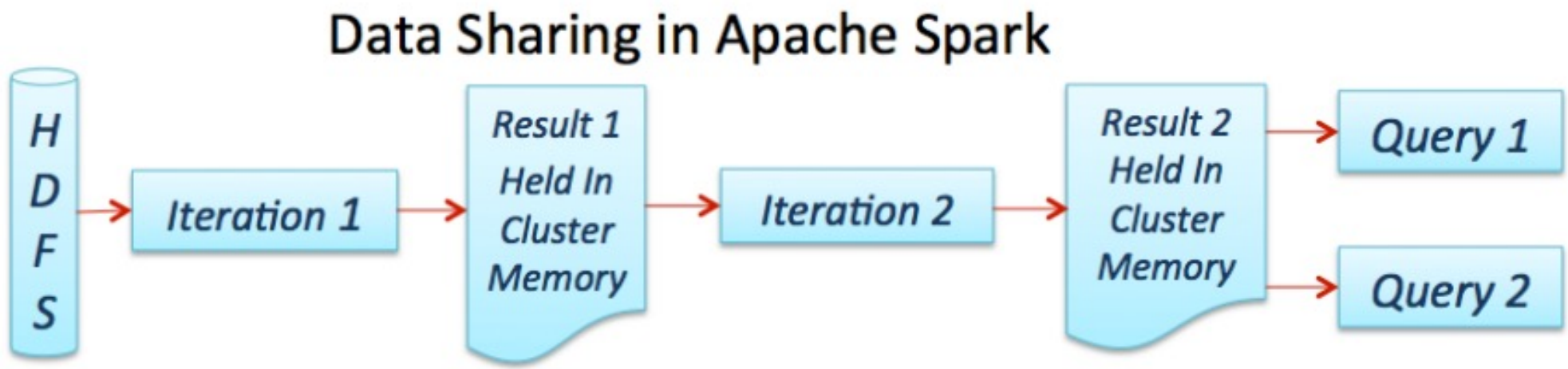
Into to Big Data and Data Science

RDDs, Data frames & SQL

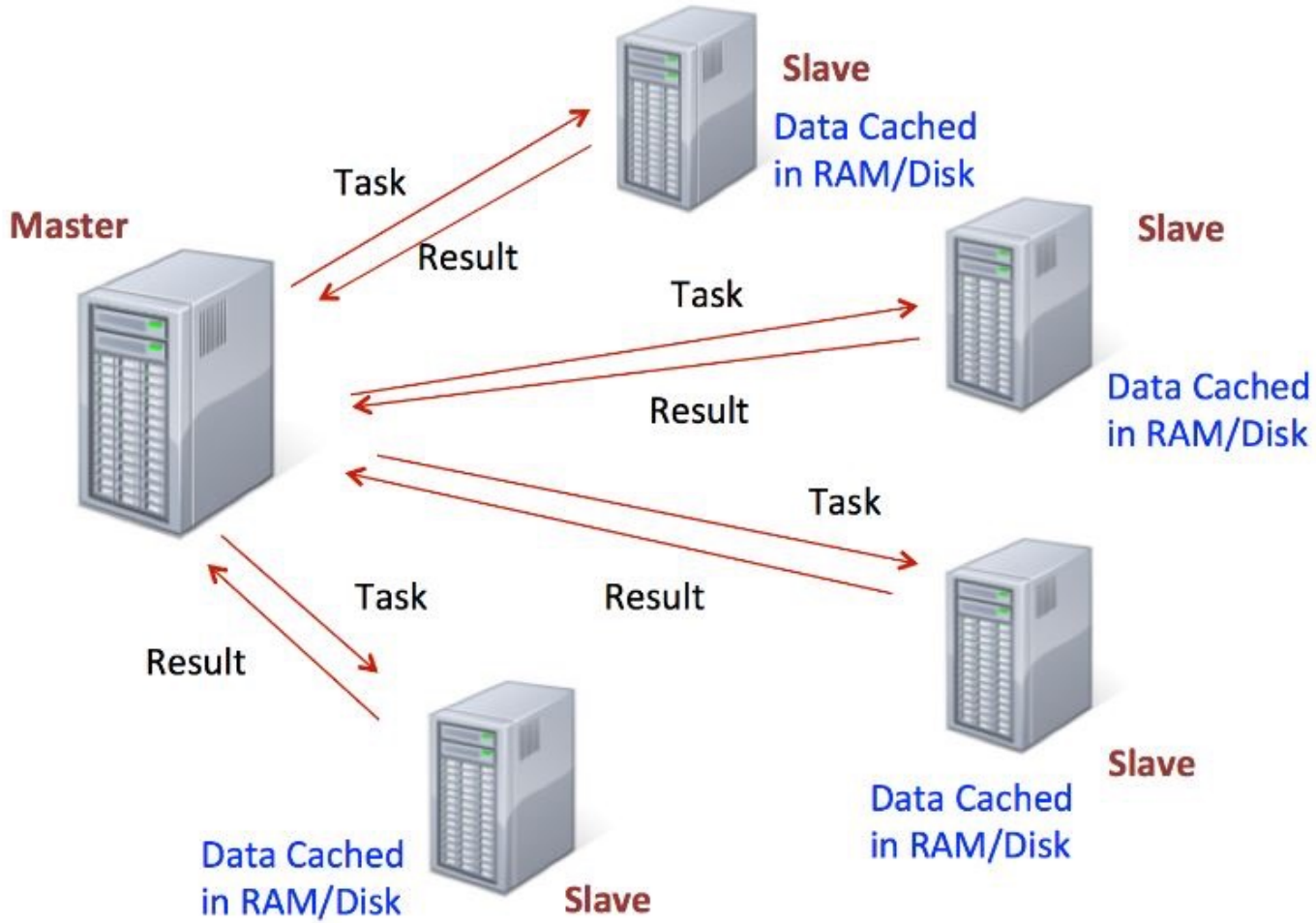
How Apache Spark works

Spark engine provides a way to process data in distributed memory over a cluster of machines. Figure 7 shows a logical diagram of how a typical Spark job processes information.



Source: https://dzone.com/refcardz/apache-spark

How does Spark execute a job



The Master controls how data is partitioned, and it takes advantage of data locality while keeping track of all the distributed data computation on the Slave machines. If a certain Slave machine is unavailable, the data on that machine is reconstructed on other available machine(s). "Master" is currently a single point of failure, but it will be fixed in upcoming releases.

Source: https://dzone.com/refcardz/apache-spark

Learning Spark Programming

- Easiest way: Spark interpreter (`spark-shell` or `pyspark`)
- Runs in local mode on 1 thread by default, but can control with `MASTER` environment var:

```
MASTER=local     ./spark-shell           # local, 1 thread
MASTER=local[2]  ./spark-shell           # local, 2 threads
MASTER=spark://host:port ./spark-shell   # Spark standalone cluster
```

First Stop: SparkContext

- Main entry point to Spark functionality
- Created for you in Spark shells as variable `sc`

Creating RDDs

```
# Turn a local collection into an RDD
sc.parallelize([1, 2, 3]) #Python

sc.parallelize(Array(1, 2, 3)) #Scala

# Load text file from local FS, HDFS, or S3
sc.textFile("file.txt")
sc.textFile("directory/*.txt")
sc.textFile("hdfs://namenode:9000/path/file")

