Lecture 04

Spark
RDDs, Data Frames, Datasets, SQL

Zoran B. Djordjević

Three API-s, RDD, DataFrames & Datasets

- In Spark 2.x, there are three sets of APIs—RDDs, DataFrames, and Datasetes.
- We need to understand why and when to use each set; outline their performance and optimization benefits; and enumerate scenarios when to use DataFrames and Datasets instead of RDDs.
- In Apache Spark 2.0 DataFrames and Datasets APIs are unified.

RDDs, When to Use Them

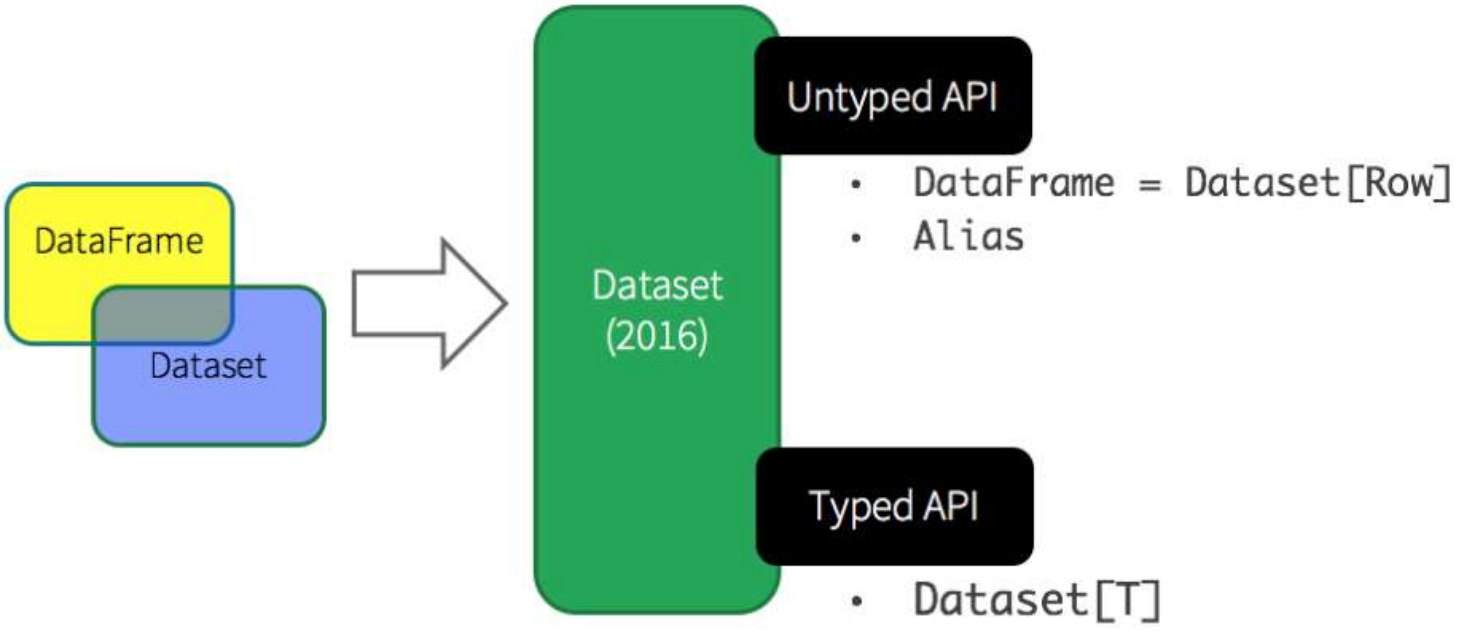
- RDD was the primary user-facing API in Spark since its inception. At the core, an RDD is an immutable distributed collection of elements of your data, partitioned across nodes in your cluster that can be operated in parallel with a low-level API that offers *transformations* and *actions*.
- Common use cases for using RDDs are:
  - you want low-level transformation and actions and control on your data;
  - your data is unstructured, such as media streams or streams of text;
  - you want to manipulate your data with functional programming constructs rather than domain specific expressions;
  - you don't care about imposing a schema, such as columnar format, while processing or accessing data attributes by name or column; and
  - you can forgo some optimization and performance benefits available with DataFrames and Datasets for structured and semi-structured data.
- Are RDDs being relegated as second class citizens? Are they being deprecated?
- The answer is a **NO!**
- We can seamlessly move between DataFrame or Dataset and RDDs at will—by simple API method calls—and DataFrames and Datasets are built on top of RDDs.

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.
- In Spark 2.0, DataFrame APIs merged with Datasets APIs, unifying data processing capabilities across libraries.
- Because of this unification, developers now have fewer concepts to learn or remember, and work with a single high-level and type-safe API called Dataset.

