RAMAIAH
Institute of Technology

Cloud Computing and Big Data

Subject Code: CS71 (Credits: 4:0:0)

Textbook:

1. Cloud Computing Theory and Practice – **DAN C**. Marinescu – Morgan Kaufmann Elsevier.
2. Cloud Computing A hands - on approach – Arshdeep Bahga & **Vijay madisetti** Universities press
3. Big Data Analytics, Seema Acharya and Subhashini Chellappan. 2nd edition, Wiley India Pvt. Ltd. 2019

NOTE: I declare that the PPT content is picked up from the prescribed course text books or reference material prescribed in the syllabus book and Online Portals.

Unit V

Analysing Big Data:

- The Challenges of Data Science,
- Introducing Apache Spark.

Introduction to Data Analysis with Scala and Spark :

- Scala for Data Scientists,
- The Spark Programming Model, Record Linkage,
- Getting Started: The Spark Shell and Spark Context,
- Bringing Data from the Cluster to the Client,
- Shipping Code from the Client to the Cluster,
- Structuring Data with Tuples and Case Classes,
- Aggregations, Creating Histograms, Summary Statistics for Continuous Variables,
- Creating Reusable Code for Computing Summary Statistics,
- Simple Variable Selection and Scoring

The Challenges of Data Science:

First, the vast majority of work that goes into conducting successful analyses lies in **preprocessing data.**

- Data is messy, and **cleansing, munging, fusing, mushing**, and many other verbs are prerequisites to doing anything useful with it.

- Large data sets in particular, because they are not amenable to direct examination by humans, can **require computational methods** to even discover what preprocessing steps are required.

- Even when it comes time to **optimize model performance**, a typical data pipeline requires spending far more time in feature engineering and selection than in choosing and writing algorithms

For example, when **building a model that attempts to detect fraudulent purchases on a website**, the data scientist must choose from a wide variety of potential features:

- fields that users are required to fill out, IP location info, login times, and click logs as users navigate the site.
- Each of these comes with its own challenges when converting to vectors fit for machine learning algorithms.
- A system needs to support more flexible transformations than turning a 2D array of doubles into a mathematical model.

Second, **iteration is a fundamental part of data science**.

- Modeling and analysis typically require multiple passes over the same data.

- One aspect of this lies within **machine learning algorithms** and **statistical procedures.** Popular optimization procedures like **stochastic gradient descent** and expectation maximization involve repeated scans over their inputs to reach convergence.

- Iteration also matters within the data scientist's own workflow.

- When data scientists are initially investigating and trying to get a feel for a data set, usually the results of a query inform the next query that should run. When building models, data scientists do not try to get it right in one try.

- **Choosing the right features, picking the right algorithms**, running the right significance tests, and **finding the right hyperparameters** all require experimentation

Third, the task isn’t over when **a well-performing model has been built**.

If the point of data science is to make data useful to non–data scientists, then a model stored as a list of regression weights in a text file on the data scientist’s computer has not really accomplished this goal.

Uses of **data recommendation engines** and **real-time fraud detection systems** culminate in data applications.

- **In the lab, data scientists engage in exploratory analytics**. They try to understand the nature of the data they are working with. They visualize it and test wild theories. They **experiment with different classes of features a**nd auxiliary sources they can use to augment it. They cast a wide net of algorithms in the hopes that one or two will work.

- **In the factory, in building a data application, data scientists engage in operational analytics**. They package their models into services that can inform real-world decisions. They **track their models’ performance over time** and obsess about how they can make small tweaks to squeeze out another percentage point of accuracy. They care about SLAs and uptime

Introducing Apache Spark.

Apache Spark is an **open-source cluster computing framework** for real-time processing. It is of the most successful projects in the Apache Software Foundation. Spark has clearly evolved as the market leader for Big Data processing

- Spark provides an interface for programming entire clusters with implicit data parallelism and fault-tolerance.

- It was built on top of Hadoop MapReduce and it extends the MapReduce model to efficiently use more types of computations.

- Spark provides high-level APIs in Java, Scala, Python and R. Spark code can be written in any of these four languages. **It provides a shell in Scala and Python.**

- Spark supports multiple data sources such as JSON, Hive and Cassandra apart from the usual formats such as text files, CSV and RDBMS tables.

Spark Shell:

- Spark’s shell provides a simple way to learn the API, as well as a powerful tool to analyze data interactively.

RDD:

- **Resilient Distributed Dataset (RDD)** is a fundamental data structure of Spark. It is an immutable distributed collection of objects.

- Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster.

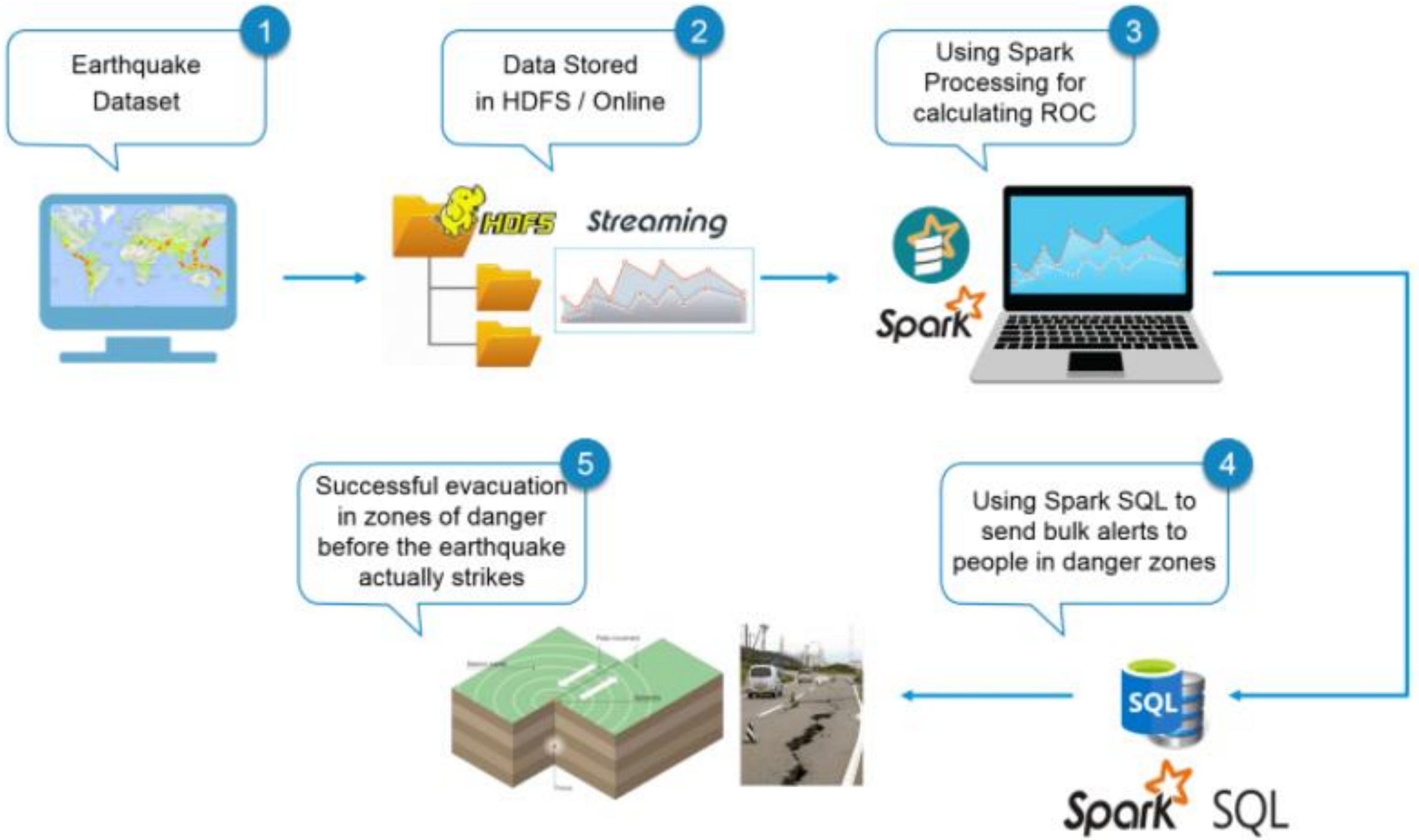
- RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes.

DataFrames:

- A DataFrame is a Dataset 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.

Use Case – Flow Diagram



1 Earthquake Dataset

2 Data Stored in HDFS / Online

3 Using Spark Processing for calculating ROC

4 Using Spark SQL to send bulk alerts to people in danger zones

5 Successful evacuation in zones of danger before the earthquake actually strikes

Figure: *Use Case – Flow diagram of Earthquake Detection using Apache Spark*

Scala for Data Scientists: advantages

It reduces performance overhead.

- Whenever we’re running an algorithm in R or Python on top of a JVM-based language like Scala, we have to do some work to pass code and data across the different environments, and oftentimes, things can get lost in translation.

- When you’re writing data analysis algorithms in Spark with the Scala API, you can be far more confident that your program will run as intended.

It gives you access to the latest and greatest.

- All of Spark’s machine learning, stream processing, and graph analytics libraries are written in Scala, and the Python and R bindings tend to get support this new functionality much later.

- If you want to take advantage of all the features that Spark has to offer (without waiting for a port to other language bindings), you will need to learn at least a little bit of Scala; and if you want to be able to extend those functions to solve new problems you encounter, you’ll need to learn a little bit more.

It will help you understand the Spark philosophy.

- Even when you’re using Spark from Python or R, the APIs reflect the underlying computation philosophy that Spark inherited from the language in which it was developed—Scala.

- If you know how to use Spark in Scala—even if you primarily use it from other languages—you’ll have a better understanding of the system and will be in a better position to “think in Spark.”

The Spark Programming Model

Spark programming starts with a data set, usually residing in some form of distributed, persistent storage like HDFS.

Writing a Spark program typically consists of a few related steps:

1. Define a set of transformations on the input data set.

2. Invoke actions that output the transformed data sets to persistent storage or return results to the driver's local memory.

3. Run local computations that operate on the results computed in a distributed fashion. These can help you decide what transformations and actions to under- take next.

