Lecture 06

Spark
RDDs, Data Frames, SQL

Zoran B. Djordjević

Closure

- One of the harder things about Spark is understanding the scope and life cycle of variables and methods when executing code across a cluster.
- RDD operations that modify variables outside of their scope can be a frequent source of confusion. In the example below we'll look at code that uses `foreach()` to increment a counter, but similar issues can occur for other operations as well.
- The naive RDD element sum behaves completely differently depending on whether execution is happening within the same JVM or several.
- A common example of this is when running Spark in local mode (`--master = local[n]`) versus deploying a Spark application to a cluster (e.g. via spark-submit to YARN):

```
counter = 0
rdd = sc.parallelize(data)
# Wrong: Don't do this!!
rdd.foreach(lambda x: counter += x)
print("Counter value: " + counter)
```

Local vs. Cluster Mode

- The primary challenge is that the behavior of the above code is undefined. In local mode with a single JVM, the above code will sum the values within the RDD and store it in counter. This is because both the RDD and the variable counter are in the same memory space on the driver node.
- However, in cluster mode the above may not work as intended. To execute jobs, Spark breaks up the processing of RDD operations into tasks - each of which is operated on by an executor. Prior to execution, Spark computes the closure. The closure is the set of variables and methods which must be visible for the executor to perform its computations on the RDD (in this case `foreach()`). This closure is serialized and sent to each executor. In local mode, there is only the one executors so everything shares the same closure. In other modes however, this is not the case and the executors running on separate worker nodes each have their own copy of the closure.
- What is happening here is that the variables within the closure sent to each executor are now copies and thus, when counter is referenced within the `foreach()` function, it's no longer the counter on the driver node. The counter in the memory of the driver node is no longer visible to the executors! The executors only see the copy from the serialized closure. Thus, the final value of counter will still be zero since all operations on counter were referencing the value within the serialized closure.
- To ensure well-defined behavior in these sorts of scenarios one should use an Accumulator. Accumulators in Spark are used specifically to provide a mechanism for safely updating a variable when execution is split up across worker nodes in a cluster.

Shared Variables

- Normally, when a function passed to a Spark operation (such as `map()` or `reduce()`) is executed on a remote cluster node, it works on separate copies of all the variables used in the function.
- These variables are copied to each machine, and no updates to the variables on the remote machine are propagated back to the driver program.
- Supporting general, read-write shared variables across tasks would be inefficient.
- Spark does provide two limited types of shared variables for two common usage patterns: broadcast variables and accumulators.

Accumulators

- Accumulators are variables that are only “added” to through an associative operation and can therefore be efficiently supported in parallel. They can be used to implement counters (as in MapReduce) or sums. Spark natively supports accumulators of numeric types, and programmers can add support for new types.
- An accumulator is created from an initial value v by calling SparkContext.accumulator(v). Tasks running on the cluster can then add to it using the add method or the += operator (in Scala and Python). However, they cannot read its value. Only the driver program can read the accumulator's value, using its value method.
- The code below shows an accumulator being used to add up the elements of an array:

```
>>> accum = sc.accumulator(0)
Accumulator<id=0, value=0>
>>> sc.parallelize([1, 2, 3, 4]).foreach(lambda x:
accum.add(x))
>>> accum.value
```

Broadcast Variable

- Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks.
- They can be used, for example, to give every node a copy of a large input dataset in an efficient manner. Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce transfer costs.
- Spark actions are executed through stages, separated by distributed "shuffle" operations. Spark automatically broadcasts the common data needed by tasks within each stage. The data is cached in serialized form and deserialized before running each task. 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.
- Broadcast variables are created from a variable `v` by calling `sc.broadcast(v)`. The broadcast variable is a wrapper around v, and its value can be accessed by calling the `value` method.

```
>>> broadcastVar = sc.broadcast([1, 2, 3])
>>> broadcastVar.value
[1, 2, 3]
```

- Broadcast variable should be used instead of the value `v` in any functions run on the cluster so that v is not shipped to the nodes more than once. The object `v` should not be modified after it is broadcast in order to ensure that all nodes get the same value of the broadcast variable (e.g. if the variable is shipped to a new node later).

Programming with RDDs

- Spark RDD is an immutable distributed collection of objects.
- Each RDD is split into multiple *partitions*, which may be computed on different nodes of the cluster.
- RDDs can contain any type of Python, Java, or Scala objects, including user defined classes.
- Users create RDDs in two ways: by loading an external dataset, or by distributing a collection of objects (e.g., a list or set) in their driver program.
- We have already seen loading of a text file as an RDD of strings using `SparkContext.textFile()`.
- Once created, RDDs offer two types of operations: *transformations* and *actions*.

Partitions

- Parallelized collections are created by calling `SparkContext`'s `parallelize()` method on an existing iterable or collection in your driver program.
- The elements of the collection are copied to form a distributed dataset that can be operated on in parallel.
- For example, here is how to create a parallelized collection holding the numbers 1 to 5:

```
data = [1, 2, 3, 4, 5]
distData = sc.parallelize(data)
```

- Once created, the distributed dataset (distData) can be operated on in parallel. For example, we can call `distData.reduce(lambda a, b: a + b)` to add up the elements of the list.
- One important parameter for parallel collections is the number of partitions to cut the dataset into. Spark will run one task for each partition of the cluster.
- Typically you want 2-4 partitions for each CPU in your cluster. Normally, Spark tries to set the number of partitions automatically based on your cluster. However, you can also set it manually by passing it as a second parameter to parallelize (e.g. `sc.parallelize(data, 10)`).
- By default, Spark creates one partition for each block of the file (blocks being 64MB by default in HDFS). You can also for a higher number of partitions by passing a larger value. You cannot have fewer partitions than blocks.

Creating RDDs

- We already know how to create RDDs by loading data from external files.
- Another way to create RDDs is to take an existing collection in your program and pass it to SparkContext's `parallelize()` method. Like:
- *`parallelize()` method in Python*

```
lines = sc.parallelize(["Heaven", "Earth"])
numbers = sc.parallize(range(1,6))
print(numbers)
ParallelCollectionRDD[9] at parallelize at PythonRDD.scala:391
>>> numbers.collect()
[1, 2, 3, 4, 5]
```

- *`parallelize()` method in Scala*

```
val lines = sc.parallelize(List ("Heaven", "Earth"))
```

- *`parallelize()` method in Java*

```
JavaRDD<String> lines = sc.parallelize(Arrays.asList("Heaven", "Earth"));
```

Transformations and Actions

- *Transformations* construct a new RDD from a previous one. For example, one common transformation is filtering data that matches a predicate.
- In our text file example, we used a transformation (`filter()`) to create a new RDD holding just the strings that contain the word *Heaven* . Transformations always return an RDD.
- *Actions*, on the other hand, compute a result based on an RDD, and either return it to the driver program or save it to an external storage system (e.g., HDFS).
- One example of an action we called earlier is method `first()`, which returns the first element in an RDD. Actions could return other types and not only RDD-s.

Lazy Compute

- Spark computes RDDs only in a *lazy* fashion—that is, the first time they are used in an action.
- If Spark were to load and store all the lines in the file as soon as we define an RDD, it would waste a lot of storage space, given that we might filter out many lines. Instead, once Spark sees the whole chain of transformations, it can compute just the data needed for its result.
- For example, for the `first()` action, Spark scans the file only until it finds the first matching line; it doesn't even read the whole file.
- Finally, Spark's RDDs are by default recomputed each time you run an action on them.

