Apache Spark and Delta Lake Under the Hood

APACHE Spark + DELTA LAKE

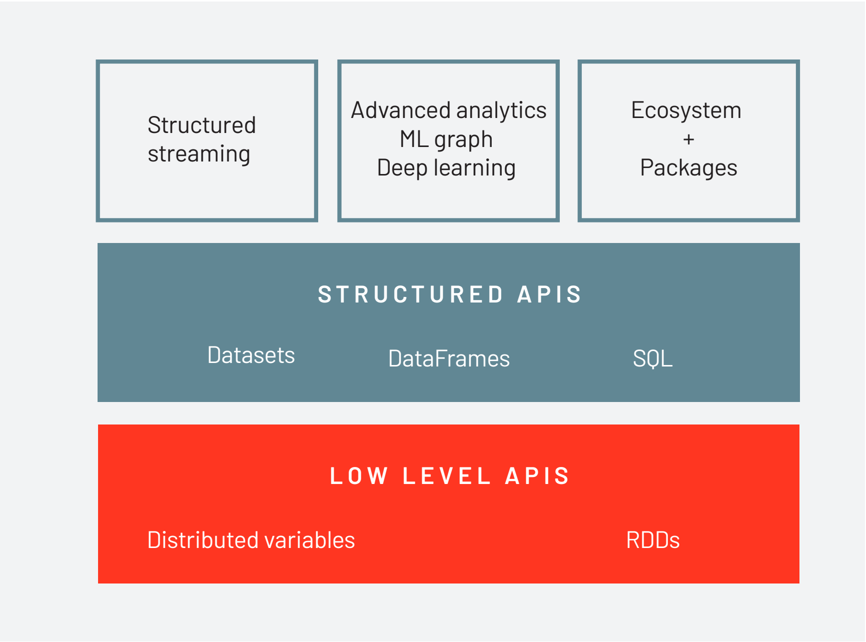
databricks

Table of Contents

Apache Spark™ has seen immense growth over the past several years, including its compatibility with Delta Lake.

Delta Lake is an open-source storage layer that sits on top of your existing data lake file storage, such as AWS S3, Azure Data Lake Storage, or HDFS. Delta Lake brings reliability, performance, and lifecycle management to data lakes. Databricks is proud to share excerpts from the Delta Lake Quickstart and the book, ***Spark: The Definitive Guide***.

CHAPTER 1: Defining Spark



WHAT IS APACHE SPARK?

Apache Spark is a unified *computing engine* and a set of *libraries* for parallel data processing on computer clusters. As of the time this writing, Spark is the most actively developed open source engine for this task; making it the de facto tool for any developer or data scientist interested in big data. Spark supports multiple widely used programming languages (Python, Java, Scala and R), includes libraries for diverse tasks ranging from SQL to streaming and machine learning, and runs anywhere from a laptop to a cluster of thousands of servers. This makes it an easy system to start with and scale up to big data processing or incredibly large scale.

◀ Here's a simple illustration of all that Spark has to offer an end user.

You'll notice the boxes roughly correspond to the different parts of this book. That should really come as no surprise, our goal here is to educate you on all aspects of Spark and Spark is composed of a number of different components.

Given that you opened this book, you may already know a little bit about Apache Spark and what it can do. Nonetheless, in this chapter, we want to cover a bit about the overriding philosophy behind Spark, as well as the context it was developed in (why is everyone suddenly excited about parallel data processing?) and its history. We will also cover the first few steps to running Spark.

Apache Spark's Philosophy

Let's break down our description of Apache Spark — a unified computing engine and set of libraries for big data — into its key components.

1. UNIFIED: Spark's key driving goal is to offer a *unified* platform for writing big data applications. What do we mean by unified? Spark is designed to support a wide range of data analytics tasks, ranging from simple data loading and SQL queries to machine learning and streaming computation, over the same *computing engine* and with a consistent set of *APIs*. The main insight behind this goal is that real-world data analytics tasks — whether they are interactive analytics in a tool such as a Jupyter notebook, or traditional software development for production applications — tend to combine many different processing types and libraries. Spark's unified nature makes these tasks both easier and more efficient to write. First, Spark provides consistent, composable APIs that can be used to build an application out of smaller pieces or out of existing libraries, and makes it easy for you to write your own analytics libraries on top. However, composable APIs are not enough: Spark's APIs are also designed to enable *high performance* by optimizing across the different libraries and functions composed together in a user program. For example, if you load data using a SQL query and then evaluate a machine learning model over it using Spark's ML library, the engine can combine these steps into one scan over the data. The combination of general APIs and high-performance execution no matter how you combine them makes Spark a powerful platform for interactive and production applications.

Spark's focus on defining a unified platform is the same idea behind unified platforms in other areas of software. For example, data scientists benefit from a unified set of libraries (e.g., Python or R) when doing modeling, and web developers benefit from unified frameworks such as Node.js or Django. Before Spark, no open source systems tried to provide this type of unified engine for parallel data processing, meaning that users had to stitch together an application out of multiple APIs and systems. Thus, Spark quickly became the standard for this type of development. Over time, Spark has continued to expand its built-in APIs to cover more workloads. At the same time the project's developers have continued to refine its theme of a unified engine. In particular, one major focus of this book will be the "structured APIs" (DataFrames, Datasets and SQL) that were finalized in Spark 2.0 to enable more powerful optimization under user applications.

2. COMPUTING ENGINE: At the same time that Spark strives for unification, Spark carefully limits its scope to a *computing engine*. By this, we mean that Spark only handles loading data from storage systems and performing computation on it, not permanent storage as the end itself. Spark can be used with a wide variety of persistent storage systems, including cloud storage systems such as Azure Storage and Amazon S3, distributed file systems such as Apache Hadoop, key-value stores such as Apache Cassandra, and message buses such as Apache Kafka. However, Spark neither stores data long-term itself, nor favors one of these. The key motivation here is that most data already resides in a mix of storage systems. Data is expensive to move so Spark focuses on performing computations over the data, no matter where it resides. In user-facing APIs, Spark works hard to make these storage systems look largely similar so that applications do not have to worry about where their data is.

Spark's focus on computation makes it different from earlier big data software platforms such as Apache Hadoop. Hadoop included both a storage system (the Hadoop file system, designed for low-cost storage over clusters of commodity servers) and a computing system (MapReduce), which were closely integrated together. However, this choice makes it hard to run one of the systems without the other, or even more importantly, to write applications that access data stored anywhere else. While Spark runs well on Hadoop storage, it is also now used broadly in environments where the Hadoop architecture does not make sense, such as the public cloud (where storage can be purchased separately from computing) or streaming applications.

3. LIBRARIES: Spark's final component is its libraries, which build on its design as a unified engine to provide a unified API for common data analysis tasks. Spark supports both standard libraries that ship with the engine, and a wide array of external libraries published as third-party packages by the open source communities. Today, Spark's standard libraries are actually the bulk of the open source project: the Spark core engine itself has changed little since it was first released, but the libraries have grown to provide more and more types of functionality. Spark includes libraries for SQL and structured data (Spark SQL), machine learning (MLlib), stream processing (Spark Streaming and the newer Structured Streaming), and graph analytics (GraphX). Beyond these libraries, there hundreds of open source external libraries ranging from connectors for various storage systems to machine learning algorithms. One index of external libraries is available at spark-packages.org.

Context: The Big Data Problem

Why do we need a new engine and programming model for data analytics in the first place? As with many trends in computer programming, this is due to changes in the economic trends that underlie computer applications and hardware.

For most of their history, computers got faster every year through processor speed increases: the new processors each year could run more instructions per second than last year's. As a result, applications also automatically got faster every year, without having to change their code. This trend led to a large and established ecosystem of applications building up over time, most of which were only designed to run on a single processor, and rode the trend of improved processor speeds to scale up to larger computations or larger volumes of data over time.

Unfortunately, this trend in hardware stopped around 2005: due to hard limits in heat dissipation, hardware developers stopped making individual processors faster, and switched towards adding more parallel CPU cores all running at the same speed. This change meant that, all of a sudden, applications needed to be modified to add parallelism in order to run faster, and already started to set the stage for new programming models such as Apache Spark.

On top of that, the technologies for *storing* and *collecting* data did not slow down appreciably in 2005, when processor speeds did. The cost to store 1 TB of data continues to drop by roughly 2x every 14 months, meaning that it is very inexpensive for organizations of all sizes to store large amounts of data. Moreover, many of the technologies for collecting data (sensors, cameras, public datasets, etc) continue to drop in cost and improve in resolution. For example, camera technology continues to improve in resolution and drop in cost per pixel every year, to the point where a 12-megapixel webcam only costs 3-4 US dollars; this has made it inexpensive to collect a wide range of visual data, whether from people filming video or automated sensors in an industrial setting. Moreover, cameras are themselves the key sensors in other data collection devices, such as telescopes and even gene sequencing machines, driving the cost of these technologies down as well.

The end result is a world where collecting data is extremely inexpensive — many organizations might even consider it negligent not to log data of possible relevance to the business — but processing it requires large, parallel computations, often on clusters of machines. Moreover, in this new world, the software developed in the past 50 years cannot automatically scale up, and neither can the traditional programming models for data processing applications, creating the need for new programming models. It is this world that Apache Spark was created for.

History of Spark

Apache Spark began in 2009 as the Spark research project at UC Berkeley, which was first published in a research paper in 2010 by **Matei Zaharia**, **Mosharaf Chowdhury**, **Michael Franklin**, **Scott Shenker** and **Ion Stoica** of the *UC Berkeley AMPlab*. At the time, Hadoop MapReduce was the dominant parallel programming engine for clusters, being the first open source system to tackle data-parallel processing on clusters of thousands of nodes. The AMPlab had worked with multiple early MapReduce users to understand the benefits and drawbacks of this new programming model, and was therefore able to synthesize a list of problems across several use cases and start designing more general computing platforms. In addition, Zaharia had also worked with Hadoop users at UC Berkeley to understand their needs for the platform — specifically, teams that were doing large-scale machine learning using iterative algorithms that need to make multiple passes over the data.

Across these conversations, two things were clear. First, cluster computing held tremendous potential: at every organization that used MapReduce, brand new applications could be built using the existing data, and many new groups started using the system after its initial use cases. Second, however, the MapReduce engine made it both challenging and inefficient to build large applications. For example, the typical machine learning algorithm might need to make 10 or 20 passes over the data, and in MapReduce, each pass had to be written as a separate MapReduce job, which had to be launched separately on the cluster and load the data from scratch.

To address this problem, the Spark team first designed an API based on functional programming that could succinctly express multi-step applications, and then implemented it over a new engine that could perform efficient, in-memory data sharing across computation steps. They also began testing this system with both Berkeley and external users.

