DataBricks Spark – Azure DataBricks

Contents

Spark RDD

- RDDs are part of core Spark
- Resilient Distributed Dataset (RDD)
  - Resilient: If data in memory is lost, it can be recreated
  - Distributed: Processed across the cluster
  - Dataset: Initial data can come from a source such as a file, or it can be created programmatically
  - Despite the name, RDDs are not Spark SQL Dataset objects

- RDDs predate Spark SQL and the DataFrame/Dataset API

Spark RDD

- RDDs are unstructured
  - No schema defining columns and rows
  - Not table-like; cannot be queried using SQL-like transformations such as where and select
  - RDD transformations use lambda functions

- RDDs can contain any type of object
  - DataFrames are limited to Row objects
  - Datasets are limited to Row objects, case class objects (products), and primitive types

- Each RDD is split into multiple partitions, which may be computed on different nodes of the cluster

RDD Data Types

- RDDs can hold any serializable type of element
  - Primitive types such as integers, characters, and booleans
  - Collections such as strings, lists, arrays, tuples, and dictionaries (including nested collection types)
  - Scala/Java Objects (if serializable)
  - Mixed types
- Some RDDs are specialized and have additional functionality
  - Pair RDDs
    - RDDs consisting of key-value pairs
  - Double RDDs
    - RDDs consisting of numeric data

RDD Data Sources

- There are several types of data sources for RDDs
  - Files, including text files and other formats
  - Data in memory
  - Other RDDs
  - Datasets or DataFrames

Creating RDD from Files

- Use SparkContext object, not SparkSession
  - SparkContext is part of the core Spark library
  - SparkSQL is part of the Spark SQL library
  - One Spark context per application
  - Use sparkContext to access the Spark context
  - Called sc in the Spark shell

Creating RDD from Files

- Create file-based RDDs using the Spark context
  - Use textFile or wholeTextFiles to read text files
  - Use hadoopFile or newAPIHadoopFile to read other formats
  - The Hadoop “new API” was introduced in Hadoop .20
  - Spark supports both for backward compatibility

Creating RDD from Text File

- SparkContext.textFile reads newline-terminated text files
  - Accepts a single file, a directory of files, a wildcard list of files, or a commaseparated list of files

Examples

```
textFile("myfile.txt")
textFile("mydata/")
textFile("mydata/*.log")
textFile("myfile1.txt,myfile2.txt")
```

Sample Python Code

```
myRDD = sparkContext.textFile("mydata/")
```

Creating RDD from Text File

- textFile maps each line in a file to a separate RDD element
  - Only supports newline-terminated text

```
myRDD = spark. \
  sparkContext. \
  textFile("purplecow.txt")
```

Language: *Python*

```
I've never seen a purple cow.\n
I never hope to see one;\n
But I can tell you, anyhow,\n
I'd rather see than be one.\n
```

myRDD

I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.

wholeTextFile usage

```
userRDD = spark.sparkContext. \
  wholeTextFiles("userFiles")
```

Language: *Python*

userRDD

("user1.json",{"firstName":"Fred", "lastName":"Flintstone","userid":"123"})
("user2.json",{"firstName":"Barney", "lastName":"Rubble","userid":"234"})
("user3.json",…)
("user4.json",…)

Creating RDD from Collection

- You can create RDDs from collections instead of files
  - SparkContext.parallelize(collection)
- Useful when
  - Testing
  - Generating data programmatically
  - Integrating with other systems or libraries
  - Learning

```
myData = ["Alice","Carlos","Frank","Barbara"]
myRDD = sc.parallelize(myData)
```

Language: *Python*

Saving RDD

- You can save RDDs to the same data source types supported for reading RDDs
  - Use RDD.saveAsTextFile to save as plain text files in the specified directory
  - Use RDD.saveAsHadoopFile or saveAsNewAPIHadoopFile with a specified Hadoop OutputFormat to save using other formats
  - The specified save directory cannot already exist

Python Example

```
myRDD.saveAsTextFile("mydata/")
```

RDD Operations

- Two general types of RDD operations
  - Actions return a value to the Spark driver or save data to a data source
  - Transformations define a new RDD based on the current one(s)

- RDDs operations are performed lazily
  - Actions trigger execution of the base RDD transformations

RDD Action Operations

- Common RDD operations are
  - count returns the number of elements
  - first returns the first element
  - take(n) returns an array (Scala) or list (Python) of the first n elements
  - collect returns an array (Scala) or list (Python) of all elements
  - saveAsTextFile(dir) saves to text files

RDD Action Operations

- Basic RDD Actions are
  - collect() - Returns all the elements of the RDD. Ex - rdd.collect()
  - count() - Number of elements in RDD. Ex - rdd.count()
  - countByValue() - Number of times each element occurs in RDD. Ex - rdd.countByValue()
  - take(num) - Return num of elements of the RDD. Ex - rdd.take(10)
  - top(num) - Returns the top num elements in RDD Ex - rdd.top(10)
  - reduce(func) - Combines the elements of RDD together in parallel. Ex - rdd.reduce((x,y) => x + y)

RDD Transformation Operations

- Transformations create a new RDD from an existing one
- RDDs are immutable
  - Data in an RDD is never changed
  - Transform data to create a new RDD

- A transformation operation executes a transformation function
  - The function transforms elements of an RDD into new elements
  - Some transformations implement their own transformation logic
  - For many, you must provide the function to perform the transformation

RDD Transformation Operations

- Transformation operations include
  - distinct creates a new RDD with duplicate elements in the base RDD removed
  - union(rdd) creates a new RDD by appending the data in one RDD to another
  - map(function) creates a new RDD by performing a function on each record in the base RDD
  - filter(function) creates a new RDD by including or excluding each record in the base RDD according to a Boolean function

RDD Transformation Operations

- Transformation operations include
  - distinct creates a new RDD with duplicate elements in the base RDD removed
  - union(rdd) creates a new RDD by appending the data in one RDD to another
  - map(function) creates a new RDD by performing a function on each record in the base RDD
  - filter(function) creates a new RDD by including or excluding each record in the base RDD according to a Boolean function

Example Transformations

cities1.csv

```
Boston,MA
Palo Alto,CA
Santa Fe,NM
Palo Alto,CA
```

cities2.csv

```
Calgary,AB
Chicago,IL
Palo Alto,CA
```

```
distinctRDD = sc.\
  textFile("cities1.csv").distinct()
for city in distinctRDD.collect(): \
  print city
Boston,MA
Palo Alto,CA
Santa Fe,NM

unionRDD = sc.textFile("cities2.csv"). \
  union(distinctRDD)
for city in unionRDD.collect():
  print city
Calgary,AB
Chicago,IL
Palo Alto,CA
Boston,MA
Palo Alto,CA
Santa Fe,NM
```

Language: *Python*

RDD Transformation Procedures

- RDD transformations execute a transformation procedure
  - Transforms elements of an RDD into new elements
  - Runs on executors

- A few transformation operations implement their own transformation logic
  Examples: distinct and union

- Most transformation operations require you to pass a function
  - Function implements your own transformation procedure
  - Examples: map and filter

Passing Functions

- Passed functions can be named or anonymous
- Anonymous functions are defined inline without an identifier
  - Best for short, one-off functions
  - Supported in many programming languages
  - Python: lambda x: …
  - Scala: x => …
  - Java 8: x -> …

Passing Named Functions in Python

```
def toUpper(s):
  return s.upper()

myRDD = sc. \
  textFile("purplecow.txt")

myUpperRDD = myRDD.map(toUpper)

for line in myUpperRDD.take(2):
  print line
I'VE NEVER SEEN A PURPLE COW.
I NEVER HOPE TO SEE ONE;
```

I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.