# Use any existing Hadoop InputFormat
sc.hadoopFile(keyClass, valClass, inputFmt, conf)
```

Basic Transformations (Python)

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

# Pass each element through a function
squares = nums.map(lambda x: x*x)   # => {1, 4, 9}

# Keep elements passing a predicate
even = squares.filter(lambda x: x % 2 == 0) # => {4}

# Map each element to zero or more others
nums.flatMap(lambda x: range(0, x))  # => {0, 0, 1, 0,
1, 2}
```

Range object (sequence of numbers 0, 1, …, x-1)

Basic Transformations (Scala)

```scala
System.setProperty("hadoop.home.dir", "F:\\winutils");

val sparkConf = new SparkConf().setAppName("SparkTransformation").setMaster("local[*]")

val sc = new SparkContext(sparkConf)

val nums = sc.parallelize(Array(1, 2, 3))
// Pass each element through a function
val squares = nums.map(x => (x * x)) // => {1, 4, 9}

// Keep elements passing a predicate
val even = squares.filter(x => x % 2 == 0) // => {4}

// Map each element to zero or more others
val result = nums.flatMap(x => Array.range(0, x)) //=> {0, 0, 1, 0, 1, 2}

result.foreach(println(_))
```

Basic Actions (Python)

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

# Retrieve RDD contents as a local collection
nums.collect() # => [1, 2, 3]

# Return first K elements
nums.take(2)   # => [1, 2]

# Count number of elements
nums.count()   # => 3

# Merge elements with an associative function
nums.reduce(lambda x, y: x + y)  # => 6

# Write elements to a text file
nums.saveAsTextFile("hdfs://file.txt")
```

Note:

1. ***Collect() and count() throws the Py4JavaError, to use take() or saveAsTextFile() in it's place***
2. ***Reduce() and sortbyKey() might throw the same error.***

