## Spark:

Apache Spark is a unified computing engine and a set of libraries for parallel data processing on computer clusters.

## Unified?

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.

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.

## Computing Engine:

We mean that Spark handles loading data from storage systems and performing computation on it, not permanent storage as the end itself. You can use Spark 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 over another. 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 need 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 difficult to run one of the systems without the other. More important, this choice also makes it a challenge to write applications that access data stored anywhere else. Although Spark runs well on Hadoop storage, today it is also used broadly in environments for which the Hadoop architecture does not make sense, such as the public cloud (where storage can be purchased separately from computing) or streaming applications

## 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 as well as 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 are hundreds of open source external libraries ranging from connectors for various storage systems to machine learning algorithms.

## How Spark is helpful?

A *cluster*, or group, of computers, pools the resources of many machines together, giving us the ability to use all the cumulative resources as if they were a single computer. Now, a group of machines alone is not powerful, you need a framework to coordinate work across them. Spark does just that, managing and coordinating the execution of tasks on data across a cluster of computers

The cluster of machines that Spark will use to execute tasks is 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.

## The Spark driver

The driver is the process "in the driver seat" of your Spark Application. It is the controller of the execution of a Spark Application and maintains all of the state of the Spark cluster (the state and tasks of the executors). It must interface with the cluster manager in order to actually get physical resources and launch executors. At the end of the day, this is just a process on a physical machine that is responsible for maintaining the state of the application running on the cluster.

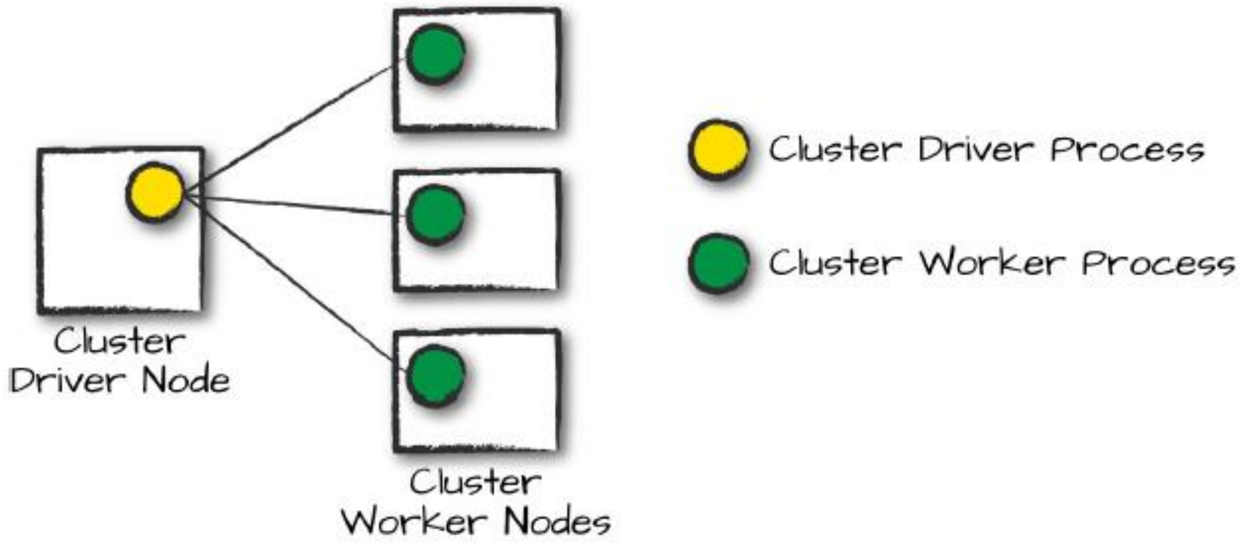
## The Spark executors

Spark executors are the processes that perform the tasks assigned by the Spark driver. Executors have one core responsibility: take the tasks assigned by the driver, run them, and report back their state (success or failure) and results. Each Spark Application has its own separate executor processes.

## The cluster manager

The Spark Driver and Executors do not exist in a void, and this is where the cluster manager comes in. The cluster manager is responsible for maintaining a cluster of machines that will run your Spark Application(s). Somewhat confusingly, a cluster manager will have its own "driver" (sometimes called master) and “worker” abstractions. The core difference is that these are tied to physical machines rather than processes (as they are in Spark)

The machine on the left of the illustration is the *Cluster Manager Driver Node*. The circles represent daemon processes running on and managing each of the individual worker nodes. There is no Spark Application running as of yet—these are just the processes from the cluster manager.