↓

myUpperRDD

I'VE NEVER SEEN A PURPLE COW.
I NEVER HOPE TO SEE ONE;
BUT I CAN TELL YOU, ANYHOW,
I'D RATHER SEE THAN BE ONE.

Passing Named Functions in Scala

```
def toUpper(s: String):
  String = { s.toUpperCase }

val myRDD =
 sc.textFile("purplecow.txt")

val myUpperRDD =
  myRDD.map(toUpper)

myUpperRDD.take(2).
  foreach(println)
I'VE NEVER SEEN A PURPLE COW.
I NEVER HOPE TO SEE ONE;
```

I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.

↓

myUpperRDD

I'VE NEVER SEEN A PURPLE COW.
I NEVER HOPE TO SEE ONE;
BUT I CAN TELL YOU, ANYHOW,
I'D RATHER SEE THAN BE ONE.

Passing Anonymous Functions

Python: Use the lambda keyword to specify the name of the input parameter(s) and the function that returns the output
myUpperRDD = myRDD.map(lambda line: line.upper())

Scala: Use the => operator to specify the name of the input parameter(s) and the function that returns the output
val myUpperRDD = myRDD.map(line => line.toUpperCase)

Scala shortcut: Use underscore (_) to stand for anonymous input parameters
val myUpperRDD = myRDD.map(_.toUpperCase)

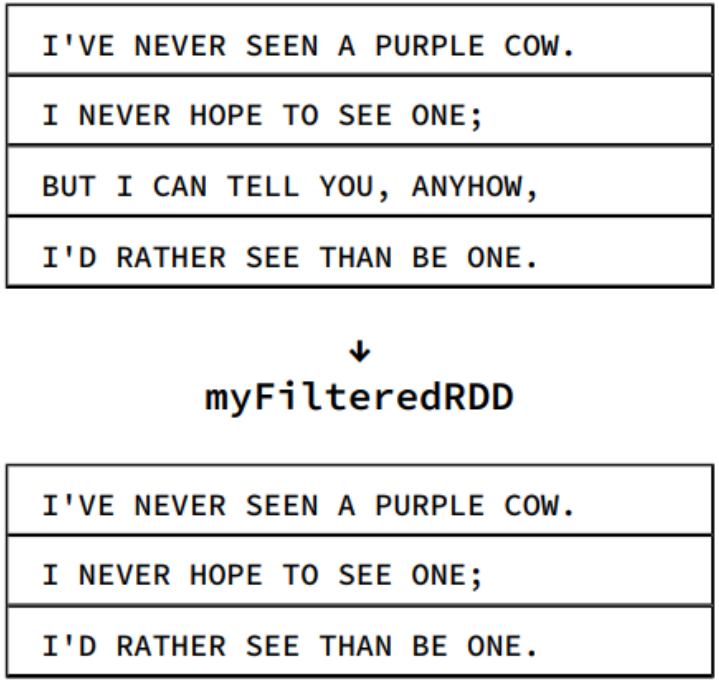
Map and Filter Transformation

```
myFilteredRDD = myRDD. \
  map(lambda line: line.upper()). \
  filter(lambda line: \
    line.startswith('I'))
```

Language: *Python*

```
val myFilteredRDD = myRDD.
  map(line => line.toUpperCase).
  filter(line =>
    line.startsWith("I"))
```

Language: *Scala*



RDD Execution

- An RDD query consists of a sequence of one or more transformations completed by an action
- RDD queries are executed lazily
  - When the action is called

- RDD queries are executed differently than DataFrame and Dataset queries
  - DataFrames and Datasets scan their sources to determine the schema eagerly (when created)
  - RDDs do not have schemas and do not scan their sources before loading

RDD Lineage

- Transformations create a new RDD based on one or more existing RDDs
  - Result RDDs are considered children of the base (parent) RDD
  - Child RDDs depend on their parent RDD

- An RDD's lineage is the sequence of ancestor RDDs that it depends on
  - When an RDD executes, it executes its lineage starting from the source

- Spark maintains each RDD's lineage – Use toDebugString to view the lineage

RDD Persistence(Caching)

- Since RDDs are lazily evaluated, sometimes we may want to use the same RDD multiple times. If we do this normally in spark, everytime the ful execution will happen.

- To avoid the redundant operations to be executed, we can persist the RDD for iterative algorithms in which the same intermediate dataset will be accessed multiple times.

- This will cache the RDD and provide it for the future execution within the application.

rdd.persist() or rdd.cache()

RDD Persistence(Caching)

- RDD can be persisted in different storage levels. They are
  - MEMORY_ONLY - Space used in-memory will be more, CPU time will be less, No Disk involved
  - MEMORY_ONLY_SER - Serializing the RDD will reduce the space in-memory, But CPU time will be more because it has to deserialize the data, No Disk involved
  - MEMORY_AND_DISK - Space used in-memory will be more, CPU time will be less, Data will be spilled to DIsk if there is too much data to fit in-memory.

RDD Persistence(Caching)

- MEMORY_AND_DISK_SER - in-memory used will be less, CPU time will be more due to deserialization, Stores the data in-memory and spilled to DISK with serialization involved.

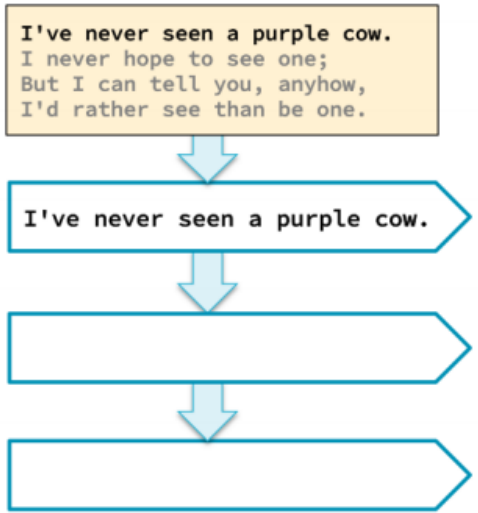
- DISK_ONLY - Data will be persisted in Disk.

Example - rdd.persist(StorageLevel.MEMORY_ONLY)
rdd.cache(StorageLevel.MEMORY_ONLY)

RDD Pipeline

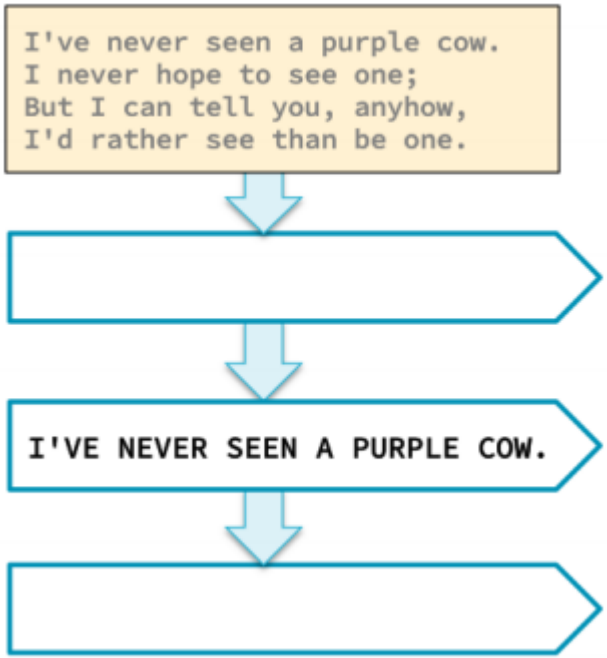
- When possible, Spark will perform sequences of transformations by element so no data is stored

```
myFilteredRDD = sc. textFile("purplecow.txt").
map(line => line.toUpperCase).
filter(line => line.startsWith("I"))
myFilteredRDD.take(2)
```