Transformations

- Suppose that we have a logfile, *log.txt*, with a number of messages, and we want to select only the error messages. We can use the `filter()` transformation.
- *`filter()` transformation in Python*

```
inputRDD = sc.textFile("log.txt")
errorsRDD = inputRDD.filter(lambda x: "error" in x)
```

- *`filter()` transformation in Scala*

```
val inputRDD = sc.textFile("log.txt")
val errorsRDD = inputRDD.filter(line =>
line.contains("error"))
```

- *`filter()` transformation in Java*

```
JavaRDD<String> inputRDD = sc.textFile("log.txt");
JavaRDD<String> errorsRDD = inputRDD.filter(
   new Function<String, Boolean>() {
      public Boolean call(String x) { return x.contains("error"); }
   }
});
```

- `filter()` operation does not mutate the existing `inputRDD`. Instead, it returns a pointer to an entirely new RDD. `inputRDD` can still be reused later in the program

RDD Persistence

- One of the most important capabilities in Spark is *persisting* (or *caching*) a dataset in memory across operations. When you persist an RDD, each node stores any partitions of it that it computes in memory and reuses them in other actions on that dataset (or datasets derived from it). This allows future actions to be much faster (often by more than 10x).
- Caching is a key tool for iterative algorithms and fast interactive use. If you would like to reuse an RDD in multiple actions, you can ask Spark to *persist* it using `RDD.persist()` or `RDD.cache`.
- Method `cache()` works only for memory storage of moderately small objects.
- Method `persist()` can ask Spark to persist our data in a number of different places. Persisting RDDs on disk instead of memory is also possible.
- After computing an RDD the first time, Spark will store the RDD contents in memory (partitioned across the machines in your cluster), and reuse them in future actions. In practice, you will often use `persist()` to load a subset of your data into memory and query it repeatedly.

```
>>> pythonLines.persist(StorageLevel.MEMORY_ONLY)
>>> pythonLines.count()
2
>>> pythonLines.first()
```

Persisting RDDs

Storage Level	Meaning
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER	Store RDD as serialized Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as the levels above, but replicate each partition on two cluster nodes.
OFF_HEAP (experimental)	Store RDD in serialized format in Tachyon. Compared to MEMORY_ONLY_SER, OFF_HEAP reduces garbage collection overhead and allows executors to be smaller and to share a pool of memory, making it attractive in environments with large heaps or multiple concurrent application

- *In Python, stored objects will always be serialized with the Pickle library, for any storage level.*
- Spark also automatically persists intermediate results in shuffle operations without users calling persist (e.g. `reduceByKey()`).

`union()` Transformation

- If we need to print the number of lines that contained either *error* or *warning*, we could use union() function, which is identical in all three languages. Text that follows is in Python:

```
errorsRDD = inputRDD.filter(lambda x: "error" in x)
warningsRDD = inputRDD.filter(lambda x: "warning" in x)
badLinesRDD = errorsRDD.union(warningsRDD)
```

- `union()` is a bit different than `filter()`, in that it operates on two RDDs instead of one.
- Transformations can operate on any number of input RDDs.

Actions

- At some point, we'll want to actually *do something* with our dataset. Actions are the second type of RDD operation. They are the operations that return a final value to the driver program or write data to an external storage system.
- Actions force the evaluation of the transformations required for the RDD they were called on, since they need to actually produce output.
- Continuing the log example from the previous section, we might want to print out some information about the `badLinesRDD`. To do that, we'll use two actions, `count()`, which returns the count as a number, and `take()`, which fetches a number of elements from the RDD and gives us an iterable collection.
- In the following example, we will use method `take()` to retrieve a small number of elements (sample of 10) in the RDD at the driver program. We then iterate over them locally to print out information at the driver.
- RDDs also have a `collect()` function to retrieve the entire RDD. This can be useful if your program filters RDDs down to a very small size and you'd like to deal with it locally.
- Entire dataset must fit in memory on a single machine to use `collect()` on it.

Actions, `count()`, `take()`

- *Python error count and sample display using actions*

```
print "Input had " + badLinesRDD.count() + " concerning lines"
print "Here are 10 examples:"
for line in badLinesRDD.take(10):
print line
```

- *Scala error count and sample display using actions*

```
println("Input had " + badLinesRDD.count() + " concerning lines")
println("Here are 10 examples:")
badLinesRDD.take(10).foreach(println)
```

- *Java error count and sample display using actions*

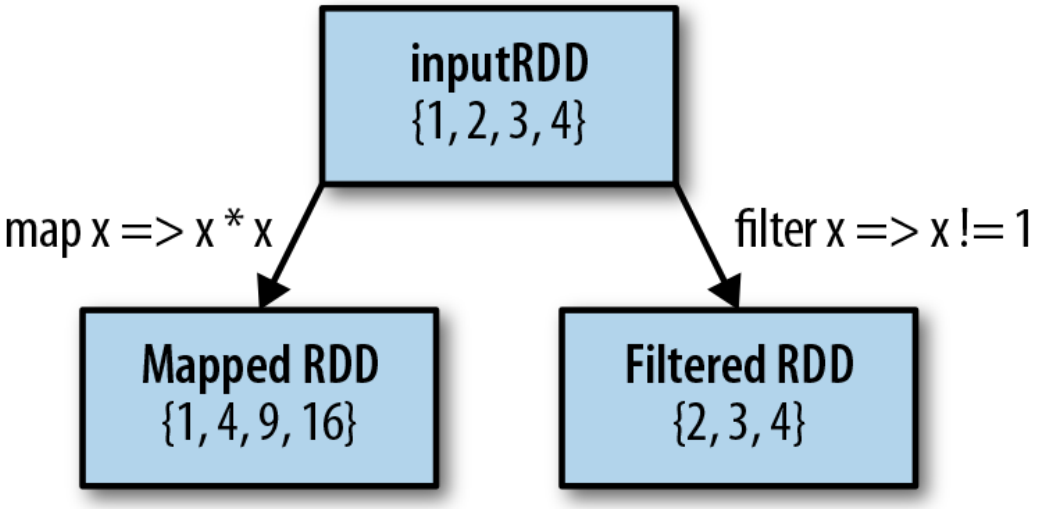
```
System.out.println("Input had " + badLinesRDD.count() + "
concerning lines")
System.out.println("Here are 10 examples:")
for (String line: badLinesRDD.take(10)) {
   System.out.println(line);
}
```

- If RDDs can't be `collect()`–ed because they are too large, it is common to write data out to a distributed storage system such as HDFS or Amazon S3.
- You can save the contents of an RDD using the `saveAsTextFile()` action, `saveAsSequenceFile()`, or any of a number of actions for various built-in formats.

Common Transformations and Actions

Element-wise transformations:

- Two most common transformations we use are `map()` and `filter()`.
- `map()` transformation takes in a function and applies it to each element in the RDD with the result of the function being the new value of each element in the resulting RDD.
- `filter()` transformation takes in a function and returns an RDD that only has elements that pass the filter() function.



- Return type of a `map()` does not have to be the same as its input type.

map() Examples

- *Python squaring the values in an RDD*

```
nums = sc.parallelize([1, 2, 3, 4])
squared = nums.map(lambda x: x * x).collect()
for num in squared:
    print "%i " % (num)
```

- *Scala squaring the values in an RDD*

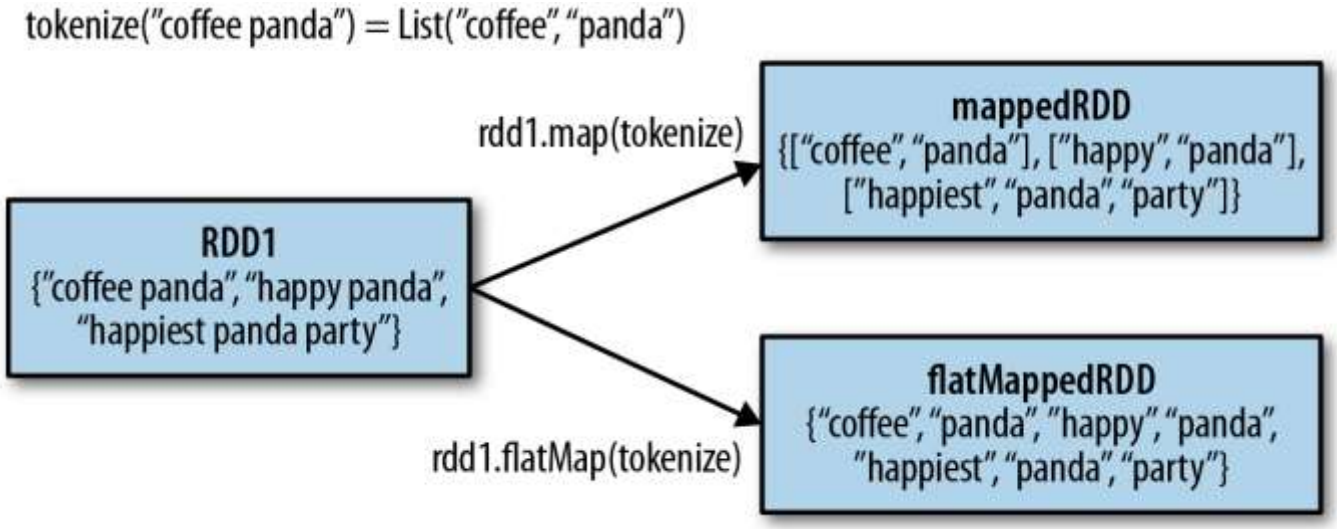
```
val input = sc.parallelize(List(1, 2, 3, 4))
val result = input.map(x => x * x)
println(result.collect().mkString(","))
```