Basic Actions (Scala)

```scala
System.setProperty("hadoop.home.dir", "F:\\winutils");

val sparkConf = new SparkConf().setAppName("SparkActions").setMaster("local[*]")

val sc = new SparkContext(sparkConf)

val nums = sc.parallelize(Array(1, 2, 3))
// Retrieve RDD contents as a local collection
nums.collect() // => [1, 2, 3]
//Return first K elements
nums.take(2) // => [1, 2]
//Count number of elements
nums.count() // => 3
//Merge elements with an associative function
nums.reduce((x, y) => (x + y)) // => 6
//Write elements to a text file
nums.saveAsTextFile("file.txt")
```

Working with Key-Value Pairs

- Spark's "distributed reduce" transformations act on RDDs of *key-value pairs*
- Python:

```
pair = (a, b)
pair[0] # => a
pair[1] # => b
```

- Scala:

```
val pair = (a, b)
pair._1 // => a
pair._2 // => b
```

- Java:

```
Tuple2 pair = new Tuple2(a, b);  // class scala.Tuple2
pair._1 // => a
pair._2 // => b
```

Some Key-Value Operations (Python)

```
pets = sc.parallelize([(“cat”, 1), (“dog”, 1), (“cat”, 2)])

pets.reduceByKey(lambda x, y: x + y)
# => {(cat, 3), (dog, 1)}

pets.groupByKey()
# => {(cat, Seq(1, 2)), (dog, Seq(1)}

pets.sortByKey()
# => {(cat, 1), (cat, 2), (dog, 1)}
```

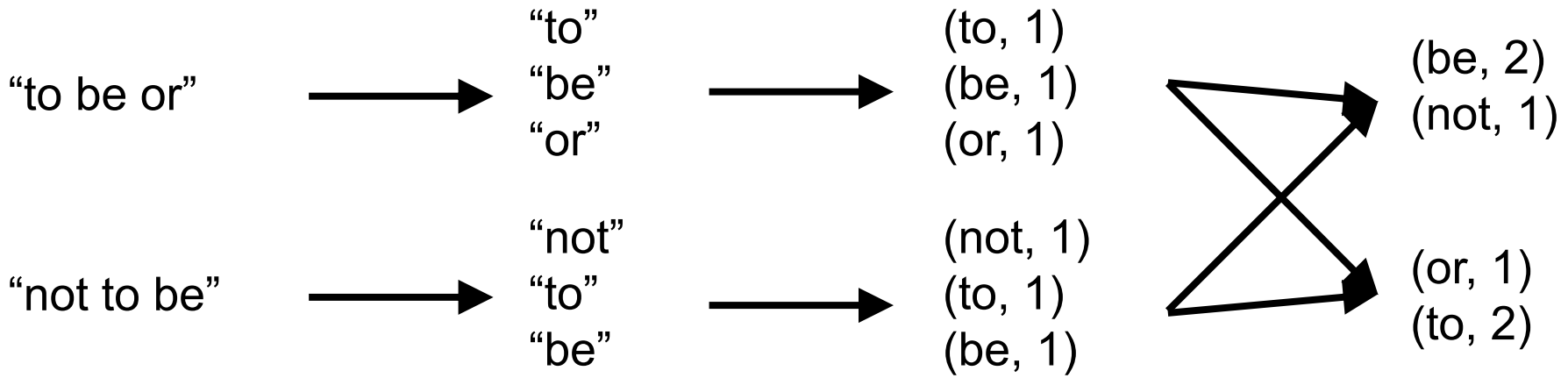
`reduceByKey` also automatically implements combiners on the map side

Some Key-Value Operations (Scala)

```scala
System.setProperty("hadoop.home.dir", "F:\\winutils");

val sparkConf = new SparkConf().setAppName("SparkActions").setMaster("local[*]")

val sc = new SparkContext(sparkConf)

val pets = sc.parallelize(Array(("cat", 1), ("dog", 1), ("cat", 2)))
pets.reduceByKey((x, y) => x + y) // => {(cat, 3), (dog, 1)}
pets.groupByKey() // => {(cat, Seq(1, 2)), (dog, Seq(1)}
pets.sortByKey() // => {(cat, 1), (cat, 2), (dog, 1)}
```

Example: Word Count (Python)

```
lines = sc.textFile(“hamlet.txt”)

counts = lines.flatMap(lambda line: line.split(“ ”)) \
              .map(lambda word: (word, 1)) \
              .reduceByKey(lambda x, y: x + y)
```