Record Linkage

The problem that we’re going to study in this chapter goes by a lot of different names in the literature and in practice:

- entity resolution, record deduplication, merge-and purge, and list washing

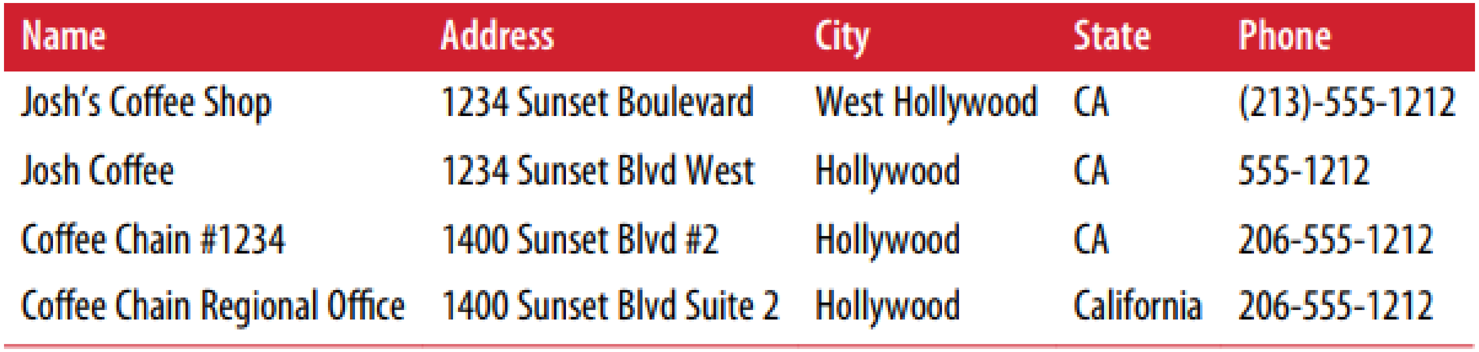
- The general structure of the problem is something like this: we have a large collection of records from one or more source systems, and it is likely that multiple records refer to the same underlying entity, such as a customer, a patient, or the location of a business or an event.

- Each entity has a number of attributes, such as a name, an address, or a birthday, and we will need to use these attributes to find the records that refer to the same entity.

- Unfortunately, the values of these attributes aren’t perfect: **values might have different formatting, typos, or missing information that means that a simple equality test on the values of the attributes will cause us to miss a significant number of duplicate records.**

- For example, let’s compare the business listings shown in Table 2-1.

Table 2-1. The challenge of record linkage



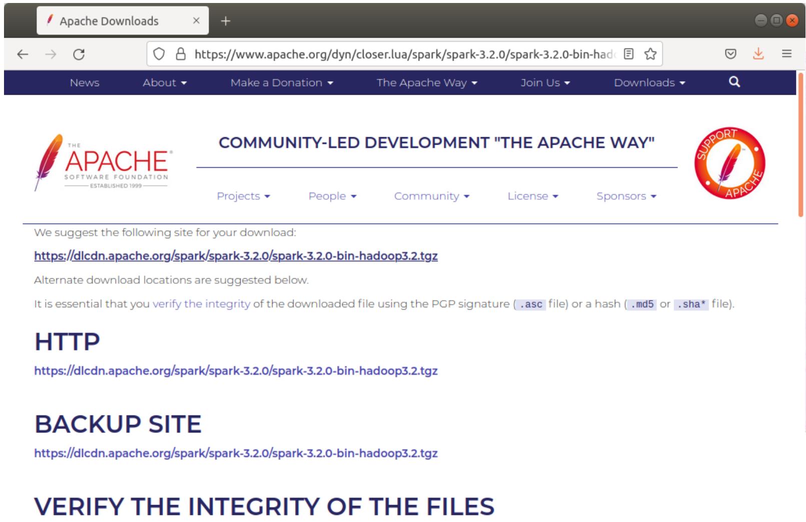
Name	Address	City	State	Phone
Josh's Coffee Shop	1234 Sunset Boulevard	West Hollywood	CA	(213)-555-1212
Josh Coffee	1234 Sunset Blvd West	Hollywood	CA	555-1212
Coffee Chain #1234	1400 Sunset Blvd #2	Hollywood	CA	206-555-1212
Coffee Chain Regional Office	1400 Sunset Blvd Suite 2	Hollywood	California	206-555-1212

The first two entries in this table refer to the same small coffee shop, even though a data entry error makes it look as if they are in two different cities (West Hollywood and Hollywood). The second two entries, on the other hand, are actually referring to different business locations of the same chain of coffee shops that happen to share a common address: one of the entries refers to an actual coffee shop, and the other one refers to a local corporate office location. Both of the entries give the official phone number of corporate headquarters in Seattle.

This example illustrates everything that makes record linkage so difficult: even though both pairs of entries look similar to each other, the criteria that we use to make the duplicate/not-duplicate decision is different for each pair. This is the kind of distinction that is easy for a human to understand and identify at a glance, but is difficult for a computer to learn.

Getting Started: The Spark Shell and SparkContext

```
sudo apt search scala
sudo apt install scala
```


Apache Downloads
https://www.apache.org/dyn/closer.lua/spark/spark-3.2.0/spark-3.2.0-bin-had
News
About
Make a Donation
The Apache Way
Join Us
Downloads
THE APACHE SOFTWARE FOUNDATION ESTABLISHED 1999
SUPPORT APACHE
Projects
People
Community
License
Sponsors

COMMUNITY-LED DEVELOPMENT "THE APACHE WAY"

We suggest the following site for your download:

https://dlcdn.apache.org/spark/spark-3.2.0/spark-3.2.0-bin-hadoop3.2.tgz

Alternate download locations are suggested below.

It is essential that you verify the integrity of the downloaded file using the PGP signature (`.asc` file) or a hash (`.md5` or `.sha*` file).

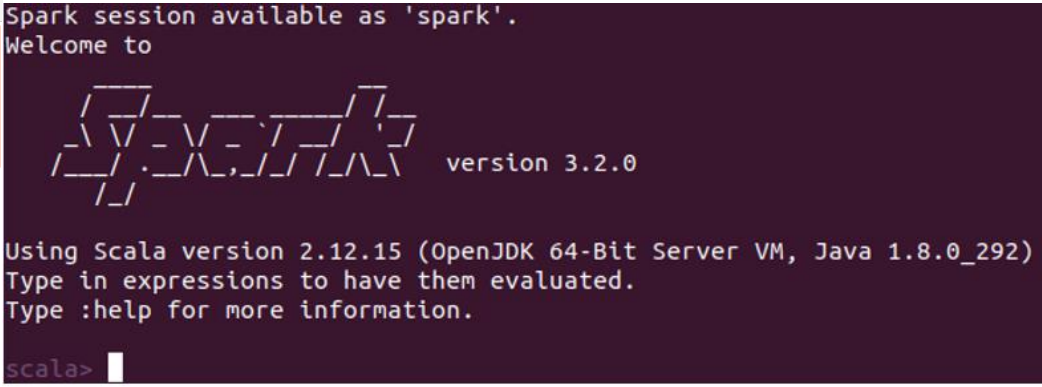
HTTP

https://dlcdn.apache.org/spark/spark-3.2.0/spark-3.2.0-bin-hadoop3.2.tgz

BACKUP SITE

https://dlcdn.apache.org/spark/spark-3.2.0/spark-3.2.0-bin-hadoop3.2.tgz

VERIFY THE INTEGRITY OF THE FILES



```
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 3.2.0
      /_/

Using Scala version 2.12.15 (OpenJDK 64-Bit Server VM, Java 1.8.0_292)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

```
tar -xvzf spark-3.1.1-bin-hadoop2.7.tgz
cd spark-3.1.1-bin-hadoop2.7/bin
source ~/.profile   vi  ~/.bashrc

spark-shell          ./spark shell
```

Resilient Distributed Datasets (RDD):

The Spark log messages indicated “Spark context available as sc.” This is a reference to the **Spark Context**, which coordinates the execution of Spark jobs on the cluster. Go ahead and type **sc** at the command line:

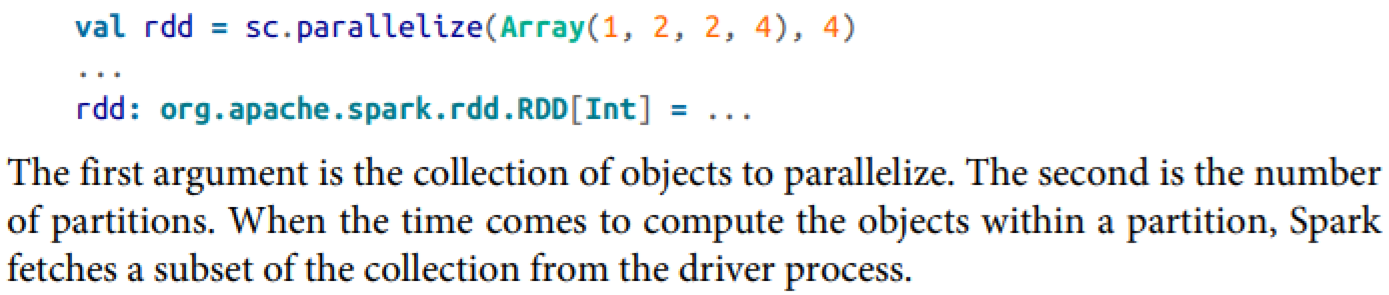
The SparkContext has a long list of methods, but the ones that we’re going to use most often allow us to create Resilient Distributed Datasets, or RDDs.

An RDD is Spark’s **fundamental abstraction for representing a collection of objects that can be distributed across multiple machines in a cluster**. **There are two ways to create an RDD in Spark:**

- Using the SparkContext to create an RDD from an external data source, like a file in HDFS, a database table via JDBC, or a local collection of objects that we create in the Spark shell

- Performing a transformation on one or more existing RDDs, like filtering records, aggregating records by a common key, or joining multiple RDDs together

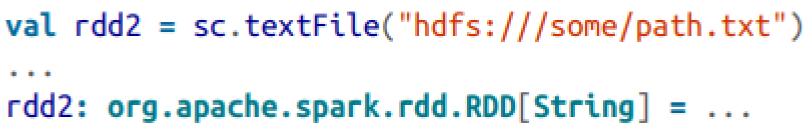
One of the simplest ways to create an RDD is to **use the parallelize method on SparkContext with a local collection of objects:**



```
val rdd = sc.parallelize(Array(1, 2, 2, 4), 4)
...
rdd: org.apache.spark.rdd.RDD[Int] = ...
```

The first argument is the collection of objects to parallelize. The second is the number of partitions. When the time comes to compute the objects within a partition, Spark fetches a subset of the collection from the driver process.

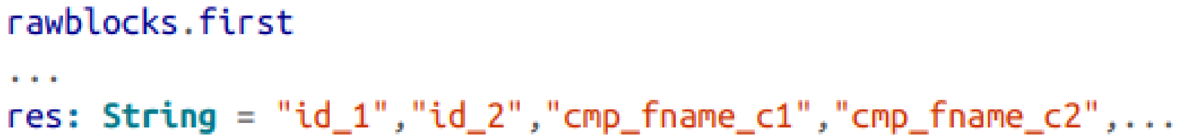
To create an RDD from a text file or directory of text files residing in a distributed filesystem like HDFS, we can pass the name of the file or directory to the textFile method:



```
val rdd2 = sc.textFile("hdfs:///some/path.txt")
...
rdd2: org.apache.spark.rdd.RDD[String] = ...
```

Bringing Data from the Cluster to the Client

RDDs have a number of methods that allow us to read data from the cluster into the Scala REPL on our client machine. Perhaps the simplest of these is first, which returns the first element of the RDD into the client:

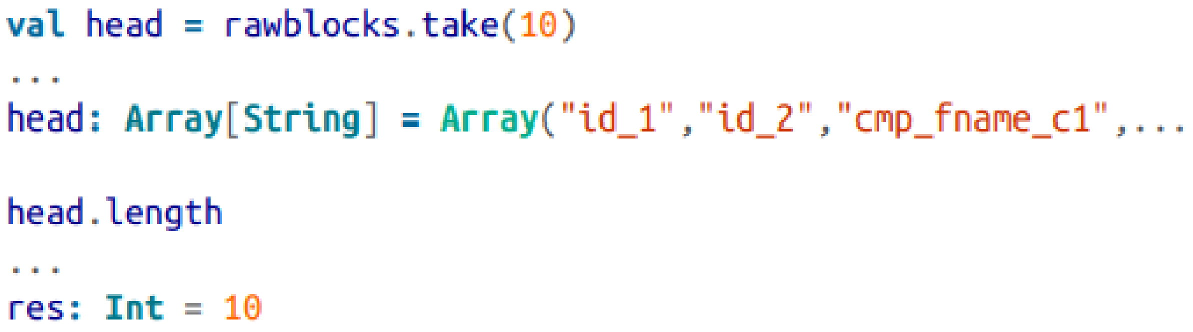


```
rawblocks.first
...
res: String = "id_1","id_2","cmp_fname_c1","cmp_fname_c2",...
```

The first method can be useful for sanity checking a data set, but we’re generally interested in bringing back larger samples of an RDD into the client for analysis.

When we know that an RDD only contains a small number of records, we can use the collect method to return all the contents of an RDD to the client as an array. Because we don’t know how big the linkage data set is just yet, we’ll hold off on doing this right now.

We can strike a balance between first and collect with the take method, which allows us to read a given number of records into an array on the client. Let's use take to get the first 10 lines from the linkage data set

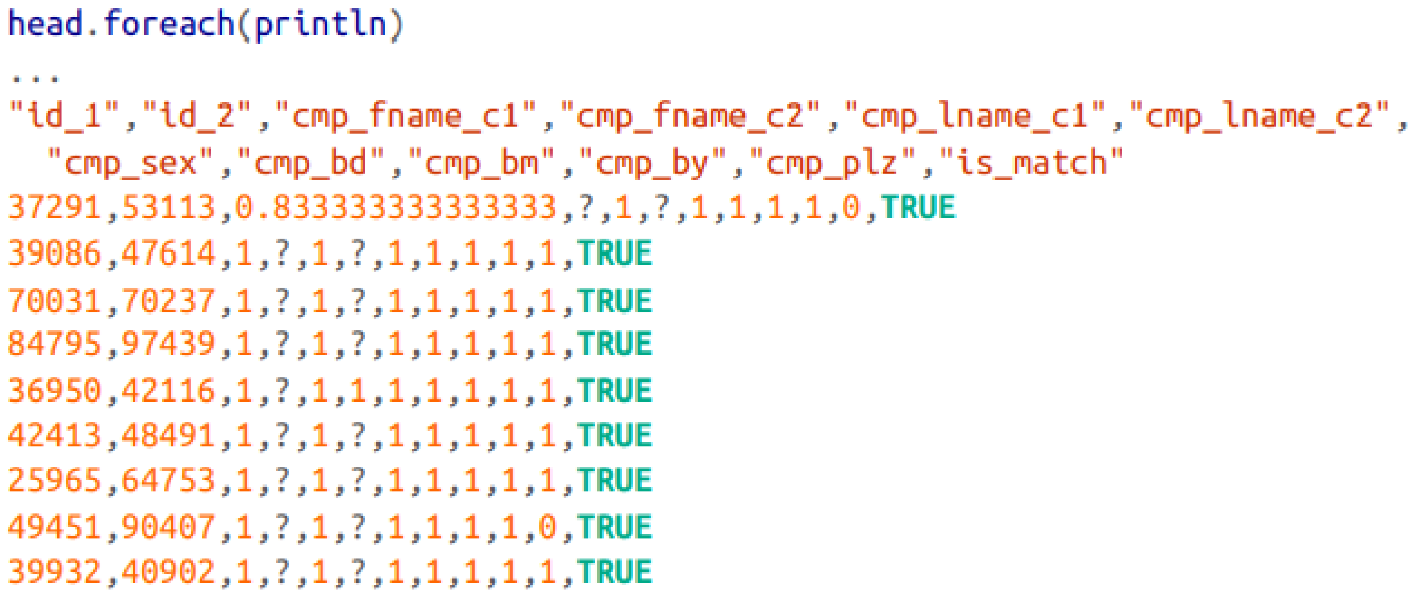


```
val head = rawblocks.take(10)
...
head: Array[String] = Array("id_1","id_2","cmp_fname_c1",...

head.length
...
res: Int = 10
```

The raw form of data returned by the Scala REPL can be somewhat hard to read, especially for arrays that contain more than a handful of elements.

To make it easier to read the contents of an array, we can use the foreach method in conjunction with println to print out each value in the array on its own line:



```
head.foreach(println)
...
"id_1","id_2","cmp_fname_c1","cmp_fname_c2","cmp_lname_c1","cmp_lname_c2",
  "cmp_sex","cmp_bd","cmp_bm","cmp_by","cmp_plz","is_match"
37291,53113,0.833333333333333,?,1,?,1,1,1,1,0,TRUE
39086,47614,1,?,1,?,1,1,1,1,1,TRUE
70031,70237,1,?,1,?,1,1,1,1,1,TRUE
84795,97439,1,?,1,?,1,1,1,1,1,TRUE
36950,42116,1,?,1,1,1,1,1,1,1,TRUE
42413,48491,1,?,1,?,1,1,1,1,1,TRUE
25965,64753,1,?,1,?,1,1,1,1,1,TRUE
49451,90407,1,?,1,?,1,1,1,1,0,TRUE
39932,40902,1,?,1,?,1,1,1,1,1,TRUE
```

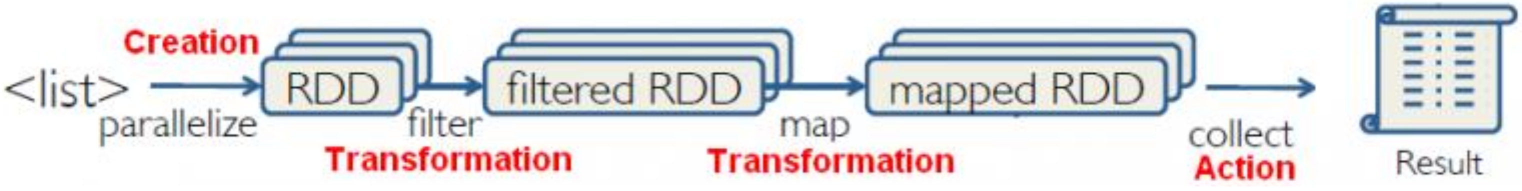
The foreach(println) pattern is one that we will frequently use in this book. It’s an example of a common functional programming pattern, where we pass one function (println) as an argument to another function (foreach) in order to perform some action.

Actions

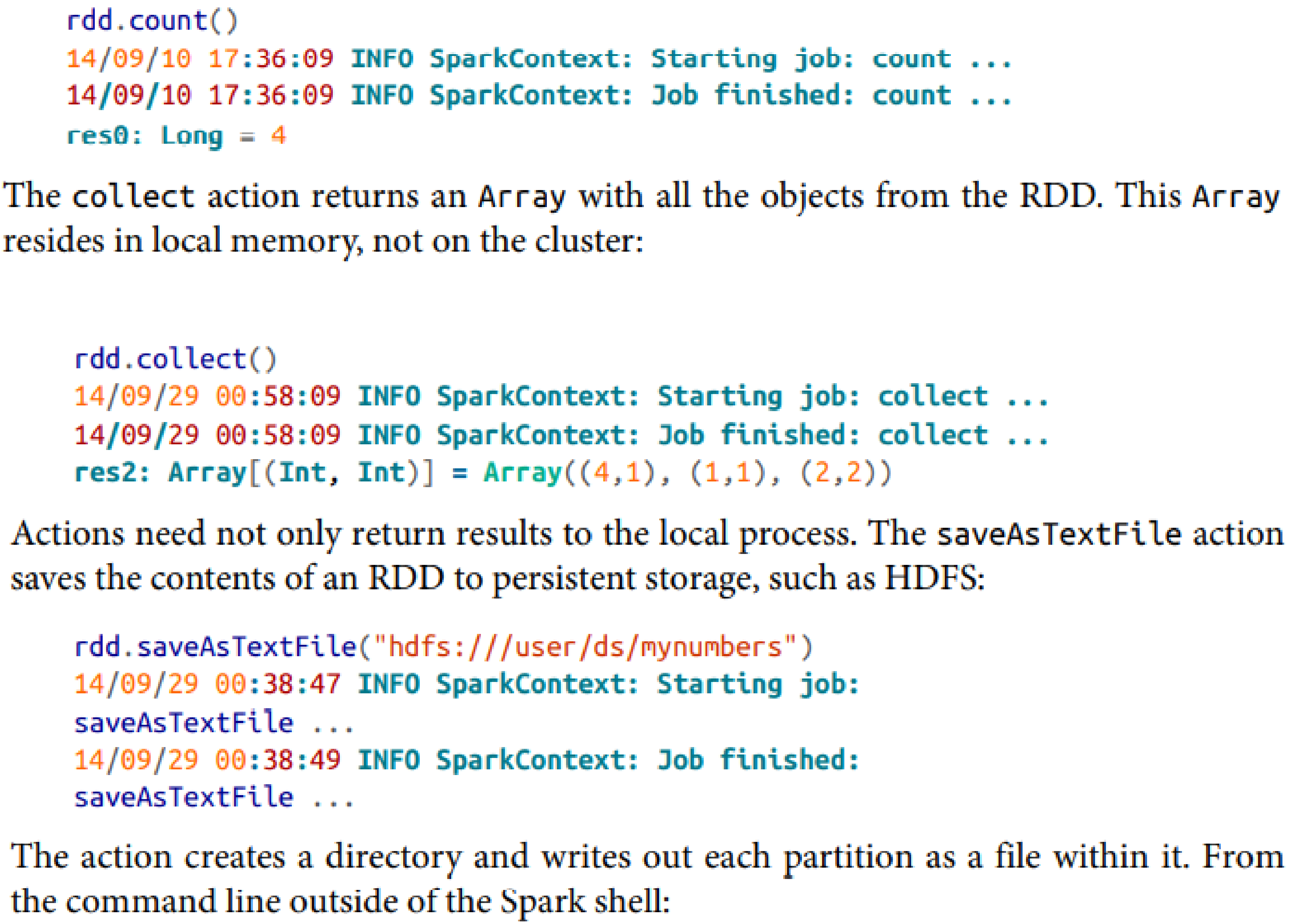
The act of creating an RDD does not cause any distributed computation to take place on the cluster. Rather, RDDs define logical data sets that are intermediate steps in a computation. Distributed computation occurs upon invoking an action on an RDD.

- **Actions return final results of RDD computations.** Actions triggers execution using lineage graph to load the data into original RDD, carry out all intermediate transformations and return final results to Driver program or write it out to file system.

- **RDD actions are operations that return the raw values,** In other words, any RDD function that returns other than RDD[T] is considered as an action in spark programming.



RDD Actions Example