- *Java squaring the values in an RDD*

```
JavaRDD<Integer> rdd = sc.parallelize(
    Arrays.asList(1, 2, 3, 4));
JavaRDD<Integer> result = rdd.map(
  new Function<Integer, Integer>() {
   public Integer call(Integer x) { return x*x; }
});
System.out.println(StringUtils.join(result.collect(),
","));
```

flatMap()

- Sometimes we want to produce multiple output elements for each input element. This is accomplished with operation `flatMap()`.
- As with `map()`, the function we provide to `flatMap()` is called individually for each element in the input RDD. Instead of returning a single element, `flatMap()` returns an iterator with the return values.
- `flatMap()` does not produce an RDD of iterators, we get back an RDD that consists of the elements from all of the iterators. A simple usage of `flatMap()` is splitting up an input text into words.
- The difference between `map()` and `flatMap()` is illustrated bellow.



`flatMap()` Examples

- *`flatMap()` in Python, splitting lines into words*

```
lines = sc.parallelize(["hello world", "hi"])
words = lines.flatMap(lambda line: line.split(" "))
words.first() # returns "hello"
```

- *`flatMap()` in Scala, splitting lines into multiple words*

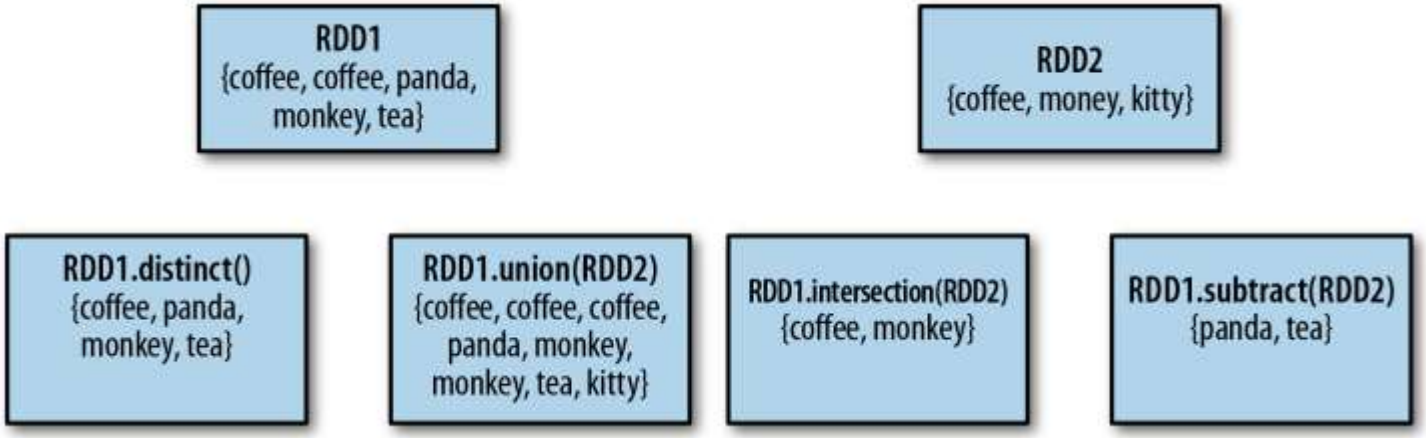
```
val lines = sc.parallelize(List("hello world", "hi"))
val words = lines.flatMap(line => line.split(" "))
words.first() // returns "hello"
```

- *`flatMap()` in Java, splitting lines into multiple words*

```
JavaRDD<String> lines = sc.parallelize(
     Arrays.asList("hello world", "hi"));
JavaRDD<String> words = lines.flatMap(
 new FlatMapFunction<String, String>() {
  public Iterable<String> call(String line) {
    return Arrays.asList(line.split(" "));
  }
});
words.first(); // returns "hello"
```

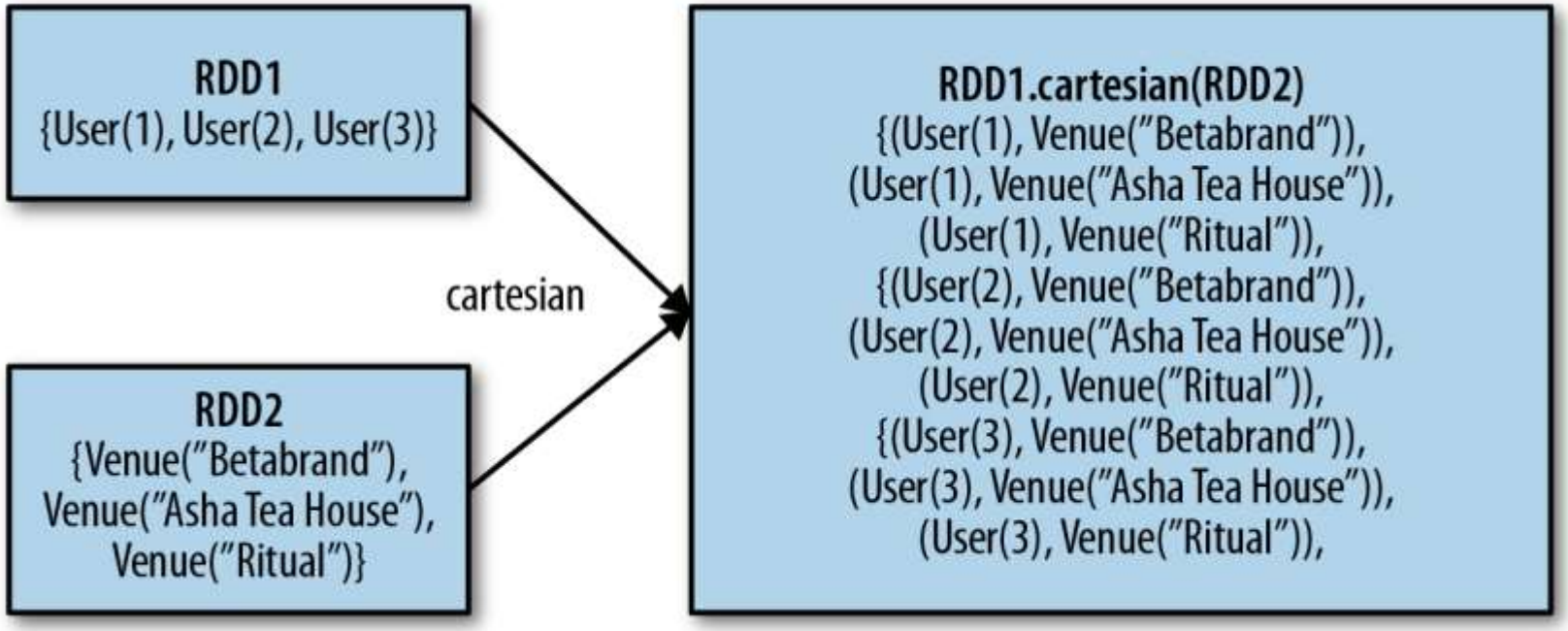
Pseudo set operations

- RDDs support many of the operations of mathematical sets, such as union and intersection, even when the RDDs themselves are not proper sets.
- All set operations require that the RDDs being operated on have elements of the same type.
- The set property most frequently missing from RDDs is the uniqueness of elements, as we often have duplicates.
- Below we illustrate four set operations: `distinct()`, `union()`, `intersection()` and `subtract()`



Cartesian Product between RDDs

- The `cartesian(other)` transformation returns all possible pairs of (a, b) where a is in the source RDD and b is in the other RDD.