The first version of Spark only supported batch applications, but soon enough, another compelling use case became clear: interactive data science and ad-hoc queries. By simply plugging the Scala interpreter into Spark, the project could provide a highly usable *interactive* system for running queries on hundreds of machines. The AMPlab also quickly built on this idea to develop Shark, an engine that could run SQL queries over Spark and enable interactive use by analysts as well as data scientists. Shark was first released in 2011.

After these initial releases, it quickly became clear that the most powerful additions to Spark would be new libraries, and so the project started to follow the "standard library" approach it has today. In particular, different AMPlab groups started MLlib (Apache Spark's machine learning library), Spark Streaming, and GraphX (a graph processing API). They also ensured that these APIs would be highly interoperable, enabling writing end-to-end big data applications in the same engine for the first time.

In 2013, the project had grown to widespread use, with over 100 contributors from more than 30 organizations outside UC Berkeley. The AMPlab contributed Spark to the Apache Software Foundation as a long-term, vendor-independent home for the project. The early AMPlab team also launched a startup company, Databricks, to harden the project, joining the community of other companies and organizations contributing to Spark. Since that time, the Apache Spark community released Spark 1.0 in 2014 and Spark 2.0 in 2016, and continues to make regular releases bringing new features into the project.

Finally, Spark's core idea of composable APIs has also been refined over time. Early versions of Spark (before 1.0) largely defined this API in terms of *functional operations* — parallel operations such as maps and reduces over collections of Java objects. Starting in 1.0, the project added Spark SQL, a new API for working with *structured data* — tables with a fixed data format that is not tied to Java's in-memory representation. Spark SQL enabled powerful new optimizations across libraries and APIs by understanding both the data format and the user code that runs on it in more detail. Over time, the project added a plethora of new APIs that build on this more powerful structured foundation, including DataFrames, machine learning pipelines, and Structured Streaming, a high-level, automatically optimized streaming API. In this book, we will spend a significant amount of time explaining these next-generation APIs, most of which are marked as production ready.

The Present Day and Future of Spark

Spark has been around for a number of years at this point but continues to gain massive popularity. Many new projects within the Spark ecosystem continue to push the boundaries of what's possible with the system. For instance a streaming engine (Structured Streaming) was built and introduced into Spark in 2017. This technology is a huge apart of companies solving massive scale data challenges, from technology companies like Uber and Netflix leveraging Spark's streaming engines and machine learning tools to institutions like the Broad Institute of MIT and Harvard leveraging Spark for genetic data analysis.

Spark will continue to be a cornerstone of companies doing big data analysis for the foreseeable future, especially since Spark is still in its infancy. Any data scientist or data engineer that needs to solve big data problems needs a copy of Spark on their machine and a copy of this book on their bookshelf!

Running Spark

This book contains an abundance of Spark related code. Therefore, as a reader, it is essential that you're prepared to run the code in this book. For the most part, you'll want to run the code interactively so that you can experiment with the code. Let's go over some of your options before we get started with the coding parts of the book.

Spark can be used from Python, Java, or Scala, R, or SQL. Spark itself is written in Scala, and runs on the Java Virtual Machine (JVM) and therefore to run Spark either on your laptop or a cluster, all you need is an installation of Java 6 or newer. If you wish to use the Python API you will also need a Python interpreter (version 2.6 or newer). If you wish to use R you'll also need a version of R on your machine.

There are two versions we'll recommend using to get started with Spark, you can either download it or run it for free on the web on Databricks Community Edition. We explain both of those options below.

Downloading Spark

The first step to using Spark is to download and unpack it. Let's start by downloading a recent precompiled released version of Spark. Visit spark.apache.org/downloads, select the package type of "Pre-built for Hadoop 2.7 and later" and click "Direct Download." This will download a compressed TAR file, or tarball. The majority of this book was written during the release of Spark 2.1 and 2.2 so downloading any version 2.2 or greater should be a good starting point.

DOWNLOADING SPARK FOR YOUR HADOOP VERSION: You don't need to have Hadoop, but if you have an existing Hadoop cluster or HDFS installation, download the matching version. You can do so from spark.apache.org/downloads by selecting a different package type, but they will have slightly different filenames. We discuss how Spark runs on clusters in later chapters, but at this point we recommend running a pre-compiled binary on your local machine to start out.

NOTE | *In Spark 2.2, Spark maintainers added the ability to install Spark via pip. This is brand new at the time of this writing. This will likely be the best way to install Pyspark in the future but because it's a brand new, the community has not had a chance to test it thoroughly across a multitude of machines and environments.*

BUILDING SPARK FROM SOURCE: We won't cover this in the book but you can also build Spark from source. You can find the latest source code on GitHub or select the package type of "Source Code" when downloading.

Once you've downloaded Spark, you'll want to open up a command line prompt and unzip the package. In our case, we're unzipping Spark 2.1. The following is a code snippet that you can run on any unix style command line in order to unzip the file you downloaded from Spark and move into the directory.

```
cd ~/Downloads
tar -xf spark-2.1.0-bin-hadoop2.7.tgz
cd spark-2.1.0-bin-hadoop2.7.tgz
```

Now Spark has a large number of directories and files within the project. Don't be intimidated! We will cover the important ones in the book but for right now. Spark also has a large number of deployment modes, we will cover those at the end of the book. You will be able to run everything in this book on your local machine. So only jump to those when necessary.

Launching Spark's Consoles

The first step you should take is launching Spark and this all depends on our language familiarity. The majority of this book is written with Scala, Python, and SQL in mind and those are our recommended starting points.

LAUNCHING THE PYTHON CONSOLE: You'll need Python 2 or 3 installed in order to achieve this. From Spark's home directory, run the following code.

```
./bin/pyspark
```

Once you've done that, type `spark` and hit enter. You'll see the SparkSession (which we will cover in the next chapter.)

LAUNCHING THE SCALA CONSOLE: In order to launch the scala console, you'll need to have some java runtime installed on your machine.

```
./bin/spark-shell
```

Once you've done that, type `spark` and hit enter. You'll see the SparkSession (which we will cover in the next chapter.)

LAUNCHING THE SQL CONSOLE: Parts of this book will cover a large amount of Spark SQL. For those, you may want to launch the SQL console. We'll revisit some of the more relevant details once we actually cover these topics in the book.

```
./bin/spark-sql
```

Running Spark in the Cloud

Now for the more adventurous, it's worth running Spark on your local machine. For those that would rather have a clean, interactive notebook experience with Spark. It's worth using Databricks' Community Version. Databricks, as we mentioned above, is the company founded by the authors of Apache Spark and in an effort to continue to serve the Spark community, they released a free community edition that anyone can use to run Spark without the overhead and management of managing your own Spark cluster. Additionally, Databricks makes all of the data used in this book accessible from the Community Edition so that it's readily accessible. You can access and import all the code from this book into Community Edition by following these instructions in the github repository for this book to get started.

Data used in this book

We'll use a number of data sources in this book for our examples. You can download them from the official repository in this book by following these instructions on the github repository for this book. In short, you'll download the data, put it in a folder, and then run the code snippets in this book!

CHAPTER 2: A Gentle Introduction to Spark

Now that we took our history lesson on Apache Spark, it's time to start using it and applying it! This chapter will present a gentle introduction to Spark — we will walk through the core architecture of a cluster, Spark Application, and Spark's Structured APIs using DataFrames and SQL. Along the way we will touch on Spark's core terminology and concepts so that you are empowered start using Spark right away. Let's get started with some basic background terminology and concepts.

Spark's Basic Architecture

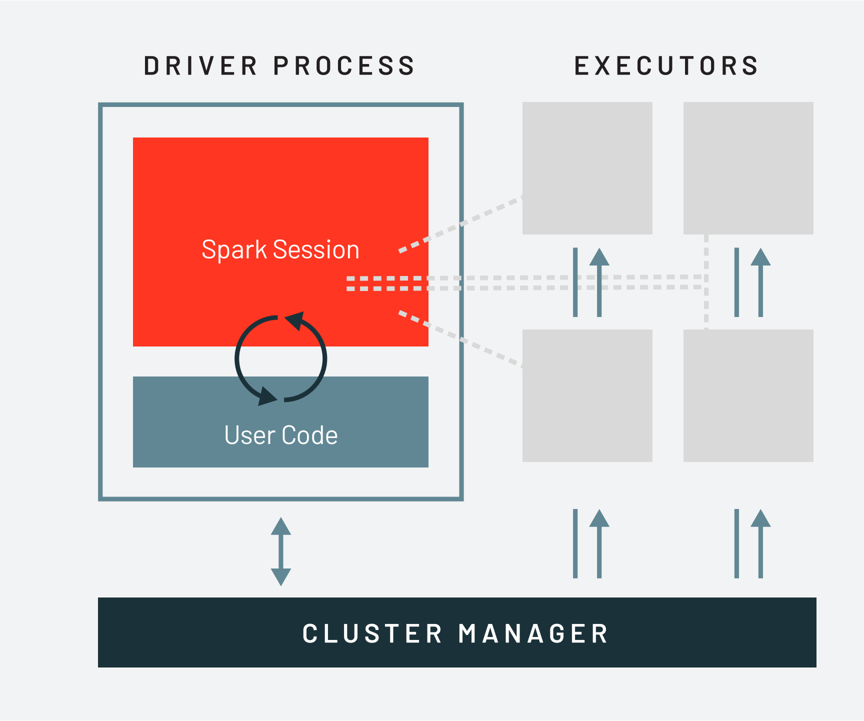
Typically when you think of a "computer" you think about one machine sitting on your desk at home or at work. This machine works perfectly well for watching movies or working with spreadsheet software. However, as many users likely experience at some point, there are some things that your computer is not powerful enough to perform. One particularly challenging area is data processing. Single machines do not have enough power and resources to perform computations on huge amounts of information (or the user may not have time to wait for the computation to finish). A cluster, or group of machines, pools the resources of many machines together allowing us to use all the cumulative resources as if they were one. Now a group of machines alone is not powerful, you need a framework to coordinate work across them. Spark is a tool for just that, managing and coordinating the execution of tasks on data across a cluster of computers.

The cluster of machines that Spark will leverage to execute tasks will be managed by a cluster manager like Spark's Standalone cluster manager, YARN, or Mesos. We then submit Spark Applications to these cluster managers which will grant resources to our application so that we can complete our work.

Spark Applications

Spark Applications consist of a *driver* process and a set of *executor* processes. The driver process runs your `main()` function, sits on a node in the cluster, and is responsible for three things: maintaining information about the Spark Application; responding to a user's program or input; and analyzing, distributing, and scheduling work across the executors (defined momentarily). The driver process is absolutely essential — it's the heart of a Spark Application and maintains all relevant information during the lifetime of the application.

The *executors* are responsible for actually executing the work that the driver assigns them. This means, each executor is responsible for only two things: executing code assigned to it by the driver and reporting the state of the computation, on that executor, back to the driver node.



The cluster manager controls physical machines and allocates resources to Spark Applications. This can be one of several core cluster managers: Spark's standalone cluster manager, YARN, or Mesos. This means that there can be multiple Spark Applications running on a cluster at the same time. We will talk more in depth about cluster managers in ***Part IV: Production Applications*** of this book.