Multiple Datasets (Python)

```
visits = sc.parallelize([(“index.html”, “1.2.3.4”),
                         (“about.html”, “3.4.5.6”),
                         (“index.html”, “1.3.3.1”)])

pageNames = sc.parallelize([(“index.html”, “Home”), (“about.html”, “About”)])

visits.join(pageNames)
# (“index.html”, (“1.2.3.4”, “Home”))
# (“index.html”, (“1.3.3.1”, “Home”))
# (“about.html”, (“3.4.5.6”, “About”))

visits.cogroup(pageNames)
# (“index.html”, (Seq(“1.2.3.4”, “1.3.3.1”), Seq(“Home”)))
# (“about.html”, (Seq(“3.4.5.6”), Seq(“About”)))
```

Multiple Datasets (Scala)

```scala
System.setProperty("hadoop.home.dir", "F:\\winutils");

val sparkConf = new SparkConf().setAppName("SparkActions").setMaster("local[*]")

val sc = new SparkContext(sparkConf)

val visits = sc.parallelize(Array(("index.html", "1.2.3.4"),("about.html", "3.4.5.6"),("index.html", "1.3.3.1")))
val pageNames = sc.parallelize(Array(("index.html", "Home"), ("about.html", "About")))
visits.join(pageNames)
// ("index.html", ("1.2.3.4", "Home"))
// ("index.html", ("1.3.3.1", "Home"))
// ("about.html", ("3.4.5.6", "About"))
visits.cogroup(pageNames)
// ("index.html", (Seq("1.2.3.4", "1.3.3.1"), Seq("Home")))
// ("about.html", (Seq("3.4.5.6"), Seq("About")))
```

Spark Data Frames

Write Less Code: High-Level Operations

Solve common problems concisely using Data Frame functions:

1. Selecting columns and filtering
2. Joining different data sources
3. Aggregation (count, sum, average, etc.)
4. Plotting results (e.g., with Pandas)

What are DataFrames?

DataFrames are a recent addition to Spark (early 2015).

The DataFrames API:

1. Is intended to enable wider audiences beyond "Big Data" engineers to leverage the power of distributed processing
2. Is inspired by data frames in R and Python (Pandas)
3. Designed from the ground-up to support modern big data and data science applications
4. An extension to the existing RDD API

What are DataFrames?

DataFrames have the following features:

1. Ability to scale from kilobytes of data on a single laptop to petabytes on a large cluster
2. Support for a wide array of data formats and storage systems
3. State-of-the-art optimization and code generation through the Spark SQL Catalyst optimizer
4. Seamless integration with all big data tooling and infrastructure via Spark
5. APIs for Python, Java, Scala, and R

Construct a DataFrame

```python
# Construct a DataFrame from a "users" table in Hive.
df = sqlContext.table("users")

# Construct a DataFrame from a log file in S3.
df = sqlContext.load("s3n://someBucket/path/to/data.json", "json")
```

```scala
val people = sqlContext.read.parquet("...")
```

```java
DataFrame people = sqlContext.read().parquet("...")
```

Use DataFrames

```
# Create a new DataFrame that contains only "young" users
young = users.filter(users["age"] < 21)

# Alternatively, using a Pandas-like syntax
young = users[users.age < 21]

# Increment everybody's age by 1
young.select(young["name"], young["age"] + 1)

# Count the number of young users by gender
young.groupBy("gender").count()

# Join young users with another DataFrame, logs
young.join(log, logs["userId"] == users["userId"], "left_outer")
```

DataFrames and Spark SQL

```
young.registerTempTable("young")
sqlContext.sql("SELECT count(*) FROM young")
```

DataFrames and Spark SQL

DataFrames are fundamentally tied to Spark SQL.

1. The DataFrames API provides a *programmatic* interface really, a *domain-specific language* (DSL) for interacting with your data.
2. Spark SQL provides a *SQL-like* interface.
3. What you can do in Spark SQL, you can do in Data frames … and vice versa.

What, exactly, is Spark SQL?

Spark SQL allows you to manipulate distributed data with SQL queries. Currently, two SQL dialects are supported.

- If you're using a Spark SQLContext, the only supported dialect is "sql", a rich subset of SQL 92.
- If you're using a HiveContext, the default dialect is "hiveql", corresponding to Hive's SQL dialect. "sql" is also available, but "hiveql" is a richer dialect.

Spark SQL

- You issue SQL queries through a `SQLContext` or `HiveContext`, using the `sql()` method.
- The `sql()` method returns a `DataFrame`.
- You can mix DataFrame methods and SQL queries in the same code.
- To use SQL, you *must* either:
  - query a persisted Hive table, or
  - make a *tablealias* for a DataFrame, using `registerTempTable()`

DataFrames

Like Spark SQL, the DataFrames API assumes that the data has a table-like structure.