Unified Apache Spark 2 API

- Starting in Spark 2.0, Dataset takes on two distinct APIs characteristics: a ***strongly-typed*** API and an ***untyped*** API, as shown in the table below.
- Conceptually, DataFrame is an *alias* for a collection of generic objects *Dataset[Row]*, where a *Row* is a generic ***untyped*** JVM object. Dataset, by contrast, is a collection of ***strongly-typed*** JVM objects, dictated by a case class you define in Scala or a class in Java.



Typed and Untyped APIs

- Python and R have no compile-time type-safety, they only have untyped APIs, i.e. DataFrames. Scala has both DataFrames and Datasets and Java Datasets only.

Language	Main Abstraction
Scala	Dataset[T], DataFrame (alias for Dataset[Row])
Java	Dataset<T>
Python	DataFrame
R	DataFrame

- A Dataset<T> is a strongly typed collection of domain-specific objects that can be transformed in parallel using functional or relational operations.
- `Row` represents one row of output from a relational operator:

```
// Create a Row from values.
Row(value1, value2, value3, ...)
// Create a Row from a Seq of values.
Row.fromSeq(Seq(value1, value2, ...
```

- Row allows both generic access by ordinal, as well as native primitive access.

Benefits of Dataset, Static Typing and Runtime Type Safety

- If we treat static-typing and runtime safety as a spectrum, SQL is least restrictive and Dataset is the most restrictive.
- For instance, in your Spark SQL string queries, you won't know a syntax error until runtime (which could be costly), whereas in DataFrames and Datasets you can catch errors at compile time (which saves developer-time and costs).
- At the far end of the spectrum is Dataset, most restrictive. Dataset APIs are all expressed as lambda functions and JVM typed objects, any mismatch of typed-parameters will be detected at compile time. Also, your analysis error can be detected at compile time too, when using Datasets, hence saving developer-time and costs.
- In your Spark code, Datasets as most restrictive yet productive for a developer.

	SQL	DataFrames	Datasets
Syntax Error	Runtime	Compile Time	Compile Time
Logical Error	Runtime	Runtime	Compile Time

Custom view into structured & semi-structured data

- DataFrames as a collection of *Datasets[Row]* render a structured custom view into your semi-structured data.
- For example, let's say, we have a huge IoT device event dataset, expressed as JSON. Since JSON is a semi-structured format, it lends itself well to employing Dataset as a collection of type-specific `Row[DeviceIoTData]` objects.
- If we have JSON file with data that reads like:

```
{"device_id": 198164, "device_name": "sensor-pad-198164owomcJZ", "ip":
"80.55.20.25", "cca2": "PL", "cca3": "POL", "cn": "Poland", "latitude":
53.080000, "longitude": 18.620000, "scale": "Celsius", "temp": 21, "humidity":
65, "battery_level": 8, "c02_level": 1408, "lcd": "red", "timestamp"
:1458081226051}
```

- We could save such data in a JSON file, e.g. `deviceIoTdata.json` and then read such data directly into a `DataFrame`.

```
>>>Using Python version 2.7.5 (default, Aug  4 2017 00:39:18)
SparkSession available as 'spark'.
>>> ds = spark.read.json("/home/centos/deviceIoTdata.json")
>>> type(ds)
<class 'pyspark.sql.dataframe.DataFrame'>
>>> ds.collect()
```

This is useful since you need something to collect data from all machines

```
[Row(battery_level=8, c02_level=1408, cca2=u'PL', cca3=u'POL', cn=u'Poland',
device_id=198164, device_name=u'sensor-pad-198164owomcJZ', humidity=65,
ip=u'80.55.20.25', latitude=53.08, lcd=u'red', longitude=18.62, scale=u'Celsius',
temp=21, timestamp=1458081226051)]
Ds.columns
```

Custom view into structured & semi-structured data

- We could for example ask the DataFrame what are its columns :

```
>>> ds.columns
['battery_level', 'c02_level', 'cca2', 'cca3', 'cn', 'device_id',
'device_name', 'humidity', 'ip', 'latitude', 'lcd', 'longitude', 'scale',
'temp', 'timestamp']
```

- Or we could create a temporary view (table) called devices:

```
>>> ds.createGlobalTempView("devices")
```

- From which we could perform regular SQL queries, like:

```
>>> ds2 = spark.sql("select battery_level, latitude, device_id
from global_temp.devices")
>>> ds2.collect()
[Row(battery_level=8, latitude=53.08, device_id=198164)]
```

- In other words, DataFrame treated or exposed semi-structured, JSON, data as ordered, typed data. Three things happen here under the hood in the code above:

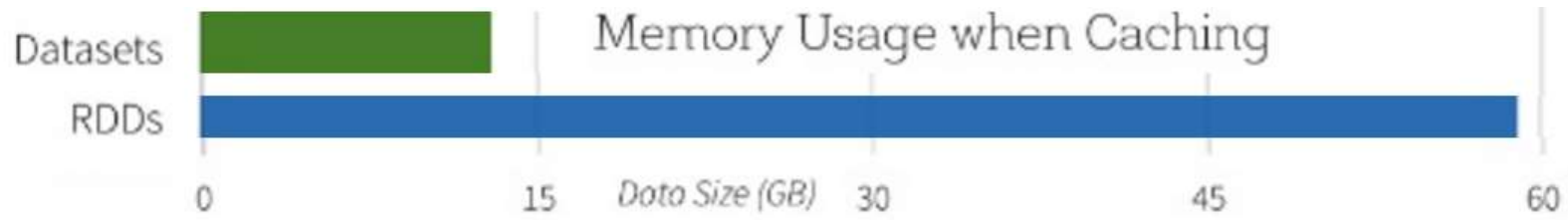
1. Spark reads the JSON, infers the schema, and creates a collection of DataFrames.
2. At this point, Spark converts your data into *`DataFrame = Dataset[Row]`*, a collection of generic Row object, since it does not know the exact type.
3. In Scala, Spark converts the *`Dataset[Row] -> Dataset[DeviceIoTData]`* ***type-specific*** Scala JVM object, as dictated by the **class** *DeviceIoTData*.

Ease-of-use of APIs with structure

- When data are rendered as a DataFrame, it is much simpler to perform `agg`, `select`, `sum`, `avg`, `map`, `filter`, or `groupBy` operations by accessing a Dataset typed object's `DeviceIoTData` than using RDD rows' data fields.
- You could extract all those data from an RDD using some kind of regular expression process but that is much more tedious.

Performance, Optimization, Space Efficiency

- Spark developers assert that there is a great space efficiency and performance gain in using DataFrames and Dataset APIs over RDD-s.
- DataFrame and Dataset APIs are built on top of the Spark SQL engine which generate an optimized logical and physical query plan.
- Across R, Java, Scala, or Python DataFrame/Dataset APIs, all relation type queries undergo the same code optimizer, providing the space and speed efficiency. Whereas the *Dataset[T]* typed API is optimized for data engineering tasks, the untyped *Dataset[Row]* (an alias of DataFrame) is even faster and suitable for interactive analysis.
- Spark as a compiler understands Dataset type JVM object and can efficiently serialize/deserialize JVM objects as well as generate compact bytecode that can execute at superior speeds.



When to Use DataFrames, Datasets

Semi-Structured: dataset/dataframe, unstructured: RDD

- If you want rich semantics, high-level abstractions, and domain specific APIs, use DataFrame or Dataset.
- If your processing demands high-level expressions, filters, maps, aggregation, averages, sum, SQL queries, columnar access and use of lambda functions on semi-structured data, use DataFrame or Dataset.
- If you want higher degree of type-safety at compile time, want typed JVM objects, take advantage of optimization, and benefit from efficient code generation, use Dataset.
- If you want unification and simplification of APIs across Spark Libraries, use DataFrame or Dataset.
- If you are an R user, use DataFrames.
- If you are a Python user, use DataFrames and resort back to RDDs if you need more control. Dataset doesn't exist in Python

DataFrames

- A *DataFrame* is a table of data with rows and columns. The list of columns and the types in those columns is the *schema*. Structured
- A simple analogy to Spark DataFrame is a spreadsheet with named columns. The fundamental difference is that while a spreadsheet sits on one computer in one specific location, a Spark DataFrame can span thousands of computers.
- One is putting the data on more than one computer because either the data is too large to fit on one machine or it would simply take too long to perform that computation on one machine.
- The DataFrame concept is not unique to Spark. The concept is borrowed from R. Python has similar concepts. However, Python/R DataFrames (with some exceptions) exist on one machine rather than multiple machines. This limits what you can do with a given DataFrame in Python and R to the resources that exist on that specific machine.
- Spark has language interfaces for both Python and R, and it is easy to convert to Spark DataFrames to Pandas (Python) DataFrames

Core Abstractions, Partitions

- In order to allow every executor to perform work in parallel, Spark breaks up the data into chunks, called partitions. A *partition* is a collection of rows that sit on one physical machine in our cluster.
- If you have one partition, Spark will only have a parallelism of one even if you have thousands of executors. If you have many partitions, but only one executor Spark will still only have a parallelism of one because there is only one computation resource.
- An important thing to note, is that with DataFrames, we do not (for the most part) manipulate partitions individually. We simply specify high level transformations of data in the physical partitions and Spark determines how this work will actually execute on the cluster.

Transformations, Immutability

- In Spark, the core data structures are *immutable* meaning they cannot be changed once created. In order to "change" a DataFrame you have to instruct Spark how to modify the DataFrame you have into the one that you want.
- These instructions are called *transformations*.
- In Python, we create a simple DataFrame with command `spark.range()` and then name its single column with method `toDF()`

```
>>> myRange = spark.range(1000).toDF("number")
```

- We transform DF `myRange` by applying `filter("number % 2 = 0")`.