In the previous illustration we see on the left, our driver and on the right the four executors on the right. In this diagram, we removed the concept of cluster nodes. The user can specify how many executors should fall on each node through configurations.

NOTE | *Spark, in addition to its cluster mode, also has a local mode. The driver and executors are simply processes, this means that they can live on the same machine or different machines. In local mode, these both run (as threads) on your individual computer instead of a cluster. We wrote this book with local mode in mind, so everything should be runnable on a single machine.*

As a short review of Spark Applications, the key points to understand at this point are that:

- Spark has some cluster manager that maintains an understanding of the resources available.
- The driver process is responsible for executing our driver program's commands across the executors in order to complete our task.

Now while our executors, for the most part, will always be running Spark code. Our driver can be "driven" from a number of different languages through Spark's Language APIs.

Spark's Language APIs

Spark's language APIs allow you to run Spark code from other languages. For the most part, Spark presents some core "concepts" in every language and these concepts are translated into Spark code that runs on the cluster of machines. If you use the Structured APIs (Part II of this book), you can expect all languages to have the same performance characteristics.

NOTE | *This is a bit more nuanced than we are letting on at this point but for now, it's the right amount of information for new users. In Part II of this book, we'll dive into the details of how this actually works.*

SCALA

Spark is primarily written in Scala, making it Spark's "default" language. This book will include Scala code examples wherever relevant.

JAVA

Even though Spark is written in Scala, Spark's authors have been careful to ensure that you can write Spark code in Java. This book will focus primarily on Scala but will provide Java examples where relevant.

PYTHON

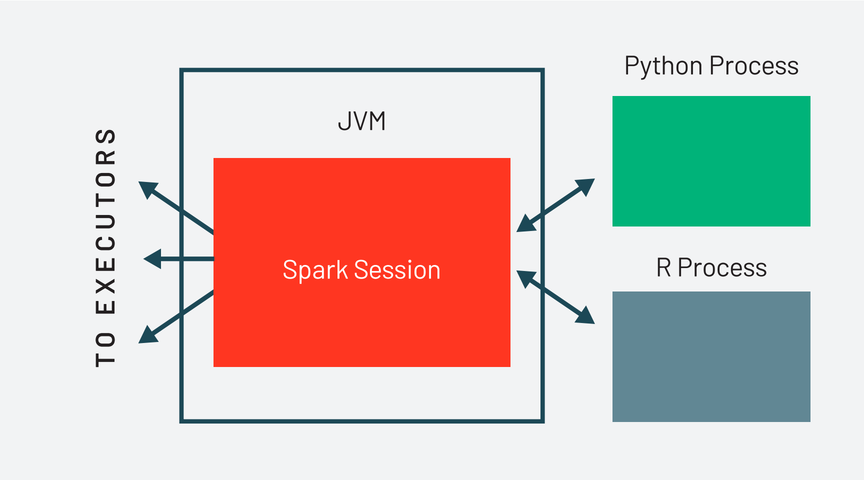
Python supports nearly all constructs that Scala supports. This book will include Python code examples whenever we include Scala code examples and a Python API exists.

SQL

Spark supports ANSI SQL 2003 standard. This makes it easy for analysts and non-programmers to leverage the big data powers of Spark. This book will include SQL code examples wherever relevant

R

Spark has two commonly used R libraries, one as a part of Spark core (SparkR) and another as an R community driven package (sparklyr). We will cover these two different integrations in Part VII: Ecosystem.



Here's a simple illustration of this relationship.

Each language API will maintain the same core concepts that we described above. There is a SparkSession available to the user, the SparkSession will be the entrance point to running Spark code. When using Spark from a Python or R, the user never writes explicit JVM instructions, but instead writes Python and R code that Spark will translate into code that Spark can then run on the executor JVMs.

Spark's APIs

While Spark is available from a variety of languages, what Spark makes available in those languages is worth mentioning. Spark has two fundamental sets of APIs: the low level "Unstructured" APIs and the higher level Structured APIs. We discuss both in this book but these introductory chapters will focus primarily on the higher level APIs.

Starting Spark

Thus far we covered the basic concepts of Spark Applications. This has all been conceptual in nature. When we actually go about writing our Spark Application, we are going to need a way to send user commands and data to the Spark Application. We do that with a SparkSession.

NOTE | *To do this we will start Spark's local mode, just like we did in the previous chapter. This means running `./bin/spark-shell` to access the Scala console to start an interactive session. You can also start Python console with `./bin/pyspark`. This starts an interactive Spark Application. There is also a process for submitting standalone applications to Spark called `spark-submit` where you can submit a precompiled application to Spark. We'll show you how to do that in the next chapter.*

When we start Spark in this interactive mode, we implicitly create a SparkSession which manages the Spark Application. When we start it through a job submission, we must go about creating it or accessing it.

The SparkSession

As discussed in the beginning of this chapter, we control our Spark Application through a driver process. This driver process manifests itself to the user as an object called the SparkSession. The SparkSession instance is the way Spark executes user-defined manipulations across the cluster. There is a one to one correspondence between a SparkSession and a Spark Application. In Scala and Python the variable is available as `spark` when you start up the console. Let's go ahead and look at the `SparkSession` in both Scala and/or Python.

```
spark
```

In **Scala**, you should see something like:

```
res0: org.apache.spark.sql.SparkSession = org.apache.spark.sql.SparkSession@27159a24
```

In **Python** you'll see something like:

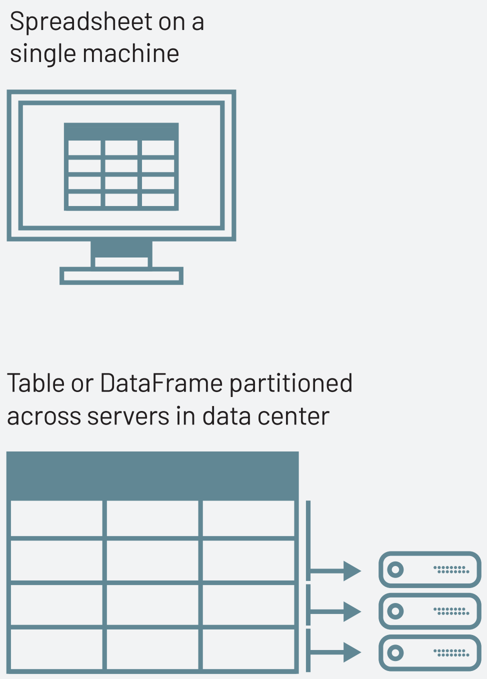
```
<pyspark.sql.session.SparkSession at 0x7efda4c1ccd0>
```

Let's now perform the simple task of creating a range of numbers. This range of numbers is just like a named column in a spreadsheet.

```
%scala
val myRange = spark.range(1000).toDF(“number”)
```

```
%python
myRange = spark.range(1000).toDF(“number”)
```

You just ran your first Spark code! We created a DataFrame with one column containing 1000 rows with values from 0 to 999. This range of number represents a *distributed collection*. When run on a cluster, each part of this range of numbers exists on a different executor. This is a Spark DataFrame.


Spreadsheet on a single machine
Table or DataFrame partitioned across servers in data center

DataFrames

A *DataFrame* is the most common Structured API and simply represents a table of data with rows and columns. The list of columns and the types in those columns the *schema*. A simple analogy would be 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. The reason for putting the data on more than one computer should be intuitive: 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. R and Python both have 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. However, since Spark has language interfaces for both Python and R, it's quite easy to convert to Pandas (Python) DataFrames to Spark DataFrames and R DataFrames to Spark DataFrames (in R).

NOTE | *Spark has several core abstractions: Datasets, DataFrames, SQL Tables, and Resilient Distributed Datasets (RDDs). These abstractions all represent distributed collections of data however they have different interfaces for working with that data. The easiest and most efficient are DataFrames, which are available in all languages. We cover Datasets at the end of Part II and RDDs in Part III of this book. The following concepts apply to all of the 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. A DataFrame's partitions represent how the data is physically distributed across your cluster of machines during execution. 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 manually (on an individual basis). We simply specify high-level transformations of data in the physical partitions and Spark determines how this work will actually execute on the cluster. Lower level APIs do exist (via the Resilient Distributed Datasets interface) and we cover those in Part III of this book.

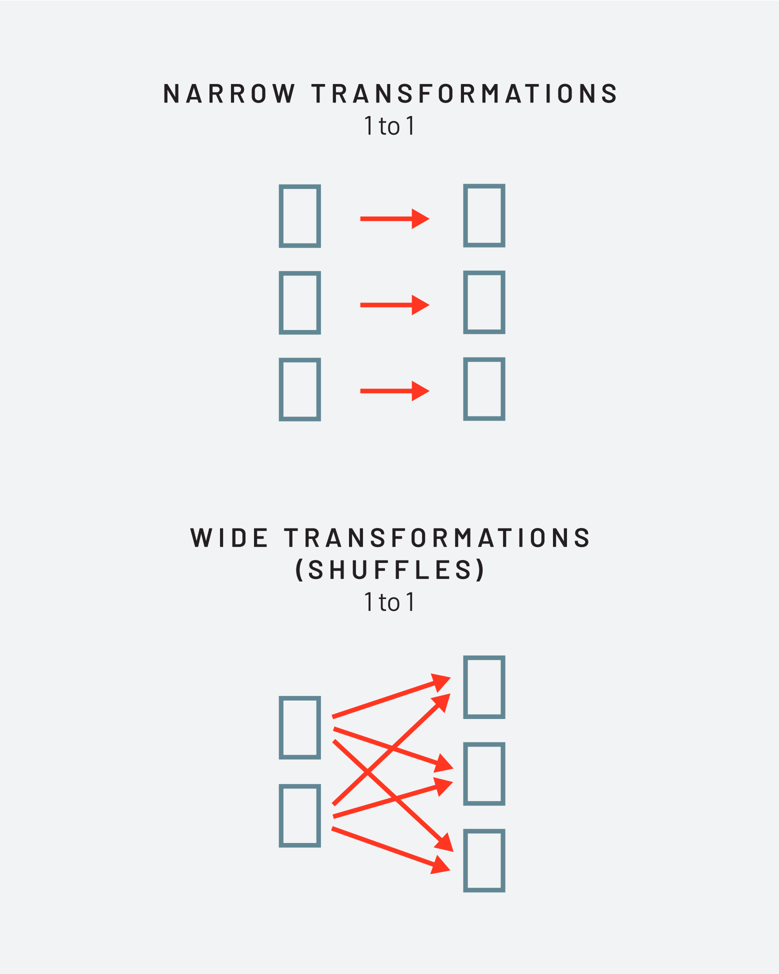
Transformations

In Spark, the core data structures are *immutable* meaning they cannot be changed once created. This might seem like a strange concept at first, if you cannot change it, how are you supposed to use it? In order to "change" a DataFrame you will have to instruct Spark how you would like to modify the DataFrame you have into the one that you want. These instructions are called *transformations*. Let's perform a simple transformation to find all even numbers in our current DataFrame.

```
%scala
val divisBy2 = myRange.where(“number % 2 = 0”)
```

```
%python
divisBy2 = myRange.where(“number % 2 = 0”)
```

You will notice that these return no output, that's because we only specified an abstract transformation and Spark will not act on transformations until we call an action, discussed shortly. Transformations are the core of how you will be expressing your business logic using Spark. There are two types of transformations, those that specify narrow dependencies and those that specify wide dependencies.