When it comes time to actually run a Spark Application, we request resources from the cluster manager to run it. Depending on how our application is configured, this can include a place to run the Spark driver or might be just resources for the executors for our Spark Application. Over the course of Spark Application execution, the cluster manager will be responsible for managing the underlying machines that our application is running on.
Spark currently supports three cluster managers: a simple built-in standalone cluster manager, Apache Mesos, and Hadoop YARN. However, this list will continue to grow, so be sure to check the documentation for your favorite cluster manager. Now that we’ve covered the basic components of an application, let’s walk through one of the first choices you will need to make when running your applications: choosing the execution mode.

## Execution Modes

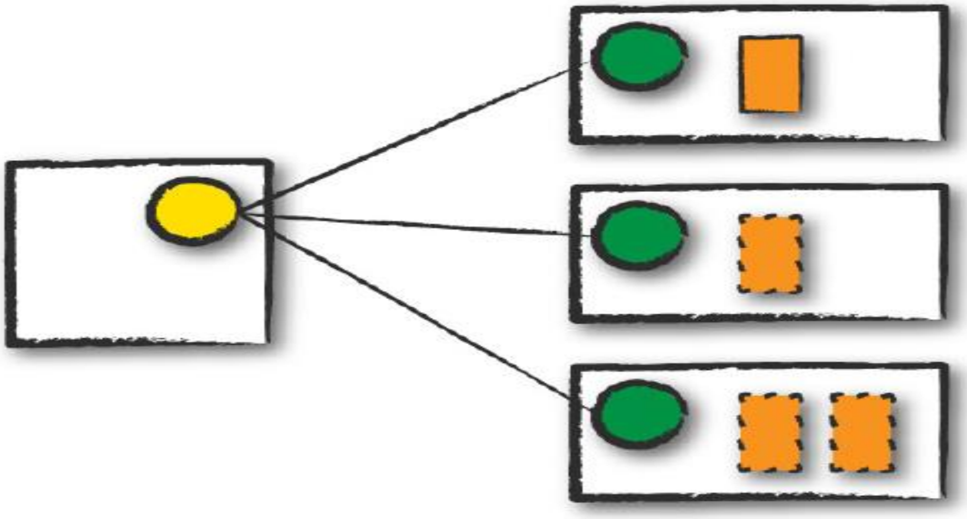
An execution *mode* gives you the power to determine where the aforementioned resources are physically located when you go to run your application. You have three modes to choose from:

1. Cluster mode
2. Client mode
3. Local mode

We will walk through each of these in detail using above figure as a template. In the following section, rectangles with solid borders represent Spark *driver process* whereas those with dotted borders represent the *executor processes*.

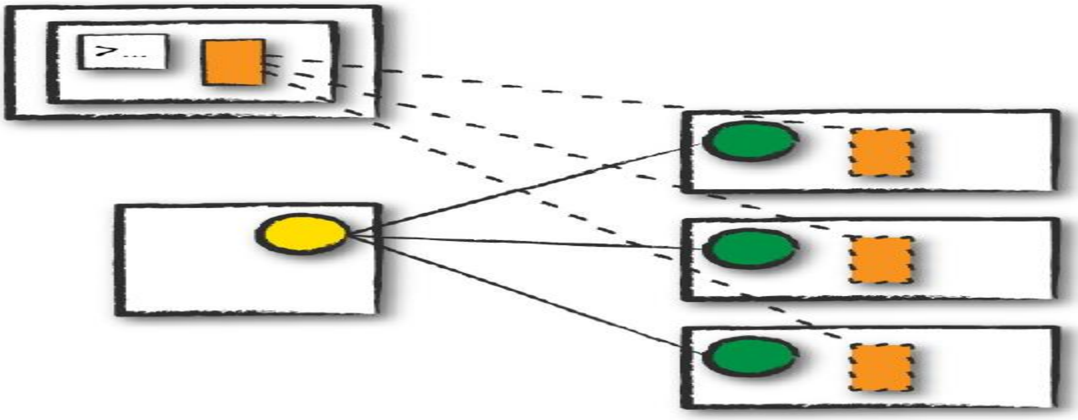
## Cluster mode

Cluster mode is probably the most common way of running Spark Applications. In cluster mode, a user submits a pre-compiled JAR, Python script, or R script to a cluster manager. The cluster manager then launches the driver process on a worker node inside the cluster, in addition to the executor processes. This means that the cluster manager is responsible for maintaining all Spark Application–related processes. Shows that the cluster manager placed our driver on a worker node and the executors on other worker nodes.