`where()` is an alias for `filter()`

```
>>> divisBy2 = myRange.where("number % 2 = 0")
```

- In Scala:

```
>>> val myRange = spark.range(1000).toDF("number")
>>> val divisBy2 = myRange.where("number % 2 = 0")
```

- You notice that these return no output, because we only specified an abstract transformation and Spark will not act on transformations until we call an action. Transformations are the core of how you express your business logic using Spark.

Narrow and wide Dependencies

- There are two types of transformations, those that specify narrow dependencies and those that specify wide dependencies.
- Transformations consisting of *narrow dependencies* are those where each input partition will contribute to only one output partition.
- Our `where()` clause specifies a narrow dependency, where only one partition contributes to at most one output partition.
- A *wide dependency* style transformation will have input partitions contributing to many output partitions. The operation where Spark exchanges partitions across the cluster is called a *shuffle* .
- Spark will automatically perform an operation called pipelining on narrow dependencies, meaning that if we specify multiple filters on DataFrames they'll all be performed in memory.
- The same cannot be said for shuffles. When we perform a shuffle, Spark will write the results to disk.

Lazy Evaluation

- *Lazy evaulation* means that Spark will wait until the very last moment to execute your transformations.
- In Spark, instead of modifying the data quickly, we build up a plan of transformations that we would like to apply to our source data.
- Spark, by waiting until the last minute to execute the code, will compile this plan from your raw, DataFrame transformations, to an efficient physical *plan* that will run as efficiently as possible across the cluster. This provides immense benefits to the end user because Spark can optimize the entire data flow from end to end.
- An example of this might be “predicate pushdown”. If we build a large Spark job consisting of narrow dependencies, but specify a filter at the end that only requires us to fetch one row from our source data, the most efficient way to execute this is to access the single record that we need. Spark will actually optimize this calculation by pushing the filter down automatically.

Actions

- Transformations allow us to build up our logical transformation plan. To trigger the computation, we run an *action*.
- An *action* instructs Spark to compute a result from a series of transformations. The simplest action is `count()` which gives us the total number of records in the DataFrame.

```
>>> divisBy2.count()
500
```

- We now see a result! There are 500 number divisible by two from 0 to 999. count() is not the only action. There are three kinds of actions:
  - actions to view data in the console;
  - actions to collect data to native objects in the respective language; and
  - actions to write to output data sources.
- In specifying an action, we started a Spark job that runs our filter transformation (a narrow transformation), then an aggregation (a wide transformation) that performs the counts on a per partition basis, then a collect will brings our result to a native object in the respective language.

Converting from a DataFrame to RDD

- We can always seamlessly interoperate or convert from DataFrame and/or Dataset to an RDD, by simple method call `.rdd`. For instance we can design a new DataFrame by selecting several columns and rows from previous data set `ds`: Not only does spark support sql queries(seen in prior slide), but it also has sql-like function calls

```
>>> deviceEventsDS = ds.select("device_name", "cca3",
"c02_level").where("c02_level > 1300")
```

- Here we see that we can apply regular SQL on a DataFrame and pass predicates (as strings) through a `where (filter)` clause.

```
>>> deviceEventsDS.collect()
[Row(device_name=u'sensor-pad-198164owomcJZ', cca3=u'POL',
c02_level=1408)]
```

Now, transform this DataFrame, `deviceEventsDS`, into an RDD using method `rdd`

```
>>> eventsRDD = deviceEventsDS.rdd
>>> type(eventsRDD)
<class 'pyspark.rdd.RDD'>
>>> eventsRDD.collect()
[Row(device_name=u'sensor-pad-198164owomcJZ', cca3=u'POL',
c02_level=1408)]
>>>
```

Features of RDDs

- We open `pyspark` and load data from the local operating system.
- We happen to have the `ulysses10.txt` file in `/home/centos` directory and could do the following:

```
>>> blines = sc.textFile("file:///home/centos/ulysses10.txt")
>>> blines.count()
32742
>>> blines.first()
u'The Project Gutenberg Etext of Ulysses'
```

- What we did above was create an RDD named `blines` and populate that RDD with data from the local file `/home/centos/ulysses10.txt`.
- We also see another method of RDD-s, an action, `first()`, which tells us that the first line in RDD `blines` is a collection of characters (**`u'. . .'`**).
- Please note that we are not terminating our commands with a semi-colon (";") or anything else aside from the carriage return. Savings on typing all those semi-colons is one of the greatest contributions of Python to the computer science. That is why our commands are in Python.