Transformations consisting of *narrow dependencies* (we'll call them narrow transformations) are those where each input partition will contribute to only one output partition. In the preceding code snippet, our `where` statement specifies a narrow dependency, where only one partition contributes to at most one output partition.

A *wide dependency* (or wide transformation) style transformation will have input partitions contributing to many output partitions. You will often hear this referred to as a *shuffle* where Spark will exchange partitions across the cluster. With narrow transformations, Spark will automatically perform an operation called pipelining on narrow dependencies, this means 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. You'll see lots of talks about shuffle optimization across the web because it's an important topic but for now all you need to understand are that there are two kinds of transformations.

We now see how transformations are simply ways of specifying different series of data manipulation. This leads us to a topic called lazy evaluation.

Lazy Evaluation

Lazy evaluation means that Spark will wait until the very last moment to execute the graph of computation instructions. In Spark, instead of modifying the data immediately when we express some operation, 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 is something called "predicate pushdown" on DataFrames. If we build a large Spark job 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 for us 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()
```

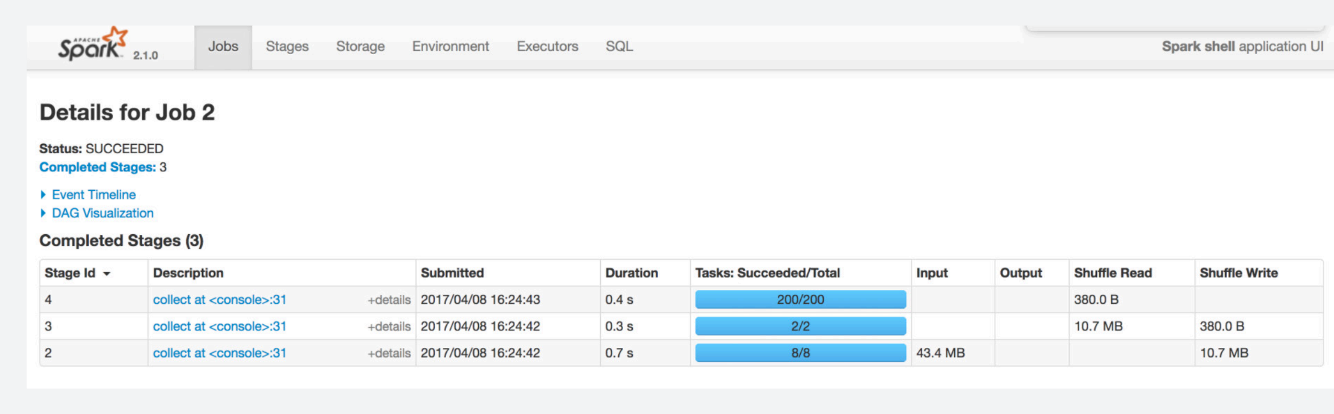
We now see a result! There are 500 numbers divisible by two from 0 to 999 (big surprise!). Now `count` is not the only action. There are three kinds of actions:

- 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 our 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 with brings our result to a native object in the respective language. We can see all of this by inspecting the Spark UI, a tool included in Spark that allows us to monitor the Spark jobs running on a cluster.

Spark UI

During Spark's execution of the previous code block, users can monitor the progress of their job through the Spark UI. The Spark UI is available on port 4040 of the driver node. If you are running in local mode this will just be the `http://localhost:4040`. The Spark UI maintains information on the state of our Spark jobs, environment, and cluster state. It's very useful, especially for tuning and debugging. In this case, we can see one Spark job with two stages and nine tasks were executed.



Details for Job 2

Status: SUCCEEDED
Completed Stages: 3

Event Timeline
DAG Visualization

Completed Stages (3)

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
4	collect at <console>:31	+details	2017/04/08 16:24:43	0.4 s	200/200			380.0 B	
3	collect at <console>:31	+details	2017/04/08 16:24:42	0.3 s	2/2			10.7 MB	380.0 B
2	collect at <console>:31	+details	2017/04/08 16:24:42	0.7 s	8/8	43.4 MB			10.7 MB

This chapter avoids the details of Spark jobs and the Spark UI, we cover the Spark UI in detail in **Part IV: Production Applications**. At this point you should understand that a Spark job represents a set of transformations triggered by an individual action and we can monitor that from the Spark UI.

An End to End Example

In the previous example, we created a DataFrame of a range of numbers; not exactly groundbreaking big data. In this section we will reinforce everything we learned previously in this chapter with a worked example and explaining step by step what is happening under the hood. We'll be using some flight data available here from the United States Bureau of Transportation statistics.

Inside of the CSV folder linked above, you'll see that we have a number of files. You will also notice a number of other folders with different file formats that we will discuss in **Part II: Reading and Writing Data**. We will focus on the CSV files.

Each file has a number of rows inside of it. Now these files are CSV files, meaning that they're a semi-structured data format with a row in the file representing a row in our future DataFrame.

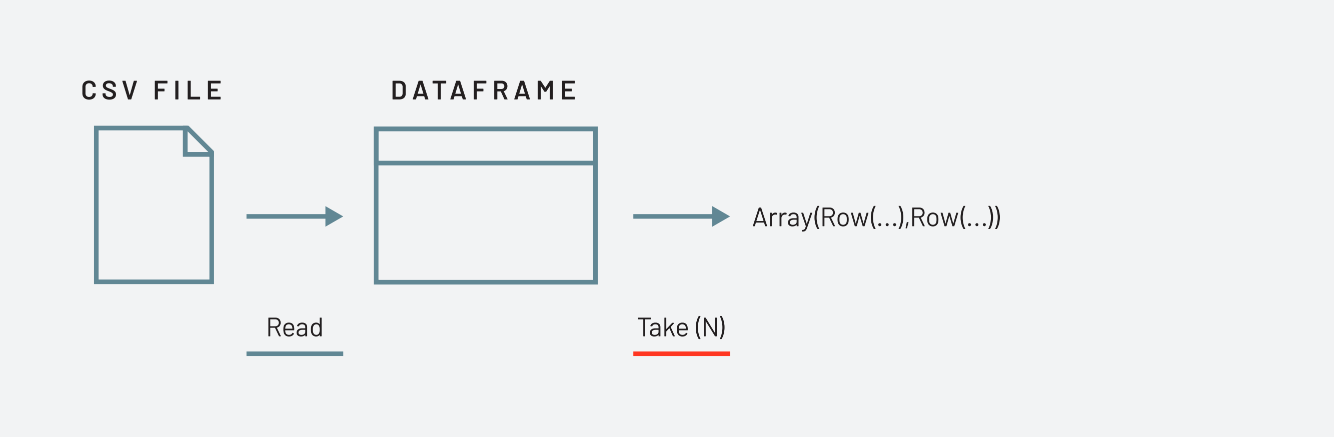
```
$ head /mnt/defg/flight-data/csv/2015-summary.csv
DEST_COUNTRY_NAME,ORIGIN_COUNTRY_NAME,count
United States,Romania,15
United States,Croatia,1
United States,Ireland,344
```

Spark includes the ability to read and write from a large number of data sources. In order to read this data in, we will use a DataFrameReader that is associated with our SparkSession. In doing so, we will specify the file format as well as any options we want to specify. In our case, we want to do something called schema inference, we want Spark to take the best guess at what the schema of our DataFrame should be. The reason for this is that CSV files are not completely structured data formats. We also want to specify that the first row is the header in the file, we'll specify that as an option too.

To get this information Spark will read in a little bit of the data and then attempt to parse the types in those rows according to the types available in Spark. You'll see that this works just fine. We also have the option of strictly specifying a schema when we read in data (which we recommend in production scenarios).

```
%scala
val flightData2015 = spark
  .read
  .option("inferSchema", "true")
  .option("header", "true")
  .csv("/mnt/defg/flight-data/csv/2015-summary.csv")
```

```
%python
flightData2015 = spark\
  .read\
  .option("inferSchema", "true")\
  .option("header", "true")\
  .csv("/mnt/defg/flight-data/csv/2015-summary.csv")
```



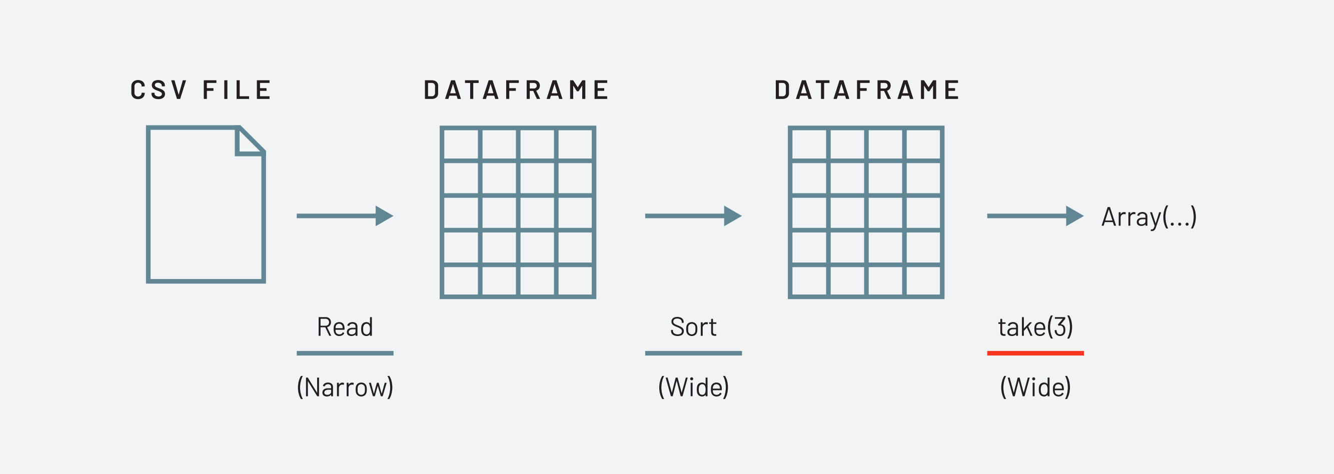
Each of these DataFrames (in Scala and Python) each have a set of columns with an unspecified number of rows. The reason the number of rows is "unspecified" is because reading data is a transformation, and is therefore a lazy operation. Spark only peeked at a couple of rows of data to try to guess what types each column should be.

If we perform the `take` action on the DataFrame, we will be able to see the same results that we saw before when we used the command line.