```
rdd.count()
14/09/10 17:36:09 INFO SparkContext: Starting job: count ...
14/09/10 17:36:09 INFO SparkContext: Job finished: count ...
res0: Long = 4
```

The collect action returns an Array with all the objects from the RDD. This Array resides in local memory, not on the cluster:

```
rdd.collect()
14/09/29 00:58:09 INFO SparkContext: Starting job: collect ...
14/09/29 00:58:09 INFO SparkContext: Job finished: collect ...
res2: Array[(Int, Int)] = Array((4,1), (1,1), (2,2))
```

Actions need not only return results to the local process. The saveAsTextFile action saves the contents of an RDD to persistent storage, such as HDFS:

```
rdd.saveAsTextFile("hdfs:///user/ds/mynumbers")
14/09/29 00:38:47 INFO SparkContext: Starting job:
saveAsTextFile ...
14/09/29 00:38:49 INFO SparkContext: Job finished:
saveAsTextFile ...
```

The action creates a directory and writes out each partition as a file within it. From the command line outside of the Spark shell:

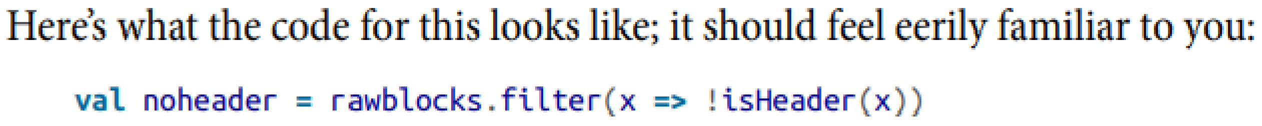
```
head.foreach(println)
...
"id_1","id_2","cmp_fname_c1","cmp_fname_c2","cmp_lname_c1","cmp_lname_c2",
  "cmp_sex","cmp_bd","cmp_bm","cmp_by","cmp_plz","is_match"
37291,53113,0.83333333333333,?,1,?,1,1,1,1,0,TRUE
39086,47614,1,?,1,?,1,1,1,1,1,TRUE
70031,70237,1,?,1,?,1,1,1,1,1,TRUE
84795,97439,1,?,1,?,1,1,1,1,1,TRUE
36950,42116,1,?,1,1,1,1,1,1,1,TRUE
42413,48491,1,?,1,?,1,1,1,1,1,TRUE
25965,64753,1,?,1,?,1,1,1,1,1,TRUE
49451,90407,1,?,1,?,1,1,1,1,0,TRUE
39932,40902,1,?,1,?,1,1,1,1,1,TRUE
```

The foreach(println) pattern is one that we will frequently use in this book. It's an example of a common functional programming pattern, where we pass one function (println) as an argument to another function (foreach) in order to perform some action. This kind of programming style will be familiar to data scientists who have worked with R and are used to processing vectors and lists by avoiding for loops and instead using higher-order functions like apply and lapply. Collections in Scala are similar to lists and vectors in R in that we generally want to avoid for loops and instead process the elements of the collection using higher-order functions.

Shipping Code from the Client to the Cluster

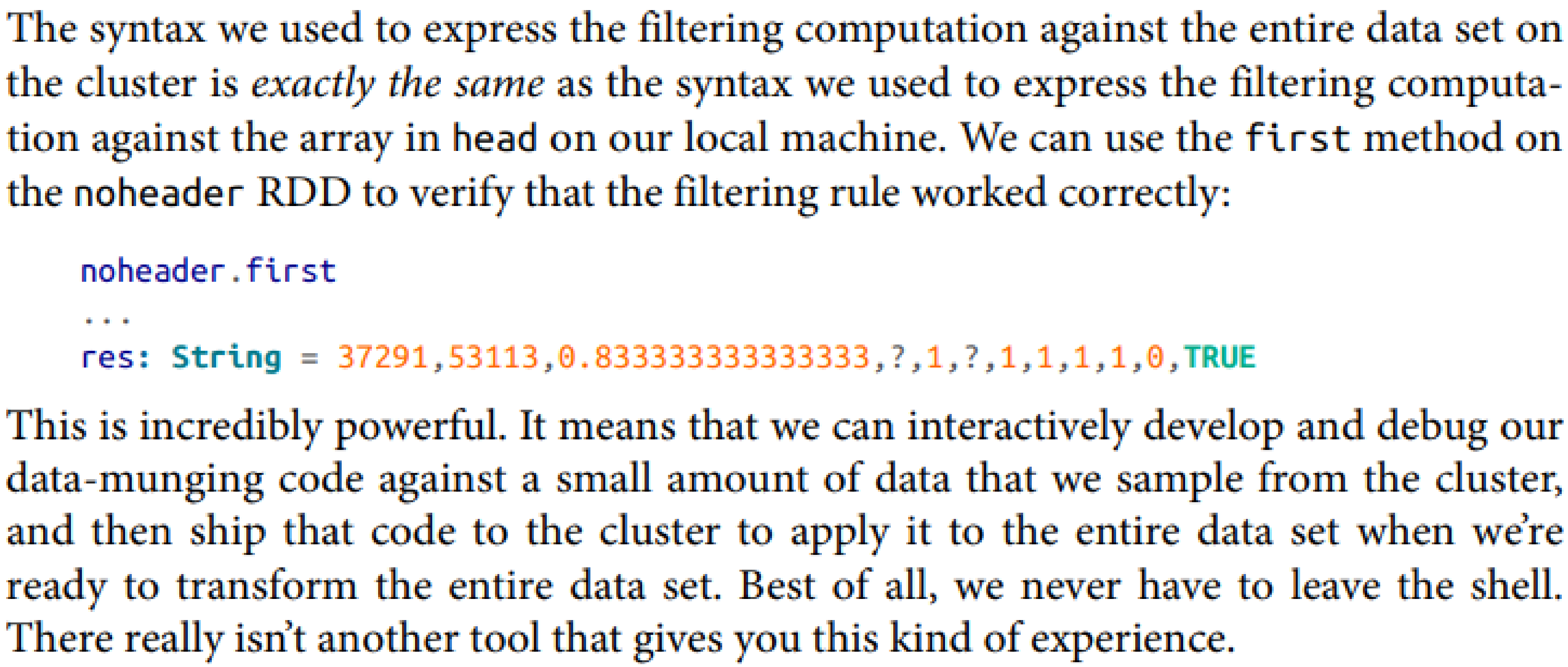
We just saw a **wide variety of ways to write and apply functions to data in Scala**. All the code that we executed was done against the data inside the head array, which was contained on our client machine.

- Now we're going to take the code that we just wrote and apply it to the millions of linkage records contained in our cluster and represented by the rawblocks RDD in Spark.



Here's what the code for this looks like; it should feel eerily familiar to you:

```
val noheader = rawblocks.filter(x => !isHeader(x))
```

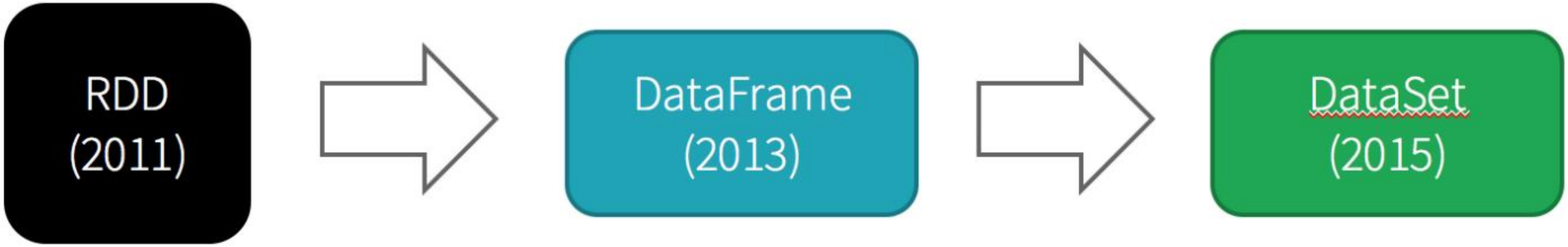


The syntax we used to express the filtering computation against the entire data set on the cluster is *exactly the same* as the syntax we used to express the filtering computation against the array in `head` on our local machine. We can use the `first` method on the `noheader` RDD to verify that the filtering rule worked correctly:

```
noheader.first
...
res: String = 37291,53113,0.833333333333333,?,1,?,1,1,1,1,0,TRUE
```

This is incredibly powerful. It means that we can interactively develop and debug our data-munging code against a small amount of data that we sample from the cluster, and then ship that code to the cluster to apply it to the entire data set when we're ready to transform the entire data set. Best of all, we never have to leave the shell. There really isn't another tool that gives you this kind of experience.

History of Spark APIs



RDD
(2011)

DataFrame
(2013)

DataSet
(2015)

Distribute collection
of JVM objects

Functional Operators (map,
filter, etc.)

Distribute collection
of Row objects

Expression-based operations
and UDFs

Logical plans and optimizer

Fast/efficient internal
representations

Internally rows, externally
JVM objects

Almost the "Best of both
worlds": **type safe + fast**

But slower than DF
Not as good for interactive
analysis, especially Python

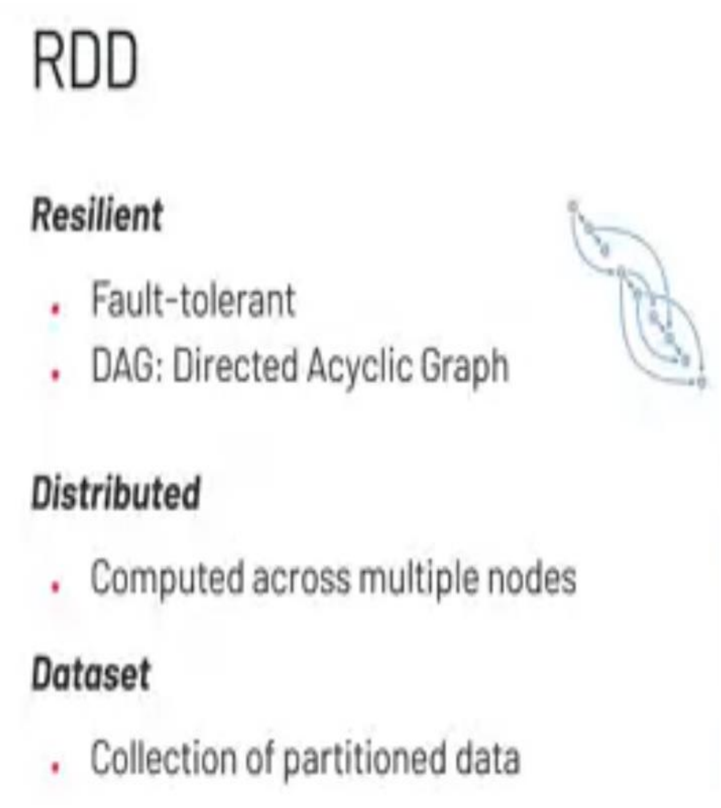
From RDDs to Data Frames

In Spark, the DataFrame is an abstraction built on top of RDDs for data sets that have a regular structure in which each record is a row made up of a set of columns, and each column has a well-defined data type.

- You can think of a data frame as the Spark analogue of a table in a relational database.

- Even though the naming convention might make you think of a data.frame object in R or a pandas.DataFrame object in Python, Spark's DataFrames are a different beast.

- **To create a data frame** for our record linkage data set, we're going to use the other object that was created for us when we started the Spark REPL, the SparkSession object named spark:



RDD

Resilient

- Fault-tolerant
- DAG: Directed Acyclic Graph

Distributed

- Computed across multiple nodes

Dataset

- Collection of partitioned data

DataFrame > RDD

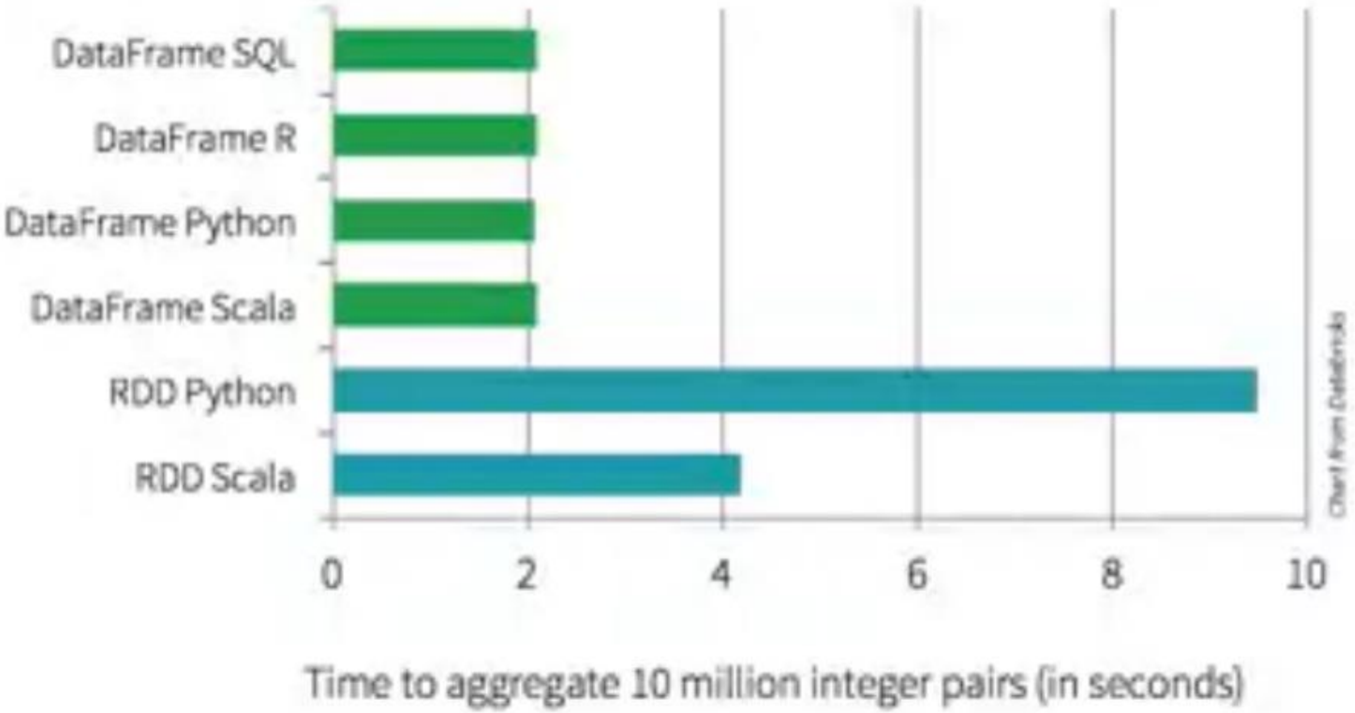
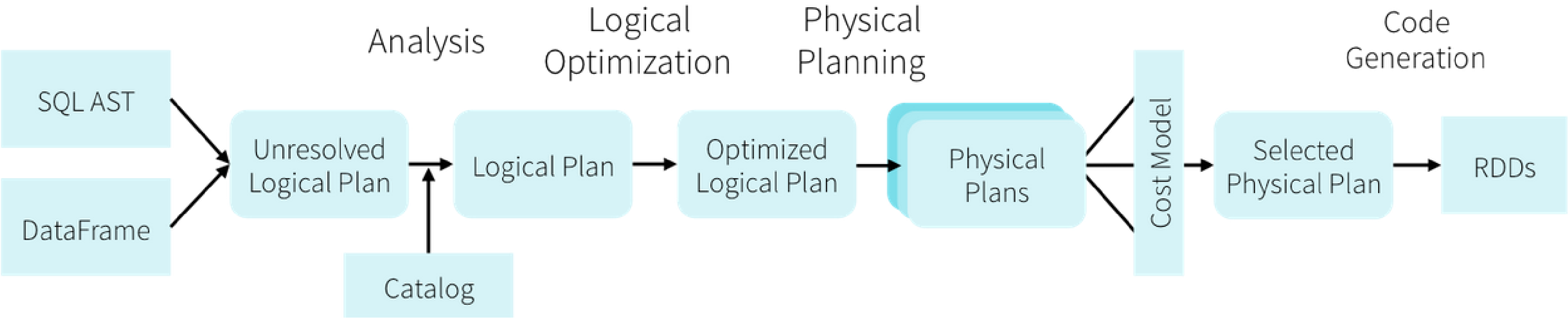


Chart from Databricks



SQL AST

DataFrame

Analysis

Unresolved Logical Plan

Catalog

Logical Optimization

Logical Plan

Physical Planning

Optimized Logical Plan

Physical Plans

Cost Model

Selected Physical Plan

Code Generation

RDDs

```
spark
...
res: org.apache.spark.sql.SparkSession = ...
```

SparkSession is a replacement for the now deprecated SQLContext object that was originally introduced in Spark 1.3. Like SQLContext, SparkSession is a wrapper around the SparkContext object, which you can access directly from the SparkSession:

```
spark.sparkContext
...
res: org.apache.spark.SparkContext = ...
```

You should see that the value of spark.sparkContext is identical to the value of the sc variable that we have been using to create RDDs thus far. To create a data frame from the SparkSession, we will use the csv method on its Reader API:

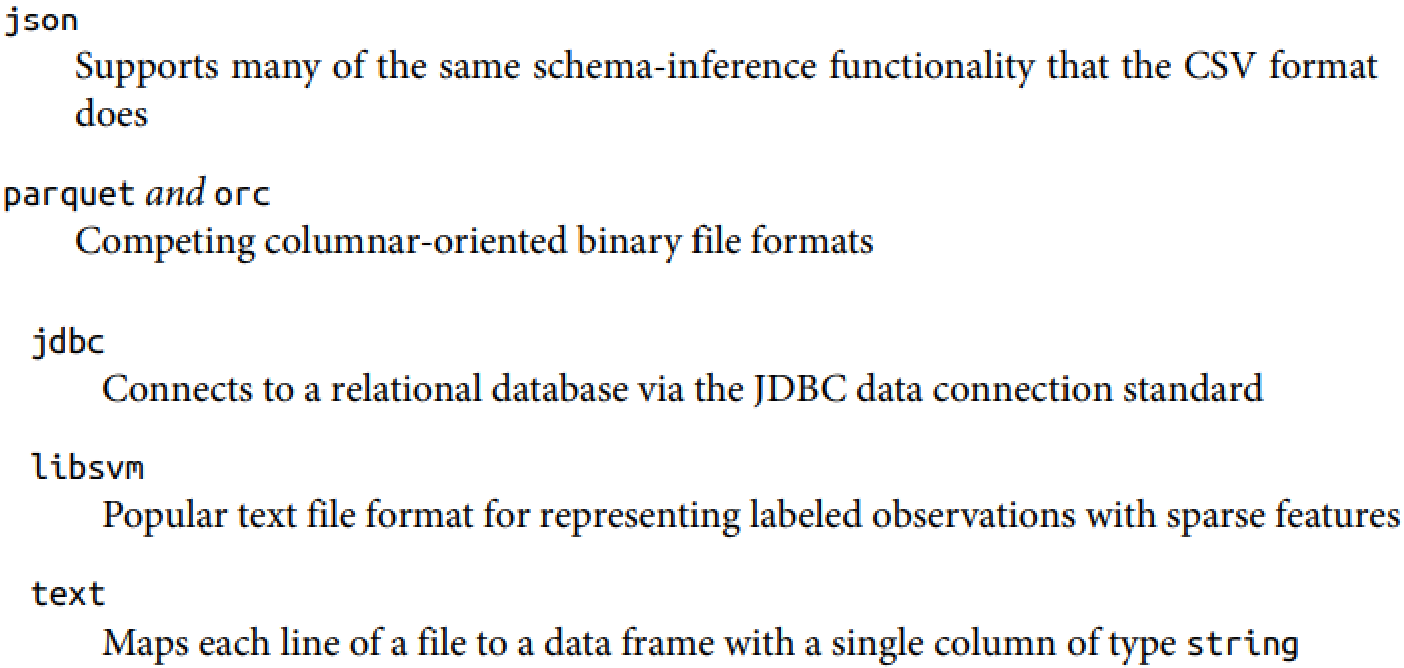
```
val prev = spark.read.csv("linkage")
...
prev: org.apache.spark.sql.DataFrame = [_c0: string, _c1: string, ...
```

By default, every column in a CSV file is treated as a string type, and the column names default to _c0, _c1, _c2, ... We can look at the head of a data frame in the shell by calling its show method:

```
prev.show()
```

Data Formats and Data Frames

Spark 2.0 ships with built-in support for reading and writing data frames in a variety of formats via the **DataFrameReader** and **DataFrameWriter** APIs. In addition to the CSV format discussed here, you can also read and write structured data from the following sources:



`json`
: Supports many of the same schema-inference functionality that the CSV format does

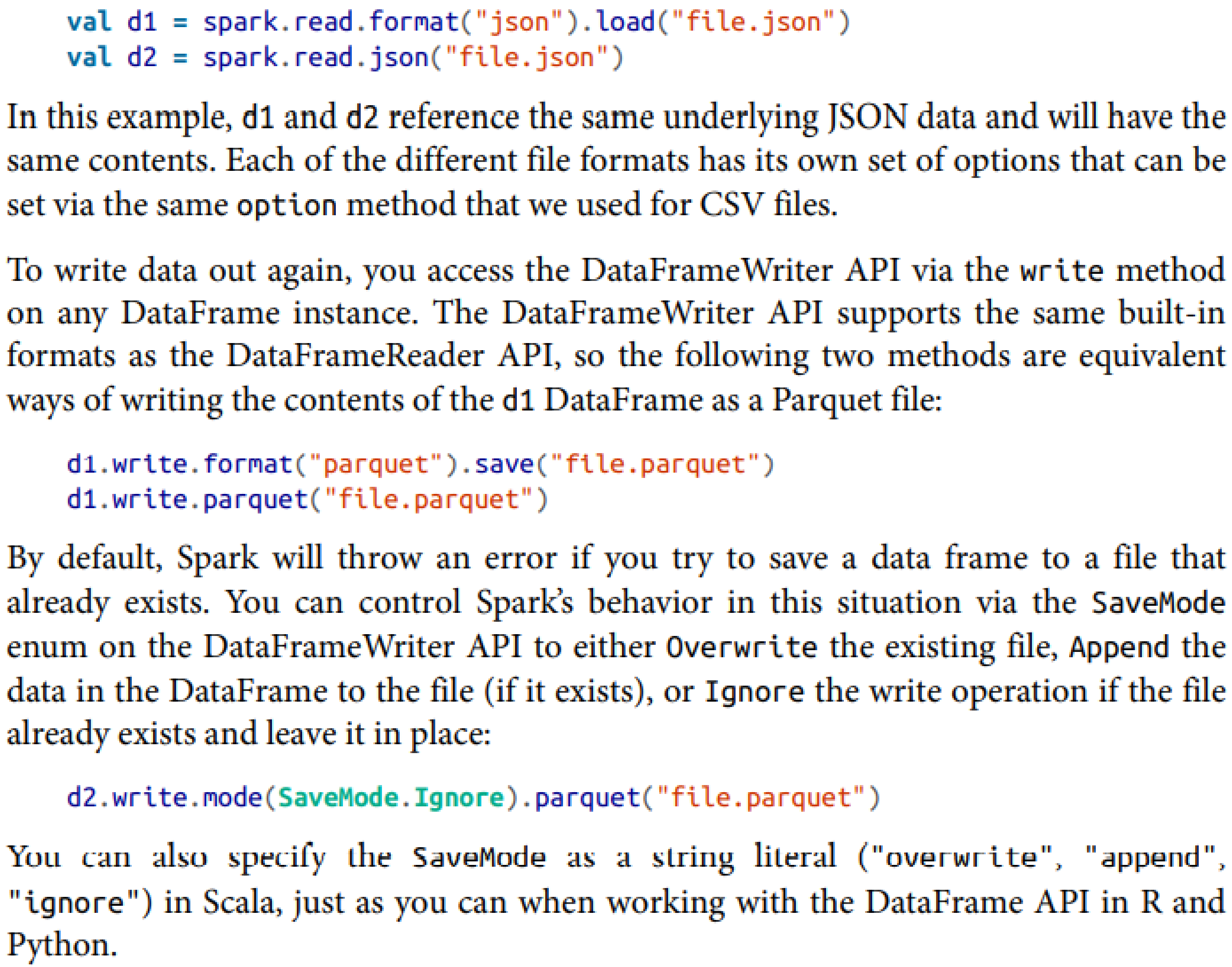
`parquet` *and* `orc`
: Competing columnar-oriented binary file formats

`jdbc`
: Connects to a relational database via the JDBC data connection standard

`libsvm`
: Popular text file format for representing labeled observations with sparse features

`text`
: Maps each line of a file to a data frame with a single column of type `string`

You **access the methods of the DataFrameReader API** by calling the read method on a SparkSession instance, and you can load data from a file using either the format and load methods, or one of the shortcut methods for built-in formats:



```
val d1 = spark.read.format("json").load("file.json")
val d2 = spark.read.json("file.json")
```

In this example, d1 and d2 reference the same underlying JSON data and will have the same contents. Each of the different file formats has its own set of options that can be set via the same option method that we used for CSV files.

To write data out again, you access the DataFrameWriter API via the write method on any DataFrame instance. The DataFrameWriter API supports the same built-in formats as the DataFrameReader API, so the following two methods are equivalent ways of writing the contents of the d1 DataFrame as a Parquet file:

```
d1.write.format("parquet").save("file.parquet")
d1.write.parquet("file.parquet")
```

By default, Spark will throw an error if you try to save a data frame to a file that already exists. You can control Spark's behavior in this situation via the SaveMode enum on the DataFrameWriter API to either Overwrite the existing file, Append the data in the DataFrame to the file (if it exists), or Ignore the write operation if the file already exists and leave it in place:

```
d2.write.mode(SaveMode.Ignore).parquet("file.parquet")
```

You can also specify the SaveMode as a string literal ("overwrite", "append", "ignore") in Scala, just as you can when working with the DataFrame API in R and Python.

Caching

Although the contents of DataFrames and RDDs are transient by default, Spark provides a mechanism for persisting the underlying data:

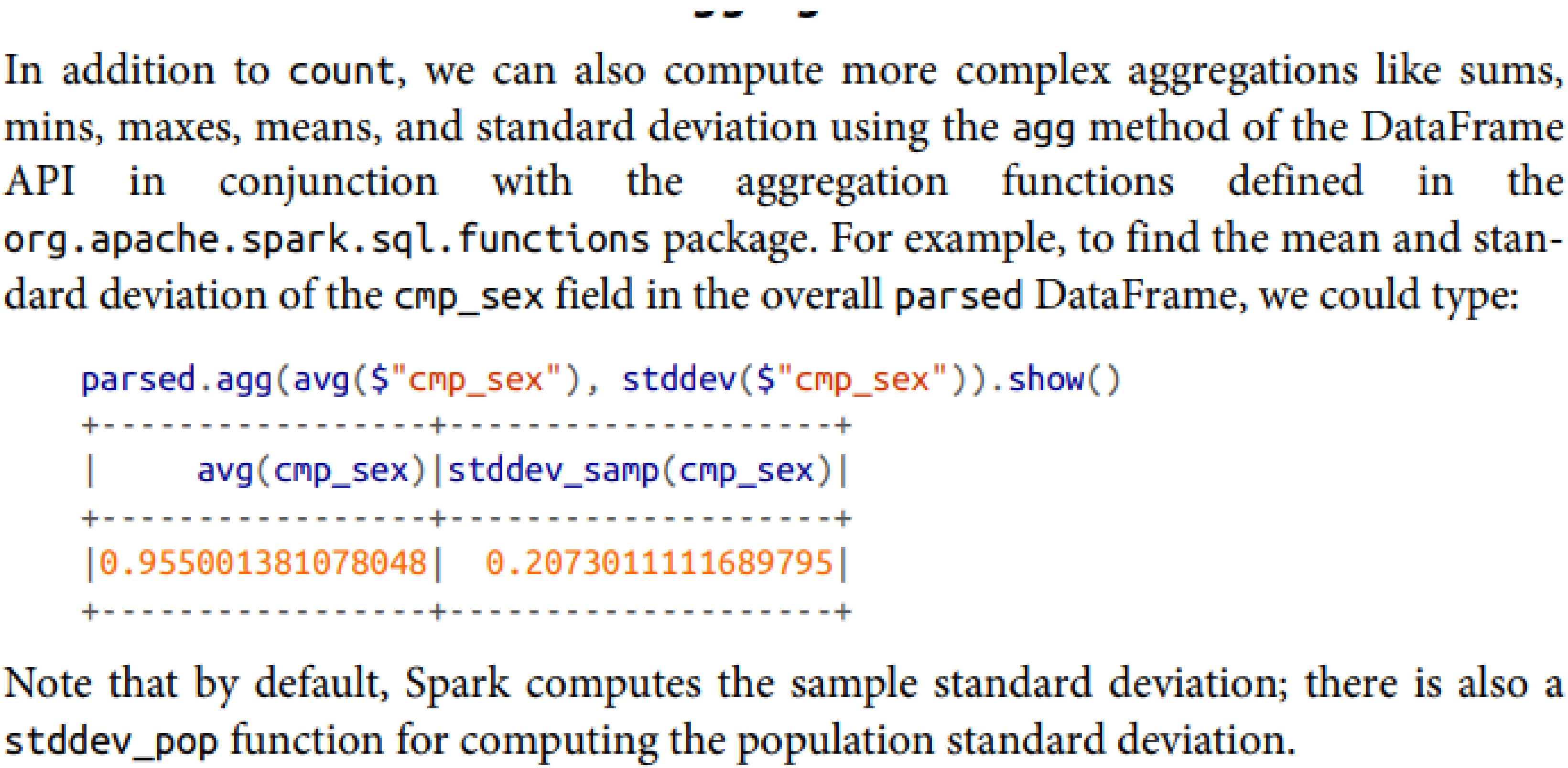
```
cached.cache()
cached.count()
cached.take(10)
```

The call to cache indicates that the contents of the DataFrame should be stored in memory the next time it's computed. In this example, the call to count computes the contents initially, and the take action returns the first 10 elements of the DataFrame as a local Array[Row]. When take is called, it accesses the cached elements of cached instead of recomputing them from their dependencies.

Spark defines a few different mechanisms, or StorageLevel values, for persisting data. cache() is shorthand for persist(StorageLevel.MEMORY), which stores the rows as unserialized Java objects. When Spark estimates that a partition will not fit in memory, it simply will not store it, and it will be recomputed the next time it's needed. This level makes the most sense when the objects will be referenced frequently and/or require low-latency access, because it avoids any serialization overhead. Its drawback is that it takes up larger amounts of memory than its alternatives. Also, holding on to many small objects puts pressure on Java's garbage collection, which can result in stalls and general slowness.

DataFrame Aggregation Functions

Apache Spark / Spark SQL Functions Spark SQL provides built-in standard Aggregate functions defines in DataFrame API, these come in handy when we need to make aggregate operations on DataFrame columns. Aggregate functions operate on a group of rows and calculate a single return value for every group.

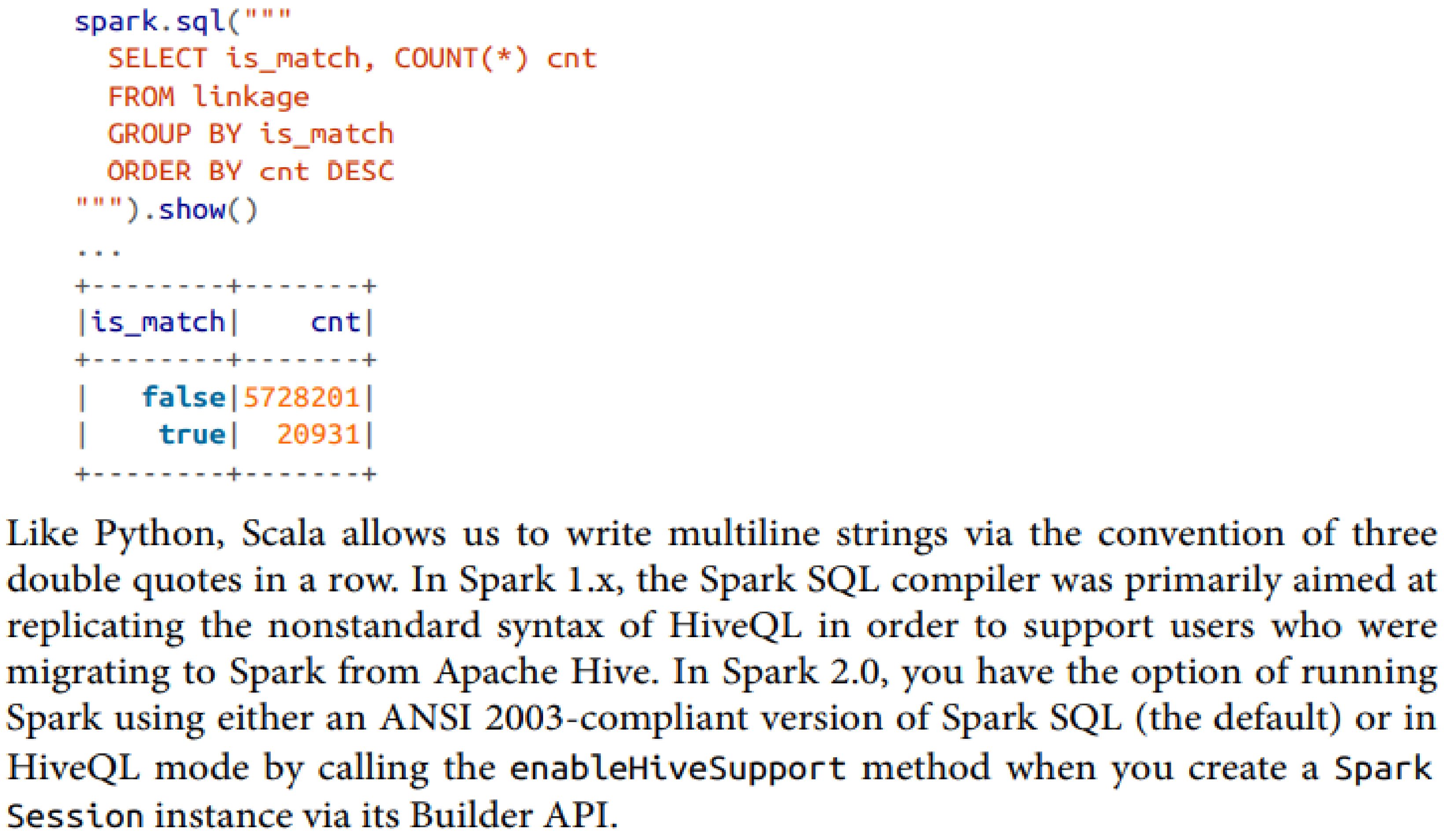


In addition to count, we can also compute more complex aggregations like sums, mins, maxes, means, and standard deviation using the agg method of the DataFrame API in conjunction with the aggregation functions defined in the org.apache.spark.sql.functions package. For example, to find the mean and standard deviation of the cmp_sex field in the overall parsed DataFrame, we could type:

```
parsed.agg(avg($"cmp_sex"), stddev($"cmp_sex")).show()
+-----------------+-------------------+
|     avg(cmp_sex)|stddev_samp(cmp_sex)|
+-----------------+-------------------+
|0.955001381078048|  0.2073011111689795|
+-----------------+-------------------+
```

Note that by default, Spark computes the sample standard deviation; there is also a stddev_pop function for computing the population standard deviation.

https://github.com/mogalmahesh/spark-basics



```
spark.sql("""
  SELECT is_match, COUNT(*) cnt
  FROM linkage
  GROUP BY is_match
  ORDER BY cnt DESC
""").show()
...
+--------+-------+
|is_match|    cnt|
+--------+-------+
|   false|5728201|
|    true|  20931|
+--------+-------+
```

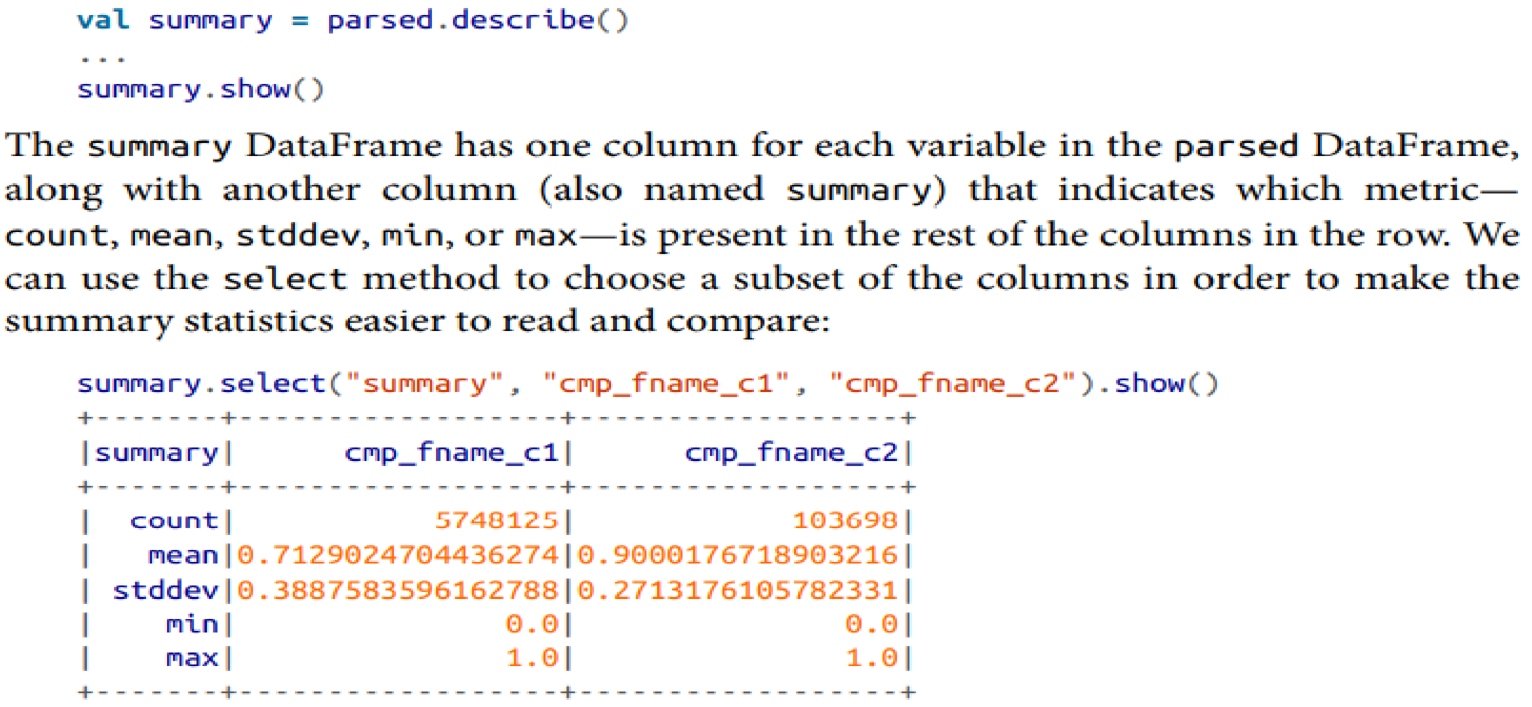
Like Python, Scala allows us to write multiline strings via the convention of three double quotes in a row. In Spark 1.x, the Spark SQL compiler was primarily aimed at replicating the nonstandard syntax of HiveQL in order to support users who were migrating to Spark from Apache Hive. In Spark 2.0, you have the option of running Spark using either an ANSI 2003-compliant version of Spark SQL (the default) or in HiveQL mode by calling the `enableHiveSupport` method when you create a `SparkSession` instance via its Builder API.

The built-in DataFrames functions provide common aggregations such as count (), countDistinct (), avg (), max (), min (), etc. While those functions are designed for DataFrames, Spark SQL also has type-safe versions for some of them in Scala and Java to work with strongly typed Datasets.

Fast Summary Statistics for DataFrames

Although there are many kinds of analyses that may be expressed equally well in SQL or with the DataFrame API, there are certain common things that we want to be able to do with data frames that can be tedious to express in SQL.

One such analysis that is especially helpful is computing the min, max, mean, and standard deviation of all the non-null values in the numerical columns of a data frame. In R, this function is named summary; and in **Spark, this function has the same name that it does in Pandas, describe:**