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 the 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)
```

Closure, 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 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 and commutative 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 that variable `v` 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 variable `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. Object `v` should not be modified after it is broadcasted 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 and `collect()` –ing 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.parallelize(range(1,60))
print(numbers)
ParallelCollectionRDD[9] at parallelize at PythonRDD.scala:391
>>> numbers.collect()
[1, 2, 3, 4, 5]
```

- To print all elements on the driver, one uses the `collect()` method to first bring the RDD to the driver node. This can cause the driver to run out of memory, though, because `collect()` fetches the entire RDD to a single machine. To print a few elements of the RDD, a safer approach is to use the `take(): rdd.take(100).foreach(println)`.
- *`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., OS or 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 determine which data are needed for the requested 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*

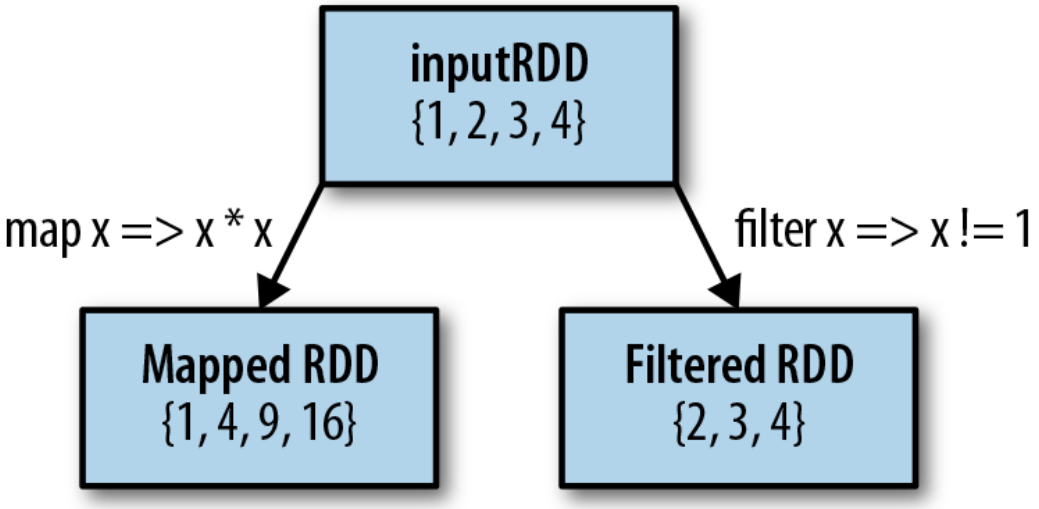
```
System.out.println("Input had " + badLinesRDD.count() + "
concerning lines")
System.out.println("Here are 10 examples:")
for (String line: badLinesRDD.take(10)) {
   System.out.println(line);
}
```

- 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*

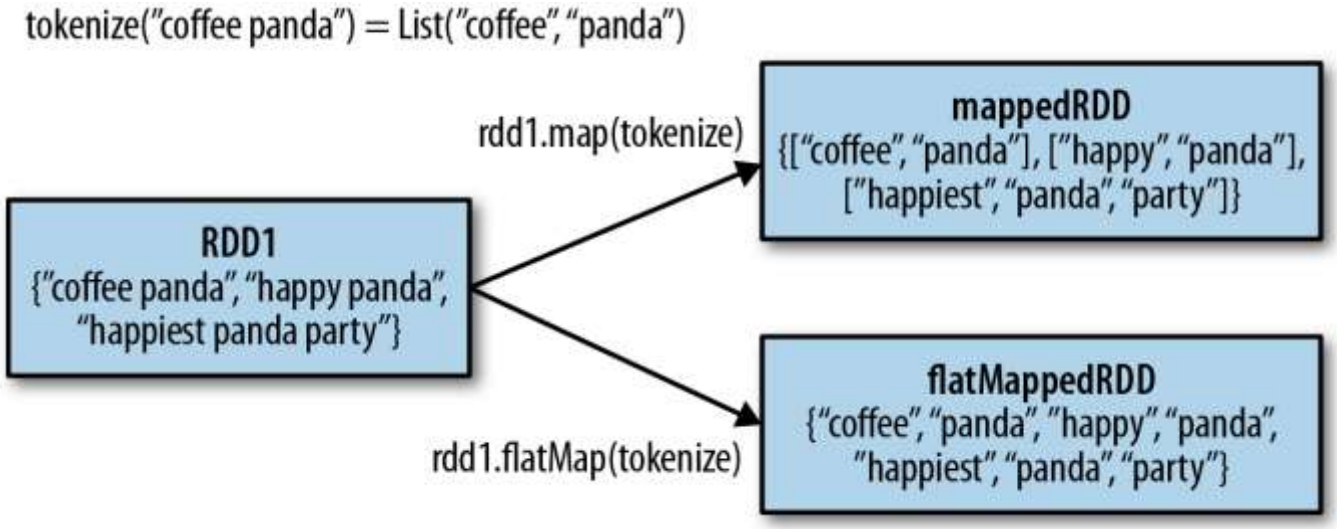
```
val input = sc.parallelize(List(1, 2, 3, 4))
val result = input.map(x => x * x)
println(result.collect().mkString(","))
```