Formally, a DataFrame is a size-mutable, potentially heterogeneous tabular data structure with labeled axes (i.e., rows and columns).

Just think of it as a table in a distributed database: a distributed collection of data organized into named, typed columns.

Transformations, Actions, Laziness

DataFrames are *lazy*. *Transformations* contribute to the query plan, but they don't execute anything.

Actions cause the execution of the query.

Transformation examples

- filter
- select
- drop
- intersect
- join

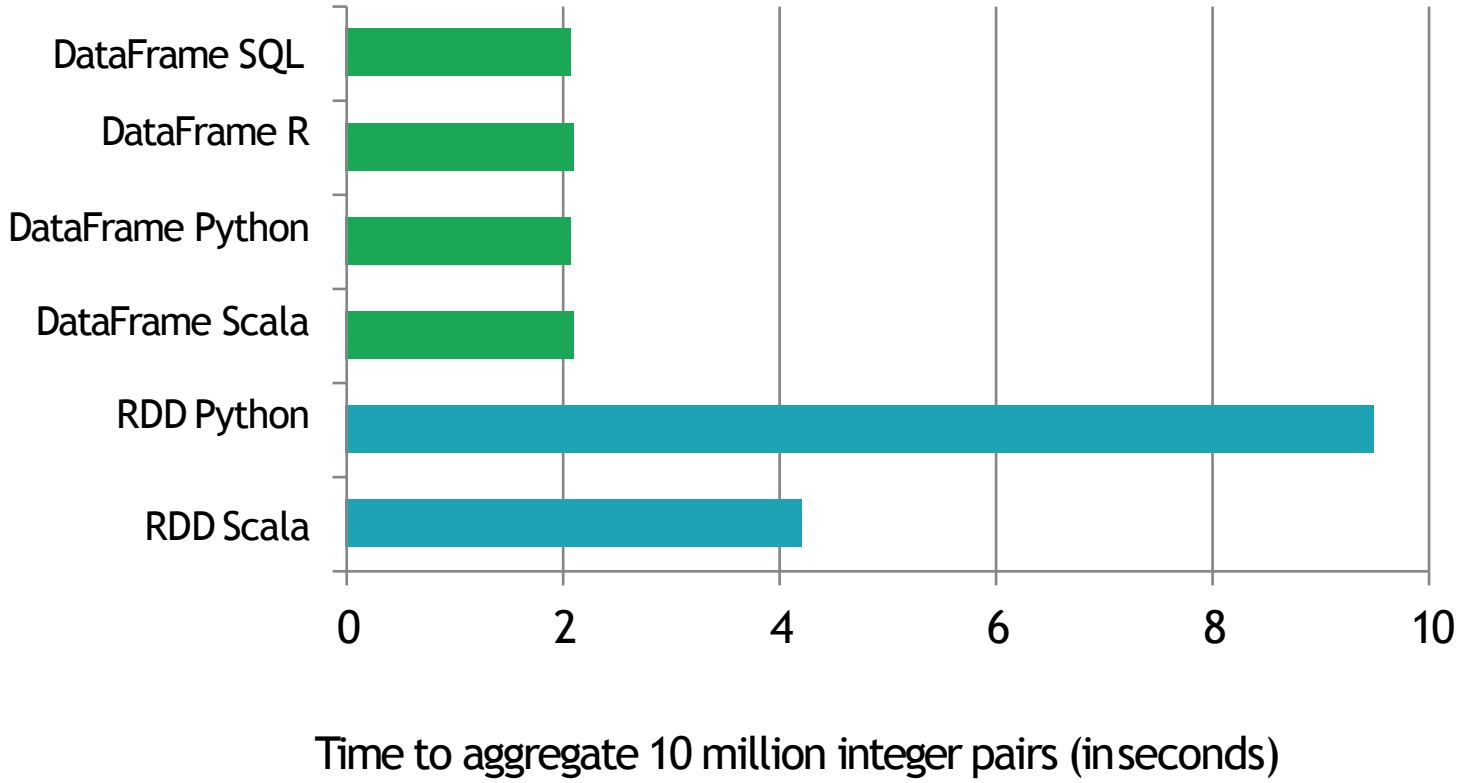
Action examples

- count
- collect
- show
- head
- take

Data frames & ResilientDistributed Datasets (RDDs)

- Data frames are built on top of the Spark RDD* API.
  - This means you can use normal RDD operations on Data frames.
- However, stick with the Data frame API, wherever possible.
  - Using RDD operations will often give you back an RDD, not a Data frame.
  - The Data frame API is likely to be more efficient, because it can optimize the underlying operations with Catalyst.