## Client mode

Client mode is nearly the same as cluster mode except that the Spark driver remains on the client machine that submitted the application. This means that the client machine is responsible for maintaining the Spark driver process, and the cluster manager maintains the executor processses. we are running the Spark Application from a machine that is not colocated on the cluster. These machines are commonly referred to as ***gateway machines*** or ***edge nodes***



## Local mode

Local mode is a significant departure from the previous two modes: it runs the entire Spark Application on a single machine. It achieves parallelism through threads on that single machine. This is a common way to learn Spark, to test your applications, or experiment iteratively with local development. However, we do not recommend using local mode for running production applications

## The Life Cycle of a Spark Application (Outside Spark)

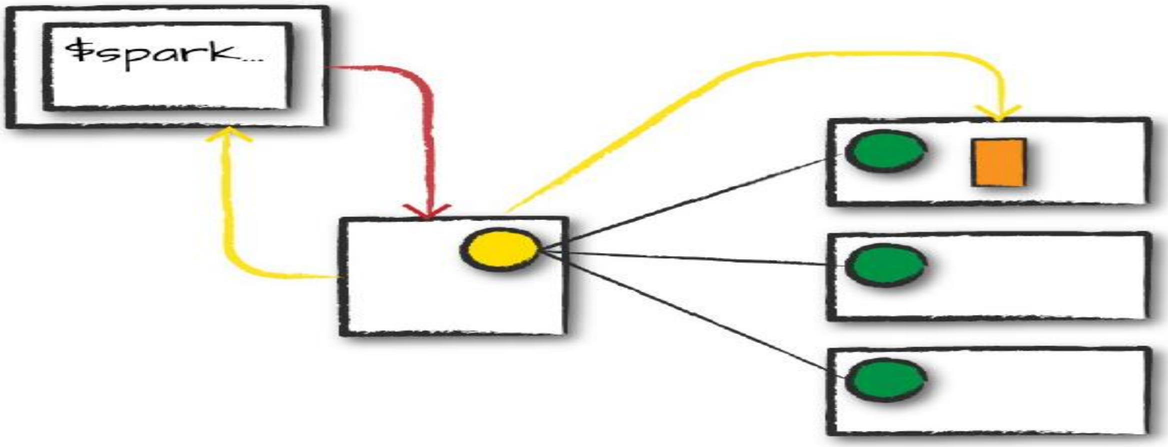
This section also makes use of illustrations and follows the same notation that we introduced previously. Additionally, we now introduce lines that represent network communication. Darker arrows represent communication by Spark or Spark-related processes, whereas dashed lines represent more general communication (like cluster management communication)

## Client Request

The first step is for you to submit an actual application. This will be a pre-compiled JAR or library. At this point, you are executing code on your local machine and you're going to make a request to the cluster manager driver node. Here, we are explicitly asking for resources for the *Spark driver process* only. We assume that the cluster manager accepts this offer and places the driver onto a node in the cluster. The client process that submitted the original job exits and the application is off and running on the cluster.

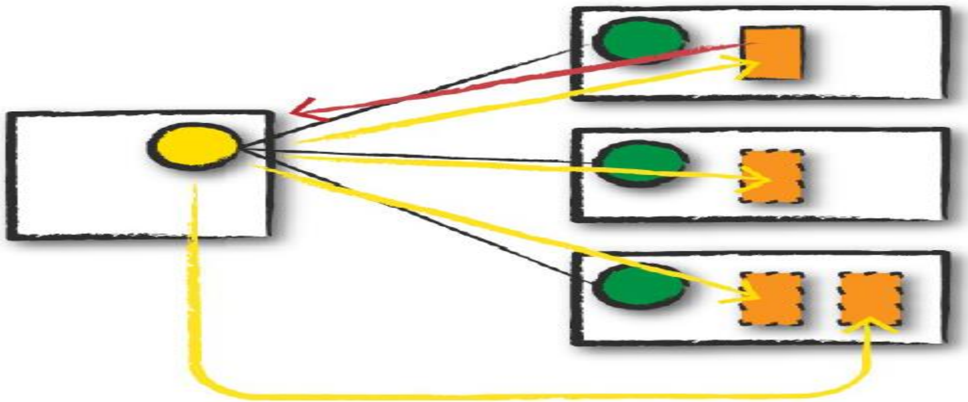
To do this, you'll run something like the following command in your terminal:

```
./bin/spark-submit \
--class <main-class> \
--master <master-url> \
--deploy-mode cluster \
--conf <key>=<value> \
... # other options
<application-jar> \
[application-arguments]
```



