To create a HashPartitioner:

Specify the desired number of partitions.

Note:

Result of partitionBy should be persisted to prevent the partitioning from being applied each time the partitioned RDD is used.

```
val hpart1 = new HashPartitioner(100)  
val pairlrrdd2 = pairl.partitionBy(hpart1).persist()
```

© 2015 MapR Technologies 

48

Notes on Slide 48:

HashPartitioner takes a single argument which defines the number of partitions

Values are assigned to partitions using hashCode of keys. If distribution of keys is not uniform you can end up in situations when part of your cluster is idle

Keys have to be hashable.

NOTE: When you use partition, it will create a shuffle. The result of partitionBy should be persisted to prevent reshuffling each time the partitioned RDD is used.





Specify Partitions in Transformations

- **Some operations accept additional argument**
 - numPartitions or type or partitioner
 - Examples: reduceByKey; aggregateByKey
- **Some operations automatically result in RDD with known partitioner**
 - sortByKey → RangePartitioner
 - groupByKey → HashPartitioner

© 2015 MapR Technologies 

49

Notes on Slide 49:

A lot of the operations on Pair RDDs accept an additional argument, such as the number of partitions or the type or partitioner.

Some operations on RDDs automatically result in an RDD with a known partitioner.

For example, by default, when using sortByKey, a RangePartitioner is used, and the default partitioner used by groupByKey, is HashPartitioner.





Change Partitions

Note:
Repartitioning your
data can be fairly
expensive

To change partitioning outside of aggregations and grouping operations:

- `repartition()`
Shuffles data across network to create new set of partitions
- `coalesce()`
Decreases the number of partitions

© 2015 MapR Technologies 

50

Notes on Slide 50:

To change partitioning outside of aggregations and grouping operations, you can use the `repartition()` function or `coalesce()`.

`repartition()`: shuffles data across the network to create a new set of partitions

`coalesce()`: decreases the number of partitions

NOTE: Repartitioning your data can be fairly expensive since repartitioning shuffles data across the network into new partitions.



Change Partitions: `coalesce()`

Initial RDD (5 partitions):

(MISSION, 1)	(TENDERLOIN, 1)	(CENTRAL, 1)	(BAYVIEW, 1)	(MISSION, 1)
(SOUTHERN, 1)	(CENTRAL, 1)	(CENTRAL, 1)	(RICHMOND, 1)	(INGLESIDE, 1)
(PARK,1)	(INGLESIDE,1)	(INGLESIDE,1)	(RICHMOND, 1)	(NORTHERN, 1)

Code:

```
val dists=sfpd.map(x=>(x(PdDistrict), 1)).reduceByKey(_+_ ,5)
```

Intermediate RDD (3 partitions):

(MISSION, 2)	(TENDERLOIN, 1)		(BAYVIEW, 1)	(NORTHERN, 1)
(SOUTHERN, 1)	(CENTRAL, 3)		(RICHMOND, 2)	
(PARK,1)	(INGLESIDE,3)			

Code:

```
dists.partitions.size  
dists.coalesce(3)
```

Final RDD (3 partitions):

(TENDERLOIN, 1)	(MISSION, 2)	(BAYVIEW,)
(CENTRAL, 3)	(SOUTHERN, 1)	(RICHMOND, 2)
(INGLESIDE,3)	(PARK,1)	(NORTHERN, 1)

© 2015 MapR Technologies  51

Notes on Slide 51:

You need to make sure that when using coalesce, you specify fewer number of partitions. Use `rdd.partition.size()` to determine the current number of partitions.

In this example, we apply the `reduceByKey` to the pair RDD specifying the number of partitions as a parameter.

To find out the number of partitions, we use `rdd.partitions.size()`. Then we apply `coalesce` specifying the decreased number of partitions, in this case 3.





Determine the Partitioner

Use `partitioner()` method to determine the RDD partitioning

```
val pairl = sfpd.map(x=>(x(PdDistrict), (x(Resolution), x(Category))))  
val rpart1 = new RangePartitioner(4, pairl)  
val partrdd1 = pairl.partitionBy(rpart1).persist()  
partrdd1.partition
```

```
res3: Option[org.apache.spark.Partitioner] =  
Some(org.apache.spark.RangePartitioner@8f60c3c6)
```

© 2015 MapR Technologies 

52

Notes on Slide 52:

In Scala and Java, you can use the method `partitioner()` on an RDD to determine how the RDD is partitioned.

Looking at the example used before, when you run the command, `partrdd1.partition`, it returns the type of partitioning – in this case `RangePartitioner`.





Operations that Benefit from Partitioning

- cogroup()
- groupWith()
- join()
- leftOuterJoin()
- rightOuterJoin()
- groupByKey()
- reduceByKey()
- combineByKey()
- lookup()

© 2015 MapR Technologies 

53

Notes on Slide 53:

Since many operations on pair RDD shuffle data by key across the network, partitioning can benefit many operations. The operations that benefit from partitioning are listed here.

For operations such as reduceByKey, when you pre-partition, the values are computed locally for each key on a single machine. Only the final locally reduced value will be sent from each worker node back to the master.

For operations that act on two pair RDDs, when you pre-partition, at least one of the pair RDD (one with the known partitioner) will not be shuffled. If both the RDDs have the same partitioner, then there is no shuffling across the network.



 Operations that Affect Partitioning

- cogroup()
- groupByKey()
- groupWith()
- reduceByKey()
- join()
- combineByKey()
- leftOuterJoin()
- partitionBy()
- rightOuterJoin()
- sort()

© 2015 MapR Technologies 

54

Notes on Slide 54:

These are operations that result in a partitioner being set on the output RDD.

All the other operations will produce a result with no partitioner.

Note that operations such as mapValues(), flatMapValues() and filter() will result in a partitioner being set on the output RDD only if the parent RDD has a partitioner.





Best Practices

- Too few partitions → idle resources
- Too many partitions → too much overhead
- Use numPartitions parameter in operations to specify partitions OR
- Use `repartition()` to increase or decrease # of partitions
- To decrease number of partitions → `coalesce()`
 - More efficient than `repartition()`
 - No shuffle

© 2015 MapR Technologies 55

Notes on Slide 55:

If you have too few number of partitions, i.e. too little parallelism, Spark will leave resources idle. If you have too many partitions or too much parallelism, you could affect the performance as the overheads for each partition can add up to provide significant total overhead.

Use `repartition()` to randomly shuffle existing RDDs to have less or more partitions.

Use `coalesce(N)` over `repartition()` to decrease number of partitions as it avoids shuffle.

Note: If N is greater than your current partitions, you need to pass “shuffle=true” to `coalesce`. If N is less than your current partitions (i.e. you are shrinking partitions) do not set `shuffle=true`, otherwise it will cause additional unnecessary shuffle overhead.



 Knowledge Check**How can you create an RDD with specific partitioning?**

1. `partitionBy()`
2. `rdd.partitioner = Hash`
3. Specify partition in transformation

Notes on Slide 56:

Answer: 1, 3





Lab 4.3: Explore Partitioning

Notes on Slide 57:

In this activity, you will see how to determine the number of partitions in an RDD, and the type of partitioner.



 Next Steps



Lesson 5

Work with Spark
DataFrames

© 2015 MapR Technologies 

58

Notes on Slide 58:

Congratulations! You have completed Lesson 4 of this course.



Lesson 5: Work with DataFrames



DEV 361: Build & Monitor Apache Spark Applications

Lesson 5: Work with DataFrames

© 2015 MapR Technologies 1

Notes on Slide 1:

Welcome to Build and Monitor Apache Spark Applications, Lesson 5 - Work with Apache Spark DataFrames.



Learning Goals

- ▶ 1. Create Apache Spark DataFrames
- 2. Explore Data in DataFrames
- 3. Create User Defined Functions
- 4. Repartition DataFrames

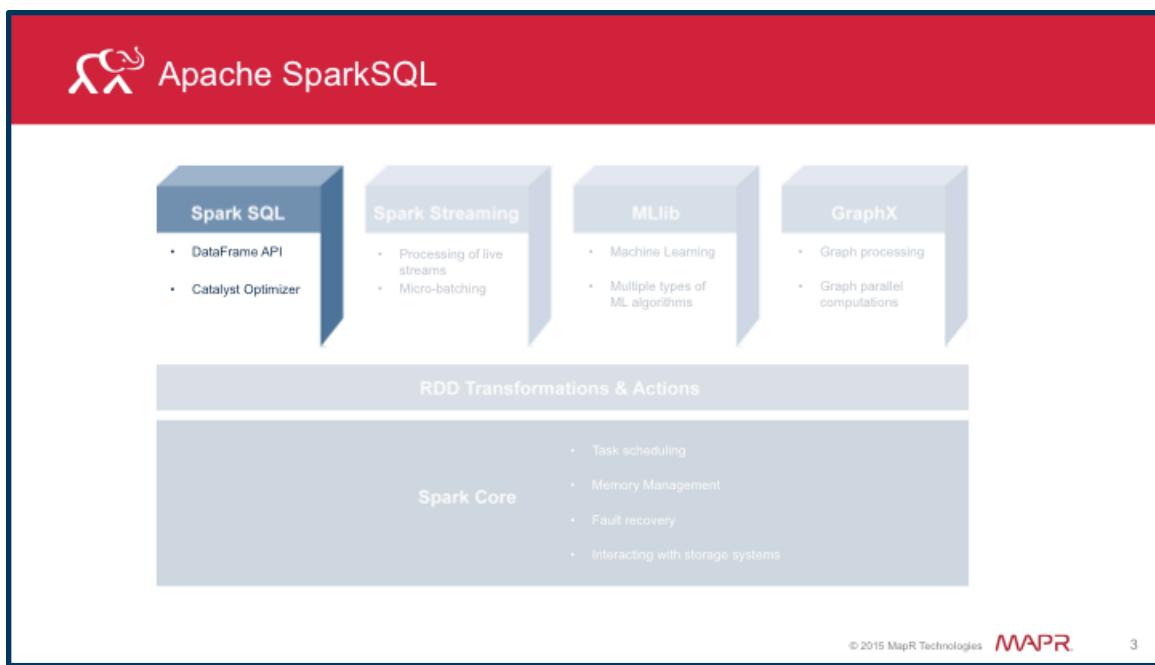
© 2015 MapR Technologies **MAPR**2**Notes on Slide 2:**

At the end of this lesson, you will be able to:

- 1. Create Apache Spark DataFrames
- 2. Work with data in DataFrames
- 3. Create User Defined Functions
- 4. Repartition DataFrames

In this section, we will create Apache Spark DataFrames.





Notes on Slide 3:

SparkSQL is a library that runs on top of the Apache Spark core. It provides access to the SQL interface via the Interactive Shell, JDBC/ODBC or through the DataFrame API.

Spark DataFrames use a relational optimizer called the Catalyst optimizer. The SQL queries that are constructed against DataFrames are optimized and executed as sequences of Spark Jobs.





Apache Spark DataFrames

- Programming abstraction in SparkSQL
- Distributed collection of data organized into named columns
- Supports wide array of data formats & storage systems
- Works in Scala, Python, Java & R

© 2015 MapR Technologies 4

Notes on Slide 4:

A DataFrame is the programming abstraction in SparkSQL. It is a distributed collection of data organized into named columns, and scales from KBs to PBs.

DataFrames supports a wide array of data formats. They can be constructed from structured data files, tables in Hive, external databases or existing RDDs. A DataFrame is equivalent to a Database table but provides a much finer level of optimization.

The DataFrames API is available in Scala, Python, Java and SparkR.





Spark DataFrames vs Spark RDD

Spark DataFrames	Spark RDD
Collection with schema	Opaque collection of objects with no information of underlying data type
Can query data using SQL	Cannot query using SQL

© 2015 MapR Technologies

5

Notes on Slide 5:

DataFrames are collections of objects that have a schema associated with them. The information about the schema makes it possible to do a lot more optimization. SQL can be used to query the data directly.

Spark RDDs on the other hand are collections of objects with no information about the underlying data format. Data can be queried using RDD transformation and actions but not directly using SQL.





What is the starting point for creating DataFrames?

(HINT: Think back to DEV 360: Lesson 2. It is the entry to Spark SQL)

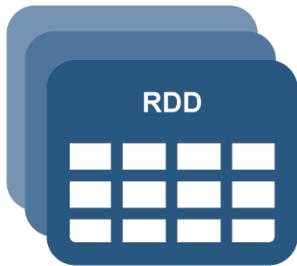
© 2015 MapR Technologies **MAPR**

6

Notes on Slide 6:

SQLContext – you must create a new SQLContext.