- *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*

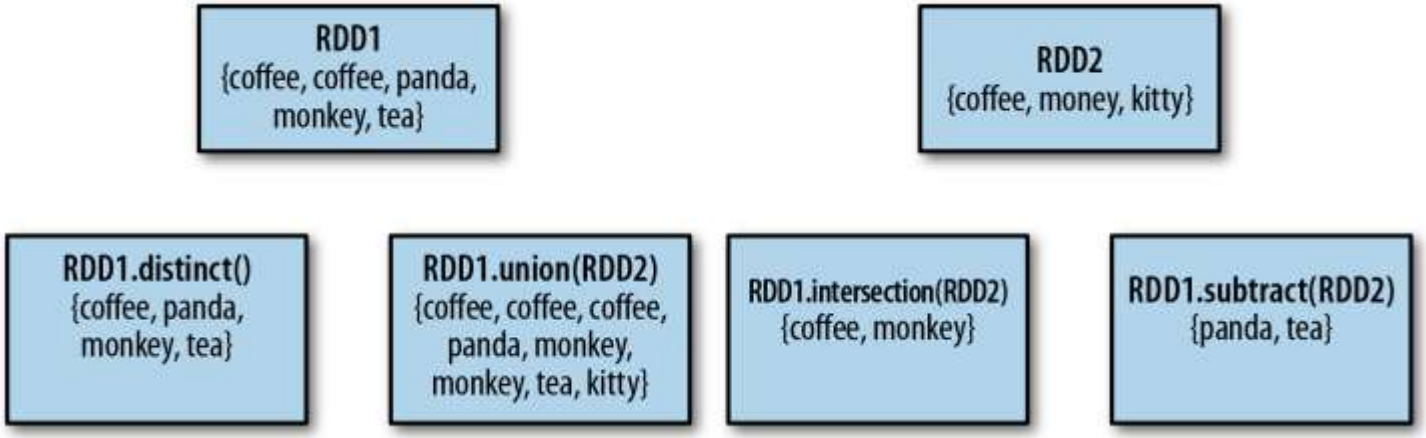
```
val lines = sc.parallelize(List("hello world", "hi"))
val words = lines.flatMap(line => line.split(" "))
words.first() // returns "hello"
```

- *`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"
```

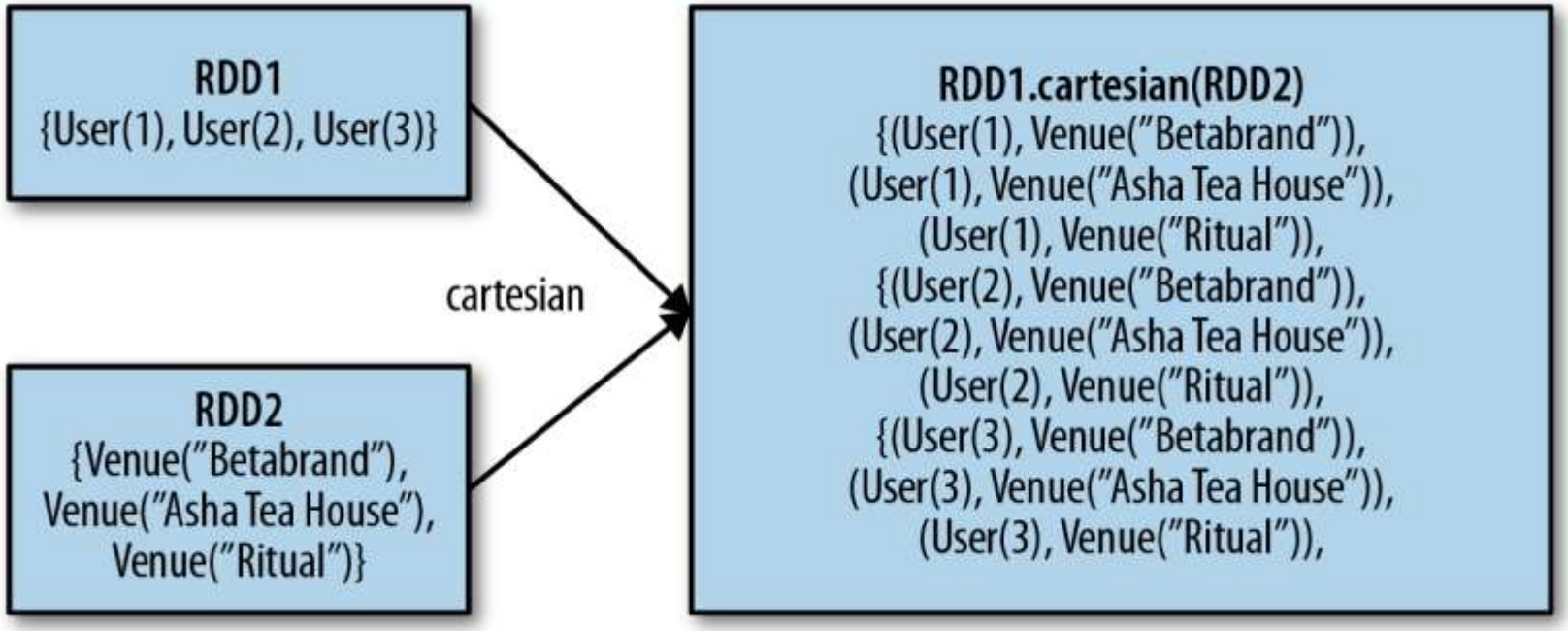
Pseudo set operations

- 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()`. `reduce()` takes a function that operates on two elements of a 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 might be faster to modify `a` to contain `a + b` and return that rather than allocate a new data structure..

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 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 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 listing different comma-separated values one after another. 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. Tuples can be sliced, concatenated, and so on.
- To access values in a 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 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("file://...")
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; }
});
```