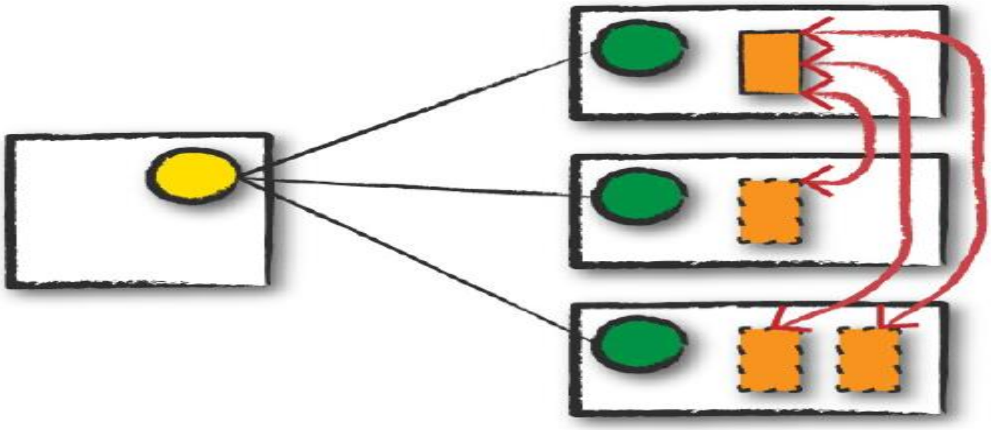
## Launch

Now that the driver process has been placed on the cluster, it begins running user code. This code must include a SparkSession that initializes a Spark cluster (e.g., driver + executors). The SparkSession will subsequently communicate with the cluster manager (the darker line), asking it to launch Spark executor processes across the cluster (the lighter lines). The number of executors and their relevant configurations are set by the user via the command-line arguments in the original spark-submit call. The cluster manager responds by launching the executor processes (assuming all goes well) and sends the relevant information about their locations to the driver process. After everything is hooked up correctly, we have a "Spark Cluster" as you likely think of it today.



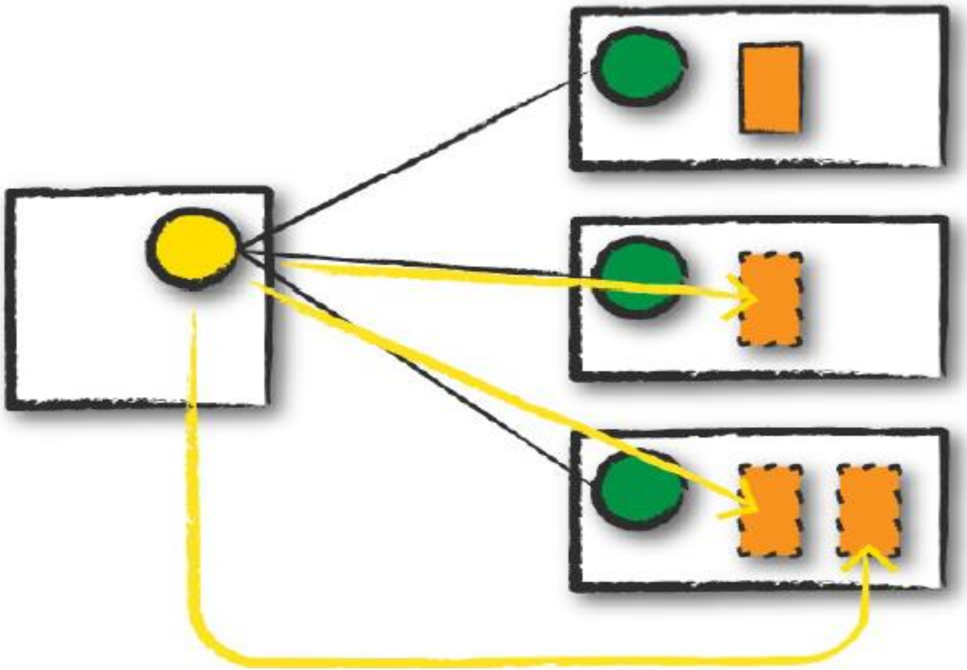
## Execution

Now that we have a "Spark Cluster," Spark goes about its merry way executing code, as shown. The driver and the workers communicate among themselves, executing code and moving data around. The driver schedules tasks onto each worker, and each worker responds with the status of those tasks and success or failure. (We cover these details shortly.)



## Completion

After a Spark Application complete, the driver processs exits with either success or failure. The cluster manager then shuts down the executors in that Spark cluster for the driver. At this point, you can see the success or failure of the Spark Application by asking the cluster manager for this information.



## The Life Cycle of a Spark Application (Inside Spark)

We just examined the life cycle of a Spark Application outside of user code (basically the infrastructure that supports Spark), but it's arguably more important to talk about what happens within Spark when you run an application. This is "user-code" (the actual code that you write that defines your Spark Application). Each application is made up of one or more *Spark jobs*. Spark jobs within an application are executed serially (unless you use threading to launch multiple actions in parallel).