```
flightData2015.take(3)
Array([United States,Romania,15], [United States,Croatia...
```

Let's specify some more transformations! Now we will sort our data according to the count column which is an integer type.

NOTE | *Remember, the `sort` does not modify the DataFrame. We use the sort is a transformation that returns a new DataFrame by transforming the previous DataFrame. Let's illustrate what's happening when we call `take` on that resulting DataFrame.*



Nothing happens to the data when we call sort because it's just a transformation. However, we can see that Spark is building up a plan for how it will execute this across the cluster by looking at the `explain` plan. We can call explain on any DataFrame object to see the DataFrame's lineage (or how Spark will execute this query).

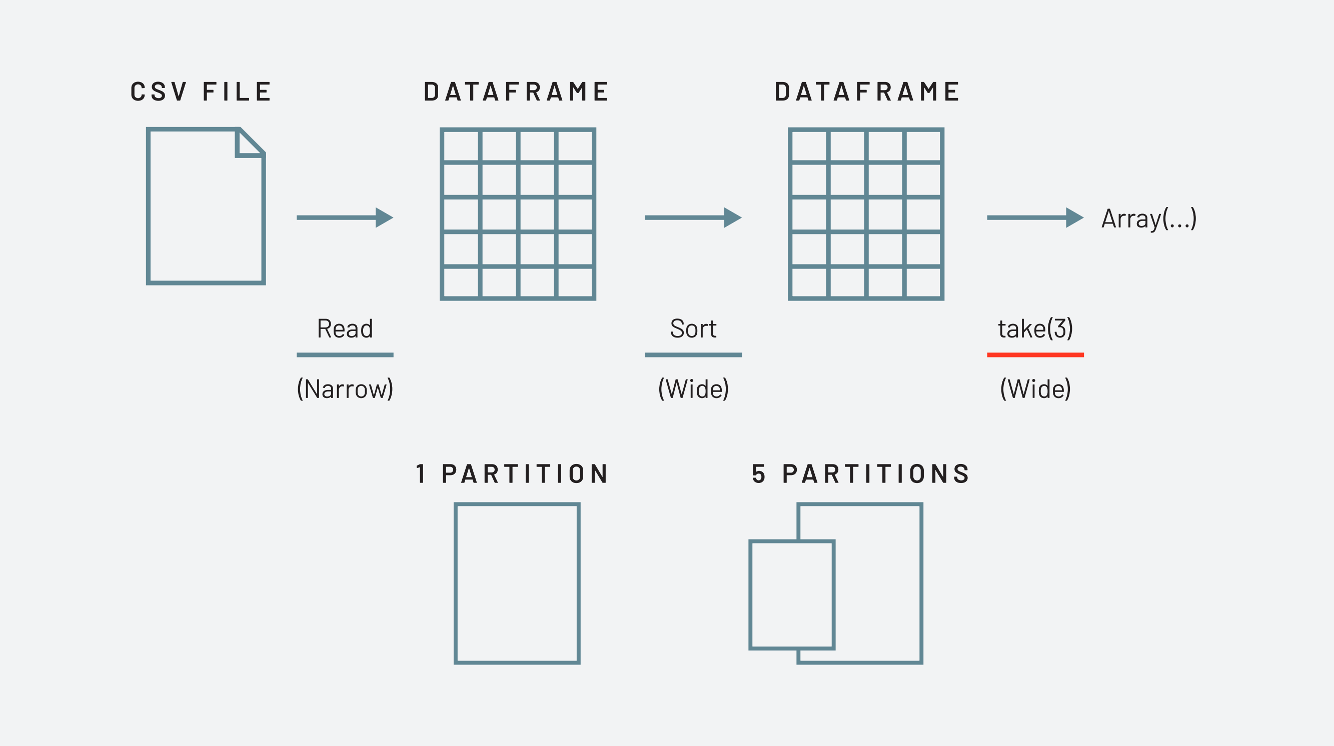
```
flightData2015.sort("count").explain()
```

Congratulations, you've just read your first explain plan! Explain plans are a bit arcane, but with a bit of practice it becomes second nature. Explain plans can be read from top to bottom, the top being the end result and the bottom being the source(s) of data. In our case, just take a look at the first keywords. You will see "sort", "exchange", and "FileScan". That's because the sort of our data is actually a wide transformation because rows will have to be compared with one another. Don't worry too much about understanding everything about explain plans at this point, they can just be helpful tools for debugging and improving your knowledge as you progress with Spark.

Now, just like we did before, we can specify an action in order to kick off this plan. However before doing that, we're going to set a configuration. By default, when we perform a shuffle Spark will output two hundred shuffle partitions. We will set this value to five in order to reduce the number of the output partitions from the shuffle from two hundred to five.

```
spark.conf.set("spark.sql.shuffle.partitions", "5")
flightData2015.sort("count").take(2)
... Array([United States,Singapore,1], [Moldova,United States,1])
```

This operation is illustrated in the following image. You'll notice that in addition to the logical transformations, we include the physical partition count as well.



The logical plan of transformations that we build up defines a lineage for the DataFrame so that at any given point in time Spark knows how to recompute any partition by performing all of the operations it had before on the same input data. This sits at the heart of Spark's programming model, functional programming where the same inputs always result in the same outputs when the transformations on that data stay constant.

We do not manipulate the physical data, but rather configure physical execution characteristics through things like the shuffle partitions parameter we set above. We got five output partitions because that's what we changed the shuffle partition value to. You can change this to help control the physical execution characteristics of your Spark jobs. Go ahead and experiment with different values and see the number of partitions yourself. In experimenting with different values, you should see drastically different run times. Remeber that you can monitor the job progress by navigating to the Spark UI on port 4040 to see the physical and logical execution characteristics of our jobs.

DataFrames and SQL

We worked through a simple example in the previous example, let's now work through a more complex example and follow along in both DataFrames and SQL. Spark the same transformations, regardless of the language, in the exact same way. You can express your business logic in SQL or DataFrames (either in R, Python, Scala, or Java) and Spark will compile that logic down to an underlying plan (that we see in the explain plan) before actually executing your code. Spark SQL allows you as a user to register any DataFrame as a table or view (a temporary table) and query it using pure SQL. There is no performance difference between writing SQL queries or writing DataFrame code, they both "compile" to the same underlying plan that we specify in DataFrame code.

Any DataFrame can be made into a table or view with one simple method call.

```
%scala
flightData2015.createOrReplaceTempView(“flight_data_2015”)
```

```
%python
flightData2015.createOrReplaceTempView(“flight_data_2015”)
```

Now we can query our data in SQL. To execute a SQL query, we'll use the `spark.sql` function (remember `spark` is our SparkSession variable?) that conveniently, returns a new DataFrame. While this may seem a bit circular in logic – that a SQL query against a DataFrame returns another DataFrame, it's actually quite powerful. As a user, you can specify transformations in the manner most convenient to you at any given point in time and not have to trade any efficiency to do so! To understand that this is happening, let's take a look at two explain plans.

```
%scala
val sqlWay = spark.sql(“””
SELECT DEST_COUNTRY_NAME, count(1)
FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
“””)
val dataFrameWay = flightData2015
   .groupBy(‘DEST_COUNTRY_NAME)
   .count()
sqlWay.explain
dataFrameWay.explain
```

```
%python
sqlWay = spark.sql(“””
SELECT DEST_COUNTRY_NAME, count(1)
FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
“””)
dataFrameWay = flightData2015\
.groupBy(“DEST_COUNTRY_NAME”)\
.count()
sqlWay.explain()
dataFrameWay.explain()
```

```
== Physical Plan ==
*HashAggregate(keys=[DEST_COUNTRY_NAME#182], functions=[count(1)])
+- Exchange hashpartitioning(DEST_COUNTRY_NAME#182, 5)
   +- *HashAggregate(keys=[DEST_COUNTRY_NAME#182], functions=[partial_
count(1)])
      +- *FileScan csv [DEST_COUNTRY_NAME#182] ...
== Physical Plan ==
*HashAggregate(keys=[DEST_COUNTRY_NAME#182], functions=[count(1)])
+- Exchange hashpartitioning(DEST_COUNTRY_NAME#182, 5)
   +- *HashAggregate(keys=[DEST_COUNTRY_NAME#182], functions=[partial_
count(1)])
      +- *FileScan csv [DEST_COUNTRY_NAME#182] ...
```

We can see that these plans compile to the exact same underlying plan!

To reinforce the tools available to us, let’s pull out some interesting statistics from our data. One thing to understand is that DataFrames (and SQL) in Spark already have a huge number of manipulations available. There are hundreds of functions that you can leverage and import to help you resolve your big data problems faster. We will use the `max` function, to find out what the maximum number of flights to and from any given location are. This just scans each value in relevant column the DataFrame and sees if it’s bigger than the previous values that have been seen. This is a transformation, as we are effectively filtering down to one row. Let’s see what that looks like.

```
spark.sql(“SELECT max(count) from flight_data_2015”).take(1)
```

```
%scala
import org.apache.spark.sql.functions.max
flightData2015.select(max(“count”)).take(1)
```

```
%python
from pyspark.sql.functions import max
flightData2015.select(max(“count”)).take(1)
```

Great, that’s a simple example. Let’s perform something a bit more complicated and find out the top five destination countries in the data? This is our first multi-transformation query so we’ll take it step by step. We will start with a fairly straightforward SQL aggregation.