 Creating DataFrames**Data Sources**

- Parquet
- JSON
- Hive tables
- Relational Databases

© 2015 MapR Technologies **MAPR**

7

Notes on Slide 7:

There are two ways to create Spark DataFrames. One way is to create it from existing RDDs and the other from data sources.

We will look at creating DataFrames from existing RDD first.





Creating DataFrames from Existing RDD

**1. Infer schema by reflection**

- Convert RDD containing case classes
- Use when schema is known

2. Construct schema programmatically

- Use to construct DataFrames when columns & their types not known until runtime

© 2015 MapR Technologies MAPR8**Notes on Slide 8:**

You can create DataFrames from existing RDD in two ways. Inferring the schema by reflection is used when the RDD has case classes that define the schema. The DataFrame will infer a fixed schema from the case classes.

Use the programmatic interface to construct the schema and apply it to the existing RDD at runtime. This method is used when the schema is dynamic based on certain conditions. You also use this method if you have more than 22 fields as the limit to the number of fields in a Case class is 22.

Let us take a look at inferring the schema by reflection next.





Infer Schema by Reflection: Case Class

- **Defines table schema**
 - Names of arguments to case class read using reflection
 - Names become name of column
- **Can be**
 - Nested
 - Contain complex data (Sequences or Arrays)

© 2015 MapR Technologies 

9

Notes on Slide 9:

The Case class defines the table schema. The names of the arguments passed to the Case class are read using reflection and the names become names of the columns.

Case classes can be nested and can also contain complex data such as sequences or arrays.

NOTE: A Case class in Scala is equivalent to plain old Java Objects (POJOs) or Java beans.





Infer Schema by Reflection

Note:
Create
SQLContext
first

- 1. Import necessary classes
- 2. Create RDD
- 3. Define case class
- 4. Convert RDD into RDD of case objects
- 5. Implicitly convert resulting RDD of case objects into DataFrame
 - Apply DataFrame operations & functions to DataFrame
- 6. Register DataFrame as table
 - Run SQL queries on table

© 2015 MapR Technologies  10

Notes on Slide 10:

Here are the steps to create a DataFrame from an existing RDD inferring schema by reflection.

1. Import the necessary classes – for example, `sqlContext.implicits` and all subclasses.

Note that you need to create a `SQLContext` first which is the starting point for creating the DataFrame.

2. Create the RDD.
3. Define the case class.
4. Convert the RDD into an RDD of case objects using the map transformation to map the case class to every element in the RDD.
5. The resulting RDD is then implicitly converted to DataFrame. We can then apply DataFrame operations and functions to this DataFrame.
6. To run SQL queries on the data in the DataFrame, register it as a table.

Let us take a look at an example next.





Example: Infer Schema by Reflection

1. Import necessary classes

```
import org.apache.spark.sql._  
import sqlContext.implicits._  
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
```

2. Create RDD

```
val sfpdRDD = sc.textFile("/path/to/file/  
sfpd.csv").map(inc=>inc.split(","))
```

© 2015 MapR Technologies

11

Notes on Slide 11:

1. First, Import the necessary classes – for example, `sqlContext.implicits` and all subclasses.
2. The data is in a csv file. Create the base RDD by importing the csv file and splitting on the delimiter “,”. We are using the SFPD data in this example.





Example: Infer Schema by Reflection:

3. Define case class

```
case class Incidents(incidentnum:String,  
category:String, description:String, dayofweek:String,  
date:String, time:String, pddistrict:String,  
resolution:String, address:String, X:Float, Y:Float,  
pdid:String)
```

4. Convert RDD into RDD of case objects

```
val sfpdCase = sfpdRDD.map(inc=>Incidents(inc(0), inc(1),  
inc(2), inc(3), inc(4),inc(5), inc(6), inc(7), inc(8),  
inc(9).toFloat, inc(10).toFloat, inc(11)))
```

© 2015 MapR Technologies

12

Notes on Slide 12:

3. Define the case class.
4. Convert the base RDD into an RDD on case objects – sfpdCase. We apply the map transformation to each element of the RDD mapping the case class to every element in the RDD.





Example: Infer Schema by Reflection

5. Implicitly convert resulting RDD of case objects into DataFrame

```
val sfpdDF = sfpdCase.toDF()
```

6. Register DataFrame as table

```
sfpdDF.registerTempTable("sfpd")
```

© 2015 MapR Technologies 

13

Notes on Slide 13:

5. We then implicitly convert sfpdCase into a DataFrame using the toDF() method.. We can apply DataFrame operations to sfpdDF now.
6. Finally we register the DataFrame as a table so we can query it using SQL. We can now query the table – sfpd – using SQL.





Construct Schema Programmatically

Note:
 Used when:

- Case class cannot be defined ahead of time
- More than 22 fields as case classes (Scala)

1. Create a Row RDD from original RDD
2. Create schema separately using:
 - StructType → table
 - StructField → field
3. Create DataFrame by applying schema to Row RDD

© 2015 MapR Technologies  14

Notes on Slide 14:

We now take a look at the other method to create DataFrames from existing RDDs. When case classes cannot be defined ahead of time. For example, we want parts of a string to represent different fields, or we want to parse a text dataset based on the user. In this case a DataFrame can be created programmatically with three steps.

1. Create an RDD of Rows from the original RDD;
2. Create the schema represented by a StructType matching the structure of Rows in the RDD created in Step 1.
3. Apply the schema to the RDD of Rows via createDataFrame method provided by SQLContext.

Another reason for using this method is when there are more than 22 fields as there is a limit of 22 fields in a case class in Scala.

The schema for the SFPD data is known and it has less than 22 fields. We will take a look at another simple example to demonstrate how to programmatically construct the schema.





Example: Construct Schema Programmatically

Note:
Sample data is
from the file
test.txt



Sample Data

150599321 Thursday **7/9/15** 23:45 **CENTRAL**

156168837 Thursday **7/9/15** 23:45 **CENTRAL**

150599321 Thursday **7/9/15** 23:45 **CENTRAL**

Import Classes

```
import sqlContext.implicits._  
import org.apache.spark.sql._  
import org.apache.spark.sql.types._
```

© 2015 MapR Technologies 15

Notes on Slide 15:

Here is sample data that we will use to create a DataFrame. We have a group of users that is only interested in a DataFrame that has data from the first, third and last “columns” – incident number, date of incident and the district.

First we need to import the necessary classes.





Example: Construct Schema Programmatically

1. Create Row RDD from input RDD

```
val rowRDD = sc.textFile("/user/user01/data/test.txt")
  .map(x=>x.split(" "))
  .map(p=>Row(p(0),p(2),p(4)))
```

© 2015 MapR Technologies 

16

Notes on Slide 16:

In this step, we load the data into an RDD, apply map to split on space and then convert that RDD into a Row RDD with the last map transformation.





Example: Construct Schema Programmatically

2. Create schema separately

```
val testsch = StructType(Array(StructField("IncNum",  
StringType,true), \ StructField("Date",StringType,true), \  
StructField("District",StringType,true)))
```

© 2015 MapR Technologies MAPR

17

Notes on Slide 17:

The StructType object defines the schema. It takes an array of StructField objects.

- StructType takes the arguments: (fields, Array[StructField])
- StructField takes the following arguments: (name, dataType , nullable – Boolean, metadata).

In this example, we are building a schema called testsch. We are defining fields IncNum, Date and District. Each field here is a String (StringType) and can be null (nullable = true).





Example: Construct Schema Programmatically

3. Create DataFrame

```
val testDF = sqlContext.createDataFrame(rowRDD, schema)
```

Register the DataFrame as a table

```
testDF.registerTempTable("test")
```

```
val incs = sql("SELECT * FROM test")
```

© 2015 MapR Technologies MAPR

18

Notes on Slide 18:

Create the DataFrame from the Row RDD by applying the schema to it.

Once the DataFrame is created, it can be registered as a table and the table can be queried using SQL as shown here.





Can you think of situations in your organization for each method?

- Infer schema by reflection
- Programmatic

© 2015 MapR Technologies **MAPR**

19

Notes on Slide 19:

Another way to think about this:

Think of situations where you have data whose schema is known and all your users are going to see the same fields in the same way?

Are there situations where some of users may require the field to be parsed differently?





Knowledge Check

You want to create a DataFrame based on HR data that will provide the dataset parsed differently for managers, employees and HR personnel. Which method will you use to create the DataFrame?

- A. Create DataFrame by programmatically constructing the schema
- B. Create the DataFrame by inferring the schema by reflection

© 2015 MapR Technologies **MAPR**

20

Notes on Slide 20:

Answer: A





Create DataFrames from Data Sources

Operate on variety of data sources through DataFrame interface

- Parquet
- JSON
- Hive tables
- Relational Databases (in Spark 1.4)

© 2015 MapR Technologies 

21

Notes on Slide 21:

Spark SQL can operate on a variety of data sources through the DataFrame interface. Spark SQL includes a schema inference algorithm for JSON and other semistructured data that enables users to query the data right away.

Spark 1.4 provides support to load data directly from Relational Databases into DataFrames.





Generic “load” Method

- **Generic load method**

```
sqlContext.load("/path/to/sfpd.parquet")
```
- **Specify format manually**

```
sqlContext.load("/path/to/sfpdjson", "json")
```

Note:
Default data source assumed to be parquet unless configured otherwise

© 2015 MapR Technologies **MAPR** 22

Notes on Slide 22:

There are generic load and save functions. The default data source - parquet will be used for all operations unless otherwise configured by spark.sql.sources.default.

You can also specify options manually as shown here. Data sources are specified by their fully qualified name (eg. org.apache.spark.sql.parquet), for built in resources use the shortened form (parquet, json, jdbc)





Methods to Load Specific Data Sources

DataFrame from database table:

```
sqlContext.jdbc
```

DataFrame from JSON file:

```
sqlContext.jsonFile
```

DataFrame from RDD containing JSON objects:

```
sqlContext.jsonRDD
```

DataFrame from parquet file:

```
sqlContext.parquetFile
```

© 2015 MapR Technologies **MAPR**

23

Notes on Slide 23:

There are methods available to load specific data sources into DataFrames. These methods have been deprecated in Spark 1.4. Refer to the Apache Spark documentation.





Methods to Load Data Sources: Spark 1.4 Onwards

```
sqlContext.read.load("path/to/file/filename.parquet")  
  
sqlContext.read.format("json").load("/path/to/file/filename.json")
```

© 2015 MapR Technologies **MAPR**

24

Notes on Slide 24:

This is the method to load from data sources to DataFrames in Spark 1.4 onwards





Knowledge Check

You can operate on the following data sources using the DataFrame interface:

- A. Parquet tables
- B. JSON
- C. Streaming data
- D. JDBC

© 2015 MapR Technologies **MAPR**

25

Notes on Slide 25:

Answer: A,B,D





Lab 5.1: Create DataFrame Using Reflection

**Notes on Slide 26:**

In this lab, you will create the DataFrame based on the SFPD dataset using reflection.



 Learning Goals

1. Create Apache Spark DataFrames
- ▶ **2. Explore Data in DataFrames**
3. Create User Defined Functions
4. Repartition DataFrames

© 2015 MapR Technologies 

27

Notes on Slide 27:

In this section we will explore data in DataFrames using DataFrame operations and SQL.



 Exploring the Data

- What are the top five addresses with most incidents?
- What are the top five districts with most incidents?
- What are the top 10 resolutions?
- What are the top 10 categories of incidents?

© 2015 MapR Technologies 

28

Notes on Slide 28:

Here are some questions we can ask of the SFPD data. Next we will discuss how to apply operations to the DataFrame to answer these questions.





DataFrame Operations

- DataFrame Actions
- DataFrame Functions
- Language Integrated Queries

© 2015 MapR Technologies 

29

Notes on Slide 29:

There are different categories of operations that can be performed on DataFrames. In addition to the ones listed here, you can also use some RDD operations on DataFrames. You can also output data from DataFrames to tables and files.





DataFrame Actions

Action	Description
<code>collect()</code>	Returns array containing all rows in DataFrame
<code>count()</code>	Returns number of rows in DataFrame
<code>describe(cols:String*)</code>	Computes statistics for numeric columns (count, mean, stddev, min and max)
<code>first()</code>	Returns the first row
<code>head()</code>	Returns the first row
<code>show()</code>	Displays first 20 rows
<code>take(n:int)</code>	Returns the first n rows

© 2015 MapR Technologies

30

Notes on Slide 30:

This table lists DataFrame actions.