## The SparkSession

The first step of any Spark Application is creating a SparkSession. In many interactive modes, this is done for you, but in an application, you must do it manually. Some of your legacy code might use the new SparkContext pattern. This should be avoided in favor of the builder method on the SparkSession, which more robustly instantiates the Spark and SQL Contexts and ensures that there is no context conflict, given that there might be multiple libraries trying to create a session in the same Spark Appication:

```scala
// Creating a SparkSession in Scala
import org.apache.spark.sql.SparkSession
val spark = SparkSession.builder().appName("Databricks Spark Example")
```

```
.config("spark.sql.warehouse.dir", "/user/hive/warehouse")
.getOrCreate()

# Creating a SparkSession in Python
from pyspark.sql import SparkSession
spark = SparkSession.builder.master("local").appName("Word Count")\
.config("spark.some.config.option", "some-value")\
.getOrCreate()
```

After you have a SparkSession, you should be able to run your Spark code. From the SparkSession, you can access all of low-level and legacy contexts and configurations accordingly, as well. Note that the SparkSession class was only added in Spark 2.X. Older code you might find would instead directly create a **SparkContext** and a **SQLContext** for the structured APIs.

## The SparkContext

A SparkContext object within the SparkSession represents the connection to the Spark cluster. This class is how you communicate with some of Spark's lower-level APIs, such as RDDs. It is commonly stored as the variable sc in older examples and documentation. Through a SparkContext, you can create RDDs, accumulators, and broadcast variables, and you can run code on the cluster. For the most part, you should not need to explicitly initialize a SparkContext; you should just be able to access it through the SparkSession. If you do want to, you should create it in the most general way, through the **getOrCreate** method:

```
// in Scala
import org.apache.spark.SparkContext
val sc = SparkContext.getOrCreate()
```

## THE SPARKSESSION, SQLCONTEXT, AND HIVECONTEXT

In previous versions of Spark, the SQLContext and HiveContext provided the ability to work with DataFrames and Spark SQL and were commonly stored as the variable sqlContext in examples, documentation, and legacy code. As a historical point, Spark 1.X had effectively two contexts. The SparkContext and the SQLContext. These two each performed different things. The former focused on more fine-grained control of Spark's central abstractions, whereas the latter focused on the higher-level tools like Spark SQL. In Spark 2.X, the communtiy combined the two APIs into the centralized SparkSession that we have today. However, both of these APIs still exist and you can access them via the **SparkSession**. It is important to note that you should never need to use the SQLContext and rarely need to use the SparkContext.

## A Spark Job

In general, there should be one Spark job for one action. Actions always return results. Each job breaks down into a series of *stages*, the number of which depends on how many shuffle operations need to take place.

## Stages

Stages in Spark represent groups of tasks that can be executed together to compute the same operation on multiple machines. In general, Spark will try to pack as much work as possible (i.e., as many transformations as possible inside your job) into the same stage, but the engine starts new stages after operations called *shuffles*. A shuffle represents a physical repartitioning of the data—for example, sorting a DataFrame, or grouping data that was loaded from a file by key (which requires sending records with the same key to the same node). This type of repartitioning requires coordinating across executors to move data around. Spark starts a new stage after each shuffle, and keeps track of what order the stages must run in to compute the final result.

**TIP**

We cover the number of partitions in a bit more detail in Chapter 19 because it's such an important parameter. This value should be set according to the number of cores in your cluster to ensure efficient execution. Here's how to set it:

```
spark.conf.set("spark.sql.shuffle.partitions", 50)
```

## Tasks

Stages in Spark consist of *tasks*. Each task corresponds to a combination of blocks of data and a set of transformations that will run on a single executor. If there is one big partition in our dataset, we will have one task. If there are 1,000 little partitions, we will have 1,000 tasks that can be executed in parallel. A task is just a unit of computation applied to a unit of data (the partition). Partitioning your data into a greater number of partitions means that more can be executed in parallel. This is not a panacea, but it is a simple place to begin with optimization.

## Pipelining

With pipelining, any sequence of operations that feed data directly into each other, without needing to move it across nodes, is collapsed into a single stage of tasks that do all the operations together. For example, if you write an RDD-based program that does a map, then a filter, then another map, these will result in a *single* stage of tasks that immediately read each input record, pass it through the first map, pass it through the filter, and pass it through the last map function if needed. This pipelined version of the computation is much faster than writing the intermediate results to memory or disk after each step.