```
val summary = parsed.describe()
...
summary.show()
```

The summary DataFrame has one column for each variable in the parsed DataFrame, along with another column (also named summary) that indicates which metric—count, mean, stddev, min, or max—is present in the rest of the columns in the row. We can use the select method to choose a subset of the columns in order to make the summary statistics easier to read and compare:

```
summary.select("summary", "cmp_fname_c1", "cmp_fname_c2").show()
+-------+------------------+------------------+
|summary|      cmp_fname_c1|      cmp_fname_c2|
+-------+------------------+------------------+
|  count|           5748125|            103698|
|   mean|0.7129024704436274|0.9000176718903216|
| stddev|0.3887583596162788|0.2713176105782331|
|    min|               0.0|               0.0|
|    max|               1.0|               1.0|
+-------+------------------+------------------+
val matches = parsed.where("is_match = true")
val matchSummary = matches.describe()

val misses = parsed.filter($"is_match" === false)
val missSummary = misses.describe()
```

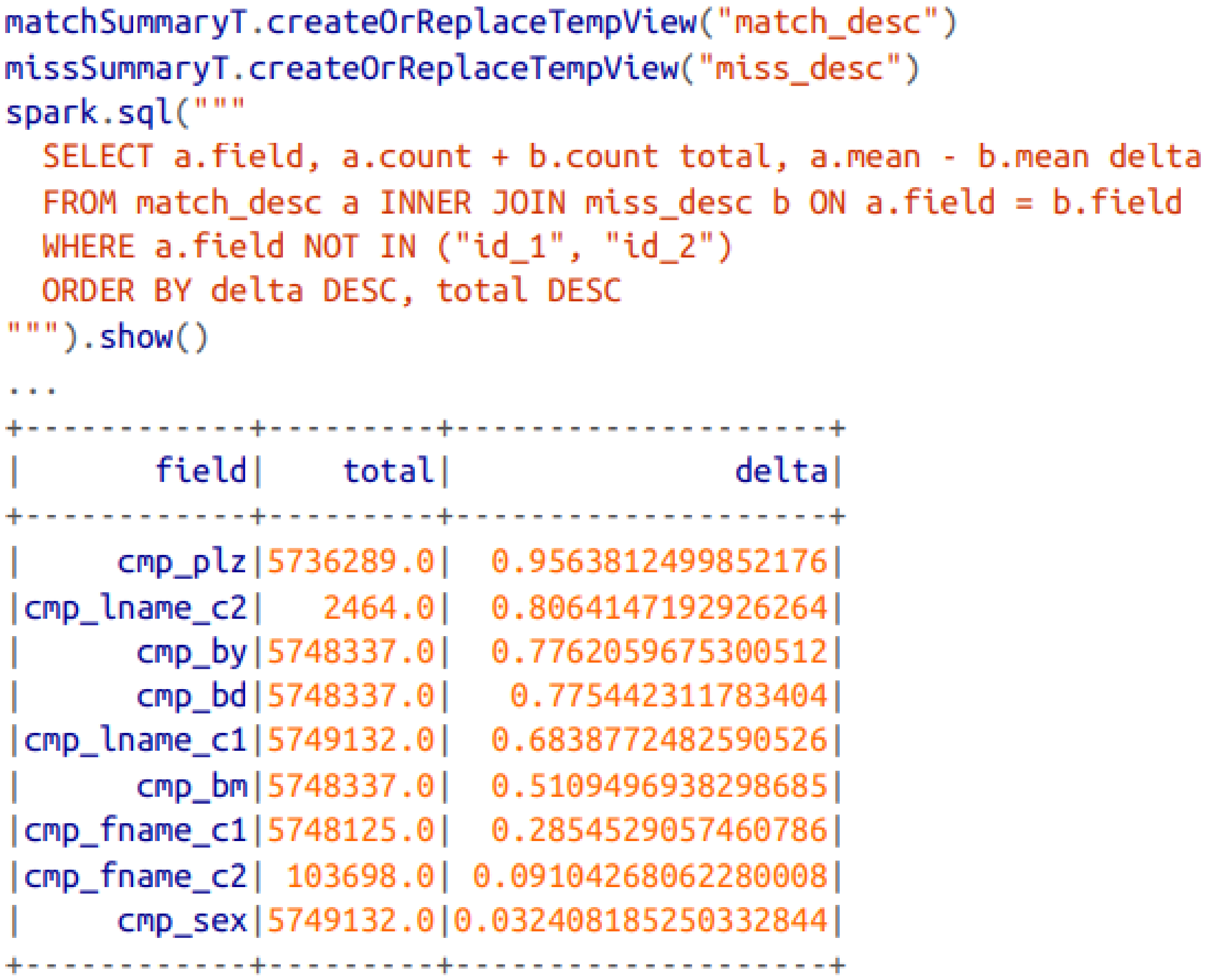
The logic inside the string we pass to the `where` function can include statements that would be valid inside a `WHERE` clause in Spark SQL. The filtering condition that uses the DataFrame API is a bit more complex: we need to use the `===` operator on the `$"is_match"` column, and we need to wrap the boolean literal `false` with the `lit` function in order to turn it into another column object that `is_match` can be compared with. Note that the `where` function is an alias for the `filter` function; we could have reversed the `where` and `filter` calls in the above snippet and everything would have worked the same way.

Joining DataFrames and Selecting Features

So far, we have only used Spark SQL and the DataFrame API to filter and aggregate the records from a data set, but we can also use these tools in order to perform joins (inner, left outer, right outer, or full outer) on DataFrames as well.

Although the Data- Frame API includes a join function, it’s often easier to express these joins using Spark SQL, especially when the tables we are joining have a large number of column names in common and we want to be able to clearly indicate which column we are referring to in our select expressions.

Let’s create temporary views for the matchSum maryT and missSummaryT DataFrames, join them on the field column, and compute some simple summary statistics on the resulting rows:



```
matchSummaryT.createOrReplaceTempView("match_desc")
missSummaryT.createOrReplaceTempView("miss_desc")
spark.sql("""
  SELECT a.field, a.count + b.count total, a.mean - b.mean delta
  FROM match_desc a INNER JOIN miss_desc b ON a.field = b.field
  WHERE a.field NOT IN ("id_1", "id_2")
  ORDER BY delta DESC, total DESC
""").show()
...
+------------+---------+-------------------+
|       field|    total|              delta|
+------------+---------+-------------------+
|     cmp_plz|5736289.0|  0.956381249985217|
|cmp_lname_c2|   2464.0|  0.806414719292626|
|      cmp_by|5748337.0|  0.776205967530051|
|      cmp_bd|5748337.0|   0.77544231178340|
|cmp_lname_c1|5749132.0|  0.683877248259052|
|      cmp_bm|5748337.0|  0.510949693829868|
|cmp_fname_c1|5748125.0|  0.285452905746078|
|cmp_fname_c2| 103698.0| 0.0910426806228000|
|     cmp_sex|5749132.0|0.03240818525033284|
+------------+---------+-------------------+
```

Model Evaluation

The final step in creating our **scoring function** is to decide on what threshold the score must exceed in order for us to predict that the two records represent a match.

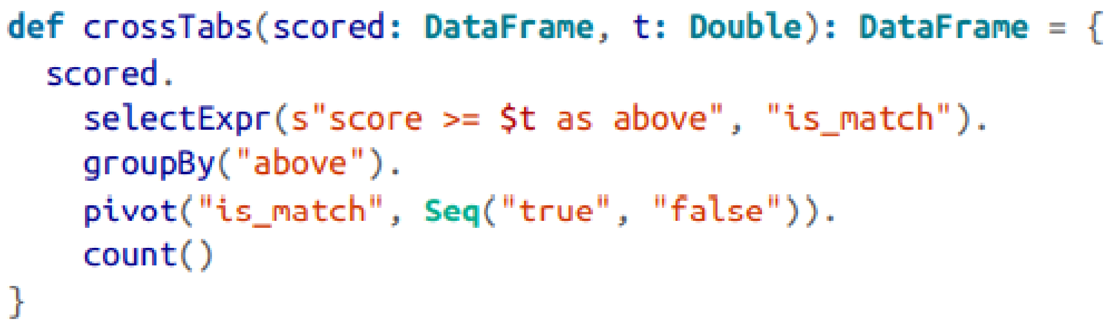
- If we set the threshold too high, then we will incorrectly mark a **matching record** as a **miss (called the false-negative rate),**

- whereas if we set the threshold too low, we will **incorrectly label misses as matches** **(the false-positive rate.)**

For any nontrivial problem, we always have to trade some false positives for some false negatives, and the question of what the threshold value should be usually comes down to the relative cost of the two kinds of errors in the situation to which the model is being applied.

To help us choose a threshold, it’s helpful to **create a 2×2 contingency table (which is sometimes called a cross tabulation, or crosstab)** that counts the number of records whose scores fall above/below the threshold value crossed with the number of records in each of those **categories that were/were not matches**.

Since we don’t know what threshold value we’re going to use yet, let’s write a function that takes the scored DataFrame and the choice of threshold as parameters and computes the crosstabs using the DataFrame API



```
def crossTabs(scored: DataFrame, t: Double): DataFrame = {
  scored.
    selectExpr(s"score >= $t as above", "is_match").
    groupBy("above").
    pivot("is_match", Seq("true", "false")).
    count()
}
```

Note that we are including the selectExpr method of the DataFrame API to dynami- cally determine the value of the field named above based on the value of the t argu- ment using Scala’s string interpolation syntax, which allows us to substitute variables by name if we preface the string literal with the letter s (yet another handy bit of Scala implicit magic). Once the above field is defined, we create the crosstab with a standard combination of the groupBy, pivot, and count methods that we used before.

```
crossTabs(scored, 4.0).show()
...
+-----+-----+-------+
|above| true|  false|
+-----+-----+-------+
| true|20871|    637|
|false|   60|5727564|
+-----+-----+-------+
```

Applying the lower threshold of 2.0, we can ensure that we capture *all* of the known matching records, but at a substantial cost in terms of false positive (top-right cell):

```
crossTabs(scored, 2.0).show()
...
+-----+-----+-------+
|above| true|  false|
+-----+-----+-------+
| true|20931| 596414|
|false| null|5131787|
+-----+-----+-------+
```

Even though the number of false positives is higher than we want, this more generous filter still removes 90% of the nonmatching records from our consideration while including every positive match. Even though this is pretty good, it's possible to do even better; see if you can find a way to use some of the other values from `MatchData` (both missing and not) to come up with a scoring function that successfully identifies every `true` match at the cost of less than 100 false positives.