```
%scala
val maxSql = spark.sql(“””
SELECT DEST_COUNTRY_NAME, sum(count) as destination_total
FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
ORDER BY sum(count) DESC
LIMIT 5
”””)
maxSql.collect()
```

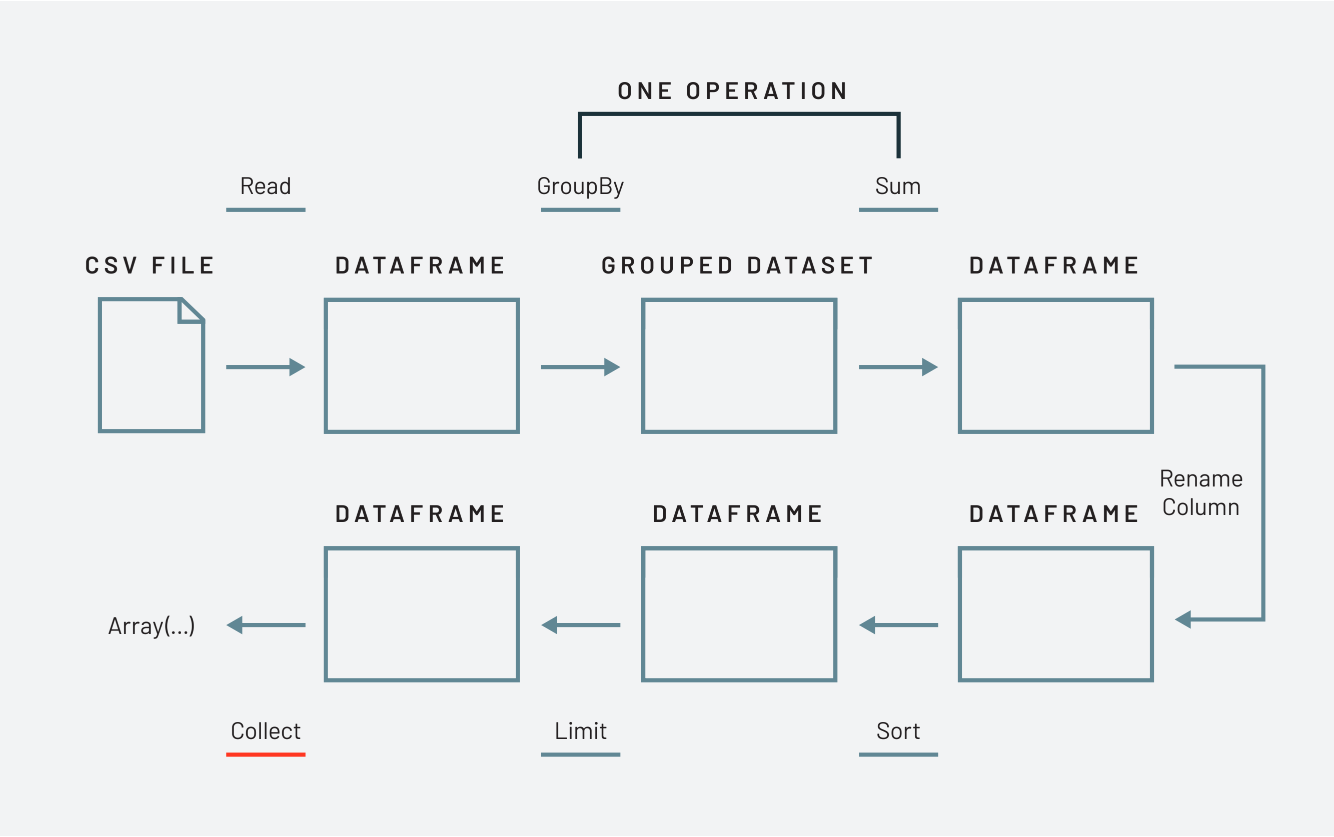
```
%python
maxSql = spark.sql(“””
SELECT DEST_COUNTRY_NAME, sum(count) as destination_total
FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
ORDER BY sum(count) DESC
LIMIT 5
”””)
maxSql.collect()
```

Now let’s move to the DataFrame syntax that is semantically similar but slightly different in implementation and ordering. But, as we mentioned, the underlying plans for both of them are the same. Let’s execute the queries and see their results as a sanity check.

```
%scala
import org.apache.spark.sql.functions.desc
flightData2015
   .groupBy(“DEST_COUNTRY_NAME”)
   .sum(“count”)
   .withColumnRenamed(“sum(count)”, “destination_total”)
   .sort(desc(“destination_total”))
   .limit(5)
   .collect()
```

```
%python
from pyspark.sql.functions import desc
flightData2015\
   .groupBy(“DEST_COUNTRY_NAME”)\
   .sum(“count”)\
   .withColumnRenamed(“sum(count)”, “destination_total”)\
   .sort(desc(“destination_total”))\
   .limit(5)\
   .collect()
```

Now there are 7 steps that take us all the way back to the source data. You can see this in the explain plan on those DataFrames. Illustrated below are the set of steps that we perform in "code". The true execution plan (the one visible in explain) will differ from what we have below because of optimizations in physical execution, however the illustration is as good of a starting point as any. This execution plan is a directed *acyclic graph (DAG)* of transformations, each resulting in a new immutable DataFrame, on which we call an action to generate a result.



The first step is to read in the data. We defined the DataFrame previously but, as a reminder, Spark does not actually read it in until an action is called on that DataFrame or one derived from the original DataFrame.

The second step is our grouping, technically when we call `groupBy` we end up with a RelationalGroupedDataset which is a fancy name for a DataFrame that has a grouping specified but needs the user to specify an aggregation before it can be queried further. We can see this by trying to perform an action on it (which will not work). We basically specified that we're going to be grouping by a key (or set of keys) and that now we're going to perform an aggregation over each one of those keys.

Therefore the third step is to specify the aggregation. Let's use the `sum` aggregation method. This takes as input a column expression or simply, a column name. The result of the sum method call is a new dataFrame. You'll see that it has a new schema but that it does know the type of each column. It's important to reinforce (again!) that no computation has been performed. This is simply another transformation that we've expressed and Spark is simply able to trace the type information we have supplied.

The fourth step is a simple renaming, we use the `withColumnRenamed` method that takes two arguments, the original column name and the new column name. Of course, this doesn't perform computation — this is just another transformation!

The fifth step sorts the data such that if we were to take results off of the top of the DataFrame, they would be the largest values found in the `destination_total` column.

You likely noticed that we had to import a function to do this, the `desc` function. You might also notice that `desc` does not return a string but a `Column`. In general, many DataFrame methods will accept Strings (as column names) or `Column` types or expressions. Columns and expressions are actually the exact same thing.

Penultimately, we'll specify a limit. This just specifies that we only want five values. This is just like a filter except that it filters by position instead of by value. It's safe to say that it basically just specifies a `DataFrame` of a certain size.

The last step is our action! Now we actually begin the process of collecting the results of our DataFrame above and Spark will give us back a list or array in the language that we're executing. Now to reinforce all of this, let's look at the explain plan for the above query.

```
%scala
flightData2015
   .groupBy(“DEST_COUNTRY_NAME”)
   .sum(“count”)
   .withColumnRenamed(“sum(count)”, “destination_total”)
   .sort(desc(“destination_total”))
   .limit(5)
   .explain()
```

```
%python
flightData2015\
.groupBy(“DEST_COUNTRY_NAME”)\
.sum(“count”)\
.withColumnRenamed(“sum(count)”, “destination_total”)\
.sort(desc(“destination_total”))\
   .limit(5)\
   .explain()
```

```
== Physical Plan ==
TakeOrderedAndProject(limit=5, orderBy=[destination_total#16194L DESC], output=[DEST_COUNTRY_NAME#7323,...
+- *HashAggregate(keys=[DEST_COUNTRY_NAME#7323], functions=[sum(count#7325L)])
  +- Exchange hashpartitioning(DEST_COUNTRY_NAME#7323, 5)
      +- *HashAggregate(keys=[DEST_COUNTRY_NAME#7323], functions=[partial
sum(count#7325L)])
         +- InMemoryTableScan [DEST_COUNTRY_NAME#7323, count#7325L]
               +- InMemoryRelation [DEST_COUNTRY_NAME#7323, ORIGIN_COUNTRY_NAME#7324, count#7325L]...
                     +- *Scan csv [DEST_COUNTRY_NAME#7578,ORIGIN_COUNTRY_NAME#7579,count#7580L]...
```

While this explain plan doesn't match our exact "conceptual plan" all of the pieces are there. You can see the limit statement as well as the `orderBy` (in the first line). You can also see how our aggregation happens in two phases, in the `partial_sum` calls. This is because summing a list of numbers is commutative and Spark can perform the sum, partition by partition. Of course we can see how we read in the DataFrame as well.

Naturally, we don't always have to collect the data. We can also write it out to any data source that Spark supports. For instance, let's say that we wanted to store the information in a database like PostgreSQL or write them out to another file.

A Unified Analytics Platform to Simplify Big Data and Data Science

The early foundations to simplify big data and AI have been laid by Apache Spark, the unified analytics engine, which combines big data processing with machine learning. Spark supports a wide range of analytics — data loading, SQL queries, machine learning and streaming computation with a consistent set of APIs.

Spark has also become the de-facto analytics engine in the enterprises today due to its speed, ease of use, and sophisticated analytics. Databricks' Unified Analytics Platform, built by the team who started the Spark research project at UC Berkeley, is based on the following fundamental beliefs:

- It's all about the data
- AI requires cross-functional collaboration
- Leverage cloud for faster Iteration

More than Just a Managed Apache Spark Platform

The Databricks Unified Analytics Platform makes AI accessible to all enterprises and accelerates AI-driven innovation.

FASTER PERFORMANCE

It's critical to process data no matter the scale as quickly as possible. Databricks has taken performance to another level through Databricks Runtime. Databricks Runtime is built on top of Spark and natively built for the cloud. Through various optimizations at the I/O and processing layer (Databricks I/O), we've made Spark faster and more performant. Recent benchmarks clock Databricks at a rate of 5x faster than vanilla Spark on AWS. Our Spark expertise is a huge differentiator in ensuring superior performance and very high reliability. These value added capabilities will increase your performance and reduce your TCO for managing Spark.

ALLEVIATE INFRASTRUCTURE COMPLEXITY HEADACHES

Infrastructure teams can stop fighting complexity and start focusing on customer-facing applications by getting out of the business of maintaining complex data infrastructure. This is thanks to Databricks' serverless, fully-managed, and highly elastic cloud service. And because Databricks has the industry's leading Spark experts, the service is fine-tuned to ensure ultra-reliable speed and reliability at scale. Data scientists no longer have to wait for an infrastructure team to provision and configure hardware for them, but instead, can be up and running in minutes, and focusing on building models and finding patterns in data that fuel innovation and accelerate time to market for transformative business outcomes.

WORK BETTER TOGETHER — BECOME A HEROIC TEAM

With a unified approach to analytics, data science teams can collaborate using the Databricks interactive workspace. They can use their preferred frameworks and libraries to interact with the data they are modeling, and then seamlessly move those models to production with a single click. By integrating and streamlining the individual elements that comprise the analytics lifecycle, these teams can create short feedback loops and work together, creating a culture of accelerated innovation. Now, thanks to Databricks, it's possible to build a model and test a prototype in hours, versus weeks or months with an older approach. Technology is nothing without people to make it great, and Databricks ensures a team can become heroes, by providing a common interface and tooling for all stakeholders, regardless of skill set, to collaborate with one another. This eliminates data silos and frees teammates to focus on what they do best, which in turn benefits their organization and increases innovation.

KEEP DATA SAFE AND SECURE

They say all press is good press, but a headline stating the company has lost valuable data is never good press. When a breach happens the enterprise grinds to a halt, and innovation and time-to-market is out the window. Databricks takes security very seriously, and by providing a common user interface as well as integrated technology set, data is protected at every level with a unified security model featuring fine grained controls, data encryption at rest and in motion, identity management, rigorous auditing, and support for compliance standards like HIPAA.

CHAPTER 3: Delta Lake Quickstart

Delta Lake is an open-source storage layer that brings data reliability to data lakes. Delta Lake provides ACID transactions, can handle metadata at scale, and can unify streaming and batch data processing. Delta Lake runs on top of your existing data lake and is fully compatible with Apache Spark APIs.

How to start using Delta Lake

The Delta Lake package is available as with the—packages option. In our example, we will also demonstrate the ability to VACUUM files and execute Delta Lake SQL commands within Apache Spark. As this is a short demonstration, we will also enable the following configurations:

- `spark.databricks.delta.retentionDurationCheck.enabled=false` to allow us to vacuum files shorter than the default retention duration of 7 days. Note, this is only required for the SQL command VACUUM.
- `spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension` to enable Delta Lake SQL commands within Apache Spark; this is not required for Python or Scala API calls.