## Shuffle Persistence

The second property you'll sometimes see is shuffle persistence. When Spark needs to run an operation that has to move data *across* nodes, such as a reduce-by-key operation (where input data for each key needs to first be brought together from many nodes), the engine can't perform pipelining anymore, and instead it performs a cross-network shuffle. Spark always executes shuffles by first having the "source" tasks (those sending data) write *shuffle files* to their local disks during their execution stage. Then, the stage that does the grouping and reduction launches and runs tasks that fetch their corresponding records from each shuffle file and performs that computation (e.g., fetches and processes the data for a specific range of keys). Saving the shuffle files to disk lets Spark run this stage later in time than the source stage (e.g., if there are not enough executors to run both at the same time), and also lets the engine re-launch reduce tasks on failure without rerunning all the input tasks. One side effect you'll see for shuffle persistence is that running a new job over data that's already been shuffled does not rerun the "source" side of the shuffle. Because the shuffle files were already written to disk earlier, Spark knows that it can use them to run the later stages of the job, and it need not redo the earlier ones. In the Spark UI and logs, you will see the preshuffle stages marked as "skipped". This automatic optimization can save time in a workload that runs multiple jobs over the same data, but of course, for even better performance you can perform your own caching with the DataFrame or RDD cache method, which lets you control exactly which data is saved and where. You'll quickly grow accustomed to this behavior after you run some Spark actions on aggregated data and inspect them in the UI.

## Responsibility of Diver in Spark:

The driver process runs your main() function, sits on a node in the cluster, and is responsible for three things:

1- maintaining information about the Spark Application
2- responding to a user's program or input
3- Analyzing, distributing, and scheduling work across the executors.

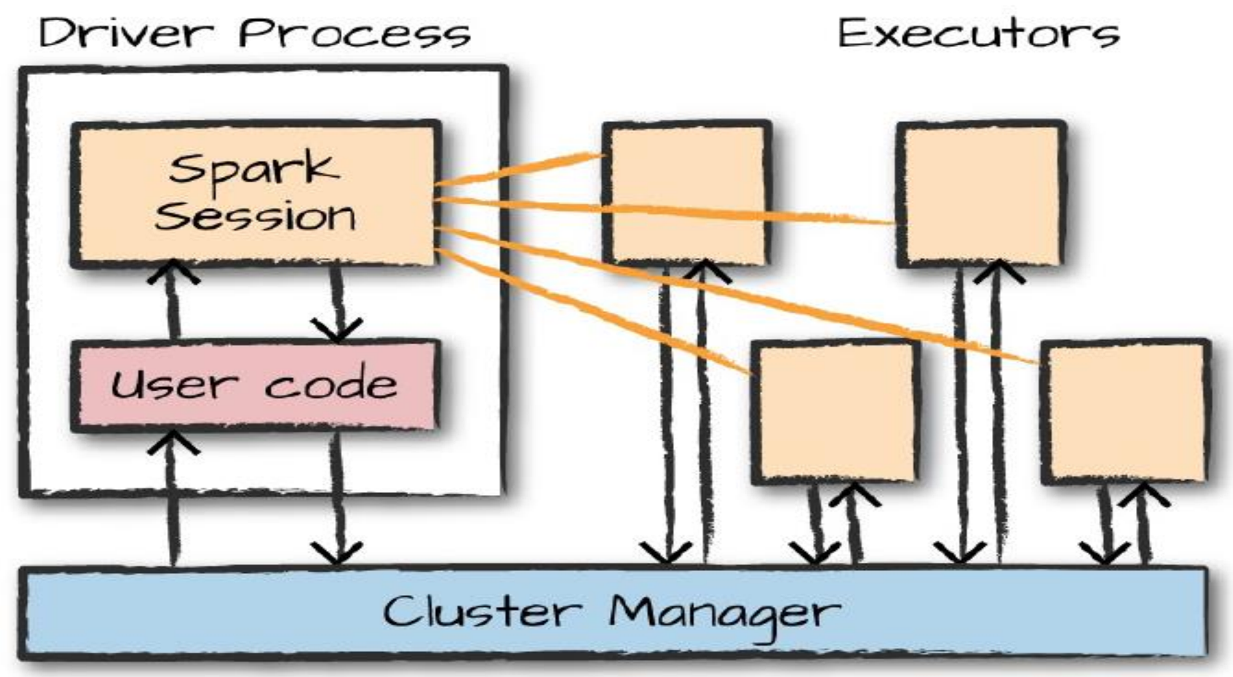
The driver process is absolutely essential—it's the heart of a Spark Application and maintains all relevant information during the lifetime of the application.

## Responsibility of Executors in Spark:

The *executors* are responsible for actually carrying out the work that the driver assigns them. This means that each executor is responsible for only two things:

1- Executing code assigned to it by the driver.
2- Reporting the state of the computation on that executor back to the driver node

demonstrates how the cluster manager controls physical machines and allocates resources to Spark Applications. This can be one of three 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.