Actions, `reduce()`

- The most common action on basic RDDs we will likely use is `reduce()`, which takes a function that operates on two elements of the type in your RDD and returns a new element of the same type.
- A simple example of such a function is +, which we can use to sum elements of an RDD.
- With `reduce()`, we can easily sum the elements of an RDD, count the number of elements, and perform other types of aggregations.
- *`reduce()` in Python*

```
sum = rdd.reduce(lambda x, y: x + y)
```

- *reduce() in Scala*

```
val sum = rdd.reduce((x, y) => x + y)
```

- *reduce() in Java*

```
Integer sum = rdd.reduce(
  new Function2<Integer, Integer, Integer>() {
     public Integer call(Integer x, Integer y) { return x + y; }
  });
```

- `reduce()` requires that the return type of the result be the same type as that of the elements in the RDD we are operating over.

`fold()`

- Similar to `reduce()` is `fold()`, which also takes a function with the same signature as needed for `reduce()`, but in addition takes a "zero value" to be used for the initial call on each partition.
- The zero value you provide should be the identity element for your operation; that is, applying it multiple times with your function should not change the value (e.g., 0 for +, 1 for *, or an empty list for concatenation).
- `fold()` requires that the return type of our result be the same type as that of the elements in the RDD we are operating over. This works well for operations like sum, but sometimes we want to return a different type.
- For example, when computing a running average, we need to keep track of both the count so far and the number of elements, which requires us to return a pair. We could work around this by first using `map()` where we transform every element into the element and the number 1, which is the type we want to return, so that the `reduce()` function can work on pairs.

`fold()` vs. `reduce()`

- `reduce()` applies some operation to pairs of elements until there is just one left. This implies the result must be of the same type as the collection.
- `reduce()` throws an exception for an empty collection.
- `fold()` lets you supply an initial zero value and does the same, so is defined for an empty collection.
- `fold()` also lets the initial value and thus result be of a different type, since the operation you give combines a source and result type into a result type.

  A note on performance: If your reduce function is `result = a + b`, and a and b are mutable types (e.g. a set), then it can be faster to modify a to contain a + b and return that rather than allocate a new data structure..

aggregate()

- The `aggregate()` function frees us from the constraint of having the return be the same type as the RDD we are working on.
- With `aggregate()`, like `fold()`, we supply an initial zero value of the type we want to return. We then supply a function to combine the elements from our RDD with the accumulator. Finally, we need to supply a second function to merge two accumulators, given that each node accumulates its own results locally.
- We can use `aggregate()` to compute the average of an RDD, avoiding a `map()` before the `fold()`.
- *`aggregate()` in Python*

```
sumCount = nums.aggregate((0, 0),
(lambda acc, value: (acc[0] + value, acc[1] + 1),
(lambda acc1, acc2: (acc1[0] + acc2[0], acc1[1] + acc2[1]))))
return sumCount[0] / float(sumCount[1])
```

- *`aggregate()` in Scala*

```
val result = input.aggregate((0, 0))(
(acc, value) => (acc._1 + value, acc._2 + 1),
(acc1, acc2) => (acc1._1 + acc2._1, acc1._2 + acc2._2))
val avg = result._1 / result._2.toDouble
```

aggregate() in Java

```
class AvgCount implements Serializable {
    public AvgCount(int total, int num) {
    this.total = total; this.num = num;
}
public int total; public int num;
    public double avg() { return total / (double) num;
    }
}
Function2<AvgCount, Integer, AvgCount> addAndCount =
    new Function2<AvgCount, Integer, AvgCount>() {
    public AvgCount call(AvgCount a, Integer x) {
    a.total += x; a.num += 1; return a;
    }
};
Function2<AvgCount, AvgCount, AvgCount> combine =
    new Function2<AvgCount, AvgCount, AvgCount>() {
    public AvgCount call(AvgCount a, AvgCount b) {
    a.total += b.total; a.num += b.num; return a;
    }
};
AvgCount initial = new AvgCount(0, 0);
AvgCount result = rdd.aggregate(initial, addAndCount, combine);
System.out.println(result.avg());
```

Examples, Actions

Basic actions on an RDD containing numbers {1, 2, 3, 3}

Function name	Purpose	Example	Result
collect()	Return all elements from the RDD.	rdd.collect()	{1, 2,3,3}
count()	Return all elements from the RDD.	rdd.count()	4
countByValue()	Number of times each element occurs in the RDD.	rdd.countByValue()	{(1,1),(2,1),(3,2)}
take(num)	Return num elements from the RDD.	rdd.take(2)	{1,2}
top(num)	Return the top num elements the RDD.	rdd.top(2)	{3,3}
takeOrdered(num)(ordering)	Return num elements based on provided ordering.	rdd.takeOrdered(2)(myOrdering)	{3,3}
takeSample(withReplacement,num,[seed])	Return num elements at random.	rdd.takeSample(false, 1)	nondeterministic
reduce()	Combine the elements of the RDD together in parallel (e.g., sum).	rdd.reduce((x, y) => x + y)	9
fold(zero)(func)	Same as reduce() but with the provided zero value.	rdd.fold(0)((x, y) => x + y)	9
aggregate(zeroValue)(seqOp, combOp)	Similar to reduce() but used to return a different type.	rdd.aggregate((0, 0)) ((x, y) => (x._1 + y, x._2 + 1), (x, y) => (x._1 + y._1, x._2 + y._2))	(9,4)
foreach(func)	Apply the provided function to each element of the RDD.	rdd.foreach(func)	

Working with `Key/Value` Pairs

- Spark tries to distinguish itself from MapReduce. Still, many or most calculation in Spark rely on key/value pairs data types.
- Key/value RDDs are commonly used to perform aggregations.
- Key/value RDDs expose new operations (e.g., counting up reviews for each product, grouping together data with the same key, and grouping together two different RDDs).
- Spark introduces an advanced feature, *partitioning,* that lets users control the layout of pair RDDs across nodes. Using controllable partitioning, applications can sometimes greatly reduce communication costs by ensuring that data will be accessed together on the same node.
- Spark provides special operations on RDDs containing key/value pairs. These RDDs are called *pair RDDs*. *Pair RDDs* are a useful building block in many programs, as they expose operations that allow you to act on each key in parallel or regroup data across the network.
- For example, pair RDDs have a `reduceByKey()` method that can aggregate data separately for each key, and a `join()` method that can merge two RDDs together by grouping elements with the same key.

Creating Pair RDDs

- There are a number of ways to get pair RDDs in Spark. Many import formats will directly return pair RDDs for their key/value data.
- In other cases we have a regular RDD that we want to turn into a pair RDD. We could do it this with a `map()` function that returns key/value pairs.
- To illustrate, we show code that starts with an RDD of lines of text and keys the data by the first word in each line.
- The way to build key-value RDDs differs by language.
- In Python, for the functions on keyed data to work we need to return an RDD composed of tuples.
- *Creating a pair RDD using the first word as the key in Python*

```
pairs = lines.map(lambda x: (x.split(" ")[0], x))
```

- In Scala, for the functions on keyed data to be available, we also need to return tuples. An implicit conversion on RDDs of tuples exists to provide the additional key/value functions.
- *Creating a pair RDD using the first word as the key in Scala*

```
val pairs = lines.map(x => (x.split(" ")(0), x))
```

Python Tuple

- A tuple is a sequence of immutable Python objects. Tuples are sequences, like lists.
- Tuples use parentheses, whereas lists use square brackets.
- Creating a tuple is as simple as putting different comma-separated values. For example

```
tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5 );
tup3 = "a", "b", "c", "d";
tup1 = ();  # this was an empty tuple
```