DataFrames can be *significantly* faster than RDDs. And they perform the same, regardless of language.



Creating a DataFrame

- You create a Data frame with SQLContext object (or one of its descendants)
- In the Spark Scala shell (*spark-shell*) or *pyspark*, you have a SQLContext available automatically, as `sqlContext`.
- In an application, you can easily create one yourself, from SparkContext.
- The Data frame *data source API* is consistent, across data formats.
  - "Opening" a data source works pretty much the same way, no matter what.

Creating a DataFrame

```
conf = SparkConf().setAppName(appName).setMaster(master)
sc = SparkContext(conf=conf)
sqlContext = SQLContext(sc)

df = sqlContext.read.parquet("/path/to/data.parquet")
df2 = sqlContext.read.json("/path/to/data.json")
```

SQLContext and Hive

Our previous examples created a default Spark SQLContext object.

If you're using a version of Spark that has Hive support, you can also create a HiveContext, which provides additional features, including:

1. The ability to write queries using the more complete HiveQL parser
2. Access to Hive user-defined functions
3. The ability to read data from Hive tables

Data Sources supported by DataFrames

50

What can I do with a DataFrame?

Once you have a DataFrame, there are a number of operations you can perform.

Let's look at a few of them.

But, first, let's talk about columns.

Columns

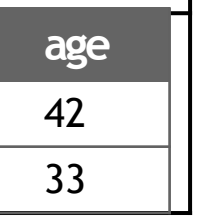
When we say "column" here, what do we mean?

A Data frame *column* is an abstraction. It provides a common column-oriented view of the underlying data, *regardless* of how the data is really organized.

Columns

Input Source Format	Data Frame Variable Name	Data

JSON	dataFrame1	<pre>[{"first": "Amy",
		"last": "Bello",
		"age": 29 },
		{"first": "Ravi",
		"last": "Agarwal",
		"age": 33 },
		...
		]</pre>

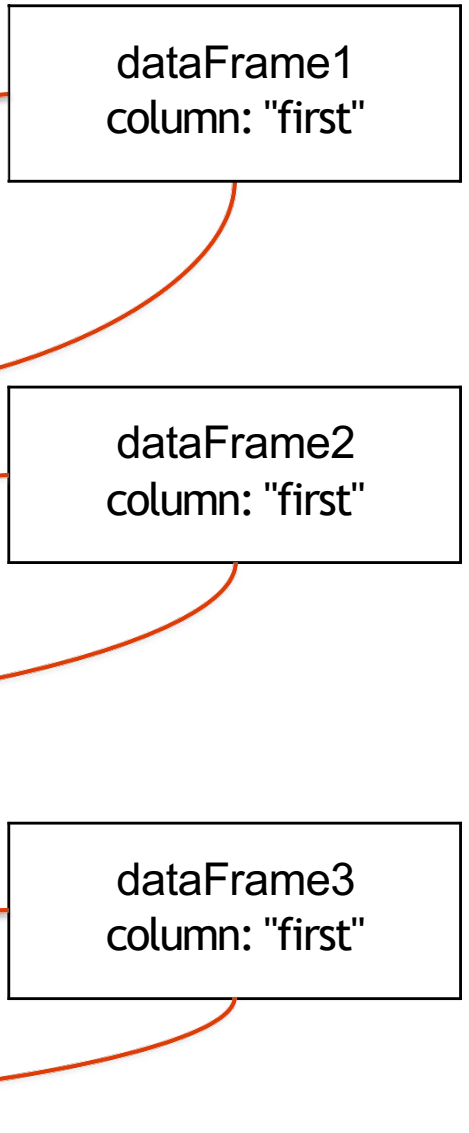
CSV	dataFrame2	first,last,age Fred,Hoover,91 Joaquin,Hernandez,24 ...
SQL Table	dataFrame3	

first	last	age
Joe	Smith	42
Jill	Jones	33

Data Frame columns map onto some common data sources.

Column

Input Source Format	Data Frame Variable Name	Data
JSON	dataFrame1	[{"first": "Amy", "last": "Bello", "age": 29 }, {"first": "Ravi", "last": "Agarwal", "age": 33 }, ...]
CSV	dataFrame2	first,last,age Fred,Hoover,91 Joaquin,Hernandez,24 ...
SQL Table	dataFrame3	first | last | age Joe | Smith | 42 Jill | Jones | 33



Columns

When we say "column" here, what do we mean?