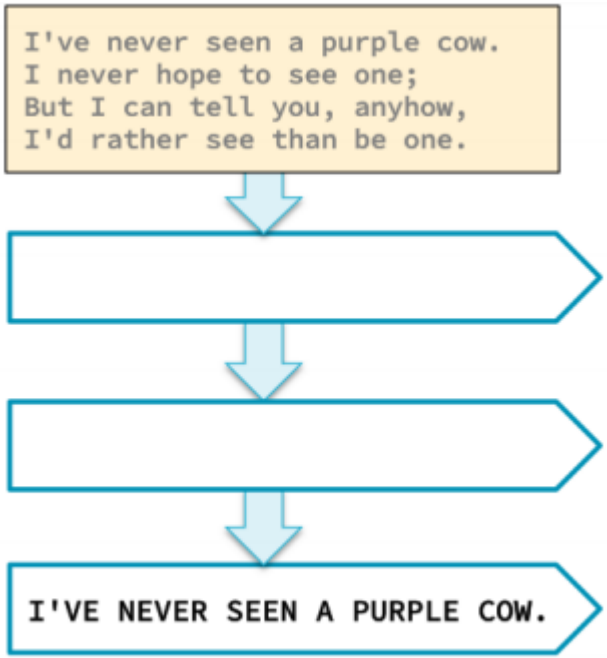
RDD Pipeline

myFilteredRDD = sc. textFile("purplecow.txt").
map(line => line.toUpperCase).
filter(line => line.startsWith("I"))
myFilteredRDD.take(2)



RDD Pipeline

```
myFilteredRDD = sc. textFile("purplecow.txt").
map(line => line.toUpperCase).
filter(line => line.startsWith("I"))
myFilteredRDD.take(2)
```



Pair RDD

- Key/Value RDDs are commonly used to perform aggregations, and often we will do ETL in this form
- Key/Value Pair exposes the operations like, grouping based upon key, grouping together two RDD, counting values based upon the key, etc.,
- Spark provides special operations on RDD containing key/value pairs. These RDDs are called as Pair RDD.

Pair RDD

- Pair RDDs are a special form of RDD
  - Each element must be a key/value pair (a two-element tuple)
  - Keys and values can be any type

- Why?
  - Use with map-reduce algorithms
  - Many additional functions are available for common data processing needs
  - Such as sorting, joining, grouping, and counting

Creating Paired RDD

- The first step in most workflows is to get the data into key/value form
  - What should the RDD should be keyed on?
  - What is the value?

- Commonly used functions to create pair RDDs
  - map
  - flatMap/flatMapValues
  - keyBy

Example - Pair RDD(Python)

```
usersRDD = sc.textFile("userlist.tsv"). \
  map(lambda line: line.split('\t')). \
  map(lambda fields: (fields[0],fields[1]))
```

```
user001\tFred Flintstone
user090\tBugs Bunny
user111\tHarry Potter
…
```

→

("user001","Fred Flintstone")
("user090","Bugs Bunny")
("user111","Harry Potter")
…

Example - Pair RDD(Scala)

```
val usersRDD = sc.textFile("userlist.tsv").
  map(line => line.split('\t').
  map(fields => (fields(0),fields(1)))
```

```
user001\tFred Flintstone
user090\tBugs Bunny
user111\tHarry Potter
…
```

→

("user001","Fred Flintstone")
("user090","Bugs Bunny")
("user111","Harry Potter")
…

Example - Mapping Single Element to Multiple

```
00001 sku010:sku933:sku022
00002 sku912:sku331
00003 sku888:sku022:sku010:sku594
00004 sku411
```

→

("00001","sku010")
("00001","sku933")
("00001","sku022")
("00002","sku912")
("00002","sku331")
("00003","sku888")
…

Example - Mapping Single Element to Multiple

```
val ordersRDD = sc.textFile("orderskus.txt")
```

Language: *Scala*

"00001 sku010:sku933:sku022"
"00002 sku912:sku331"
"00003 sku888:sku022:sku010:sku594"
"00004 sku411"

Example - Mapping Single Element to Multiple

```
val ordersRDD = sc.textFile("orderskus.txt").
  map(line => line.split(' '))
```

Language: *Scala*

Array("00001","sku010:sku933:sku022")
Array("00002","sku912:sku331")
Array("00003","sku888:sku022:sku010:sku594")
Array("00004","sku411")

`split` returns two-element arrays

Example - Mapping Single Element to Multiple

```
val ordersRDD = sc.textFile("orderskus.txt").
  map(line => line.split(' ')).
  map(fields => (fields(0),fields(1)))
```

Language: *Scala*

("00001","sku010:sku933:sku022")
("00002","sku912:sku331")
("00003","sku888:sku022:sku010:sku594")
("00004","sku411")

Map array elements to tuples to produce a pair RDD

Example - Mapping Single Element to Multiple

```
val ordersRDD = sc.textFile("orderskus.txt").
  map(line => line.split(' ')).
  map(fields => (fields(0),fields(1))).
  flatMapValues(skus => skus.split(':'))
```

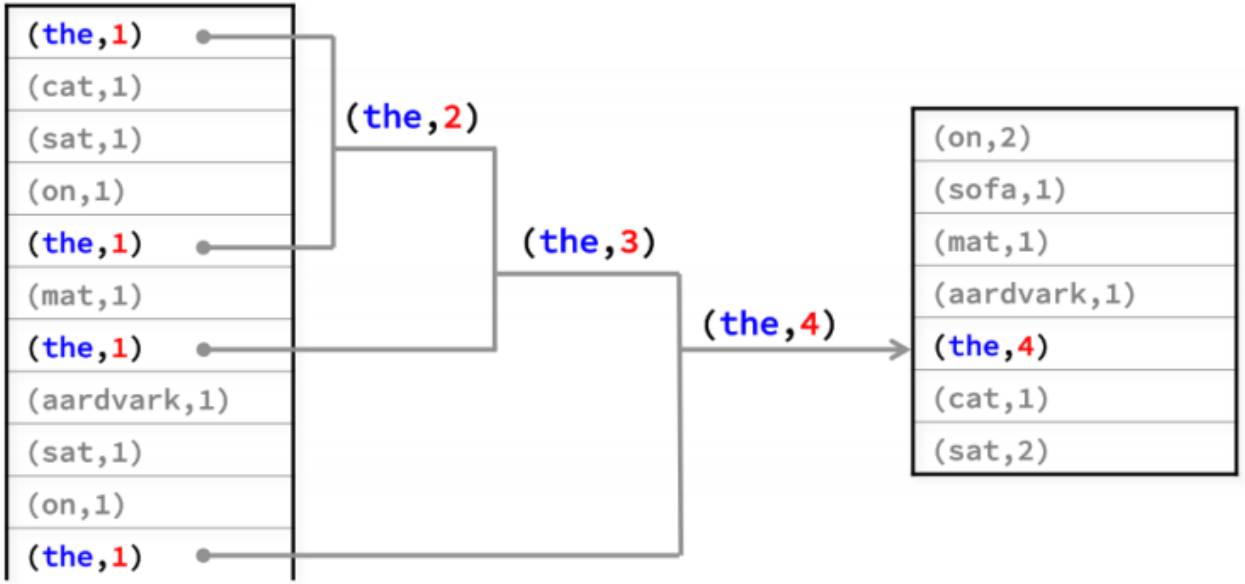
Language: *Scala*

("00001","sku010")
("00001","sku933")
("00001","sku022")
("00002","sku912")
("00002","sku331")
("00003","sku888")
…

`flatMapValues` splits a single value (a colon-separated string of SKUs) into multiple elements