Here are the key points to understand about Spark Applications at this point:

- Spark employs a cluster manager that keeps track of the resources available.
- The driver process is responsible for executing the driver program's commands across the executors to complete a given task.
- The executors, for the most part, will always be running Spark code. However, the driver can be "driven" from a number of different languages through Spark's language APIs. Let's take a look at those in the next section.

## Partitions

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 your cluster. A DataFrame's partitions represent how the data is physically distributed across the cluster of machines during execution. If you have one partition, Spark will have a parallelism of only one, even if you have thousands of executors. If you have many partitions but only one executor, Spark will still have a parallelism of only one because there is only one computation resource. An important thing to note is that with DataFrames you do not (for the most part) manipulate partitions manually or individually. You simply specify high-level transformations of data in the physical partitions, and Spark determines how this work will actually execute on the cluster.

## Transformations

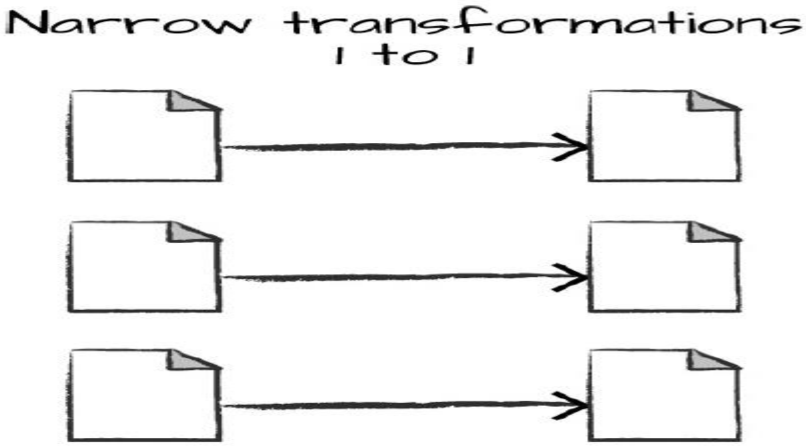
In Spark, the core data structures are *immutable*, meaning they cannot be changed after they're created. This might seem like a strange concept at first: if you cannot change it, how are you supposed to use it? To "change" a DataFrame, you need to instruct Spark how you would like to modify it to do what you want. These instructions are called *transformations*.

There are two types of Transformations

1- Narrow Dependencies
2- Wide Dependencies

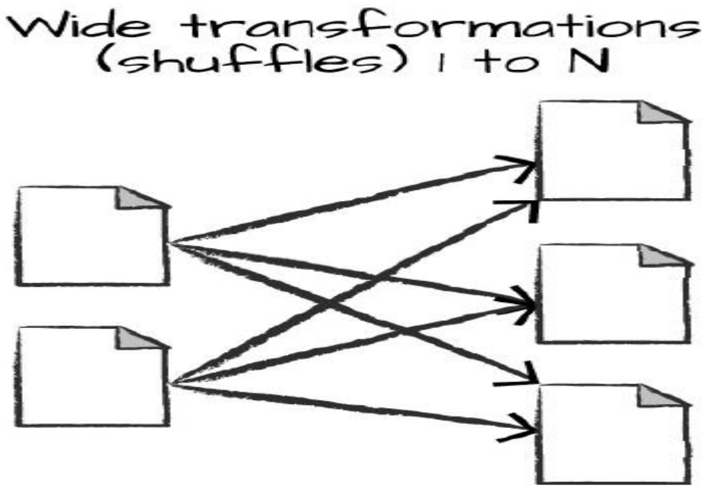
## Narrow Dependencies:

Transformations consisting of narrow dependencies (we'll call them narrow transformations) are those for which each input partition will contribute to only one output partition.



## Wide Dependencies:

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* whereby Spark will exchange partitions across the cluster. With narrow transformations, Spark will automatically perform an operation called *pipelining*, meaning that if we specify multiple filters on DataFrames, they'll all be performed in-memory. The same cannot be said for shuffles. When we perform a shuffle, Spark writes the results to disk.



## Lazy Evaluation

Lazy evaulation 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 you express some operation, you build up a *plan* of transformations that you would like to apply to your source data. By waiting until the last minute to execute the code, Spark compiles this plan from your raw DataFrame transformations to a streamlined physical plan that will run as efficiently as possible across the cluster. This provides immense benefits 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.

## Predicate Push Down:

The predicate push down is a logical optimization rule that consists on sending filtering operation directly to the data source. For instance, in the case of RDBMS, its  translated by executing “WHERE” clause directly on database level.
https://www.waitingforcode.com/apache-spark-sql/predicate-pushdown-spark-sql/read

## 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.

There are three kinds of actions:

1- Actions to view data in the console
2- Actions to collect data to native objects in the respective language
3- Actions to write to output data sources

Note: By default, when we perform a shuffle, Spark outputs 200 shuffle partitions.