Several things:

- A place (a *cell*) for a data value to reside, within a row of data. This cell can have several states:
  - empty (null)
  - missing (not there at all)
  - contains a (typed) value (non-null)
- A collection of those cells, from multiple rows
- A syntactic construct we can use to *specify* or *target* a cell (or collections of cells) in a Data frame query

select()

You can also use SQL. (This is the Python API, but you issue SQL the same way in Scala and Java)

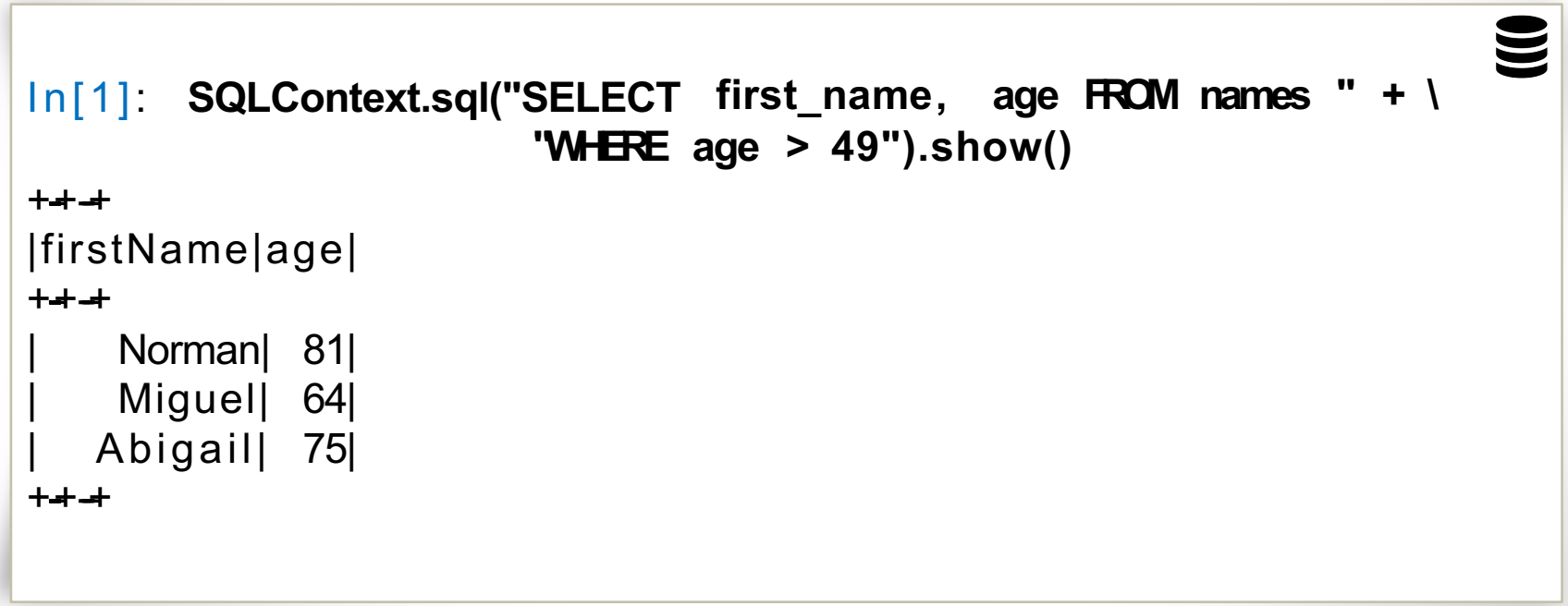
```
In[1]: df.registerTempTable("names")
In[2]: sqlContext.sql("SELECT first_name, age, age > 49 FROM names").\
                  show(5)
+----------+---+-----+
|first_name|age|  _c2|
+----------+---+-----+
|      Erin| 42|false|
|    Claire| 23|false|
|    Norman| 81| true|
|    Miguel| 64| true|
|  Rosalita| 14|false|
+----------+---+-----+
```

filter()

```
In[1]: df.filter(df['age'] > 49).\
           select(df['first_name'], df['age']).\
           show()
+---+
|firstName|age|
+---+
|     Norman|  81|
|     Miguel|  64|
|    Abigail|  75|
+---+
```

filter()

Here's the SQLversion.



```
In[1]: SQLContext.sql("SELECT first_name, age FROM names " + \
                      "WHERE age > 49").show()
+---+
|firstName|age|
+---+
|     Norman| 81|
|     Miguel| 64|
|    Abigail| 75|
+---+
```

Filter()