 DataFrame Functions

Function	Description
<code>cache()</code>	Cache this DataFrame
<code>columns</code>	Returns an array of all column names
<code>printSchema()</code>	Prints schema to console in tree format
<code>persist()</code>	Persist this DataFrame
<code>explain()</code>	Prints physical plan (Catalyst optimizer) to console

© 2015 MapR Technologies 

31

Notes on Slide 31:

This table lists DataFrame functions.





Language Integrated Queries

L I Query	Description
<code>agg(expr, exprs)</code>	Aggregates on entire DataFrame
<code>distinct</code>	Returns new DataFrame with unique rows
<code>except(other)</code>	Returns new DataFrame with rows from this DataFrame not in other DataFrame
<code>filter(expr); where(condition)</code>	Filter based on the SQL expression or condition
<code>groupBy(cols: Columns)</code>	Groups DataFrame using specified columns
<code>join (DataFrame, joinExpr)</code>	Joins with another DataFrame using given join expression
<code>sort(sortcol)</code>	Returns new DataFrame sorted by specified column
<code>select(col)</code>	Selects set of columns

© 2015 MapR Technologies

32

Notes on Slide 32:

This table lists Language Integrated queries.

Now let us use these actions, functions and language integrated queries to find the answers to questions posed earlier.





Top Five Addresses with Most Incidents

1. val incByAdd=sfpdDF.groupBy("address")
2. val numAdd=incByAdd.count
3. val numAddDesc=numAdd.sort(\$"count".desc)
4. val top5Add=numAddDesc.show(5)

© 2015 MapR Technologies

33

Notes on Slide 33:

Which are the five addresses with the most number of incidents?

To answer this:

1. Create a DataFrame by grouping the incidents by **address**.
2. Count the number of incidents for each address
3. Sort the result of previous step in descending order
4. Show the first five which is the top five addresses with the most incidents.





Top Five Addresses with Most Incidents

```
val incByAdd = sfpdDF.groupBy("address")
    .count
    .sort($"count".desc)
    .show(5)
```

address	count
800_Block_of_BRYA...	10852
800_Block_of_MARK...	3671
1000_Block_of_POT...	2027
2000_Block_of_MIS...	1585
16TH_ST/MISSION_ST	1512

© 2015 MapR Technologies 

34

Notes on Slide 34:

You can combine the statements into one statement. The result is shown here.





Top Five Addresses with Most Incidents: SQL

```
val top5Addresses = sqlContext.sql("SELECT address,  
count(incidentnum) AS inccount FROM sfpd GROUP BY address ORDER  
BY inccount DESC LIMIT 5").show
```

address	inccount
800_Block_of_BRYA...	10852
800_Block_of_MARK...	3671
1000_Block_of_POT...	2027
2000_Block_of_MIS...	1585
16TH_ST/MISSION_ST	1512

© 2015 MapR Technologies

35

Notes on Slide 35:

We can answer the same question using SQL as shown here. Note that we are selecting from “sfpd” which is the table that the DataFrame is registered as.



 Exploring the Data

- ✓ What are the top five addresses with most incidents?
 - What are the top five districts with most incidents?
 - What are the top 10 resolutions?
 - What are the top 10 categories of incidents?

© 2015 MapR Technologies 

36

Notes on Slide 36:

We have just answered the first question. We can use similar logic to answer the remaining questions which will be done in the Lab.



Output Operations: `save()`

Operation	Description
<code>Save (source, mode, options)</code>	Saves contents of DataFrame based on given data sources, savemode and set of options
<code>insertIntoJDBC (url,name, overwrite)</code>	Saves contents of DataFrame to JDBC at url under table name table
<code>saveAsParquetFile (path)</code>	Saves contents of DataFrame as parquet file
<code>saveAsTable (tablename, source, mode, options)</code>	Creates a table from contents of DataFrame using data source, options & mode

© 2015 MapR Technologies 

37

Notes on Slide 37:

There are times when we want to save the results of our queries. We can use save operations to save data from resulting DataFrames using Spark Data Sources.



Output Operations: `save()`

To save contents of top5Addresses DataFrame:

```
top5Addresses.toJSON.saveAsTextFile("/user/user01/test")
```

```
{"address":"800_Block_of_BRYANT_ST","inccount":10852}  
 {"address":"800_Block_of_MARKET_ST","inccount":3671}  
 {"address":"1000_Block_of_POTRERO_AV","inccount":2027}  
 {"address":"2000_Block_of_MISSION_ST","inccount":1585}  
 {"address":"16TH_ST/MISSION_ST","inccount":1512}
```

© 2015 MapR Technologies MAPR

38

Notes on Slide 38:

The contents of the top5Addresses DataFrame is saved in JSON format in a folder called test. In this statement, if the folder exists, you will get an error.

In Spark 1.4, the methods have been deprecated. See the link here.

<https://spark.apache.org/docs/1.4.1/sql-programming-guide.html#generic-loadsave-functions>





Output Operations – Spark 1.4 Onwards

To save contents of top5Addresses DataFrame:

```
top5Addresses.write.format("json").mode("overw  
rite").save("/user/user01/test")
```

© 2015 MapR Technologies MAPR

39

Notes on Slide 39:

In Spark 4.x onwards, **save** operations have been replaced by the “**write**” method.

The contents of the top5Addresses DataFrame is saved in JSON format in a folder called test that is created. If the folder exists, you will get an error.



 Knowledge Check

Which of the following (in Scala) will give the top 10 resolutions to the console assuming that sfpdDF is the DataFrame registered as a table – sfpd?

- A. `sqlContext.sql("SELECT resolution, count(incidentnum) AS inccount FROM sfpd GROUP BY resolution ORDER BY inccount DESC LIMIT 10")`
- B. `sfpdDF.select("resolution").count.sort($"count".desc).show(10)`
- C. `sfpdDF.groupBy("resolution").count.sort($"count".desc).show(10)`

© 2015 MapR Technologies 

40

Notes on Slide 40:

Answer: A, C

A → using SQL

C → using DataFrame operations





Lab 5.2: Explore Data in DataFrames

Notes on Slide 41:

In this lab, you will explore the dataset using DataFrame operations and SQL queries. You will also save from the DataFrame.



 Learning Goals

1. Create Apache Spark DataFrames
2. Explore data in DataFrames
- ▶ **3. Create User Defined Functions**
4. Repartition DataFrames

© 2015 MapR Technologies 

42

Notes on Slide 42:

In this section, we will create and use User Defined functions.





User Defined Functions (UDF)

- UDFs allow developers to define custom functions
- In Spark, can define UDF inline
- No complicated registration or packaging process
- Two types of UDF:
 - To use with Scala DSL (DataFrame operations)
 - To use with SQL

© 2015 MapR Technologies 

43

Notes on Slide 43:

User defined functions allow developers to define custom functions. Spark provides the ability to create UDFs like other query engines.

In Spark, you can define the UDF inline. There is no complicated registration or packaging process.

There are two types of UDFs – one that can be used with the Scala DSL (with the DataFrame operations) and the other that can be used with SQL.





User Defined Functions (Scala DSL)

- Inline creation
 - Use `udf()`
- Function can be used with DF operations

```
val func1 = udf((arguments)=>{function definition})
```

© 2015 MapR Technologies 

44

Notes on Slide 44:

You can define the function as a udf inline. Here we are creating a user defined function called func1 by using udf





User Defined Functions in SQL Query

```
def funcname  
sqlContext.udf.register("funcname", funcname _)
```

partially applied
function in Scala

Inline registration and creation:

```
sqlContext.udf.register("funcname", func def)
```

© 2015 MapR Technologies MAPR

45

Notes on Slide 45:

There are two ways to register the udf to be used with SQL.

1. You can define the function first and then register using `sqlContext.udf.register`. This accepts two arguments: the function name as a literal, and the second parameter is the function name followed by an underscore as shown. The underscore (must have a space in between function and underscore) turns the function into a partially applied function that can be passed to register. The underscore tells Scala that we don't know the parameter yet but want the function as a value. The required parameter will be supplied later.
2. The second way is to define the function inline in the registration.





Example: Want to Find Incidents by Year

- Date in the format: “dd/mm/yy”
- Need to extract string after last slash
- Can then compute incidents by year

© 2015 MapR Technologies 

46

Notes on Slide 46:

The date in the SFPD data is a string of the form “dd/mm/yy”. In order to group or do any aggregation by year, we have to extract the year from the string.





Example: Want to Find Incidents by Year

Note:
This is
Scala DSL



- **Defining UDF**

```
val getStr = udf((s:String)=>{
    val lastS = s.substring(s.lastIndexOf('/')+1)
    lastS
})
```

- **Use UDF in DataFrame Operations**

```
val yy = sfpdDF.groupBy(getStr(sfpdDF("date")))
    .count
    .show
```

© 2015 MapR Technologies

47

Notes on Slide 47:

In this example, we are defining a function getStr using udf.



 Example: Want to Find Incidents by Year

Note:
This is
Scala DSL 

	scalaUDF(date) count
13	152830
14	150185
15	80760

© 2015 MapR Technologies  48

Notes on Slide 48:

This is the number of incidents by year.





Example: Want to Find Incidents by Year

Note:
This is SQL



- **Define UDF**

```
def getStr(s:String) = {val  
strAfter=s.substring(s.lastIndexOf('/')+1)  
strAfter}
```

- **Register UDF**

```
sqlContext.udf.register("getStr", getStr _)
```

© 2015 MapR Technologies

49

Notes on Slide 49:

In this example, we are defining a function `getStr` and then registering it as a user defined function. We tell Scala, we don't know the parameter yet, we want the function as a value and we'll supply it with the required parameter at a later time. So, we use the function name with a space and underscore.





Example: Want to Find Incidents by Year

Note:
This is SQL 

- Define & register UDF

```
sqlContext.udf.register("getStr", (s:String)=>{  
    val strAfter=s.substring(s.lastIndexOf('/')+1)  
    strafter  
})
```
- Use in SQL statement

```
val numIncByYear = sqlContext.sql("SELECT getStr(date),  
    count(incidentnum) AS countbyyear  
FROM sfpd GROUP BY getStr(date)  
ORDER BY countbyyear DESC  
LIMIT 5")
```

© 2015 MapR Technologies  50

Notes on Slide 50:

This is the same function now defined and registered as a udf to be used in SQL statements.





Example: Want to Find Incidents by Year

Note:
This is SQL



```
numIncByYear.foreach(println)  
[13,152830]  
[14,150185]  
[15,80760]
```

© 2015 MapR Technologies **MAPR**

51

Notes on Slide 51:

As you can see, we get the same result as before for the number of incidents by year.





Knowledge Check

To use a user defined function in a SQL query, the function (in Scala):

- A. Must be defined inline using udf(function definition)
- B. Must be registered using sqlContext.udf.register
- C. Only needs to be defined as a function

© 2015 MapR Technologies **MAPR**

52

Notes on Slide 52:

Answers: 2





Lab 5.3: Create & Use User Defined Functions

**Notes on Slide 53:**

In this activity, you will create and use a user defined function.





Learning Goals

1. Create Apache Spark DataFrames
2. Explore data in DataFrames
3. Create User Defined Functions
- ▶ **4. Repartition DataFrames**

© 2015 MapR Technologies **MAPR**

54

Notes on Slide 54:

In this section, we look at repartitioning DataFrames.



 Partition DataFrames

DataFrame with 4 Partitions

P1			P2			P3			P4		
Incnum (Str)	Category (Str)	PdDistrict (Str)	Incnum (Str)	Category (Str)	PdDistrict (Str)	Incnum (Str)	Category (Str)	PdDistrict (Str)	Incnum (Str)	Category (Str)	PdDistrict (Str)
150598981	ASSAULT	CENTRAL	150599183	ASSAULT	SOUTHERN	150597701	ASSAULT	MISSION	150597400	ROBBERY	TARAVAL
150599161	BURGLARY	PARK	150599246	ASSAULT	CENTRAL	150597701	ROBBERY	INGLESIDE	150596468	FRAUD	SOUTHERN
150599127	SUSPICIOUS	SOUTHERN	150599246	WARRANTS	CENTRAL	150597701	ASSAULT	SOUTHERN	150597234	BURGLARY	SOUTHERN
150603455	VANDALISM	NORTHERN	150599246	WARRANTS	CENTRAL	150597591	ROBBERY	SOUTHERN	150596468	FRAUD	TARAVAL

© 2015 MapR Technologies 

55

Notes on Slide 55:

This is a DataFrame containing the SFPD data. This DataFrame has four partitions.