- A tuple containing a single value includes a comma, `tup1 = (50,)`
- Tuple indices start at 0, and they can be sliced, concatenated, and so on.
- To access values in tuple, use the square brackets. For example

```
tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5, 6, 7 );
print "tup1[0]: ", tup1[0]
print "tup2[1:5]: ", tup2[1:5]
```

- Tuples are immutable which means you cannot update or change the values of tuple elements.
- You are able to take portions of existing tuples to create new tuples.
- # Following action is not valid for tuples

```
# tup1[0] = 100;
```

- Removing individual tuple elements is not possible.
- To explicitly remove an entire tuple, just use the del statement. For example:

```
del tup;
```

Creating Pair RDD in java

- Java does not have a built-in tuple type, so Spark's Java API forces users to create tuples using the `scala.Tuple2` class. This class is very simple: Java users can construct a new tuple by writing new `Tuple2(elem1, elem2)` and can then access its elements with the `._1()` and `._2()` methods.
- Java users also need to call special versions of Spark's functions when creating pair RDDs. For instance, the `mapToPair()` function is used in place of the basic `map()` function.
- *Creating a pair RDD using the first word as the key in Java*

```
PairFunction<String, String, String> keyData =
   new PairFunction<String, String, String>() {
     public Tuple2<String, String> call(String x) {
        return new Tuple2(x.split(" ")[0], x);
     }
   };
JavaPairRDD<String, String> pairs =
lines.mapToPair(keyData);
```

Transformations on Pair RDDs

- Since pair RDDs contain tuples, we need to pass functions that operate on tuples rather than on individual elements.
- *Transformations on one pair RDD (example: {(1, 2), (3, 4), (3, 6)})*

Function name	Purpose	Example	Result
reduceByKey(func)	Compbine values with the same key	rdd.reduceByKey(x,y) => x + y	{(1,2),(3,10)}
groupByKey()	Group values with the same key	rdd.groupByKey()	{(1,[2]),(3,[4,6])}
mapValues(func)	Apply a function to each value of a pair RDD without changing the key.	rdd.mapValues(x => x+1)	{(1, 3), (3, 5), (3,7)}
flatMapValues(func)	Apply a function that returns an iterator to each value of a pair RDD, and for each element returned, produce a key/value entry with the old key. Often used for tokenization.	rdd.flatMapValues(x => (x to 5)	{(1,2), (1,3), (1,4), (1, 5), (3, 4), (3,5)}
keys()	Return an RDD of just the keys.	rdd.keys()	{1,3,5}
values()	Return an RDD just of values	rdd.values()	{2,4,6}
sortByKey()	Return and RD sorted by the key	Rdd.sortByKey()	{(1,2),(3,4)(3,6)}

Transformations on two pair RDD

- Two pair *RDDs (rdd = {(1, 2), (3, 4), (3, 6)} other = {(3, 9)})*

Function name	Purpose	Example	Result
substractByKey	Remove elements with a key present in the other RDD	Rdd.substractByKey(other)	{(1,2)}
join	Perform an inner join between two RDDs	Rdd.join(other)	{(3,(4,9)),(3,(6,9))}
rightOuterJoin	Perform a join between two RDDs where the key must be present in the first RDD	Rdd.rightOuterJoin(other)	{(3,(Some(4),9)), (3,(Some(6),9))}
leftOuterJoin	Perform a join between two RDDs where the key must be present in the other RDD.	rdd.leftOuterJoin(other)	{(1,(2,None)), (3, (4,Some(9))), (3, (6,Some(9)))}
cogroup	Group data from both RDDs sharing the same key.	rdd.cogroup(other)	{(1,([2],[])), (3, ([4, 6],[9]))}

- Sometimes working with pairs can be awkward if we want to access only the value part of our pair RDD.
- Since this is a common pattern, Spark provides the mapValues(func) function, which is the same as map{case (x, y): (x, func(y))}.

Pair RDDs are RDDs

- Pair RDDs are also still RDDs (of Tuple2 objects in Java/Scala or of Python tuples), and thus support the same functions as RDDs. For instance, we can take our pair RDD from the previous section and filter out lines longer than 20 characters:
- *Simple filter on second element in Python*

```
result = pairs.filter(lambda keyValue: len(keyValue[1]) < 20)
```

- *Simple filter on second element in Scala*

```
pairs.filter{case (key, value) => value.length < 20}
```

- *Simple filter on second element in Java*

```
Function<Tuple2<String, String>, Boolean> longWordFilter =
  new Function<Tuple2<String, String>, Boolean>() {
    public Boolean call(Tuple2<String, String> keyValue) {
      return (keyValue._2().length() < 20);
    }
  };
JavaPairRDD<String, String> result= pairs.filter(longWordFilter);
```

Aggregations

- When datasets are described in terms of key/value pairs, it is common to want to aggregate statistics across all elements with the same key. We have looked at the `fold()`, `combine()`, and `reduce()` actions on basic RDDs, and similar per-key transformations exist on pair RDDs.
- Spark has a similar set of operations that combines values that have the same key. These operations return RDDs and thus are transformations rather than actions.
- `reduceByKey()` is quite similar to `reduce()`; both take a function and use it to combine values. `reduceByKey()` runs several parallel reduce operations, one for each key in the dataset, where each operation combines values that have the same key. Because datasets can have very large numbers of keys, `reduceByKey()` is not implemented as an action that returns a value to the user program. Instead, it returns a new RDD consisting of each key and the reduced value for that key.
- `foldByKey()` is quite similar to `fold()`; both use a zero value of the same type of the data in out RDD and combination function.

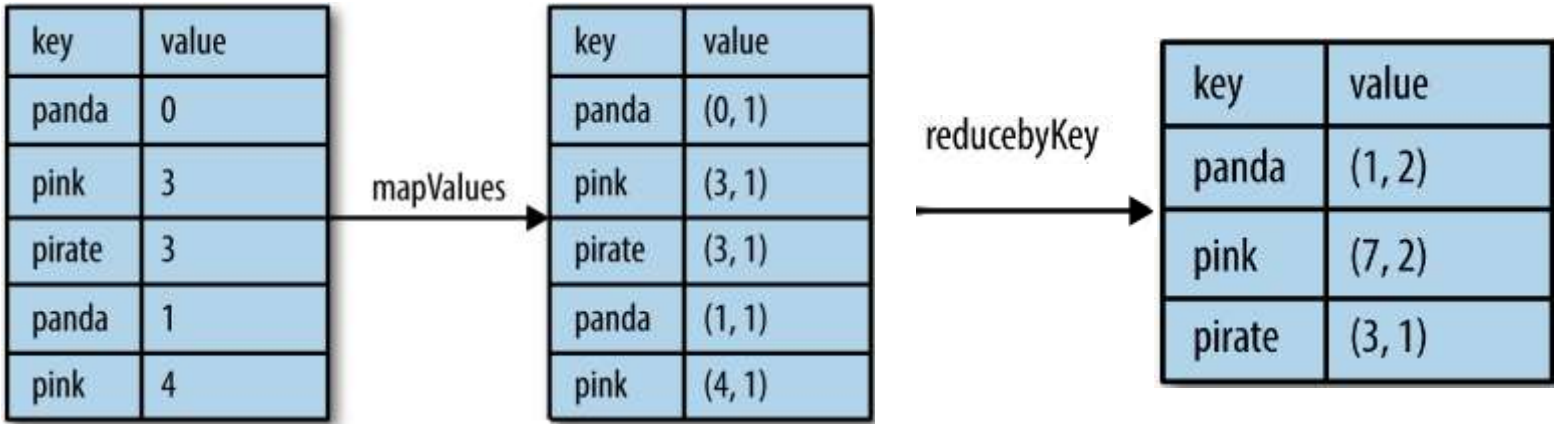
Aggregation Examples

- The following examples demonstrate, we can use `reduceByKey()` along with `mapValues()` to compute the per-key average in a very similar manner to how `fold()` and `map()` can be used to compute the entire RDD average.
- *Per-key average with `reduceByKey()` and `mapValues()` in Python*

```
rdd.mapValues(lambda x: (x, 1)).reduceByKey(
     lambda x, y: (x[0] + y[0], x[1] + y[1]))
```