```
 # Using Spark Packages
./bin/pyspark --packages io.delta:delta-core_2.11:0.4.0 --conf “spark.databricks.
 delta.retentionDurationCheck.enabled=false” --conf “spark.sql.extensions=io.delta.sql.
 DeltaSparkSessionExtension”
```

Loading and saving our Delta Lake data

This scenario will be using the On-time flight performance or Departure Delays dataset generated from the RITA BTS Flight Departure Statistics; some examples of this data in action include the 2014 Flight Departure Performance via d3.js Crossfilter and On-Time Flight Performance with GraphFrames for Apache Spark™. This dataset can be downloaded locally from this github location. Within pyspark, start by reading the dataset.

```
# Location variables
tripdelaysFilePath = “/root/data/departuredelays.csv”
pathToEventsTable = “/root/deltalake/departureDelays.delta”
# Read flight delay data
departureDelays = spark.read \
.option(“header”, “true”) \
.option(“inferSchema”, “true”) \
.csv(tripdelaysFilePath)
```

Next, let’s save our *departureDelays* dataset to a Delta Lake table. By saving this table to Delta Lake storage, we will be able to take advantage of its features including ACID transactions, unified batch and streaming, and time travel.

```
# Save flight delay data into Delta Lake format
departureDelays \
.write \
.format(“delta”) \
.mode(“overwrite”) \
.save(“departureDelays.delta”)
```

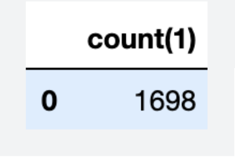
NOTE | This approach is similar to how you would normally save Parquet data; instead of specifying `format(“parquet”)`, you will now specify `format(“delta”)`. If you were to take a look at the underlying file system, you will notice four files created for the *departureDelays* Delta Lake table.

```
/departureDelays.delta$ ls -l
.
..
_delta_log
part-00000-df6f69ea-e6aa-424b-bc0e-f3674c4f1906-c000.snappy.parquet
part-00001-711bcce3-fe9e-466e-a22c-8256f8b54930-c000.snappy.parquet
part-00002-778ba97d-89b8-4942-a495-5f6238830b68-c000.snappy.parquet
part-00003-1a791c4a-6f11-49a8-8837-8093a3220581-c000.snappy.parquet
```

NOTE | *The* `_delta_log` *is the folder that contains the Delta Lake transaction log. For more information, refer to Diving Into Delta Lake: Unpacking The Transaction Log.*

Now, let’s reload the data but this time our DataFrame will be backed by Delta Lake.

```
# Load flight delay data in Delta Lake format
delays_delta = spark \
.read \
.format(“delta”) \
.load(“departureDelays.delta”)
# Create temporary view
delays_delta.createOrReplaceTempView(“delays_delta”)
# How many flights are between Seattle and San Francisco
park.sql(“select count(1) from delays_delta where origin = ‘SEA’ and destination = ‘SFO’”).show()
```



	count(1)
0	1698

Finally, let’s determine the number of flights originating from Seattle to San Francisco; in this dataset, there are 1698 flights.

In-place Conversion to Delta Lake

If you have existing Parquet tables, you have the ability to perform in-place conversions your tables to Delta Lake thus not needing to rewrite your table. To convert the table, you can run the following commands.

```
from delta.tables import *
# Convert non partitioned parquet table at path ‘/path/to/table’
deltaTable = DeltaTable.convertToDelta(spark, “parquet.`/path/to/table`”)
# Convert partitioned parquet table at path ‘/path/to/table’ and partitioned by integer column named ‘part’ partitionedDeltaTable = DeltaTable.
convertToDelta(spark, “parquet.`/path/to/table`”, “part int”)
park.sql(“select count(1) from delays_delta where origin = ‘SEA’ and destination = ‘SFO’”).show()
```

For more information, including how to do this conversion in Scala and SQL, refer to Convert to Delta Lake.

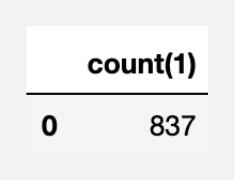
Delete our Flight Data

To delete data from your traditional Data Lake table, you will need to:

1. Select all of the data from your table not including the rows you want to delete
2. Create a new table based on the previous query
3. Delete the original table
4. Rename the new table to the original table name for downstream dependencies.

Instead of performing all of these steps, with Delta Lake, we can simplify this process by running a DELETE statement. To show this, let's delete all of the flights that had arrived early or on-time (i.e. `delay < 0`).

```
from delta.tables import *
from pyspark.sql.functions import *
# Access the Delta Lake table
deltaTable = DeltaTable.forPath(spark, pathToEventsTable
)
# Delete all on-time and early flights
deltaTable.delete("delay < 0")
# How many flights are between Seattle and San Francisco
spark.sql("select count(1) from delays_delta where origin = 'SEA' and
destination = 'SFO'").show()
```



	count(1)
0	837

After we *delete* (more on this below) all of the on-time and early flights, as you can see from the preceding query there are 837 late flights originating from Seattle to San Francisco. If you review the file system, you will notice there are more files even though you *deleted* data.

```
/departureDelays.delta$ ls -l
_delta_log
part-00000-a2a19ba4-17e9-4931-9bbf-3c9d4997780b-c000.snappy.parquet
part-00000-df6f69ea-e6aa-424b-bc0e-f3674c4f1906-c000.snappy.parquet
part-00001-711bcce3-fe9e-466e-a22c-8256f8b54930-c000.snappy.parquet
part-00001-a0423a18-62eb-46b3-a82f-ca9aac1f1e93-c000.snappy.parquet
part-00002-778ba97d-89b8-4942-a495-5f6238830b68-c000.snappy.parquet
part-00002-bfaa0a2a-0a31-4abf-aa63-162402f802cc-c000.snappy.parquet
part-00003-1a791c4a-6f11-49a8-8837-8093a3220581-c000.snappy.parquet
part-00003-b0247e1d-f5ce-4b45-91cd-16413c784a66-c000.snappy.parquet
```

In traditional data lakes, *deletes* are performed by re-writing the entire table excluding the values to be deleted. With Delta Lake, *deletes* instead are performed by selectively writing new versions of the files containing the data be deleted and only *marks* the previous files as deleted. This is because Delta Lake uses multiversion concurrency control to do atomic operations on the table: for example, while one user is deleting data, another user may be querying the previous version of the table. This multi-version model also enables us to travel back in time (i.e. time travel) and query previous versions as we will see later.

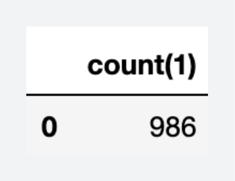
Update our Flight Data

To update data from your traditional Data Lake table, you will need to:

1. Select all of the data from your table not including the rows you want to modify
2. Modify the rows that need to be updated/changed
3. Merge these two tables to create a new table
4. Delete the original table
5. Rename the new table to the original table name for downstream dependencies.

Instead of performing all of these steps, with Delta Lake, we can simplify this process by running an UPDATE statement. To show this, let's update all of the flights originating from Detroit to Seattle.

```
Update all flights originating from Detroit to now be originating from Seattle
deltaTable.update(“origin = ‘DTW’”, { “origin”: “’SEA’” } )
# How many flights are between Seattle and San Francisco
spark.sql(“select count(1) from delays_delta where origin = ‘SEA’ and destination = ‘SFO’”).show()
```



```
   count(1)
0       986
```

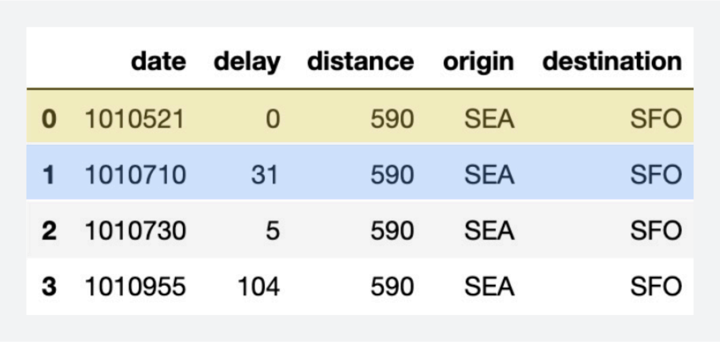
With the Detroit flights now tagged as Seattle flights, we now have 986 flights originating from Seattle to San Francisco. If you were to list the file system for your *departureDelays* folder (i.e. `$../departureDelays/ls -l`), you will notice there are now 11 files (instead of the 8 right after deleting the files and the four files after creating the table)

Merge our Flight Data

A common scenario when working with a data lake is to continuously append data to your table. This often results in duplicate data (rows you do not want inserted into your table again), new rows that need to be inserted, and some rows that need to be updated. With Delta Lake, all of this can be achieved by using the merge operation (similar to the SQL MERGE statement).

Let's start with a sample dataset that you will want to be updated, inserted, or deduplicated with the following query.

```
# What flights between SEA and SFO for these date periods
spark.sql(“select * from delays_delta where origin = ‘SEA’ and destination = ‘SFO’ and date like ‘1010%’ limit 10”).show()
```

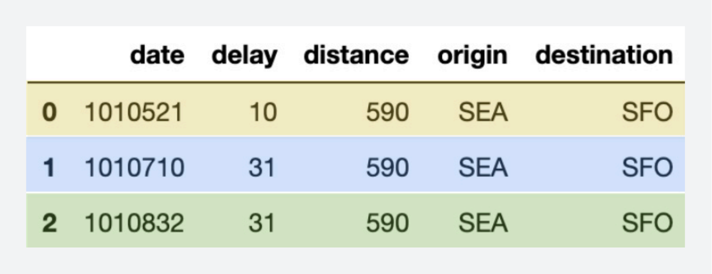


	date	delay	distance	origin	destination
0	1010521	0	590	SEA	SFO
1	1010710	31	590	SEA	SFO
2	1010730	5	590	SEA	SFO
3	1010955	104	590	SEA	SFO

The output of this query looks like the table at left. Note, the color-coding has been added to this blog to clearly identify which rows are deduplicated (blue), updated (yellow), and inserted (green).

Next, let's generate our own `merge_table` that contains data we will insert, update or de-duplicate with the following code snippet.