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());
```

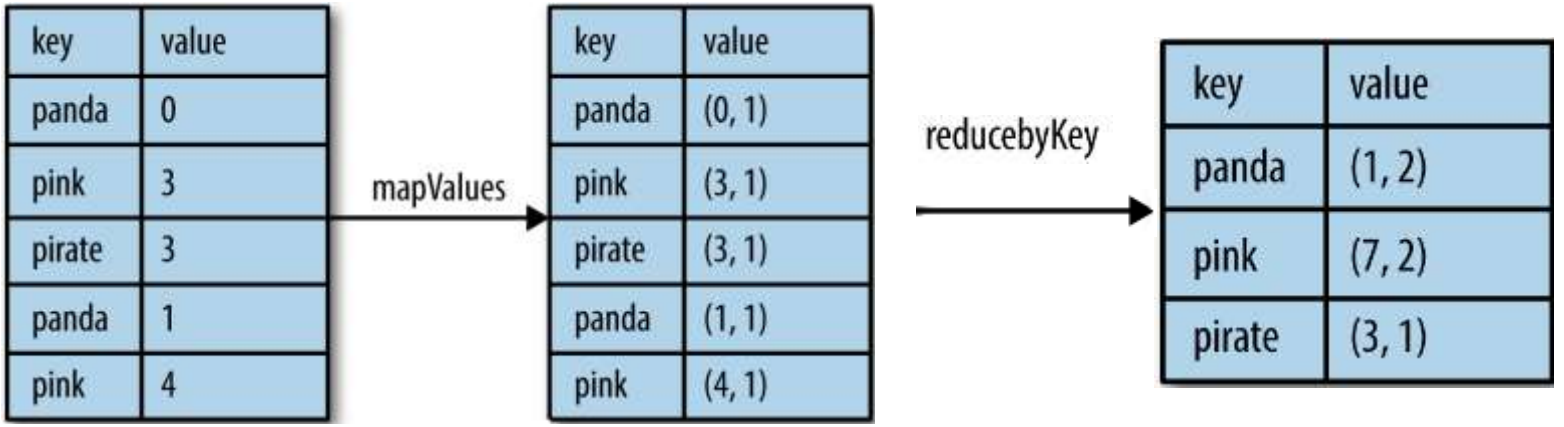
Aggregation Examples

- As 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))
```



Spark SQL

- Spark SQL used to be a separate API in Spark 1.x. Spark SQL engine is still there and implements most features of SQL 2003 standard and many sophisticated add-on.
- Spark SQL Engine is for the most part hidden behind DataFrame and Dataset API.
- 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 (Hive is a data warehouse application built atop of Hadoop)
- 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 JDBC/ODBC.
- SparkSQL could read and write data from Parquet files, JSON files, Hive, Cassandra

DataFrames

- A DataFrame is a distributed collection of data organized into named columns. It is conceptually equivalent to an Excel spreadsheet, 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 an interface that provides the benefits over 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. There is no Python or R support for Dataset API.

SQL Context, Spark 1.6

- `SQLContext` is a class and is used for initializing the functionalities of Spark SQL in Spark 1.x.
- `SparkContext` class object (`sc`) is required for initializing `SQLContext` class object in older Spark versions (1.x).
- The entry point into all relational functionality is Spark `SQLContext` class, or one of its decedents, like `HiveContext`.

Json Sample File

- We created a small JSON file
  /home/centos/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.