- *Per-key average with `reduceByKey()` and `mapValues()` in Scala*

```
rdd.mapValues(x => (x, 1)).reduceByKey(
     (x, y) => (x._1 + y._1, x._2 + y._2))
```



Aggregation Example, Word count

- We will use `flatMap()` so that we can produce a pair RDD of words and the number 1 and then sum together all of the words using `reduceByKey()`
- *Word count in Python*

```
rdd = sc.textFile("s3://...")
words = rdd.flatMap(lambda x: x.split(" "))
result = words.map(lambda x: (x, 1)).reduceByKey(lambda x, y: x + y)
```

- *Word count in Scala*

```
val input = sc.textFile("s3://...")
val words = input.flatMap(x => x.split(" "))
val result = words.map(x => (x, 1)).reduceByKey((x, y) => x + y)
```

- *Word count in Java*

```
JavaRDD<String> input = sc.textFile("s3://...")
JavaRDD<String> words = rdd.flatMap(new FlatMapFunction<String, String>() {
   public Iterable<String> call(String x) { return Arrays.asList(x.split(" ")); }
});
JavaPairRDD<String, Integer> result = words.mapToPair(
   new PairFunction<String, String, Integer>() {
     public Tuple2<String, Integer> call(String x) { return new Tuple2(x, 1); }
}).reduceByKey(
     new Function2<Integer, Integer, Integer>() {
     public Integer call(Integer a, Integer b) { return a + b; }
});
```

Spark SQL

- Spark SQL is a Spark module for structured data processing. Unlike the basic Spark RDD API, the interfaces provided by Spark SQL provide Spark with more information about the structure of both the data and the computation being performed.
- Spark SQL uses this extra information to perform extra optimizations. There are several ways to interact with Spark SQL including SQL, the DataFrames API and the Datasets API.
- One use of Spark SQL is to execute SQL queries written using either a basic SQL syntax or HiveQL.
- Spark SQL can also be used to read data from an existing Hive installation. When running SQL from within another programming language the results will be returned as a DataFrame.
- You can also interact with the SQL interface using the command-line or over JDBC/ODBC.
- SparkSQL could read and write data from Parquet files, JSON files, Hive, Cassandra

DataFrame

- A DataFrame is a distributed collection of data organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations under the hood.
- DataFrames can be constructed from a wide array of sources such as: structured data files, tables in Hive, external databases, or existing RDDs.
- The DataFrame API is available in Scala, Java, Python, and R.
- DataFrame can process the data in the size of Kilobytes to Petabytes on a single node cluster or on a large cluster.
- Supports different data formats (Avro, csv, elastic search, and Cassandra) and storage systems (HDFS, HIVE tables, mysql, etc).
- State of art optimization and code generation through the Spark SQL Catalyst optimizer (tree transformation framework).
- Can be easily integrated with all Big Data tools and frameworks via Spark-Core.

Datasets

- A Dataset is a new interface added in Spark 1.6 that tries to provide the benefits of RDDs (strong typing, ability to use powerful lambda functions) with the benefits of Spark SQL's optimized execution engine.
- A Dataset can be constructed from JVM objects and then manipulated using functional transformations (map, flatMap, filter, etc.).
- The unified Dataset API can be used both in Scala and Java. Full Python support will be added in a future release.

SQL Context

- `SQLContext` is a class and is used for initializing the functionalities of Spark SQL.
- `SparkContext` class object (sc) is required for initializing `SQLContext` class object.
- The entry point into all relational functionality in Spark `SQLContext` class, or one of its decedents, like `HiveContxt`

Json Sample File

- We created a small JSON file
  /home/cloudera/resources/employee.json
  with the following content:

```
{"id" : "3201", "name" : "Mary", "age" : "35"}
{"id" : "3202", "name" : "John", "age" : "38"}
{"id" : "3203", "name" : "Bill", "age" : "39"}
{"id" : "3204", "name" : "Mark", "age" : "33"}
{"id" : "3205", "name" : "Ann",  "age" : "33"}
```

- We also have `people.json` and `people.txt` files.
- Please note that JSON files we will ingest into Spark must have a proper JSON on every line.

```
[cloudera@quickstart resources]$ cat people.json
{"name":"Michael"}
{"name":"Andy", "age":30}
{"name":"Justin", "age":19}

[cloudera@quickstart resources]$ cat people.txt
Michael, 29
Andy, 30
Justin, 19
```

SQL Context

- To create basic `SQLContext`, all you need is a `SparkContext`.

```
>>> from pyspark.sql import SQLContext # need not do this in pyspark
>>> sqlContext = SQLContext(sc)
>>> df = sqlContext.read.json("file:///home/cloudera/resources/employee.json")
>>> df.show(3)
+---+----+----+
|age|  id|name|
+---+----+----+
| 35|3201|Mary|
| 38|3202|John|
| 39|3203|Bill|
+---+----+----+
only showing top 3 rows
```

- If you are curious

```
>>> df.collect()
[Row(age=u'35', id=u'3201', name=u'Mary'), Row(age=u'38', id=u'3202', name=u'John'), Row(age=u'39', id=u'3203', name=u'Bill'), Row(age=u'33', id=u'3204', name=u'Mark'), Row(age=u'33', id=u'3205', name=u'Ann')]
```

printSchema(), select(), filter()

- To see the schema of a DataFrame we use method `printSchema()`

```
>>> df.printSchema()
root
 |-- age: string (nullable = true)
 |-- id: string (nullable = true)
 |-- name: string (nullable = true)
```

- To select and display values in a column, use `select().show()`

```
>>> df.select("name").show(2)
+----+
|name|
+----+
|Mary|
|John|
+----+
only showing top 2 rows
```

- We can restrict the output to older employees using `filter()`.

```
>>> df.filter(df.age > 38).show()
+---+----+----+
|age|  id|name|
+---+----+----+
| 39|3203|Bill|
+---+----+----+
```

groupBy(), Column Indexing

- For counting the number of employees who are of the same age we would use `groupBy()` method.

```
>>> df.groupBy("age").count().show()
+---+-----+
|age|count|
+---+-----+
| 33|    2|
| 35|    1|
| 38|    1|
| 39|    1|
+---+-----+
```

- Select everybody, but increment the age by 1. Note how we index data frame columns with `['column-name']` notation. `df[0]` will work also.

```
>>> df.select(df['name'], df['age'] + 1).show()
+----+---------+
|name|(age + 1)|
+----+---------+
|Mary|     36.0|
|John|     39.0|
|Bill|     40.0|
|Mark|     34.0|
| Ann|     34.0|
+----+---------+
```

- We could have done this as well, but no Math.

```
>>> df.select('name', 'age').show(2)
+----+---+
|name|age|
+----+---+
|Mary| 35|
|John| 38|
+----+---+
```

Additional Query Examples

```
>>> df.filter(df['name'].like("M%")).show()
+---+----+----+
|age|  id|name|
+---+----+----+
| 35|3201|Mary|
| 33|3204|Mark|
+---+----+----+
>>> df.filter(df['name'].like("M%")).count()
2
```

Connect to local MySql DB on QuickStart VM

```
$ mysql --user=retail_dba -p
Enter password: cloudera
Mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| retail_db          |
+--------------------+
mysql> use retail_db;
Database changed
mysql> show tables;
+---------------------+
| Tables_in_retail_db |
+---------------------+
| categories          |
| customers           |
| departments         |
| order_items         |
| orders              |
| products            |
+---------------------+
```

Connect Spark to MySQL

- Download mysql java connector from:

http://dev.mysql.com/downloads/connector/j/

- You will have to create Oracle account if you do not have one.
- Unzip the ZIP file and place file `mysql-connector-java-5.1.38-bin.jar` where convenient.
- Next run `pyspark` with `–driver-class-path` option:

```
$ pyspark --driver-class-path mysql-connector-java-5.1.38-bin.jar
>>> sqlContxt = SQLContext(sc)
>>> dfm =
sqlc.read.format("jdbc").option("url","jdbc:mysql://localhost/retail_db").o
ption("driver","com.mysql.jdbc.Driver").option("dbtable","customers").optio
n("user","retail_dba").option("password","cloudera").load()
>>> dfm.printSchema()
root
 |-- customer_id: integer (nullable = false)
 |-- customer_fname: string (nullable = false)
 |-- customer_lname: string (nullable = false)
 |-- customer_email: string (nullable = false)
 |-- customer_password: string (nullable = false)
 |-- customer_street: string (nullable = false)
 |-- customer_city: string (nullable = false)
 |-- customer_state: string (nullable = false)
 |-- customer_zipcode: string (nullable = false)
```