Transformations on Pair RDD

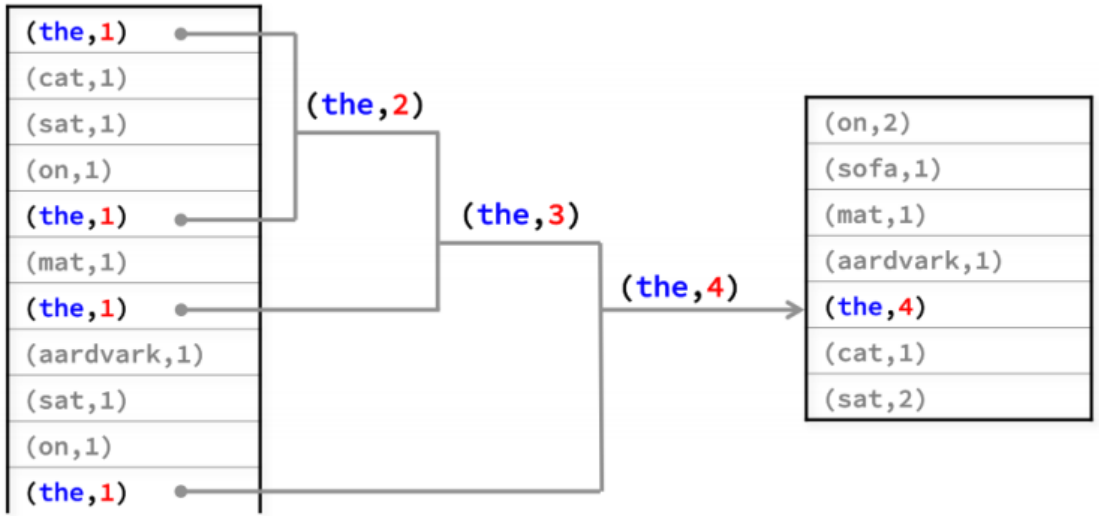
reduceByKey(func) - Combine values with the same Key



Transformations on Pair RDD

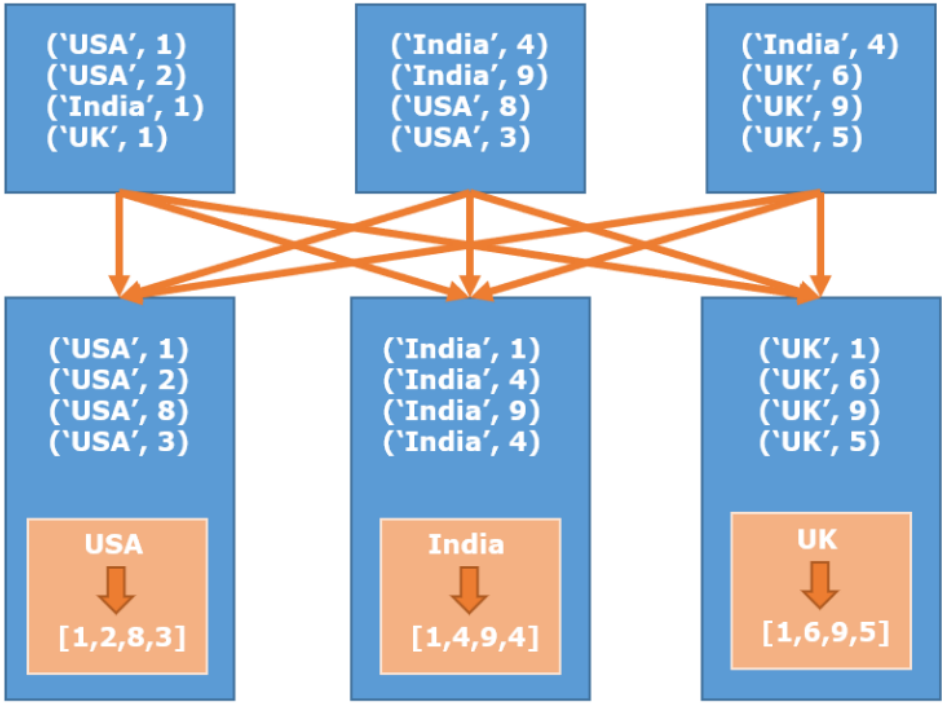
reduceByKey(func) - Combine values with the same Key.

Ex- rdd.reduceByKey((x,y) => x + y)



Transformations on Pair RDD

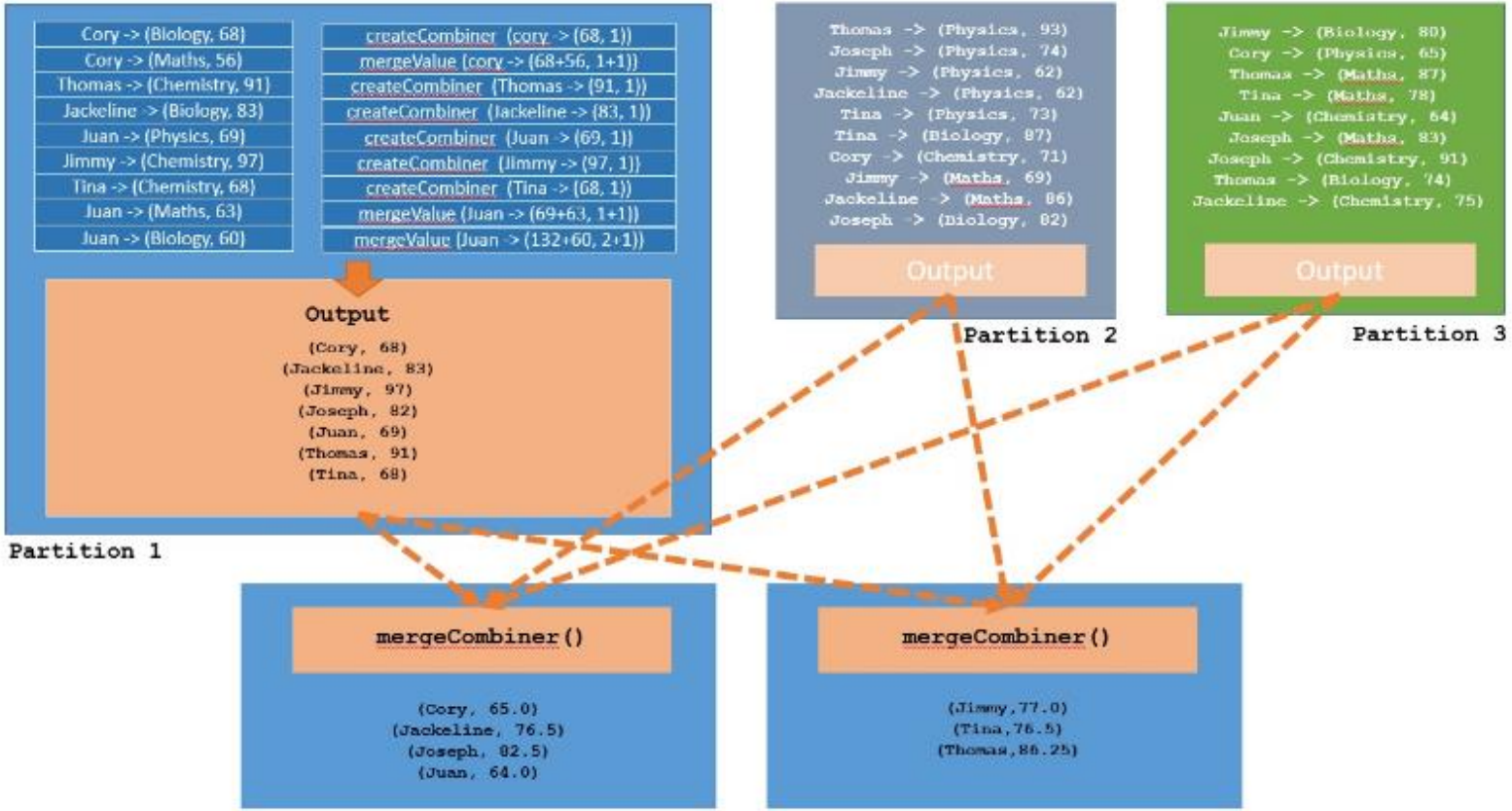
groupByKey - Group values with the same key. Ex - rdd.groupByKey()