Partition DataFrames

- Sets number of partitions in DataFrame after shuffle:
`spark.sql.shuffle.partitions`
- Default value set to 200
- Can change parameter using:
`sqlContext.setConf(key, value)`

© 2015 MapR Technologies 

56

Notes on Slide 56:

By default in Spark SQL, there is a parameter called `spark.sql.shuffle.partitions`, which sets the number of partitions in a DataFrame after a shuffle (in case the user hasn't manually specified it). Currently, Spark does not do any automatic determination of partitions, it just uses the number in that parameter. We can change this parameter using: `sqlContext.setConf(key, value)`.





Why Repartition

- Internally SparkSQL partitions data for joins and aggregations
- If applying other RDD operations on result of DataFrame operations, can manually control partitioning

© 2015 MapR Technologies 

57

Notes on Slide 57:

Internally Spark SQL will add exchange operators to make sure that data is partitioned correctly for joins and aggregations. Optimizing partitioning can improve performance.

If we want to apply other RDD operations on the result of the DataFrame operations, we have the option to manually control the partitioning.





repartition(numPartitions)

- To repartition, use
 - `df.repartition(numPartitions)`
- To determine current number of partitions:
 - `df.rdd.partitions.size`

© 2015 MapR Technologies 

58

Notes on Slide 58:

You can use the DataFrame.repartition(numPartitions) method to repartition a DataFrame. To determine the current number of partitions, use the method, df.rdd.partitions.size.





Best Practices

- Specifying the number of partitions:
- Want each partition to be 50 MB – 200 MB
- Small dataset → few partitions
- Large cluster with 100 nodes → at least 100 partitions
- Example:
 - 100 nodes with 10 slots in each executor
 - Want 1000 partitions to use all executor slots

© 2015 MapR Technologies

59

Notes on Slide 59:

Ideally we want each partition to be around 50 MB - 200 MB. If the dataset is really small, then having just a few partitions may be fine.

If we have a large cluster with 100 nodes and 10 slots in each Executor, then we want the DataFrame to have 1,000 partitions to use all of the Executor slots to process it simultaneously. In this case, it's also fine to have a DataFrame with thousands of partitions, and a thousand partitions will be processed at a time.



 Knowledge Check

**To find the number of partitions in the DataFrame,
use (in Scala)**

- A. df.numPartitions()
- B. df.rdd.partitions.size()
- C. rdd.partitions.size()

© 2015 MapR Technologies **MAPR**

60

Notes on Slide 60:

Answers: C





Lesson 6

Monitor Apache Spark Applications

© 2015 MapR Technologies **MAPR**

61

Notes on Slide 61:

Congratulations! You have completed Lesson 5 of this course—DEV 361 – Build and Monitor Apache Spark Applications



Lesson 6: Monitor Apache Spark Applications



DEV 361: Build & Monitor Apache Spark Applications

Lesson 6: Monitor Apache Spark Applications

© 2015 MapR Technologies 1

Notes on Slide 1:

Welcome to Build and Monitor Apache Spark Applications, Lesson 6- Monitor Apache Spark Applications.



Learning Goals

- ▶ 1. Describe Components of Spark Execution Model
- 2. Use Spark Web UI to Monitor Spark Applications
- 3. Debug & Tune Spark Applications

© 2015 MapR Technologies **MAPR**2**Notes on Slide 2:**

At the end of this lesson, you will be able to:

- 1. Describe components of the Spark execution model
- 2. Use Spark Web UI to monitor Spark applications
- 3. Debug and tune Spark Applications

In this section, we will look at the components of the Spark execution model.





Spark Execution Model: Logical Plan

```
1. val inputRDD = sc.textFile(".../sfpd.csv")  
2. val sfpdRDD = input.map(x=>x.split(", " ))  
3. val catRDD = sfpdRDD.map(x=>(x(Category),  
1)).reduceByKey((a,b)=>a+b)
```

© 2015 MapR Technologies 

3

Notes on Slide 3:

We are going to see how a user program translates into the units of physical execution. Let us first take a look at the logical plan.

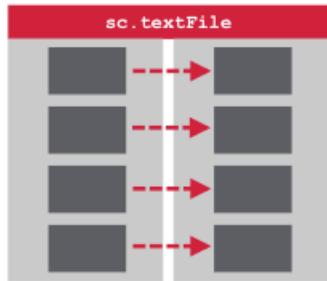
Consider the example from an earlier lesson where we load SFPD data from a csv file. We will use this as an example to walk through the components of the Spark execution model.





Spark Execution Model: Logical Plan

```
1. val inputRDD = sc.textFile("/.../sfpd.csv")
```



© 2015 MapR Technologies

4

Notes on Slide 4:

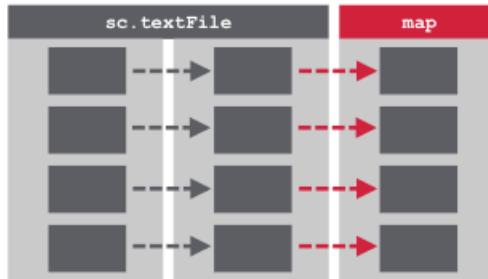
- The first line creates an RDD called inputRDD from the sfpd.csv file.





Spark Execution Model: Logical Plan

```
2. val sfpdRDD = input.map(x=>x.split(","))
```



© 2015 MapR Technologies

5

Notes on Slide 5:

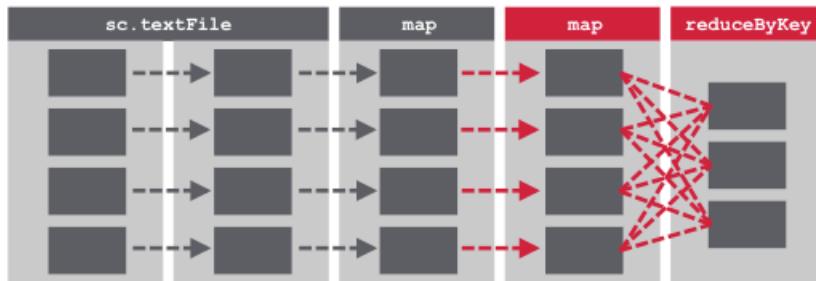
- The second line creates an RDD – sfpdRDD which splits the data in the input RDD based on the comma separator.





Spark Execution Model: Logical Plan

```
3. val catRDD = sfpdRDD.map(x=>(x(Category),
  1)).reduceByKey((a,b)=>a+b)
```



© 2015 MapR Technologies

6

Notes on Slide 6:

- The third statement creates the catRDD by applying the map and reduceByKey transformations.

No actions have been performed yet. Once Spark executes these lines, it defines a Directed Acyclic Graph (DAG) of these RDD objects. Each RDD maintains a pointer to its parent(s) along with the metadata about the type of relationship. RDDs use these pointer to trace its ancestors.





Spark Execution Model: Display Lineage

```
catRDD.toDebugString()
```

```
(2) ShuffledRDD[20] at reduceByKey at <console>:27 []
+- (2) MapPartitionsRDD[19] at map at <console>:27 []
  |  MapPartitionsRDD[16] at map at <console>:23 []
  |  /user/user01/data/sfpd.csv MapPartitionsRDD[15] at textFile at <console>:21 []
  |  /user/user01/data/sfpd.csv HadoopRDD[14] at textFile at <console>:21 []
```

© 2015 MapR Technologies 7

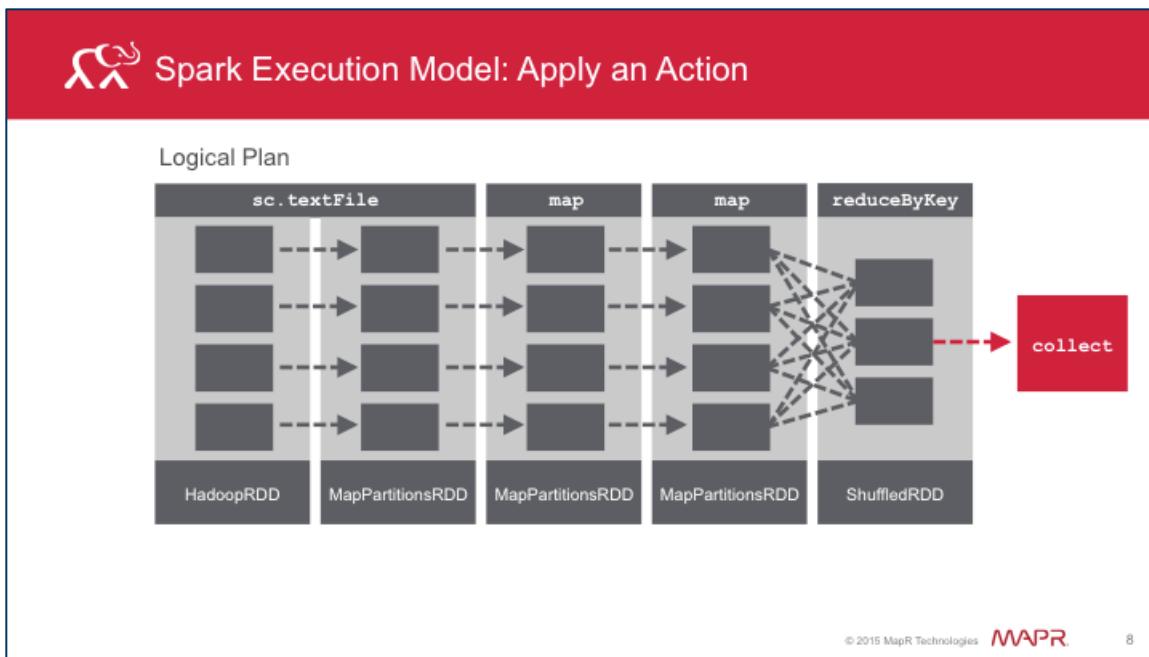
Notes on Slide 7:

To display the lineage for an RDD, use rdd.toDebugString. In this example, to display the lineage for cat RDD, use catRDD.toDebugString. The lineage displays all the ancestors of cat RDD.

sc.textFile first creates a HadoopRDD and then the MapPartitions RDD. Each time we apply the map transformation, it results in a MapPartitions RDD. When we apply the reduceByKey transformation, it results in a ShuffledRDD.

No computation has taken place yet as we have not performed any actions.





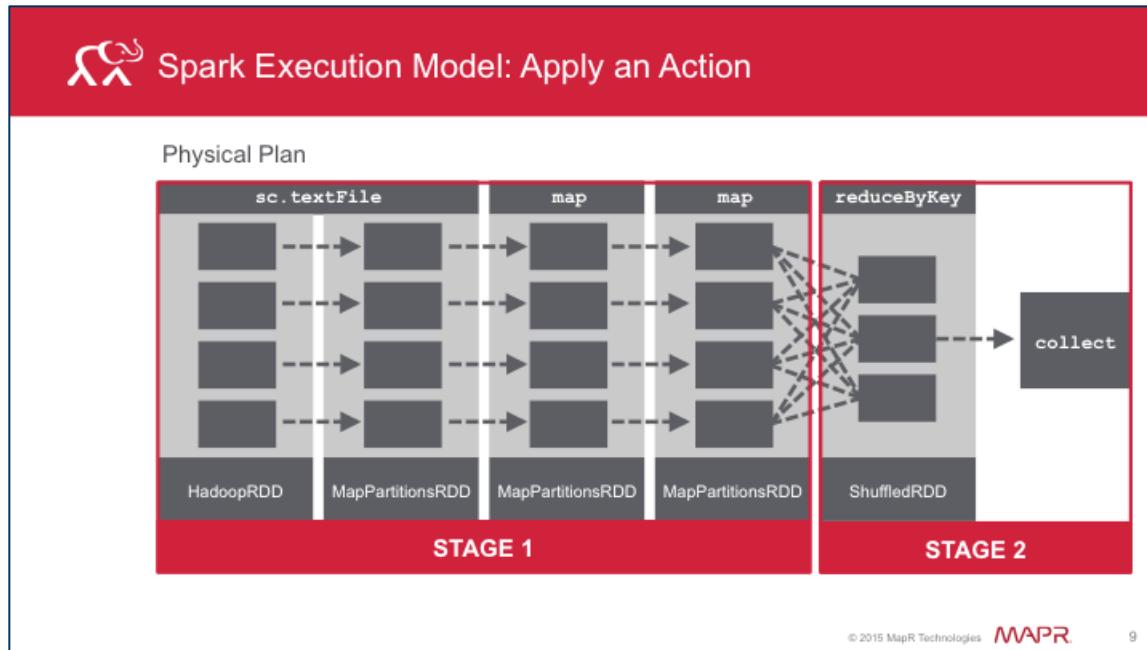
© 2015 MapR Technologies **MAPR**

8

Notes on Slide 8:

Now we add a `collect` action on the `catRDD`. The `collect` action triggers a computation. The Spark scheduler creates a physical plan to compute the RDDs needed for the computation. When the `collect` action is called, every partition of the RDD is materialized and transferred to the driver program. The Spark scheduler then works backwards from the `catRDD` to create the physical plan necessary to compute all ancestor RDDs.