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

[centos]$ cat people.txt
Michael, 29
Andy, 30
Justin, 19
```

For DataFrames use `SparkSession` (`spark`)

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

```
>>> from pyspark.sql import SQLContext #
>>> sqlContext = SQLContext(sc)
>>> df = sqlContext.read.json("file:///home/centos/employee.json")
```

need not do this in pyspark

```
df = spark.read.json("file:///home/centos/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 with no Math.

```
>>> df.select('name', 'age').show(2)
+----+---+
|name|age|
+----+---+
|Mary| 36|
|John| 39|
+----+---+
```

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
```

- On the following slide we will use files `emps.csv` and `emps.txt` with the following content:

```
Michael, 29, 3000.30
Andy, 30, 2500.25
Justin, 19, 4000.99
```

Transform an RDD to a DataFrame, Row Class

- Let us import file `emps.txt` into `RDD emps`:

```
>>> from pyspark.sql import Row
>>> sc = spark.sparkContext
>>> emps = sc.textFile("file:///home/centos/emps.csv")
>>> emps_fields = emps.map(lambda e: e.split(","))

>>> employees = emps_fields.map(lambda e: Row(name = e[0], age =
int(e[1]), salary = float(e[2])))
>>> employees.collect()
[Row(age=29, name=u'Michael', salary=3000.3000000000002),
Row(age=30, name=u'Andy', salary=2500.25), Row(age=19,
name=u'Justin', salary=4000.9899999999998)]

>>> empsDF = spark.createDataFrame(employees)
>>> empsDF.show()
+---+-------+-------+
|age|   name| salary|
+---+-------+-------+
| 29|Michael| 3000.3|
| 30|   Andy|2500.25|
| 19| Justin|4000.99|
+---+-------+-------+
```

Imposing Schema with Row class

- We imported file `emps.csv` into RDD `emps`, we broke every line in that RDD into composite fields, creating RDD `emps_fields`

```
>>> emps_fields.collect()
[[u'Michael', u' 29', u' 3000.30'], [u'Andy', u' 30', u'
2500.25'], [u'Justin', u' 19', u' 4000.99']]
```

- Next we used class `Row` to assign names and types to every element of the previous RDD. Result is an RDD `employees` which contains elements of type `Row` as its elements.
- Object `employees` is an RDD not table. If you apply method `show()` on it you will get an error. However, RDD `employees` contains `Row` elements and could be transformed into a `DataFrame` object using method `spark.createDataFrame()` applied to RDD employees.

Infer Schema and Register DataFrame as a Table

- Once we create a DataFrame and infer a schema

```
empsDF = spark.createDataFrame(employees)
```

- We could register that DataFrame as a table

```
empsDF.createOrReplaceTempView("people")
```

- SQL can be run over DataFrames that have been registered as a table.

```
teenagers = spark.sql(
 "SELECT name FROM people WHERE age >= 13 AND age <= 19")
```

- The results of SQL queries are DataFrame objects.

- Method `rdd` of DataFrame object returns the content as a `:class:`pyspark.RDD` of :class:`Row`.`

```
teenNames = teenagers.rdd.map(lambda p: "Name: " + p.name).collect()
for name in teenNames:
    print(name)
. . .
Name: Justin
```

Connect to local MySql DB

```
$ mysql -uroot -p
Enter password: Cent0s324$
Mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| classicmodels
. . . . .      |
+--------------------+
mysql> use classicmodels;
Database changed
mysql> show tables;
+---------------------+
| Tables_in_retail_db |
+---------------------+
| customers           |
| employees           |
| offices             |
| orderdetails        |
| orders              |
| payments            |
| productlines        |
| products            |
+---------------------+
```

Connect Spark to Relational DB, e.g. 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-5.1.44-bin.jar` where convenient.
- Next run `pyspark` with `–driver-class-path` option:

```
$ pyspark --driver-class-path mysql-connector-java-5.1.44-bin.jar
>>> dfm =
spark.read.format("jdbc").option("url","jdbc:mysql://localhost/retail_db").
option("driver","com.mysql.jdbc.Driver").option("dbtable","customers").opti
on("user","classicmodels").option("password", "Cent0s324$").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 DataFrame object `dfm` into a temp table `customers`, so that we could run SQL queries against it:

```
>>> dfm.registerTempTable("customers")
>>> spark.sql("select * from customers LIMIT 3")
.show(3)
+--------------+--------------------+--------------+----------------+------------+-
|customerNumber|        customerName|contactLastName|contactFirstName|       phone|
+--------------+--------------------+--------------+----------------+------------+-
|           103|   Atelier graphique|        Schmitt|         Carine |  40.32.2555|
|           112|  Signal Gift Stores|           King|            Jean|  7025551838|
|           114|Australian Collec...|      Ferguson|           Peter|03 9520 4555|
+--------------+--------------------+--------------+----------------+------------
only showing top 3 rows
```

- You drop temp table by command:

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