Transformations on Pair RDD

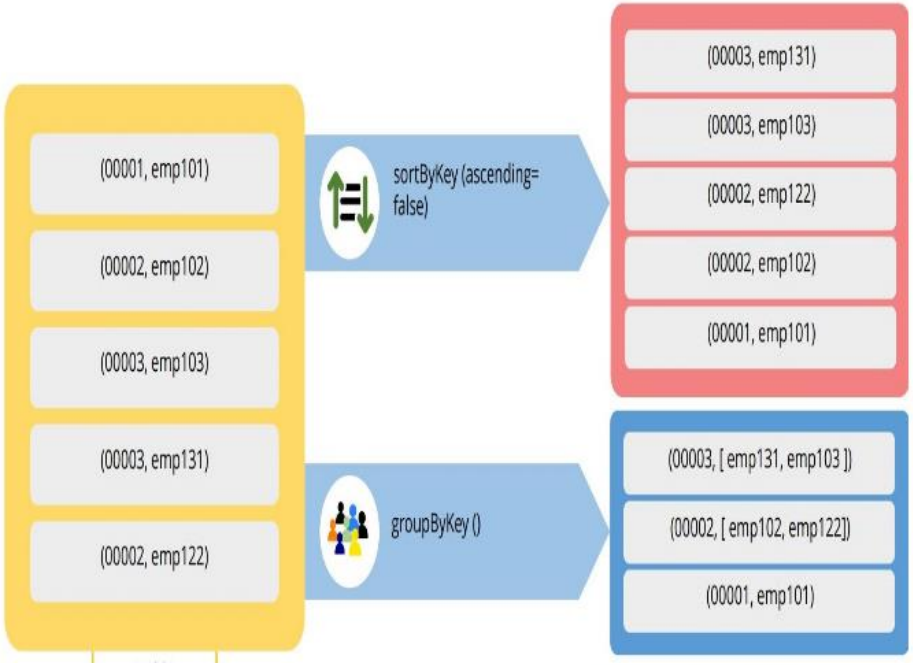
combineByKey - Combine values with the same key using a different result type.

Spark RDD


Cory -> (Biology, 68)
Cory -> (Maths, 56)
Thomas -> (Chemistry, 91)
Jackeline -> (Biology, 83)
Juan -> (Physics, 69)
Jimmy -> (Chemistry, 97)
Tina -> (Chemistry, 68)
Juan -> (Maths, 63)
Juan -> (Biology, 60)
createCombiner (cory -> (68, 1))
mergeValue (cory -> (68+56, 1+1))
createCombiner (Thomas -> (91, 1))
createCombiner (Jackeline -> (83, 1))
createCombiner (Juan -> (69, 1))
createCombiner (Jimmy -> (97, 1))
createCombiner (Tina -> (68, 1))
mergeValue (Juan -> (69+63, 1+1))
mergeValue (Juan -> (132+60, 2+1))
Output
(Cory, 68)
(Jackeline, 83)
(Jimmy, 97)
(Joseph, 82)
(Juan, 69)
(Thomas, 91)
(Tina, 68)
Partition 1
Thomas -> (Physics, 93)
Joseph -> (Physics, 74)
Jimmy -> (Physics, 62)
Jackeline -> (Physics, 62)
Tina -> (Physics, 73)
Tina -> (Biology, 87)
Cory -> (Chemistry, 71)
Jimmy -> (Maths, 69)
Jackeline -> (Maths, 86)
Joseph -> (Biology, 82)
Output
Partition 2
Jimmy -> (Biology, 80)
Cory -> (Physics, 65)
Thomas -> (Maths, 87)
Tina -> (Maths, 78)
Juan -> (Chemistry, 64)
Joseph -> (Maths, 83)
Joseph -> (Chemistry, 91)
Thomas -> (Biology, 74)
Jackeline -> (Chemistry, 75)
Output
Partition 3
mergeCombiner()
(Cory, 65.0)
(Jackeline, 76.5)
(Joseph, 82.5)
(Juan, 64.0)
mergeCombiner()
(Jimmy,77.0)
(Tina,76.5)
(Thomas,86.25)

Transformations on Pair RDD

sortByKey() - returns a RDD sortedby the key. Ex - rdd.sortByKey()



Saving Output files in Spark

- Spark can store the output files in different formats to various different data infrastructure like, HDFS, Hive, S3, Cassandra, HBase, ElasticSearch and other Databases through JDBC connection.

- File formats in the spark output writer includes text files, JSON, CSV, SequenceFiles, ObjectFiles, etc.,

Saving Output files in Spark

TextFile

resultRDD.saveAsTextFile(“/user/admin/data/spark_output/”)

Loading other structured files will be using dataframe.

Spark Shared Variables

- Shared variables are the special variables that we can use in spark task.
- While doing any functions/spark operation, it works on different variables used in that function. Generally, multiple copies of same variables copied to each worker node and the update to this variable never return to driver program. This is an inefficient way as the data transfer rate will be very high here.
- Spark Shared variables helps in reducing the data transfer.

Spark Broadcast Variables

- Broadcast Variables allow the programmers to keep a read-only variable cached on each machine rather than shipping a copy of it within the tasks.
- Without broadcast variables these variables would be shipped to each executor for every transformation and action, and this can cause network overhead.
- Explicitly creating broadcast variables is only useful when tasks across multiple stages need the same data or when caching the data in deserialized form is important.
- For Example, if your application needs to send a large, read-only lookup table to all the nodes, or even a large feature vector in a machine learning algorithm.

Spark Broadcast Variables

- We know that spark automatically sends all variables to the executors in the worker nodes. This is more convenient for processing, but it is not efficient.

- When we are broadcasting large values, it is important to choose a data serialization format for its fast and compact, because the time to send the value over the network can quickly become a bottleneck if the time taken and memory is more.

- Broadcasting very huge data to the cache will get failed because of memory constraints.

RDD Checkpointing

- Checkpointing is a process of truncating RDD lineage graph and saving it to a reliable distributed (HDFS) or local file system.