Notes on Slide 9:

The scheduler usually outputs a computation stage for each RDD in the graph. However, when an RDD can be computed from its parent without movement of data, multiple RDDs are collapsed into a single stage. The collapsing of RDDs into one stage is called **pipelining**. In the example, the map operations are not moving any data and hence the RDDs have been pipelined into stage 1. Since the reduceByKey does a shuffle, it is in the next stage. There are other reasons as to why the scheduler may truncate the lineage which are discussed next.





Situations When Lineage Truncated

1. Pipelining
2. RDD persisted in cluster memory or on disk
3. RDD materialized due to earlier shuffle

© 2015 MapR Technologies 

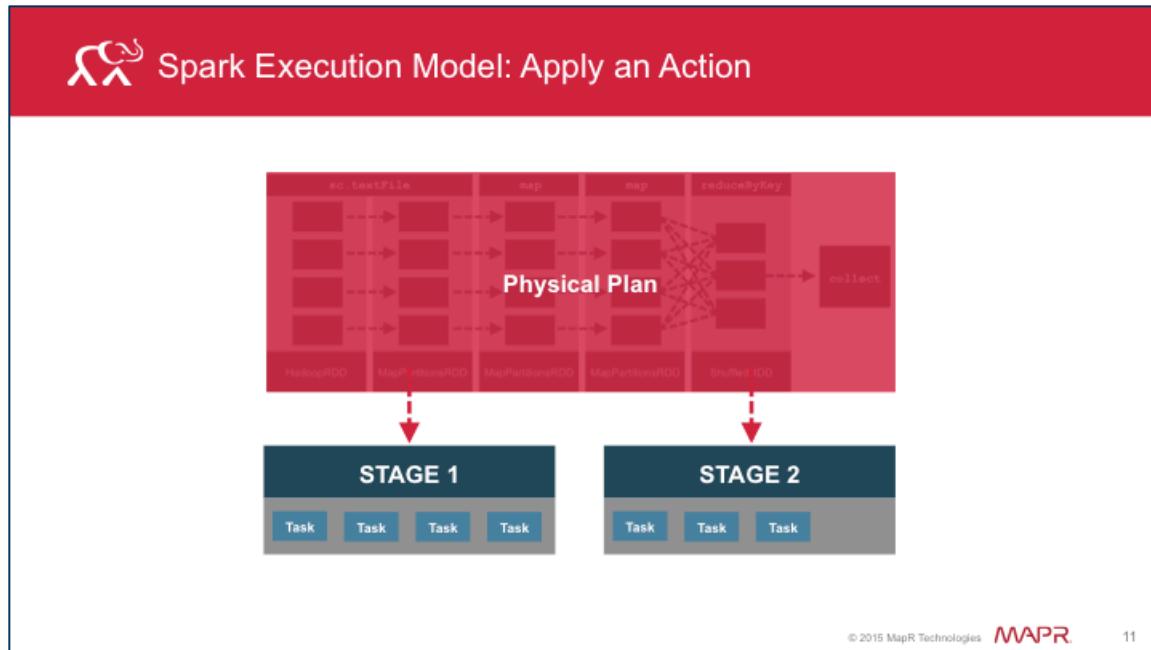
10

Notes on Slide 10:

The situations in which the scheduler can truncate the lineage of an RDD graph are listed here.

1. Pipelining: When there is no movement of data from the parent RDD, the scheduler will pipeline the RDD graph collapsing multiple RDDs into a single stage.
2. When an RDD is persisted to cluster memory or disk, the Spark scheduler will truncate the lineage of the RDD graph. It will begin computations based on the persisted RDD.
3. Another situation when the Spark scheduler will truncate the lineage is if an RDD is already materialized due to an earlier shuffle, since shuffle outputs in Spark are written to disk. This optimization is built into Spark.

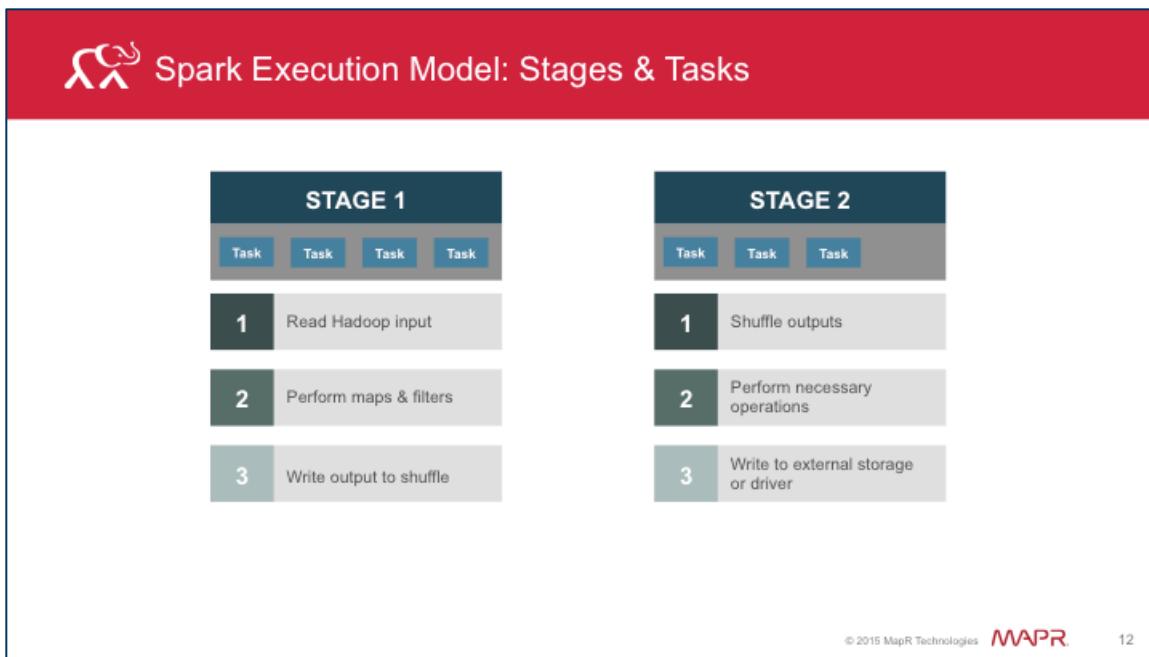




Notes on Slide 11:

When an action is encountered, the DAG is translated into a Physical plan to compute the RDDs needed for performing the action. The Spark scheduler submits a job to compute all the necessary RDDs. Each job is made up of one or more stages and each stage is composed of tasks. Stages are processed in order and individual tasks are scheduled and executed on the cluster.





Notes on Slide 12:

The stage has tasks for each partition in that RDD. A stage launches tasks that do the same thing but on specific partitions of data. Each task performs the same steps:

1. Fetch input (from data storage or existing RDD or shuffle outputs)
2. Perform necessary operations to compute required RDDs
3. Write output to shuffle, external storage or back to driver (for example, count, collect)





Spark Execution Model: Components of Spark Execution

Component	Description
Tasks	Unit of work within a stage corresponds to one RDD partition
Stages	Group of tasks which perform the same computation in parallel
Shuffle	Transfer of data between stages
Jobs	Work required to compute RDD; has one or more stages
Pipelining	Collapsing of RDDs into a single stage when RDD transformations can be computed without data movement
Directed Acyclic Graph (DAG)	Logical graph of RDD operations
Resilient Distributed Dataset (RDD)	Parallel dataset with partitions

© 2015 MapR Technologies

13

Notes on Slide 13:

- A task is a unit of work within a stage corresponding to one RDD partition.
- A stage is a group of tasks which perform the same computation in parallel.
- A shuffle is the transferring of data between stages.
- A set of stages for a particular action is a job.
- When an RDD is computed from the parent without movement of data, the scheduler will pipeline or collapse RDDs into single stage
- DAG or Directed Acyclic Graph is the logical graph of RDD operations.
- RDD or Resilient Distributed Dataset is a parallel dataset with partitions.



Phases During Spark Execution

Note: In an application, a sequence may occur many times as new RDD created

PHASE 1	User code defines the DAG or RDDs
PHASE 2	Actions responsible for translating DAG into physical execution plan
PHASE 3	Tasks scheduled and executed on cluster

© 2015 MapR Technologies **MAPR** 14

Notes on Slide 14:

1. User code defines the DAG or RDDs

The user code defines RDDs and operations on RDDs. When you apply transformations on RDDs, new RDDs are created that point to their parents resulting in a DAG.

2. Actions are responsible for translating the DAG into a physical execution plan

The RDD must be computed when an action is called on it. This results in computing the ancestor(s). The scheduler will submit a job per action to compute all the required RDDs. This job has one or more stages, which in turn is made up of tasks that operate in parallel on partitions. A stage corresponds to one RDD unless the lineage is truncated due to pipelining.

3. Tasks are scheduled and executed on the cluster

Stages are executed in order. The action is considered complete when the final stage in a job completes.

This sequence can occur many times when new RDDs are created.





Knowledge Check

The number of stages in a job is usually equal to the number of RDDs in the DAG. However, the scheduler can truncate the lineage when:

- A. There is no movement of data from the parent RDD
- B. There is a shuffle.
- C. The RDD is cached or persisted
- D. The RDD was materialized due to an earlier shuffle

© 2015 MapR Technologies **MAPR**

15

Notes on Slide 15:

Answer:A,C, D





Learning Goals

1. Describe Components of Spark Execution Model
- ▶ **2. Use Spark Web UI to Monitor Spark Applications**
3. Debug & Tune Spark Applications

© 2015 MapR Technologies **MAPR**

16

Notes on Slide 16:

In this section, we will use the Spark Web UI to monitor Apache Spark Applications.



What is Spark Web UI

Spark Jobs [?]

Total Duration: 1.8 min
Scheduling Mode: FIFO
Active Jobs: 2
Completed Jobs: 8

Active Jobs (2)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
9	print at <console>-39	2015/10/21 18:08:24	51 ms	0/2	0/4
0	start at <console>-33	2015/10/21 18:08:16	7 s	0/1	0/1

Completed Jobs (8)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
8	print at <console>-39	2015/10/21 18:08:22	26 ms	1/1 (1 skipped)	1/1 (3 skipped)
7	print at <console>-39	2015/10/21 18:08:22	0.1 s	2/2	4/4
6	print at <console>-39	2015/10/21 18:08:21	23 ms	1/1 (1 skipped)	1/1 (1 skipped)
5	print at <console>-39	2015/10/21 18:08:21	44 ms	2/2	3/2
4	print at <console>-39	2015/10/21 18:08:21	19 ms	1/1 (1 skipped)	1/1 (1 skipped)
3	print at <console>-39	2015/10/21 18:08:21	47 ms	2/2	2/2
2	print at <console>-39	2015/10/21 18:08:21	26 ms	1/1 (1 skipped)	1/1 (1 skipped)

Note:
In YARN cluster mode, access Spark web UI through YARN ResourceManager

http://<driver-node>:4040

© 2015 MapR Technologies **MAPR** 17

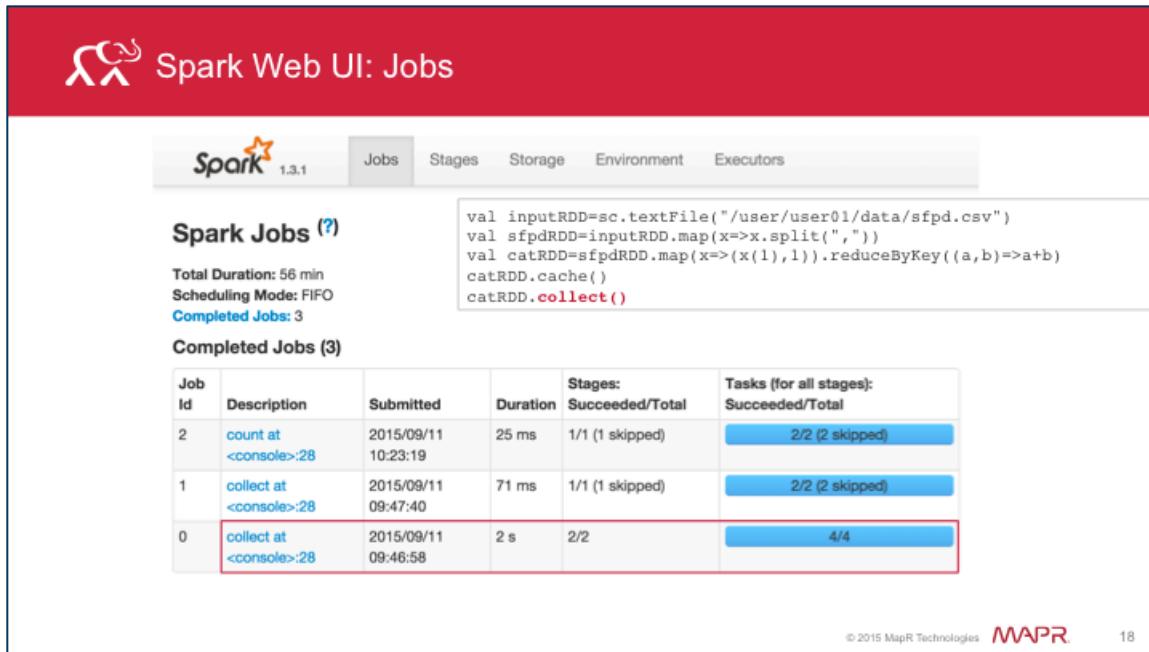
Notes on Slide 17:

The Spark Web UI provides detailed information about the progress and performance for Spark jobs. By default, this information is only available for the duration of the application. You can view the web UI after the event by setting `spark.eventLog.enabled` to true before starting the application.