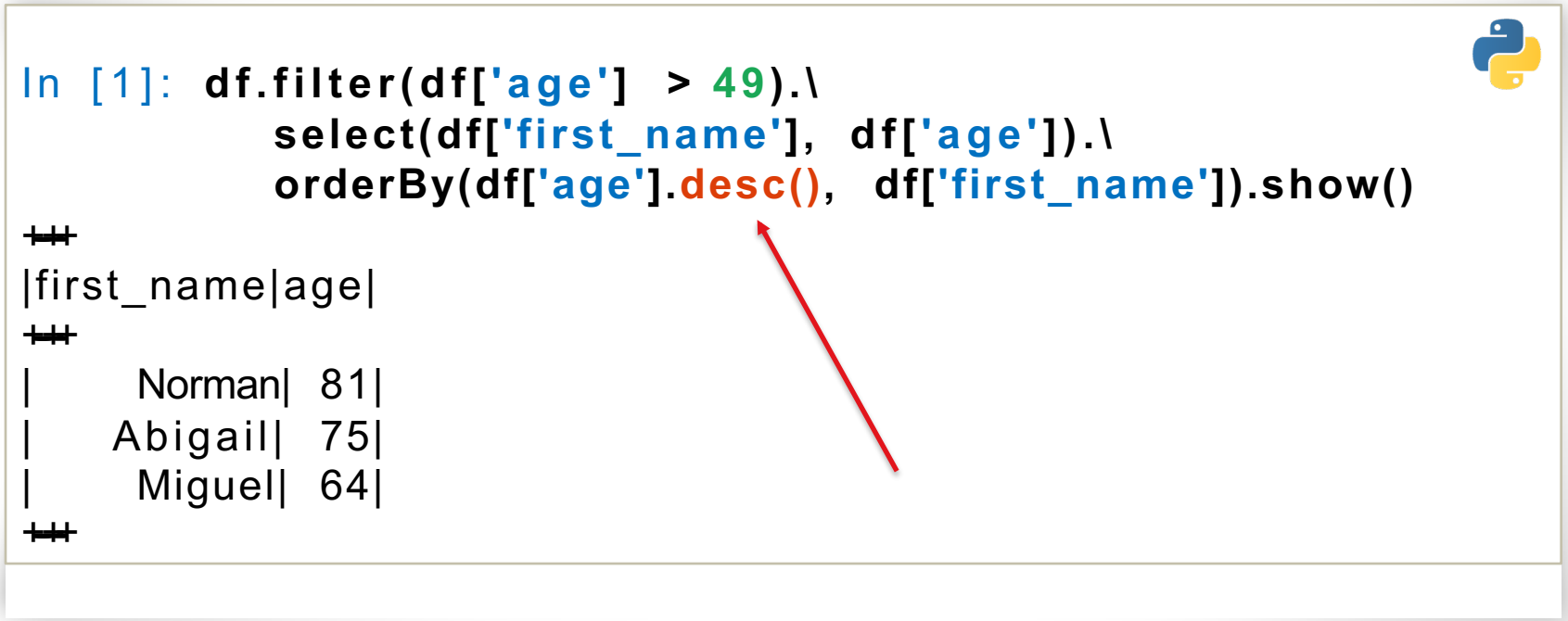
- Scala Example

```
// To create DataFrame using SQLContext
val people = sqlContext.read.parquet("...")
val department = sqlContext.read.parquet("...")

people.filter("age > 30")
  .join(department, people("deptId") === department("id"))
  .groupBy(department("name"), "gender")
  .agg(avg(people("salary")), max(people("age")))
```

orderBy()

inPython:

```
In [1]: df.filter(df['age'] > 49).\
            select(df['first_name'], df['age']).\
            orderBy(df['age'].desc(), df['first_name']).show()
+----------+---+
|first_name|age|
+----------+---+
|    Norman| 81|
|   Abigail| 75|
|    Miguel| 64|
+----------+---+
```



groupBy()

Often used with count(), groupBy() groups data items by a specific column value.

```
In [5]: df.groupBy("age").count().show()
+---+-----+
|age|count|
+---+-----+
| 39|    1|
| 42|    2|
| 64|    1|
| 75|    1|
| 81|    1|
| 14|    1|
| 23|    2|
```

References

- https://spark.apache.org/docs/latest/rdd-programming-guide.html
- https://www.tutorialspoint.com/apache_spark/advanced_spark_programming.htm
- https://www.datacamp.com/community/tutorials/apache-spark-python
- https://data-flair.training/blogs/spark-rdd-operations-transformations-actions/
- https://spark.apache.org/docs/latest/sql-programming-guide.html
- https://spark.apache.org/docs/2.2.0/sql-programming-guide.html
- https://www.tutorialspoint.com/spark_sql/spark_sql_dataframes.htm
- https://spark.apache.org/docs/1.6.3/api/java/org/apache/spark/sql/DataFrame.html