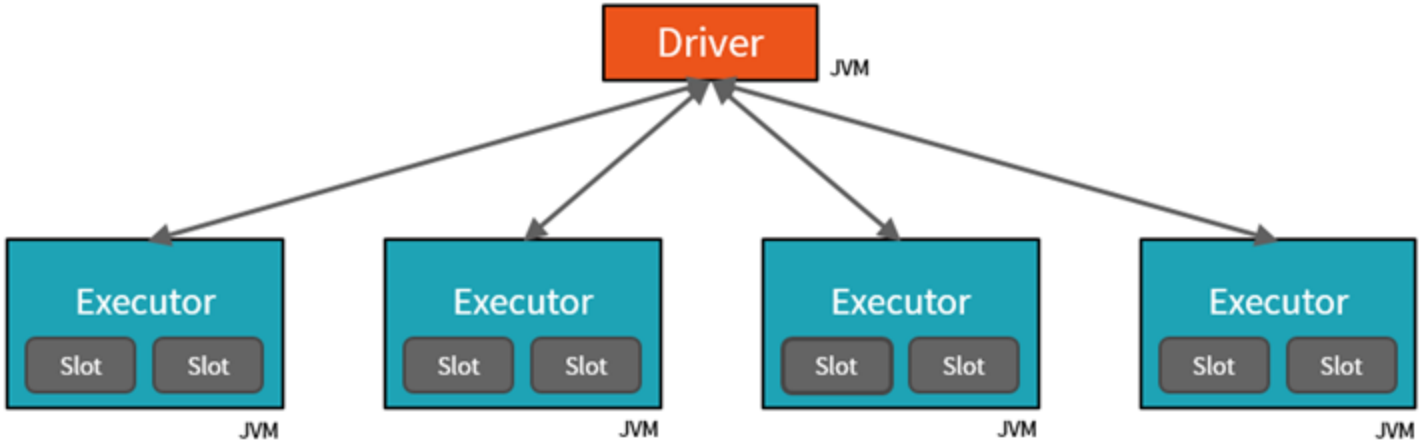
- There are two types of checkpointing:
  - reliable - in Spark (core), RDD checkpointing that saves the actual intermediate RDD data to a reliable distributed file system, e.g. HDFS.
  - local - in Spark Streaming - RDD checkpointing that truncates RDD lineage graph.

RDD Checkpointing

- It’s up to a Spark application developer to decide when and how to checkpoint using RDD.checkpoint() method.

- Before checkpointing is used, a Spark developer has to set the checkpoint directory using SparkContext.setCheckpointDir(directory: String) method.

RDD Execution



Contents

Spark Dataframes

- Like an RDD, a Dataframe is an immutable distributed collection of data
- Unlike an RDD, data is organized into named columns, like a table in a relational database.
- Designed to make large data sets processing even easier, DataFrame allows developers to impose a structure onto a distributed collection of data, allowing higher-level abstraction.
- It provides a domain specific language API to manipulate your distributed data; and makes Spark accessible to a wider audience, beyond specialized data engineers.

Spark Dataframes

- DataFrames contain an ordered collection of Row objects
  - Rows contain an ordered collection of values
  - Row values can be basic types (such as integers, strings, and floats) or collections of those types (such as arrays and lists)
  - A schema maps column names and types to the values in a row

Creating a Spark Dataframes

- The users.json text file contains sample data
  - Each line contains a single JSON record that can include a name, age, and postal code field

```
{"name":"Alice", "pcode":"94304"}
{"name":"Brayden", "age":30, "pcode":"94304"}
{"name":"Carla", "age":19, "pcode":"10036"}
{"name":"Diana", "age":46}
{"name":"Étienne", "pcode":"94104"}
```

Creating a Spark Dataframes

- DataFrames always have an associated schema
- DataFrameReader can infer the schema from the data
- Use printSchema to show the DataFrame's schema

```
> usersDF = \
  spark.read.json("users.json")

> usersDF.printSchema()
root
 |-- age: long (nullable = true)
 |-- name: string (nullable = true)
 |-- pcode: string (nullable = true)

> usersDF.show()
+----+-------+-----+
| age|   name|pcode|
+----+-------+-----+
|null|  Alice|94304|
|  30|Brayden|94304|
|  19|  Carla|10036|
|  46|  Diana| null|
|null|Etienne|94104|
+----+-------+-----+
```

Language: *Python*

Dataframe Operations

- There are two main types of DataFrame operations
  - Transformations create a new DataFrame based on existing one(s)
    - Transformations are executed in parallel by the application's executors
  - Actions output data values from the DataFrame
    - Output is typically returned from the executors to the main Spark program (called the driver) or saved to a file

Dataframe Actions

- Some common DataFrame actions include
  - count: returns the number of rows
  - first: returns the first row (synonym for head())
  - take(n): returns the first n rows as an array (synonym for head(n))
  - show(n): display the first n rows in tabular form (default is 20 rows)
  - collect: returns all the rows in the DataFrame as an array
  - write: save the data to a file or other data source

Action - Example - take Action

```
> usersDF = spark.read.json("users.json")

> users = usersDF.take(3)
[Row(age=None, name=u'Alice', pcode=u'94304'),
 Row(age=30, name=u'Brayden', pcode=u'94304'),
 Row(age=19, name=u'Carla', pcode=u'10036')]
```

Language: *Python*

```
> val usersDF = spark.read.json("users.json")

> val users = usersDF.take(3)
usersDF: Array[org.apache.spark.sql.Row] =
  Array([null,Alice,94304],
        [30,Brayden,94304],
        [19,Carla,10036])
```

Language: *Scala*

Dataframe Transformation

- Transformations create a new DataFrame based on an existing one
  - The new DataFrame may have the same schema or a different one

- Transformations do not return any values or data to the driver
  - Data remains distributed across the application's executors

- DataFrames are immutable
  - Data in a DataFrame is never modified
  - Use transformations to create a new DataFrame with the data you need

Dataframe Transformation

- Common transformations include
  - Select: only the specified columns are included
  - where: only rows where the specified expression is true are included (synonym for filter)
  - orderBy: rows are sorted by the specified column(s) (synonym for sort)
  - join: joins two DataFrames on the specified column(s)
  - limit(n): creates a new DataFrame with only the first n rows

Select and Where Transformation

```
> nameAgeDF = usersDF.select("name","age")
> nameAgeDF.show()
+-------+----+
|   name| age|
+-------+----+
|  Alice|null|
|Brayden|  30|
|  Carla|  19|
|  Diana|  46|
|Etienne|null|
+-------+----+

> over20DF = usersDF.where("age > 20")
> over20DF.show()
+---+-------+-----+
|age|   name|pcode|
+---+-------+-----+
| 30|Brayden|94304|
| 46|  Diana| null|
+---+-------+-----+
```

Language: *Python*

Defining Queries

A sequence of transformations followed by an action is a query

```
> nameAgeDF = usersDF.select("name","age")
> nameAgeOver20DF = nameAgeDF.where("age > 20")
> nameAgeOver20DF.show()
+---+-------+
|age|   name|
+---+-------+
| 30|Brayden|
| 46|  Diana|
+---+-------+
```

Chaining Transformations

```
> val nameAgeDF = usersDF.select("name","age")
> val nameAgeOver20DF = nameAgeDF.where("age > 20")
> nameAgeOver20DF.show
```

Language: *Scala*

```
> usersDF.select("name","age").where("age > 20").show
```

Language: *Scala*

Creating Dataframes from Data sources

- DataFrames read data from and write data to data sources
- Spark SQL supports a wide range of data source types and formats
  - Text files
  - CSV
  - JSON
  - Plain text
  - Binary format files
  - Apache Parquet
  - DeltaLake Tables
  - Hive Tables and metastore
  - JDBC

You can also use custom or third-party data source types

Creating Dataframes from Data sources

- spark.read returns a DataFrameReader object
- Use DataFrameReader settings to specify how to load data from the data source
  - format indicates the data source type, such as csv, json, or parquet (the default is parquet)
  - option specifies a key/value setting for the underlying data source
  - schema specifies a schema to use instead of inferring one from the data source

- Create the DataFrame based on the data source
  - load loads data from a file or files
  - table loads data from a Hive table

Creating Dataframes from Data sources

Example: Read a CSV text file – Treat the first line in the file as a header instead of data

```
myDF = spark.read. \
  format("csv"). \
  option("header","true"). \
  load("/loudacre/myFile.csv")
```

Language: *Python*

- **Example: Read a table defined in the Hive metastore**

```
myDF = spark.read.table("my_table")
```