The Spark Web UI is available on the machine where the driver is running on port 4040 by default. If multiple SparkContexts are running on the same host, they will bind to successive ports beginning with 4040, then going to 4041, 4042, and so on.

Note that in YARN cluster mode, the UI can be accessed through the YARN ResourceManager which proxies requests directly to the driver.





The screenshot shows the Spark Web UI 'Jobs' page. At the top, there's a navigation bar with tabs for 'Jobs', 'Stages', 'Storage', 'Environment', and 'Executors'. Below the navigation bar, the title 'Spark Jobs (?)' is displayed, along with the total duration (56 min), scheduling mode (FIFO), and completed jobs count (3). A code snippet is shown in a box:

```
val inputRDD=sc.textFile("/user/user01/data/sfpd.csv")
val sfpdRDD=inputRDD.map(x=>x.split(","))
val catRDD=sfpdRDD.map(x=>(x(1),1)).reduceByKey((a,b)=>a+b)
catRDD.cache()
catRDD.collect()
```

The 'Completed Jobs (3)' section contains a table with three rows, each representing a job. The columns are Job Id, Description, Submitted, Duration, Stages: Succeeded/Total, and Tasks (for all stages): Succeeded/Total.

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2	count at <console>:28	2015/09/11 10:23:19	25 ms	1/1 (1 skipped)	2/2 (2 skipped)
1	collect at <console>:28	2015/09/11 09:47:40	71 ms	1/1 (1 skipped)	2/2 (2 skipped)
0	collect at <console>:28	2015/09/11 09:46:58	2 s	2/2	4/4

At the bottom right of the page, it says '© 2015 MapR Technologies' and 'MapR'.

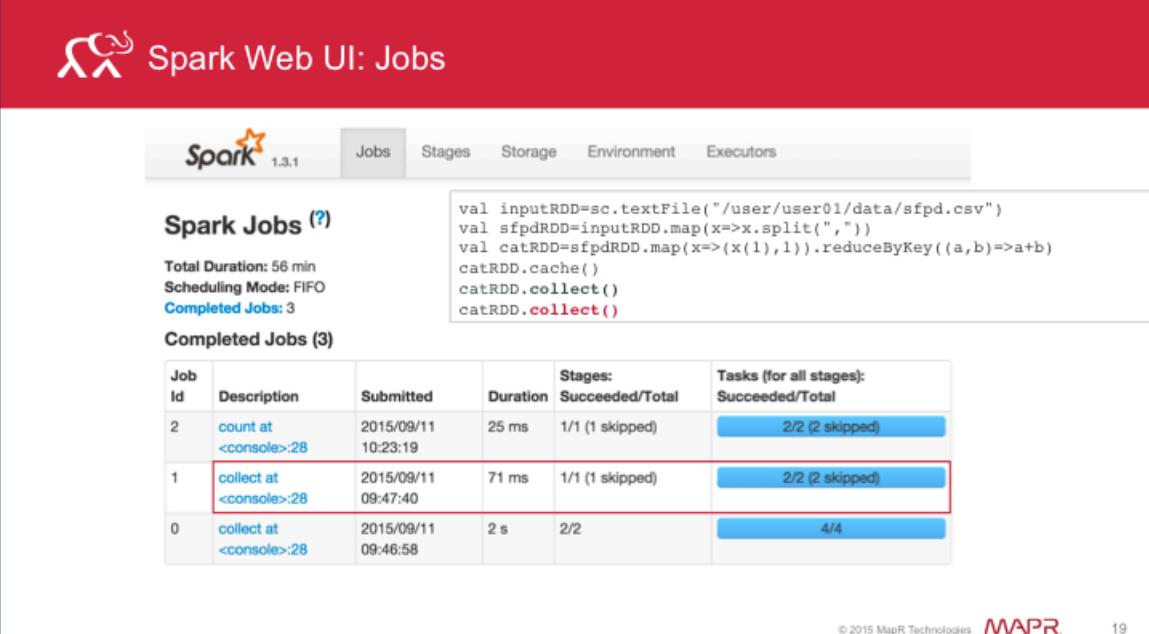
Notes on Slide 18:

To access the Spark Web UI, go to a web browser. Use the ipaddress of the driver and port 4040. The Jobs page gives you detailed execution information for active and recently completed Spark jobs. It gives you the performance of a job, and also the progress of running jobs, stages and tasks.

In this example you see the jobs and stages based on the code sample shown.

- job 0 is the first job that was executed and corresponds to the first collect() action. It consists of 2 stages and each stage consists of 4 tasks.





The screenshot shows the Spark Web UI interface. At the top, there's a red header bar with the Spark logo and the text "Spark Web UI: Jobs". Below the header, there's a navigation bar with tabs: "Jobs" (which is selected), "Stages", "Storage", "Environment", and "Executors".

Spark Jobs (?)

Total Duration: 56 min
Scheduling Mode: FIFO
Completed Jobs: 3

```
val inputRDD=sc.textFile("/user/user01/data/sfpd.csv")
val sfpdRDD=inputRDD.map(x=>x.split(","))
val catRDD=sfpdRDD.map(x=>(x(1),1)).reduceByKey((a,b)=>a+b)
catRDD.cache()
catRDD.collect()
catRDD.collect()
```

Completed Jobs (3)

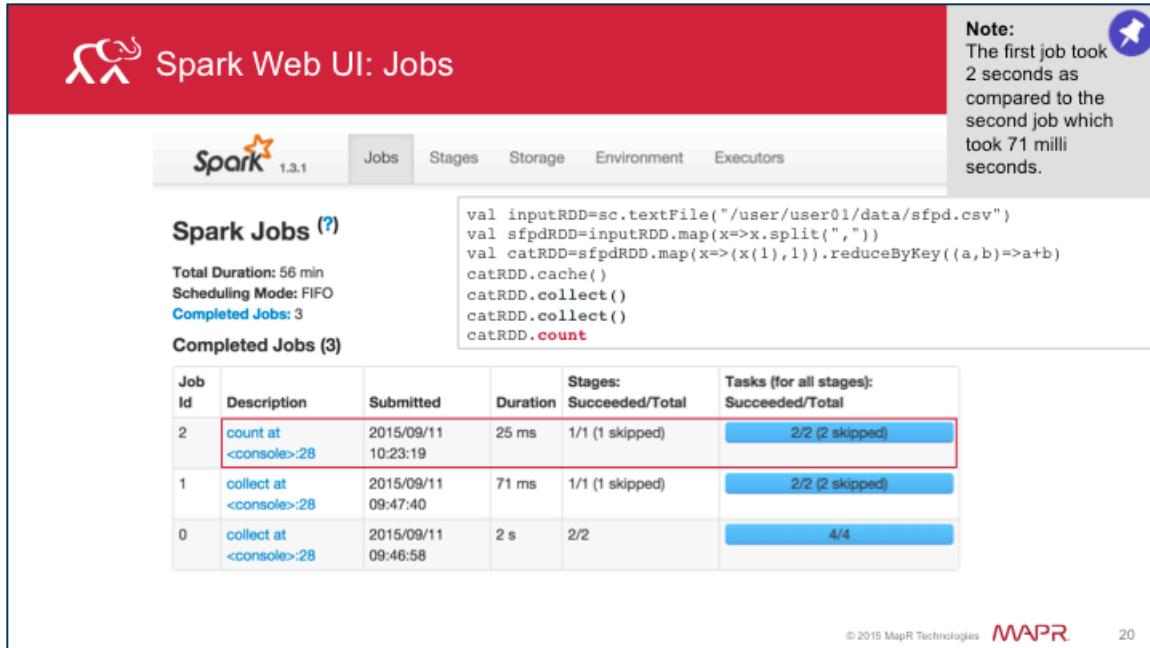
Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2	count at <console>:28	2015/09/11 10:23:19	25 ms	1/1 (1 skipped)	2/2 (2 skipped)
1	collect at <console>:28	2015/09/11 09:47:40	71 ms	1/1 (1 skipped)	2/2 (2 skipped)
0	collect at <console>:28	2015/09/11 09:46:58	2 s	2/2	4/4

© 2015 MapR Technologies **MAPR** 19

Notes on Slide 19:

- job 1 corresponds to the second collect() action. It consists of 1 stage which is made up of two tasks.





Spark Web UI: Jobs

Spark 1.3.1

Jobs Stages Storage Environment Executors

Spark Jobs (?)

Total Duration: 56 min
Scheduling Mode: FIFO
Completed Jobs: 3

Completed Jobs (3)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2	count at <console>:28	2015/09/11 10:23:19	25 ms	1/1 (1 skipped)	2/2 (2 skipped)
1	collect at <console>:28	2015/09/11 09:47:40	71 ms	1/1 (1 skipped)	2/2 (2 skipped)
0	collect at <console>:28	2015/09/11 09:46:58	2 s	2/2	4/4

Note: The first job took 2 seconds as compared to the second job which took 71 milliseconds.

© 2015 MapR Technologies **MapR** 20

Notes on Slide 20:

- job 2 corresponds to the count() action and is also consists of 1 stage containing two tasks.

Note that the first job took 2 seconds as compared to the second job which took 71 milliseconds.



 Discussion: Skipped Stages

The second collect & count jobs only consist of ONE stage (skipped 1 stage) each. Why was one stage “skipped”?

Spark 1.3.1 Jobs Stages Storage Environment Executors

Spark Jobs (?)