Connect Spark to MySQL

- We could transform data frame object into a temp table, so that we could run SQL queries against it:

```
>>> dfm.registerTempTable("people")
>>> sqlContext.sql("select * from people LIMIT 3")
.show(3)
+-----------+-------------+-------------+--------------+-------
|customer_id|customer_fname|customer_lname|customer_email|
+-----------+-------------+-------------+--------------+-------|
1|       Richard|     Hernandez|     XXXXXXXXX|         |
2|          Mary|       Barrett|     XXXXXXXXX|         |
3|           Ann|         Smith|     XXXXXXXXX|         +----------+---------
----+------------+------------+---------------+------------------+------------+-----------
```

- You drop temp table by command:

```
>>> sqlContext.dropTempTable("people")
```

Transfer Data from Hive

- For Spark to locate your Hive you have to do three things:

1. Start Hive metadata server

`$ hiveserver2 &` `# &` at the end means run in the background

2. Copy `hive-site.xml` from `$HIVE_HOME/conf`, which is `/etc/hive/conf` to `$SPARK_HOME/conf`, which happens to be `/etc/spike/conf`
3. Create `HiveContext`

```
>>> hivecontext = HiveContext(sc)
```

- In a stand alone Python script you would have to do the import first

```
from pyspark.sql import SQLContext, HiveContext
```

- Now that you are ready, you can query Hive table `shake(speare)` and create DataFrame `dfs` with the content of that Hive table.

```
>>> dfs = hive.sql("select * from shake")
>>> dfs.count()
29183
>>> dfs.first()
Row(freq=25578, word=u'the')
>>> dfs.printSchema()
root
 |-- freq: integer (nullable = true)
 |-- word: string (nullable = true)

>>>
```

Save Data Frame as persistent Parquet File

- We could save our Data Frame as a Parquet file

```
[cloudera@quickstart ~]$ hiveserver2 &
[1] 7405
[cloudera@quickstart ~]$ pyspark
>>> sqlContext = SQLContext(sc)
>>> df =
sqlContext.read.load("file:///home/cloudera/resources/people.json",format="json")
>>> df.count()
3
>>> df.show()
+----+-------+
| age|   name|
+----+-------+
|null|Michael|
|  30|   Andy|
|  19| Justin|
+----+-------+
>>> df.select("name","age").write.save("nameage.parquet",format="parquet")
SLF4J: Class path contains multiple SLF4J bindings.
>>> parquetFile = sqlContext.read.parquet("nameage.parquet")
>>> parquetFile.registerTempTable("parqf")
>>> somepeople = sqlContext.sql("select name from parqf")
>>> somepeople.show(1)
+-------+
|   name|
+-------+
| Justin|
+-------+
```

- Next we exit() Spark Session, and then enter it again

Enter Spark Session, Parquet File is still there

- Now we go Back into the `pyspark:`

```
[cloudera@quickstart ~]$ pyspark
>>> sqlContext = SQLContext(sc)
>> parquetFile = sqlContext.read.parquet("nameage.parquet")
[Stage 0:>
(0 + 0) / 8]SLF4J:
>>> parquetFile.registerTempTable("parqf")
>>> sp = sqlContext.sql("select * from parqf")
>>> sp.show()
+-------+----+
|   name| age|
+-------+----+
| Justin|  19|
|Michael|null|
|   Andy|  30|
+-------+----+
>>>
```

- Our data is still there in `nameage.parquet` file.

Sqoop

- RDBMS data are critical for operation of any enterprise and will remain so for long time to come. Enriching analysis of unstructured data on the Spark/Hadoop side with RDBMS data and vice versa is essential.
- One convenient tool for automating transfer of data from Relational Database Systems into the world of Hadoop and Spark is Sqoop.
- Sqoop transfers data directly into HDFS making them available for further analysis.
- There are 2 versions of Sqoop
  - Sqoop 1 is a "thick client" and is what you use in this tutorial. The command you run will directly submit the MapReduce jobs to transfer the data.
  - Sqoop 2 consists of a central server that submits the MapReduce jobs on behalf of clients, and a much lighter weight client that you use to connect to the server

Sqoop Commands

- To get the list of supported commands, type:

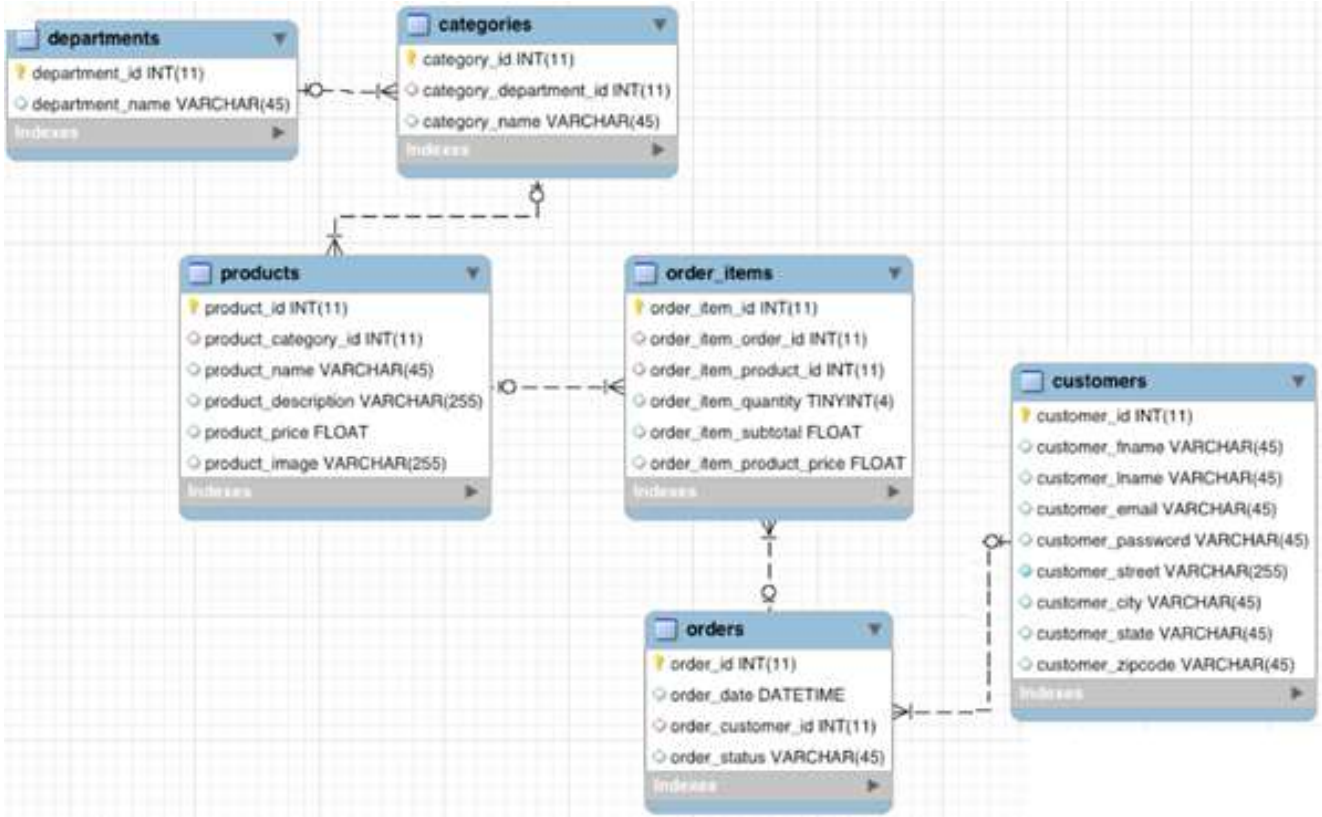
```
[cloudera@quickstart ~]$ sqoop help
Running Sqoop version: 1.4.6-cdh5.5.0
usage: sqoop COMMAND [ARGS]

Available commands:
  codegen            Generate code to interact with database records
  create-hive-table  Import a table definition into Hive
  eval               Evaluate a SQL statement and display the results
  export             Export an HDFS directory to a database table
  help               List available commands
  import             Import a table from a database to HDFS
  import-all-tables  Import tables from a database to HDFS
  import-mainframe   Import datasets from a mainframe server to HDFS
  job                Work with saved jobs
  list-databases     List available databases on a server
  list-tables        List available tables in a database
  merge              Merge results of incremental imports
  metastore          Run a standalone Sqoop metastore
  version            Display version information

See 'sqoop help COMMAND' for information on a specific command.
```

Ingesting Relational Data

- We will use tables provided in MySQL demo database `retail_db` provided with Cloudera's VM. Schema of that database is presented below:



- Various business relevant queries involving those tables even when they contain hundreds of thousands of rows could be efficiently done within modern relational databases systems.