```
items = [(1010710, 31, 590, ‘SEA’, ‘SFO’), (1010521, 10, 590, ‘SEA’, ‘SFO’), (1010822, 31, 590, ‘SEA’, ‘SFO’)]
cols = [‘date’, ‘delay’, ‘distance’, ‘origin’, ‘destination’]
merge_table = spark.createDataFrame(items, cols)
merge_table.toPandas()
```



	date	delay	distance	origin	destination
0	1010521	10	590	SEA	SFO
1	1010710	31	590	SEA	SFO
2	1010832	31	590	SEA	SFO

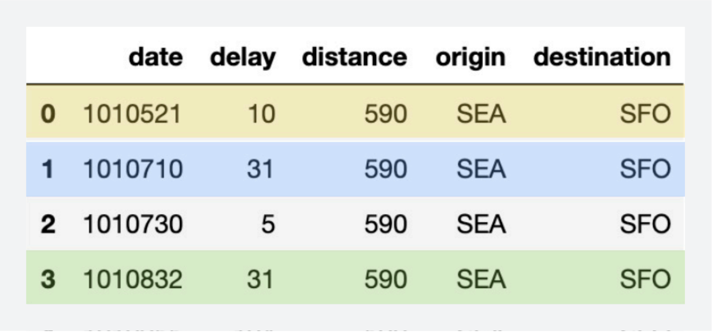
In the preceding table (`merge_table`), there are three rows that with a unique date value:

1. 1010521: this row needs to update the *flights* table with a new delay value (yellow)
2. 1010710: this row is a *duplicate* (blue)
3. 1010832: this is a new row to be *inserted* (green)

With Delta Lake, this can be easily achieved via a merge statement as noted in the following code snippet.

```
# Merge merge_table with flights
deltaTable.alias(“flights”) \
    .merge(merge_table.alias(“updates”),”flights.date = updates.date”) \
    .whenMatchedUpdate(set = { “delay” : “updates.delay” } ) \
    .whenNotMatchedInsertAll() \
    .execute()
# What flights between SEA and SFO for these date periods
spark.sql(“select * from delays_delta where origin = ‘SEA’ and destination = ‘SFO’ and date like ‘1010%’ limit 10”).show()
```

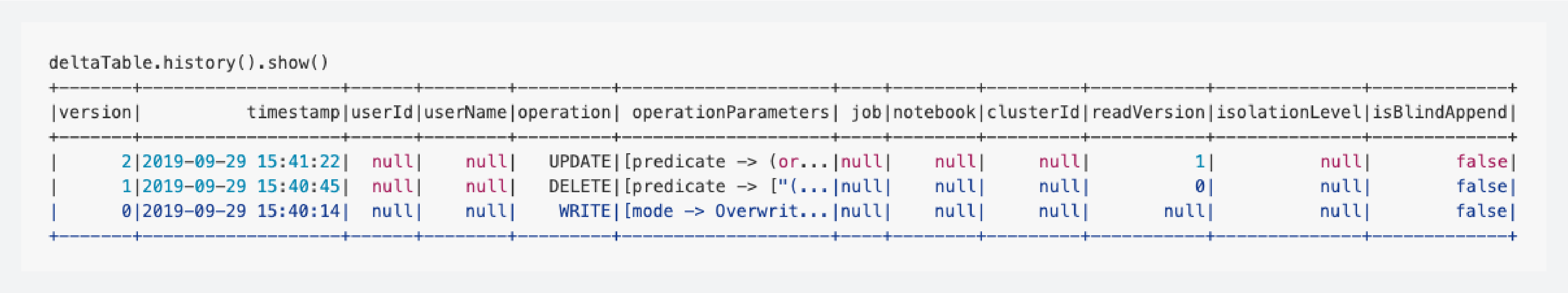
All three actions of de-duplication, update, and insert was efficiently completed with one statement.



	date	delay	distance	origin	destination
0	1010521	10	590	SEA	SFO
1	1010710	31	590	SEA	SFO
2	1010730	5	590	SEA	SFO
3	1010832	31	590	SEA	SFO

View Table History

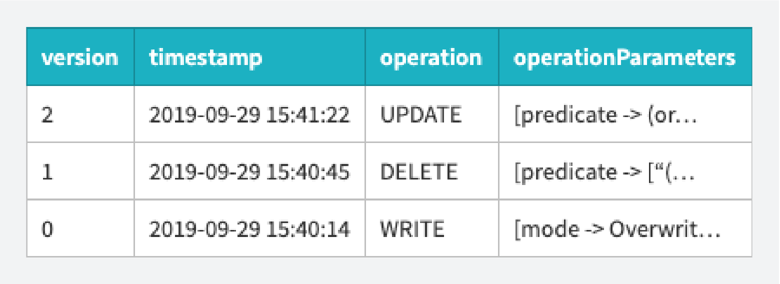
As previously noted, after each of our transactions (delete, update), there were more files created within the file system. This is because for each transaction, there are different versions of the Delta Lake table. This can be seen by using the `DeltaTable.history()` method as noted below.



```
deltaTable.history().show()
+-------+-------------------+------+--------+---------+--------------------+----+--------+---------+-----------+--------------+-------------+
|version|          timestamp|userId|userName|operation| operationParameters| job|notebook|clusterId|readVersion|isolationLevel|isBlindAppend|
+-------+-------------------+------+--------+---------+--------------------+----+--------+---------+-----------+--------------+-------------+
|      2|2019-09-29 15:41:22|  null|    null|   UPDATE|[predicate -> (or...|null|    null|     null|          1|          null|        false|
|      1|2019-09-29 15:40:45|  null|    null|   DELETE|[predicate -> ["(...|null|    null|     null|          0|          null|        false|
|      0|2019-09-29 15:40:14|  null|    null|    WRITE|[mode -> Overwrit...|null|    null|     null|       null|          null|        false|
+-------+-------------------+------+--------+---------+--------------------+----+--------+---------+-----------+--------------+-------------+
```

NOTE | *You can also perform the same task with SQL:* `spark.sql("DESCRIBE HISTORY '" + pathToEventsTable + "'").show()`

As you can see, there are three rows representing the different versions of the table (below is an abridged version to help make it easier to read) for each of the operations (create table, delete, and update):



version	timestamp	operation	operationParameters
2	2019-09-29 15:41:22	UPDATE	[predicate -> (or…
1	2019-09-29 15:40:45	DELETE	[predicate -> ["(…
0	2019-09-29 15:40:14	WRITE	[mode -> Overwrit…

Travel Back in Time with Table History

With Time Travel, you can see review the Delta Lake table as of the version or timestamp. For more information, refer to Delta Lake documentation > Read older versions of data using Time Travel. To view historical data, specify the version or Timestamp option; in the code snippet below, we will specify the version option

```
# Load DataFrames for each version
dfv0 = spark.read.format(“delta”).option(“versionAsOf”, 0).load(“departureDelays.delta”)
dfv1 = spark.read.format(“delta”).option(“versionAsOf”, 1).load(“departureDelays.delta”)
dfv2 = spark.read.format(“delta”).option(“versionAsOf”, 2).load(“departureDelays.delta”)
# Calculate the SEA to SFO flight counts for each version of history
cnt0 = dfv0.where(“origin = ‘SEA’”).where(“destination = ‘SFO’”).count()
cnt1 = dfv1.where(“origin = ‘SEA’”).where(“destination = ‘SFO’”).count()
cnt2 = dfv2.where(“origin = ‘SEA’”).where(“destination = ‘SFO’”).count()
# Print out the value
print(“SEA -> SFO Counts: Create Table: %s, Delete: %s, Update: %s” % (cnt0, cnt1, cnt2))
## Output
SEA -> SFO Counts: Create Table: 1698, Delete: 837, Update: 986
```

Whether for governance, risk management, and compliance (GRC) or rolling back errors, the Delta Lake table contains both the metadata (e.g. recording the fact that a delete had occurred with these operators) and data (e.g. the actual rows deleted). But how do we remove the data files either for compliance or size reasons?

Cleanup Old Table Versions with Vacuum

The Delta Lake vacuum method will delete all of the rows (and files) by default that are older than 7 days (reference: Delta Lake Vacuum). If you were to view the file system, you'll notice the 11 files for your table.

```
/departureDelays.delta$ ls -l
_delta_log
part-00000-5e52736b-0e63-48f3-8d56-50f7cfa0494d-c000.snappy.parquet
part-00000-69eb53d5-34b4-408f-a7e4-86e000428c37-c000.snappy.parquet
part-00000-f8edaf04-712e-4ac4-8b42-368d0bbdb95b-c000.snappy.parquet
part-00001-20893eed-9d4f-4c1f-b619-3e6ea1fdd05f-c000.snappy.parquet
part-00001-9b68b9f6-bad3-434f-9498-f92dc4f503e3-c000.snappy.parquet
part-00001-d4823d2e-8f9d-42e3-918d-4060969e5844-c000.snappy.parquet
part-00002-24da7f4e-7e8d-40d1-b664-95bf93ffeadb-c000.snappy.parquet
part-00002-3027786c-20a9-4b19-868d-dc7586c275d4-c000.snappy.parquet
part-00002-f2609f27-3478-4bf9-aeb7-2c78a05e6ec1-c000.snappy.parquet
part-00003-850436a6-c4dd-4535-a1c0-5dc0f01d3d55-c000.snappy.parquet
part-00003-b9292122-99a7-4223-aaa9-8646c281f199-c000.snappy.parquet
```

To delete all of the files so that you only keep the current snapshot of data, you will specify a small value for the vacuum method (instead of the default retention of 7 days).

```
# Remove all files older than 0 hours old.
deltaTable.vacuum(0)
```

NOTE | *You perform the same task via SQL syntax:*

```
# Remove all files older than 0 hours old
spark.sql("VACUUM '" + pathToEventsTable + "' RETAIN 0 HOURS")
```

Once the vacuum has completed, when you review the file system you will notice fewer files as the historical data has been removed.

```
/departureDelays.delta$ ls -l
_delta_log
part-00000-f8edaf04-712e-4ac4-8b42-368d0bbdb95b-c000.snappy.parquet
part-00001-9b68b9f6-bad3-434f-9498-f92dc4f503e3-c000.snappy.parquet
part-00002-24da7f4e-7e8d-40d1-b664-95bf93ffeadb-c000.snappy.parquet
part-00003-b9292122-99a7-4223-aaa9-8646c281f199-c000.snappy.parquet
```

NOTE | *The ability to time travel back to a version older than the retention period is lost after running vacuum.*

What's Next

Try out Delta Lake today by trying out the preceding code snippets on your Apache Spark 2.4.3 (or greater) instance. By using Delta Lake, you can make your data lakes more reliable (whether you create a new one or migrate an existing data lake). To learn more, refer to delta.io and join the Delta Lake community via Slack and Google Group. You can track all the upcoming releases and planned features in github milestones.

databricks
Unified Data Analytics

Get started with a free trial of Databricks and take your big data and data science projects to the next level today.

START YOUR FREE TRIAL

Contact us for a personalized demo

CONTACT