Total Duration: 56 min
Scheduling Mode: FIFO
Completed Jobs: 3

Completed Jobs (3)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2	count at <console>-28	2015/09/11 10:23:19	25 ms	1/1 (1 skipped)	2/2 (2 skipped)
1	collect at <console>-28	2015/09/11 09:47:40	71 ms	1/1 (1 skipped)	2/2 (2 skipped)
0	collect at <console>-28	2015/09/11 09:46:58	2 s	2/2	4/4

© 2015 MapR Technologies  21

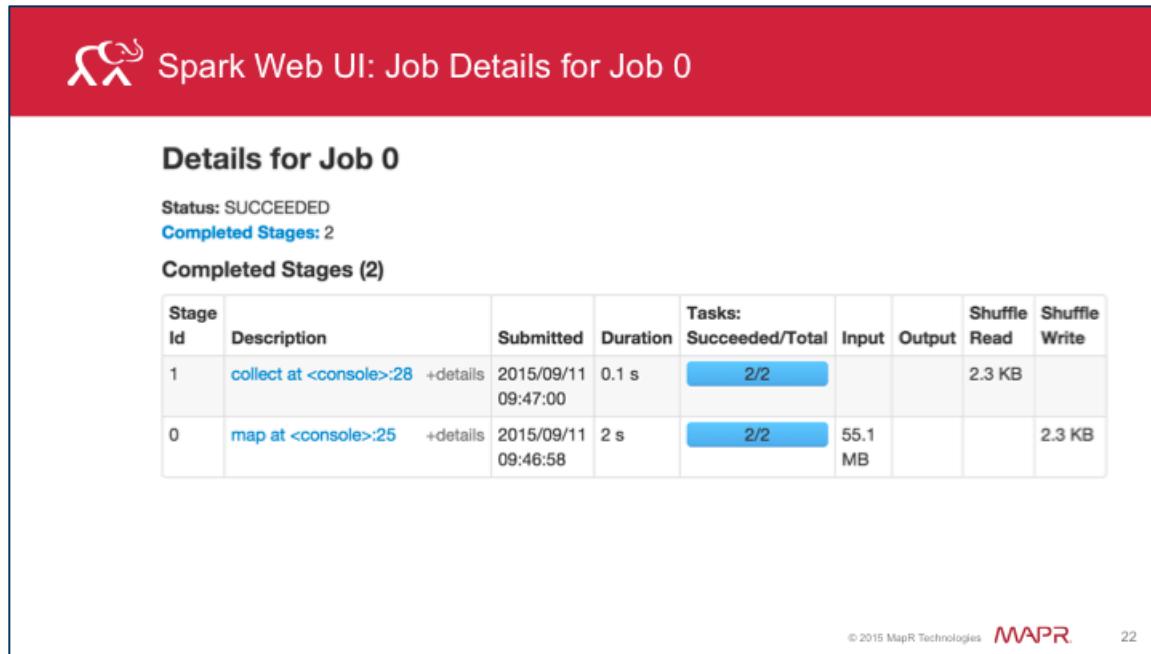
Notes on Slide 21:

Answer:

The first collect computes all RDDs and then caches catRDD and therefore has two stages.

The second `collect()` & the `count()` use the cached RDD and the scheduler truncates the lineage resulting in a skipped stage. This also results in job 1 (71 ms) being faster than job 0 (2 s).





The screenshot shows the Spark Web UI Job Details page for Job 0. The title bar says "Spark Web UI: Job Details for Job 0". Below it, the section "Details for Job 0" is displayed. It shows the status as "SUCCEEDED" and "Completed Stages: 2". A table titled "Completed Stages (2)" lists the two stages:

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
1	collect at <console>:28 +details	2015/09/11 09:47:00	0.1 s	2/2			2.3 KB	
0	map at <console>:25 +details	2015/09/11 09:46:58	2 s	2/2	55.1 MB			2.3 KB

At the bottom right of the page, there is a copyright notice: "© 2015 MapR Technologies" and the MapR logo.

Notes on Slide 22:

Clicking the link in the Description column on the Jobs page takes you to the Job Details page. This page also gives you the progress of running jobs, stages and tasks.

Note that the collect job here takes 1 s.





Spark Web UI: Job Details for Job 1

Details for Job 1

Status: SUCCEEDED
Completed Stages: 1
Skipped Stages: 1

Completed Stages (1)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
3	collect at <console>:28 +details	2015/09/11 09:47:40	31 ms	2/2	4.6 KB			

Skipped Stages (1)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
2	map at <console>:25+details	Unknown	Unknown	0/2				

© 2015 MapR Technologies  23

Notes on Slide 23:

This page provides the details for Job 1, which had the skipped stage. In this page you can see the details for the completed stage and the skipped stage.

Note that the collect here took only 31 ms.





The screenshot shows the Spark Web UI for Job 1. At the top, there's a red header bar with the MapR logo and the text "Spark Web UI: Stage Details for Job 1". Below the header, there's a section titled "Details for Stage 0" which includes summary metrics like total task time, input size, and shuffle write. There's also a link to "Show additional metrics". The main content area has a title "Summary Metrics for 2 Completed Tasks" with a table showing metrics for Duration, GC Time, Input Size / Records, and Shuffle Write Size / Records. Below this is another section titled "Aggregated Metrics by Executor" with a table showing metrics for Executor ID, Address, Task Time, Total Tasks, Failed Tasks, Succeeded Tasks, Input Size / Records, and Shuffle Write Size / Records. The table shows one row for the driver executor. At the bottom, there's a section titled "Tasks" with a table showing columns for Index, ID, Attempt, Status, Locality, Executor ID / Host, Launch Time, GC Duration, Input Size / Records, Write Time, Shuffle Write Size / Records, and a link. The footer of the UI includes copyright information and the MapR logo.

Notes on Slide 24:

Once you have identified the stage in which you are interested, you can click the link to drill down to the Stage details page.

Here we have summary metrics and aggregated metrics for completed tasks and metrics on all tasks.



Spark Web UI: Storage

Jobs Stages Storage Environment Executors

Note:
Scan this page to see if important datasets are fitting into memory

Storage

RDD Name	Storage Level
4	Replicated

RDD Storage Info for 4

Storage Level: Memory Deserialized 1x Replicated
 Cached Partitions: 2
 Total Partitions: 2
 Memory Size: 4.6 KB
 Disk Size: 0.0 B

Data Distribution on 1 Executors

Host	Memory Usage	Disk Usage
localhost:53831	4.6 KB (246.0 MB Remaining)	0.0 B

2 Partitions

Block Name	Storage Level	Size in Memory	Size on Disk	Executors
rdd_4_0	Memory Deserialized 1x Replicated	2.5 KB	0.0 B	localhost:53831
rdd_4_1	Memory Deserialized 1x Replicated	2.1 KB	0.0 B	localhost:53831

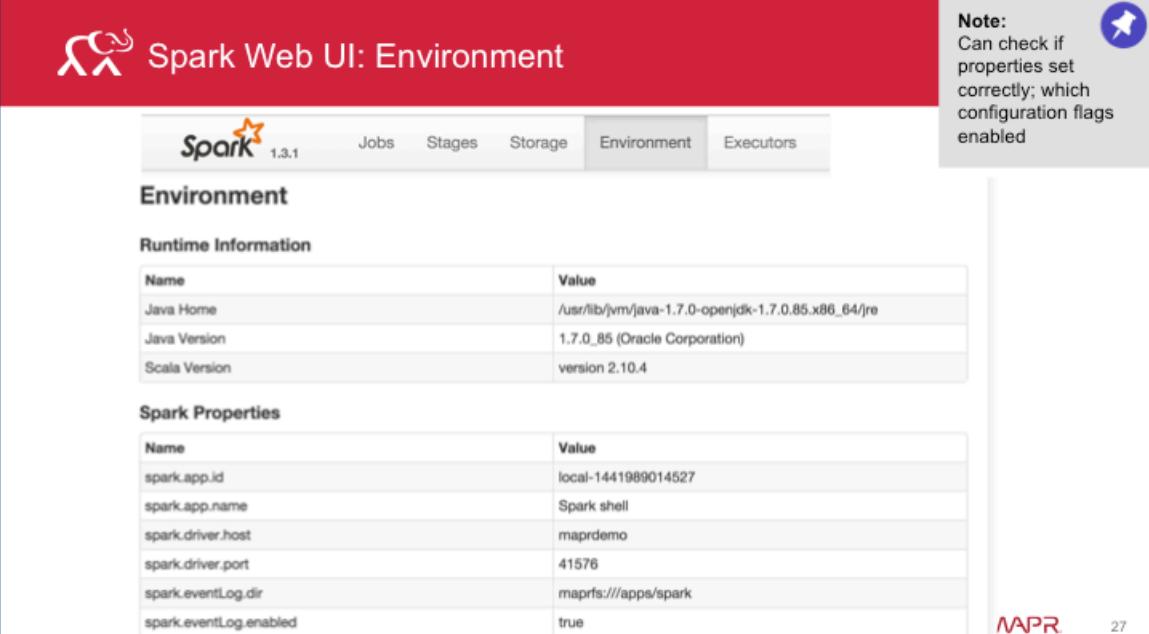
© 2015 MapR Technologies **MAPR** 25

Notes on Slide 25:

The Storage page provides information about persisted RDDs. The RDD is persisted if you called `persist()` or `cache()` on the RDD followed by an action to compute on that RDD. This page tells you which fraction of the RDD is cached and the quantity of data cached in various storage media.

Scan this page to see if important datasets are fitting into memory.





Spark Web UI: Environment

Environment

Runtime Information

Name	Value
Java Home	/usr/lib/jvm/java-1.7.0-openjdk-1.7.0.85.x86_64/jre
Java Version	1.7.0_85 (Oracle Corporation)
Scala Version	version 2.10.4

Spark Properties

Name	Value
spark.app.id	local-1441989014527
spark.app.name	Spark shell
spark.driver.host	maprdemo
spark.driver.port	41576
spark.eventLog.dir	maprfs:///apps/spark
spark.eventLog.enabled	true

Note:
Can check if properties set correctly; which configuration flags enabled

MAPR 27

Notes on Slide 27:

The Environments page lists all the active properties of your Spark application environment.

Use this page when you want to see which configuration flags are enabled.

Note that only values specified through `spark-defaults.conf`, `SparkConf`, or the command line will be displayed here. For all other configuration properties, the default value is used.



Executors (1)

Memory: 4.6 KB Used (246.0 MB Total)
Disk: 0.0 B Used

Executor ID	Address	RDD Blocks	Memory Used	Disk Used	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time	Input	Shuffle Read	Shuffle Write	Thread Dump
<driver>	localhost:53831	2	4.6 KB / 246.0 MB	0.0 B	0	0	8	8	4.3 s	55.1 MB	0.0 B	2.3 KB	Thread Dump

Access executors stack trace

© 2015 MapR Technologies **MAPR** 26

Notes on Slide 26:

The Executors page lists the active executors in the application. It also includes some metrics about the processing and storage on each executor.

Use this page to confirm that your application has the amount of resources that you were expecting.



The screenshot shows the MapR Control System (MCS) interface. At the top, there's a red header bar with the MapR logo and the text "Spark Web UI via MCS". Below this, the main interface has a green navigation bar on the left with items like Cluster, MapR FS, NFS HA, Alarms, System Settings, Hbase, Job Tracker, CLDB, SparkHistoryServer, ResourceManager, JobHistoryServer, and Nagios. The main content area has tabs for Dashboard, SparkHistoryServer, Jobs, Stages, Storage, Environment, and Executors. The "Jobs" tab is selected, showing the "Spark Jobs" section with a table of completed jobs. The table has columns for Job Id, Description, Submitted, Duration, Stages: Succeeded/Total, and Tasks (for all stages): Succeeded/Total. There are two rows of data:

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1	collect at <console>:28	2015/09/11 09:47:40	71 ms	1/1 (1 skipped)	2/2 (2 skipped)
0	collect at <console>:28	2015/09/11 09:46:58	2 s	2/2	4/4

At the bottom right of the interface, it says "© 2015 MapR Technologies MAPR".

Notes on Slide 28:

If you have a MapR distribution, you can access the Spark Web UI through the MapR Control System (MCS). Once you log into the MCS, you will see the Spark History Server in the left navigation pane as shown here.





Knowledge Check

What are some of the things you can monitor in the Spark Web UI?

- A. Which stages are running slow
- B. Your application has the resources as expected
- C. If the datasets are fitting into memory
- D. All of the above

© 2015 MapR Technologies **MAPR**

29

Notes on Slide 29:

Answer: D

You can use the Job details page and the stages tab to see which stages are running slow; to compare the metrics for a stage; look at each task.

Look at the Executors tab to see if your application has the resources as expected

Use the Storage tab to see if the datasets are fitting into memory and what fraction is cached.





Learning Goals

1. Describe Components of Spark Execution Model
2. Use Spark Web UI to Monitor Spark Applications
- ▶ **3. Debug & Tune Spark Applications**

© 2015 MapR Technologies **MAPR**

30

Notes on Slide 30:

We are now going to see how to debug and tune our Spark applications.





Detect Performance Problems

Spark Web UI: Jobs, Stages & Stage Details

- Are there any tasks that are significantly slow?
- Is issue because of skew?
 - Do some tasks read or write much more data than others?
- Are tasks running on certain nodes slow?
- How much time tasks spend on each phase: read, compute, write?

© 2015 MapR Technologies

31

Notes on Slide 31:

Here are some ways to debug performance issues.

To detect shuffle problems, look at the Spark Web UI for any tasks that are significantly slow. A common source of performance problems in data-parallel systems that occurs when some small tasks take much longer than others is called skew. To see if skew is the problem, look at the Stages Details page and see if there are some tasks that are running significantly slower than others. Drill down to see if there is a small number of tasks that read or write more data than others.

From the Stages Details page, you can also determine if tasks are running slow on certain nodes.

From the Spark Web UI, you can also find those tasks that are spending too much time on reading, computing and/or writing.