Language: *Python*

Dataframe Reader

- You can call a format-specific load function
  - A shortcut instead of setting the format and using load
- The following two code examples are equivalent

```
spark.read.format("csv").load("/loudacre/myFile.csv")
```

```
spark.read.csv("/loudacre/myFile.csv")
```

Specifying Datafile Locations

- You must specify a location when reading from a file data source
- The location can be a single file, a list of files, a directory, or a wildcard

- Examples
  - spark.read.json("myfile.json")
  - spark.read.json("mydata/")
  - spark.read.json("mydata/*.json")
  - spark.read.json("myfile1.json","myfile2.json")

Specifying Datafile Locations

- Files and directories are referenced by absolute or relative URI
  - Relative URI (uses default file system)
  - myfile.json
  - Absolute URI

Creating DataFrame from data in memory

- You can also create DataFrames from a collection of in-memory data
  - Useful for testing and integrations

```
val mydata = List(("Josiah","Bartlett"),
                  ("Harry","Potter"))

val myDF = spark.createDataFrame(mydata)
myDF.show
+------+--------+
|    _1|      _2|
+------+--------+
|Josiah|Bartlett|
| Harry|  Potter|
+------+--------+
```

Language: *Scala*

Saving Dataframes - DataFrameWriter

- The DataFrame write function returns a DataFrameWriter
  - Saves data to a data source such as a table or set of files
  - Works similarly to DataFrameReader

- DataFrameWriter methods
- format specifies a data source type
- mode determines the behaviour if the directory or table already exists
- error, overwrite, append, or ignore (default is error)
- partitionBy stores data in partitioned directories in the form column=value (as with Hive/Impala partitioning)

Saving Dataframes – DataFrameWriter

- option specifies properties for the target data source
- save saves the data as files in the specified directory
- Or use json, csv, parquet, and so on
- saveAsTable saves the data to a Hive metastore table
- Uses default table location (/user/hive/warehouse)
- Set path option to override location

Saving Dataframes - DataFrameWriter

- When you save data from a DataFrame, you must specify a directory
  - Spark saves the data to one or more part- files in the directory

```
myDF.write.json("mydata")
```

DataFrame Schemas

- Every DataFrame has an associated schema
  - Defines the names and types of columns
  - Immutable and defined when the DataFrame is created
- When creating a new DataFrame from a data source, the schema can be
  - Automatically inferred from the data source
  - Specified programmatically
- When a DataFrame is created by a transformation, Spark calculates the new schema based on the query

```
myDF.printSchema()
root
 |-- lastName: string (nullable = true)
 |-- firstName: string (nullable = true)
 |-- age: integer (nullable = true)
```

Inferred Schemas

- Spark can infer schemas from structured data, such as
  - Parquet files
  - schema is embedded in the file
  - Hive tables
  - schema is defined in the Hive metastore
  - Parent DataFrames
- Spark can also attempt to infer a schema from semi-structured data sources
  - For example, JSON and CSV

Inferred Schemas from a CSV file with no header

```
02134,Hopper,Grace,52
94020,Turing,Alan,32
94020,Lovelace,Ada,28
87501,Babbage,Charles,49
02134,Wirth,Niklaus,48
```

```
spark.read.option("inferSchema","true").csv("people.csv"). \
  printSchema()
root
 |-- _c0: integer (nullable = true)
 |-- _c1: string (nullable = true)
 |-- _c2: string (nullable = true)
 |-- _c3: integer (nullable = true)
```

Inferred Schemas from a CSV file with header

```
pcode,lastName,firstName,age
02134,Hopper,Grace,52
94020,Turing,Alan,32
94020,Lovelace,Ada,28
87501,Babbage,Charles,49
02134,Wirth,Niklaus,48
```

```
spark.read.option("inferSchema","true"). \
  option("header","true").csv("people.csv"). \
  printSchema()
root
 |-- pcode: integer (nullable = true)
 |-- lastName: string (nullable = true)
 |-- firstName: string (nullable = true)
 |-- age: integer (nullable = true)
```

Inferred Schemas vs Manual Schemas

- Drawbacks to relying on Spark's automatic schema inference
  - Inference requires an initial file scan, which may take a long time
  - The schema may not be correct for your use case
- You can define the schema manually instead
  - A schema is a StructType object containing a list of StructField objects
  - Each StructField represents a column in the schema, specifying
  - Column name
  - Column data type
  - Whether the data can be null (optional—the default is true)

Eager and lazy Evaluation

- Operations are eager when they are executed as soon as the statement is reached in the code
- Operations are lazy when the execution occurs only when the result is referenced
- Spark queries execute both lazily and eagerly
  - DataFrame schemas are determined eagerly
  - Data transformations are executed lazily
- Lazy execution is triggered when an action is called on a series of transformations

Column, Column Names and Column Expressions

- Most DataFrame transformations require you to specify a column or columns
  - select(column1, column2, …)
  - orderBy(column1, column2, …)

- For many simple queries, you can just specify the column name as a string

```
peopleDF.select("firstName","lastName")
```

- Some types of transformations use column references or column expressions instead of column name strings

Example - Column References Python

In Python, there are two equivalent ways to refer to a column

```
peopleDF = spark.read.option("header","true").csv("people.csv")

peopleDF['age']
Column<age>

peopleDF.age
Column<age>

peopleDF.select(peopleDF.age).show()
+---+
|age|
+---+
| 52|
| 32|
| 28|
```

Column Expressions

- Using column references instead of simple strings allows you to create column expressions
  - Column operations include
  - Arithmetic operators such as +, -, %, /, and *
  - Comparative and logical operators such as >, <, && and ||
  - The equality comparator is === in Scala, and == in Python
  - String functions such as contains, like, and substr
  - Data testing functions such as isNull, isNotNull, and NaN (not a number)
  - Sorting functions such as asc and desc
  - Work only when used in sort/orderBy
- For the full list of operators and functions, see the API documentation for Column

Column Expressions - Python

```
peopleDF.select("lastName", peopleDF.age * 10).show()
+--------+----------+
|lastName|(age * 10)|
+--------+----------+
|  Hopper|       520|
|  Turing|       320|
…

peopleDF.where(peopleDF.firstName.startswith("A")).show()
+-----+--------+---------+---+
|pcode|lastName|firstName|age|
+-----+--------+---------+---+
|94020|  Turing|     Alan| 32|
|94020|Lovelace|      Ada| 28|
+-----+--------+---------+---+
```

Column Alias

Use the column alias function to rename a column in a result set
– name is a synonym for alias
Example (Python): Use column name age_10 instead of (age * 10)

```
peopleDF.select("lastName",
   (peopleDF.age * 10).alias("age_10")).show()
+--------+------+
|lastName|age_10|
+--------+------+
|  Hopper|   520|
|  Turing|   320|
```

Aggregation Queries

- Aggregation queries perform a calculation on a set of values and return a single value
- To execute an aggregation on a set of grouped values, use groupBy combined with an aggregation function
- Example: How many people are in each postal code?

```
peopleDF.groupBy("pcode").count().show()
+-----+-----+
|pcode|count|
+-----+-----+
|94020|    2|
|87501|    1|
|02134|    2|
+-----+-----+
```

The groupBy Transformation

- groupBy takes one or more column names or references
  - In Scala, returns a RelationalGroupedDataset object
  - In Python, returns a GroupedData object
- Returned objects provide aggregation functions, including
  - count
  - max and min
  - mean (and its alias avg)
  - sum
  - pivot
  - agg (aggregates using additional aggregation functions)