Benefits of Merging Data with Sqoop

- A benefit of Spark, Hadoop, Impala, Hive, … platform is that you can do the same queries at greater scale at lower cost, and perform many other types of analysis not suitable for RDBMS.
- Seamless integration is important when evaluating any new infrastructure. Hence, it's important to be able to do what you normally do, and not break any regular BI reports or workloads over the dataset you plan to migrate.
- To analyze the transaction data in the new platform, we need to ingest it into the Hadoop Distributed File System (HDFS). We need a tool that easily transfers structured data from a RDBMS to HDFS, while preserving structure. That enables us to query the data, but not interfere with or break any regular workload on it.
- Apache Sqoop, is such tool. With Sqoop we can automatically load our relational data from MySQL (Oracle, DB2, etc) into HDFS, while preserving the structure of that data.

Apache Avro

- So far we mostly dealt with textual files. Data can be stored, both on regular file systems and HDFS more efficiently.
- There are several file formats that could be used in Hadoop ecosystem. Notable mentions are AVRO, Parquet, RCFile & ORC
- Avro is an efficient serialization system specially optimized for Hadoop. Several Hadoop ecosystem tools like Impala (a data warehouse engine) are actually very comfortable with Avro files.
- Avro provides:
  - Rich data structures,
  - A compact, fast, binary data format,
  - A container file, to store persistent data.
  - Remote procedure call (RPC).
  - Simple integration with dynamic languages..
- Avro relies on *schemas*. When Avro data is read, the schema used when writing it is always present. This permits each datum to be written with no per-value overheads, making serialization both fast and small. This also facilitates use with dynamic, scripting languages, since data, together with its schema, is fully self-describing.
- When Avro data is stored in a file, its schema is stored with it, so that files may be processed later by any program. If the program reading the data expects a different schema this can be easily resolved, since both schemas are present.

Apache Parquet

- How is data organized within each file is the key matter for database design.
- Apache Parquet is a columnar storage format available to the Hadoop ecosystem, regardless of the choice of data processing framework, data model or programming language.
- Parquet is built from the ground up with complex nested data structures in mind, and uses the record shredding and assembly algorithm.
- Parquet is built to support very efficient compression and encoding schemes.
- Parquet allows compression schemes to be specified on a per-column level, and is future-proofed to allow adding more encodings as they are invented and implemented.
- Parquet is an efficient columnar storage substrate without the cost of extensive and difficult to set up dependencies.
- You should experiment and find out whether you should use: Avro, Parquet, RCFile (Record Columnar File) or ORC (Optimized Row Columnar) file system.

Import all tables

- To use sqoop to import all tables of an existing database schema, at the command prompt we type all at one line:

```
$ sqoop import-all-tables -m 1 --connect
jdbc:mysql://quickstart:3306/retail_db
--username=retail_dba
--password=cloudera
--compression-codec=snappy
--as-parquetfile
--warehouse-dir=/user/hive/warehouse
--hive-import
```

- Import takes a while, more than several minutes on a fast laptop.
- You will notice that Sqoop is executing several map-reduce jobs in the process.
- Parquet is a format designed for analytical applications on Hadoop. Instead of grouping your data into rows like typical data formats, it groups your data into columns. This is ideal for many analytical queries where instead of retrieving data from specific records, you're analyzing relationships between specific variables across many records. Parquet is designed to optimize data storage and retrieval in these scenarios..

Hive Side of Things

- In Hue's Hive editor we could ask for all the tables:

```
hive> show tables;
    orders
    customers
    categories
    order_items, etc.
```

- All tables from `retail_db` database are imported and visible. We could ask for the number of records for some of those tables, for examples:

```
hive> select count(*) from order_items;
       172198
```

- We can also examine the content of our HDFS directories:

```
cloudera@quickstart ~]$ hadoop fs -ls /user/hive/warehouse
Found 14 items
drwxrwxrwx   0 2016-03-04 10:10 /user/hive/warehouse/categories
drwxrwxrwx   0 2016-03-04 10:11 /user/hive/warehouse/customers
drwxrwxrwx   0 2016-03-04 10:12 /user/hive/warehouse/departments
drwxrwxrwx   0 2016-02-26 16:00 /user/hive/warehouse/merged
```

- Every Hive table acquired a directory under `/user/hive/warehouse`

Run Queries in Impala

- Impala could query Hive tables. To save time during queries, Impala does not poll constantly for metadata changes. The first thing we must do is tell Impala that its metadata is out of date. Then we should see our tables show up, ready to be queried.

```
impala> invalidate metadata;
Impala> show tables;
```

- Since Impala now has access to our transaction data, we could use Impala to make some serious investigation of our business practices. For example, we could ask what are the most popular product categories:

```
select c.category_name, count(order_item_quantity) as count
from order_items oi
inner join products p on oi.order_item_product_id = p.product_id
inner join categories c on c.category_id = p.product_category_id
group by c.category_name
order by count desc
limit 10;
```

10 Most popular product categories

Rank	category_name	count
0	Cleats	24551
1	Men's Footwear	22246
2	Women's Apparel	21035
3	Indoor/Outdoor Games	19298
4	Fishing	17325
5	Water Sports	15540
6	Camping & Hiking	13729
7	Cardio Equipment	12487
8	Shop By Sport	10984
9	Electronics	3156

- On `http://localhost:25000/queries` we could find that the query took `4min 56 sec`. That is not correct. That must include some setup time.
- You should repeat the query and the time will be `8 sec 546 mili sec`

Do the Query on MySql Server

```
$ mysql –user=retail_dba –p
Enter password: cloudera
Mysql> use retail_db;
mysql> select count(*) from order_items;
+----------+
| count(*) |
+----------+
|   172198 |
+----------+
1 row in set (0.10 sec)
mysql> select c.category_name, count(order_item_quantity) as count
    -> from order_items oi
    -> inner join products p on oi.order_item_product_id = p.product_id
    -> inner join categories c on c.category_id = p.product_category_id
    -> group by c.category_name
    -> order by count desc
    -> limit 10;
+----------------------+-------+
| category_name        | count |
+----------------------+-------+
| Cleats               | 24551 |
| Men's Footwear       | 22246 |
| Women's Apparel      | 21035 |
| Indoor/Outdoor Games | 19298 |
| Fishing              | 17325 |
| Water Sports         | 15540 |
| Camping & Hiking     | 13729 |
| Cardio Equipment     | 12487 |
| Shop By Sport        | 10984 |
| Electronics          |  3156 |
+----------------------+-------+
10 rows in set (0.35 sec)
```

- You should as the school for your money back.

Sqoop

- We told Sqoop to import the data into Hive but used Impala to query the data.
- Hive and Impala can share both data files and the table metadata.
- Hive works by compiling SQL queries into MapReduce jobs, which makes it very flexible, whereas Impala executes queries itself and is built from the ground up to be as fast as possible, which makes it better for interactive analysis.
- You typically use Hive for an ETL (extract-transform-load) workloads.
- You use Impala for fast repetitive queries.