In this case, look at the code to see if there are any expensive operations.



Common Slow Performance Issues



Level of Parallelism Serialization Format Memory Management

© 2015 MapR Technologies  32

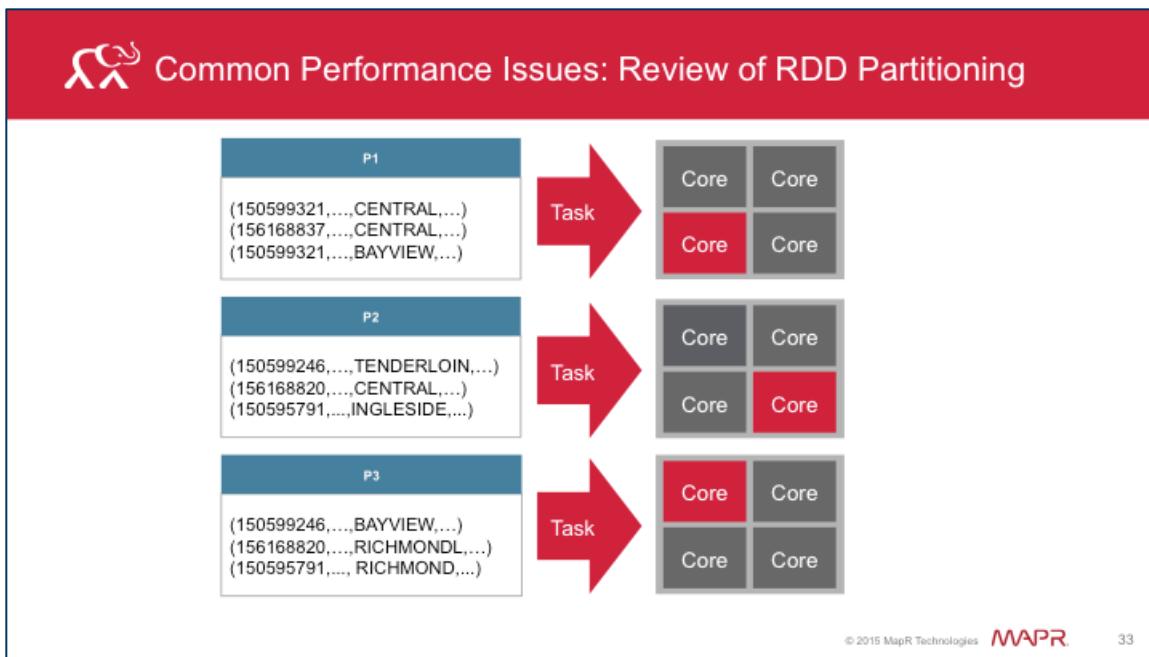
Notes on Slide 32:

Common issues leading to slow performance are:

- The level of parallelism
- The serialization format used during shuffle operations
- Managing memory to optimize your application

We will look at each one of these in more depth.



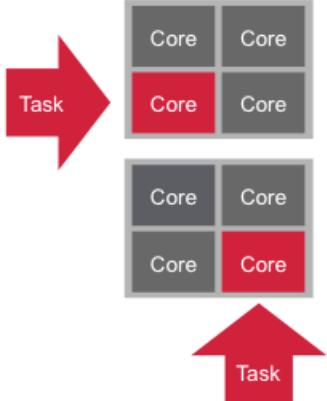


Notes on Slide 33:

An RDD is divided into a set of partitions where each partition contains a subset of the data. The scheduler will create a task for each partition. Each task requires a single core in the cluster. Spark by default will partition based on what it considers to be the best degree of parallelism.



Common Performance Issues: Level of Parallelism



© 2015 MapR Technologies  34

Notes on Slide 34:

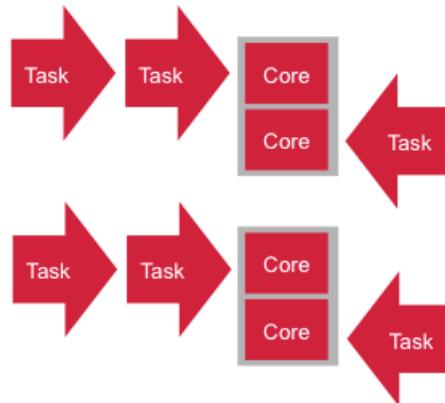
How does the level of parallelism affect performance?

If there is too little parallelism – Spark might leave resources idle.





Common Performance Issues: Level of Parallelism



© 2015 MapR Technologies 

35

Notes on Slide 35:

If there is too much parallelism, overheads associated with each partition add up to become significant.





Common Performance Issues: Tuning Level of Parallelism

Check the number of partitions

1. Use Spark Web UI Stages.

Total tasks = number of partitions

2. Use `rdd.partitions.size()`

Tasks:	Succeeded/Total
	2/2
	2/2
	2/2
	2/2

© 2015 MapR Technologies 

36

Notes on Slide 36:

How do you find the number of partitions?

You can do this through the Spark Web UI in the stages tab. Since a task in a stage maps to a single partition in the RDD, the total number of tasks will give you the number of partitions.

You can also use `rdd.partitions.size()` to get the number of partitions in the RDD.





Common Performance Issues: Tuning Level of Parallelism

Tune level of parallelism:

1. Specify degree of parallelism for operations that shuffle data
 - For example, `reduceByKey(func, numPartitions)`
2. Change the number of partitions (fewer or more) in an RDD
 - `repartition()`
 - `coalesce()`

© 2015 MapR Technologies 

37

Notes on Slide 37:

To tune the level of parallelism:

- Specify the number of partitions, when you call operations that shuffle data, for example, `reduceByKey`.
- Redistribute the data in the RDD. This can be done by increasing or decreasing the number of partitions. You can use the `repartition()` method to specify the number of partitions or `coalesce()` to decrease the number of partitions.





Common Performance Issues: Serialization Format

- During data transfer, Spark serializes data
- Happens during shuffle operations
- Java built in serializer is default
- Use Kryo serialization – often more efficient

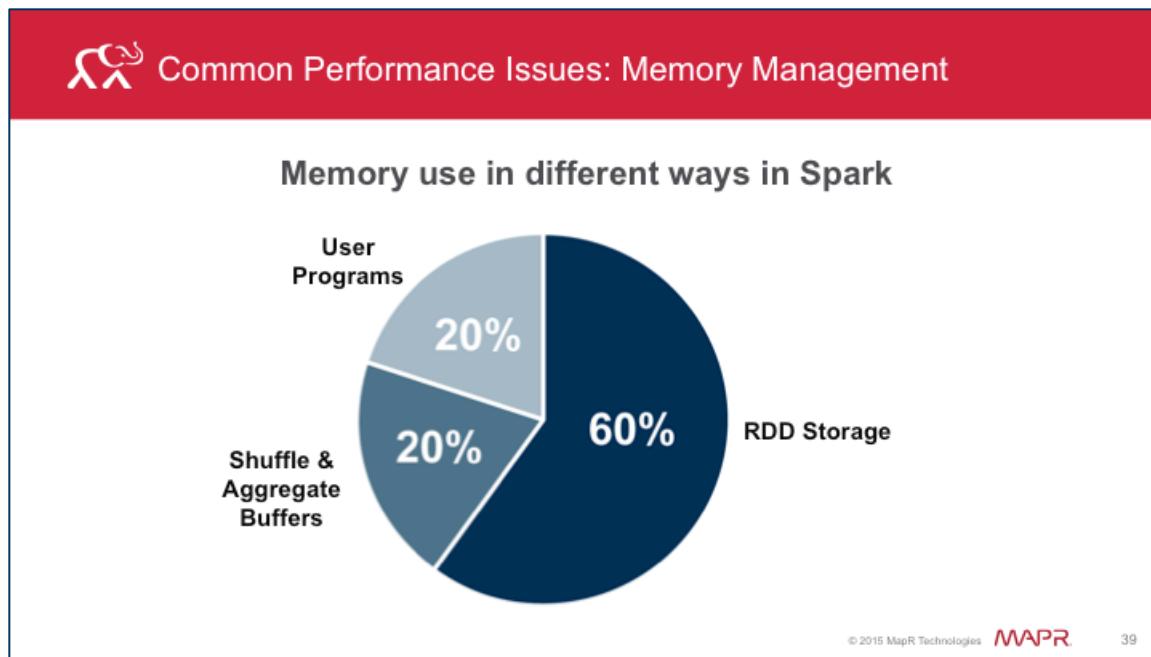
© 2015 MapR Technologies 

38

Notes on Slide 38:

When a large amount of data is transferred over the network during shuffle operations, Spark serializes objects into binary format. This can sometimes cause a bottleneck. By default Spark uses the Java built-in serializer. However, it is often more efficient to use Kryo serialization.





Notes on Slide 39:

Memory can be used in different ways in Spark. Tuning Spark's use of memory can help optimize your application.

By default, Spark will use:

- 60% of space for RDD storage
- 20% for shuffle
- 20% for user programs





Common Performance Issues: Tuning Memory Usage

- RDD storage: `cache()` and `persist()`
- Various options: `persist()`
- By default: `persist(MEMORY_ONLY)` level
- Reduce expensive computations: `persist(MEMORY_AND_DISK)`
- Reduce garbage collection: `(MEMORY_ONLY_SER)`

© 2015 MapR Technologies

40

Notes on Slide 40:

You can tune the memory usage by adjusting the memory regions used for RDD storage, shuffle and user programs.

Use `cache()` or `persist()` on RDDs. Using `cache()` on an RDD will store the RDD partitions in memory buffers.

There are various options for `persist()`. Refer to Apache Spark documentation on persist options for RDD.

By default, `persist()` is the same as `cache()` or `persist(MEMORY_ONLY)`. If there is not enough space to cache new RDD partitions, old ones are deleted and recomputed when needed.

It is better to use `persist(MEMORY_AND_DISK)`. This will store the data on disk and load it into memory when needed. This cuts down expensive computations.

Using `MEMORY_ONLY_SER` will cut down on garbage collection. Caching serialized objects may be slower than caching raw objects. However, it does decrease the time spent on garbage collection.





Spark logging subsystem based on log4j

Deployment Mode	Location
Spark Standalone	work/ directory of distribution on each worker
Mesos	work/ directory of Mesos slave
YARN	Use YARN log collection tool

© 2015 MapR Technologies  41

Notes on Slide 41:

- The Spark logging subsystem is based on log4j. The logging level or log output can be customized. An example of the log4j configuration properties is provided in the Spark conf directory which can be copied and suitably edited.
- The location of the Spark log files depends on the deployment mode.
- In the Spark Standalone mode, the log files are located in the work/ directory of the Spark distribution on each worker.
- In Mesos, the log files are in work/ directory of the Mesos slave and is accessible from the Mesos master UI.
- To access the logs in YARN, use the YARN log collection tool.





Best Practices & Tips

- Avoid shuffling large amounts of data
- Do not copy all elements of RDD to driver
- Filter sooner rather than later
- If you have many idle tasks, `coalesce()`
- If not using all slots in cluster, repartition

© 2015 MapR Technologies

42

Notes on Slide 42:

- If possible avoid having to shuffle large amounts of data. Use `aggregateByKey` when possible for aggregations.
- groupByKey on large dataset results in shuffling large amounts of data. If possible use `reduceByKey`. You can also use `combineByKey` or `foldByKey`.
- `collect()` action tries to copy every single element in the RDD to the single driver program. If you have a very large RDD, this can result in the driver crashing. The same problem occurs with `countByKey`, `countByValue` and `collectAsMap`.
 - Filter out as much as you can to have smaller datasets
 - If you have many idle tasks (~ 10k) , then `coalesce`
 - If you are not using all the slots in the cluster, then repartition





Knowledge Check

Some ways to improve performance of your Spark application include:

- A. Using Kyro serialization
- B. Tune the degree of parallelism
- C. Avoid shuffling large amounts of data
- D. Don't use `collect()` on large datasets
- E. All of the above

© 2015 MapR Technologies **MAPR**

43

Notes on Slide 43:

Answer: E





Notes on Slide 44:

In this Lab, you will create RDDs, apply transformations and actions; print the DAG to console; use the Spark UI to monitor the Spark execution components.



**DEV 362**

Create Data Pipelines
Using Apache Spark

© 2015 MapR Technologies **MAPR**

45

Notes on Slide 45:

Congratulations! You have completed Lesson 6 and this course – DEV 361 – Build and Monitor Apache Spark Applications. Go to doc.mapr.com for more information on Hadoop, Spark and other eco system components. To learn more about other courses, visit the MapR academy.