Additional Aggregation Functions

- The functions object provides several additional aggregation functions
- Aggregate functions include
  - first/last returns the first or last items in a group
  - countDistinct returns the number of unique items in a group
  - approx_count_distinct returns an approximate counts of unique items
  - Much faster than a full count
  - stddev calculates the standard deviation for a group of values
  - var_sample/var_pop calculates the variance for a group of values
  - covar_samp/covar_pop calculates the sample and population covariance of a group of values
  - corr returns the correlation of a group of values

Joining Dataframes

- Use the join transformation to join two DataFrames
- DataFrames support several types of joins
  - inner (default)
  - outer
  - left_outer
  - right_outer
  - leftsemi

- The crossJoin transformation joins every element of one DataFrame with every element of the other

Inner Join

people-no-pcode.csv

```
pcode,lastName,firstName,age
02134,Hopper,Grace,52
,Turing,Alan,32
94020,Lovelace,Ada,28
87501,Babbage,Charles,49
02134,Wirth,Niklaus,48
```

pcodes.csv

```
pcode,city,state
02134,Boston,MA
94020,Palo Alto,CA
87501,Santa Fe,NM
60645,Chicago,IL
```

```
val peopleDF = spark.read.option("header","true").
  csv("people-no-pcode.csv")

val pcodesDF = spark.read.
  option("header","true").csv("pcodes.csv")
```

Inner Join

```
peopleDF.join(pcodesDF, "pcode").show()
+-----+--------+---------+---+---------+-----+
|pcode|lastName|firstName|age|     city|state|
+-----+--------+---------+---+---------+-----+
|02134|  Hopper|    Grace| 52|   Boston|   MA|
|94020|Lovelace|      Ada| 28|Palo Alto|   CA|
|87501| Babbage|  Charles| 49| Santa Fe|   NM|
|02134|   Wirth|  Niklaus| 48|   Boston|   MA|
+-----+--------+---------+---+---------+-----+
…
```

Left Outer join

```
peopleDF.join(pcodesDF, "pcode", "left_outer").show()
+-----+--------+---------+---+---------+-----+
|pcode|lastName|firstName|age|     city|state|
+-----+--------+---------+---+---------+-----+
|02134|  Hopper|    Grace| 52|   Boston|   MA|
| null|  Turing|     Alan| 32|     null| null|
|94020|Lovelace|      Ada| 28|Palo Alto|   CA|
|87501| Babbage|  Charles| 49| Santa Fe|   NM|
|02134|   Wirth|  Niklaus| 48|   Boston|   MA|
+-----+--------+---------+---+---------+-----+
```

Joining Columns with Different names

people-no-pcode.csv

```
pcode,lastName,firstName,age
02134,Hopper,Grace,52
,Turing,Alan,32
94020,Lovelace,Ada,28
87501,Babbage,Charles,49
02134,Wirth,Niklaus,48
```

zcodes.csv

```
zip,city,state
02134,Boston,MA
94020,Palo Alto,CA
87501,Santa Fe,NM
60645,Chicago,IL
```

Joining Columns with Different names

Use column expressions when the names of the join columns are different

```
peopleDF.join(zcodesDF, $"pcode" === $"zip").show
+-----+--------+---------+---+-----+---------+-----+
|pcode|lastName|firstName|age|  zip|     city|state|
+-----+--------+---------+---+-----+---------+-----+
|02134|  Hopper|    Grace| 52|02134|   Boston|   MA|
|94020|Lovelace|      Ada| 28|94020|Palo Alto|   CA|
|87501| Babbage|  Charles| 49|87501| Santa Fe|   NM|
|02134|   Wirth|  Niklaus| 48|02134|   Boston|   MA|
+-----+--------+---------+---+-----+---------+-----+
```

Joining Columns with Different names

Use column expressions when the names of the join columns are different

```
peopleDF.join(zcodesDF, $"pcode" === $"zip").show
+-----+--------+---------+---+-----+---------+-----+
|pcode|lastName|firstName|age|  zip|     city|state|
+-----+--------+---------+---+-----+---------+-----+
|02134|  Hopper|    Grace| 52|02134|   Boston|   MA|
|94020|Lovelace|      Ada| 28|94020|Palo Alto|   CA|
|87501| Babbage|  Charles| 49|87501| Santa Fe|   NM|
|02134|   Wirth|  Niklaus| 48|02134|   Boston|   MA|
+-----+--------+---------+---+-----+---------+-----+
```

DataFrame Broadcast Join

- Spark broadcast joins are perfect for joining a large DataFrame with a small DataFrame.
- Broadcast joins cannot be used when joining two large DataFrames.
- Broadcast joins are easier to run on a cluster. Spark can “broadcast” a small DataFrame by sending all the data in that small DataFrame to all nodes in the cluster. After the small DataFrame is broadcasted, Spark can perform a join without shuffling any of the data in the large DataFrame.

DataFrame Broadcast Join

```
peopleDF.join(
  broadcast(citiesDF),
  peopleDF("city") <=> citiesDF("city")
).show()
```

DataFrame Schema Definition

```
from pyspark.sql import SparkSession

addressSchema = [StructField("Pin", IntegerType(), True),
                 StructField("city", StringType(), True),
                 StructField("street", StringType(), True)]
schemaFields = [StructField("_id", StringType(), True),
                StructField("first_name", StringType(), True),
                StructField("last_name", StringType(), True),
                StructField("address", StructType(addressSchema), True),
                StructField("interests", ArrayType(StringType()), True)]
personSchema = StructType(schemaFields)

df = spark_session.loadFromMapRDB("/tmp/user_profiles", personSchema)
```

DataFrame Schema Definition

```
df.printSchema()
 ---------------------------------
 root
 |-- _id: String (nullable = true)
 |-- first_name: String (nullable = true)
 |-- last_name: String (nullable = true)
 |-- address: Struct (nullable = true)
 |    |-- Pin: integer (nullable = true)
 |    |-- city: string (nullable = true)
 |    |-- street: string (nullable = true)
 |-- interests: array (nullable = true)
 |    |-- element: string (containsNull = true)
```

DataFrame Schema Definition

```
df.printSchema()
 ---------------------------------
 root
 |-- _id: String (nullable = true)
 |-- first_name: String (nullable = true)
 |-- last_name: String (nullable = true)
 |-- address: Struct (nullable = true)
 |    |-- Pin: integer (nullable = true)
 |    |-- city: string (nullable = true)
 |    |-- street: string (nullable = true)
 |-- interests: array (nullable = true)
 |    |-- element: string (containsNull = true)
```

DataFrame Repartition and Coalesce

- The dataset in Spark DataFrames and RDDs are segregated into smaller datasets called partitions. By default, Spark uses cluster's configuration (no. of nodes, memory etc) to decide the number of partitions.

- The partitions are distributed amongst the nodes of the cluster with redundancy. Partitions help with distributed computing and parallel processing.

DataFrame Repartition and Coalesce

- Using rdd.getNumPartitions(), we can get the number of partitions assigned.
  df.rdd.getNumPartitions()

- We can repartition and coalesce the partitions based upon the spark

DataFrame Repartition and Coalesce

- Repartition changes the number of partitions and balances the data across them
- Repartition can be used to increase or decrease the number of partition. Repartition always shuffles all the data over the network.
- RDD's repartition method takes an integer value – *numPartitions* as argument. Repartition method creates *numPartitions* number of partitions and balances the data across them.

  df.repartition(numPartitions:Int)
  df.repartition(5)

DataFrame Repartition and Coalesce

- Coalesce method takes in an integer value – *numPartitions* and returns a new RDD with *numPartitions* number of partitions.

- Coalesce can only create an RDD with fewer number of partitions. Coalesce minimizes the amount of data being shuffled.

- Coalesce doesn't do anything when the value of *numPartitions* is larger than the number of partitions.

```
Df.coalesce(2)
```

Contents


Cognixia®
THANK YOU
45%