## DataFrames and SQL

We worked through a simple transformation in the previous example, let's now work through a more complex one and follow along in both DataFrames and SQL. Spark can run 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 you can see in the explain plan) before actually executing your code. With Spark SQL, you can 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.

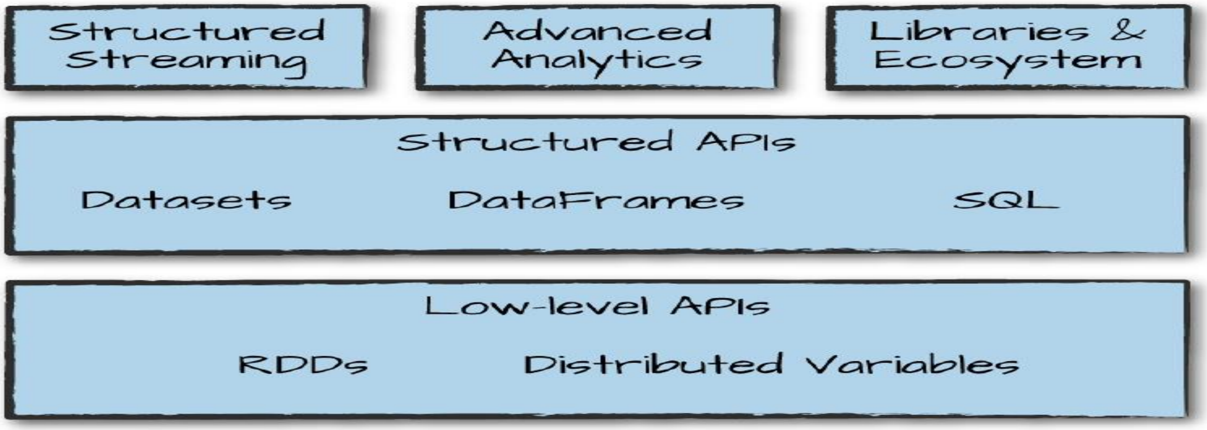


*Figure 3-1. Spark's toolset*

## Spark-submit

**spark-submit**, a built-in command-line tool. spark-submit does one thing: it lets you send your application code to a cluster and launch it to execute there. Upon submission, the application will run until it exits (completes the task) or encounters an error. You can do this with all of Spark's support cluster managers including Standalone, Mesos, and YARN. spark-submit offers several controls with which you can specify the resources your application needs as well as how it should be run and its command-line arguments

## Dataset API

The Dataset API gives users the ability to assign a Java/Scala class to the records within a DataFrame and manipulate it as a collection of typed objects, similar to a Java ArrayList or Scala Seq. The APIs available on Datasets are *type-safe*, meaning that you cannot accidentally view the objects in a Dataset as being of another class than the class you put in initially. This makes

Datasets especially attractive for writing large applications, with which multiple software engineers must interact through well-defined interfaces.

## Streaming Action

Streaming actions are a bit different from our conventional static action because we're going to be populating data somewhere instead of just calling something like count (which doesn't make any sense on a stream anyways). The action we will use will output to an in-memory table that we will update after each *trigger*. In this case, each trigger is based on an individual file (the read option that we set). Spark will mutate the data in the in-memory table such that we will always have the highest value

Before proceeding, let's review the fundamental concepts and definitions that we covered in Part I.
Spark is a distributed programming model in which the user specifies *transformations*. Multiple transformations build up a directed acyclic graph of instructions. An action begins the process of executing that graph of instructions, as a single job, by breaking it down into stages and tasks to execute across the cluster. The logical structures that we manipulate with transformations and actions are DataFrames and Datasets. To create a new DataFrame or Dataset, you call a transformation. To start computation or convert to native language types, you call an action.

## Structured API Overview

The Structured APIs are a tool for manipulating all sorts of data, from unstructured log files to semi-structured CSV files and highly structured Parquet files. These APIs refer to three core types of distributed collection APIs:

1- Datasets
2- DataFrames
3- SQL tables and views

Structured APIs apply to both *batch* and *streaming* computation. This means that when you work with the Structured APIs, it should be simple to migrate from batch to streaming (or vice versa) with little to no effort.

fundamental concepts that you should understand: the typed and untyped APIs (and their differences); what the core terminology is; and, finally, how Spark actually takes your Structured API data flows and executes it on the cluster.

## SPARK TYPES

Spark uses an engine called *Catalyst* that maintains its own type information through the planning and processing of work. In doing so, this opens up a wide variety of execution optimizations that make significant differences. Spark types map directly to the different language APIs that Spark maintains and there exists a lookup table for each of these in Scala, Java, Python, SQL, and R. Even if we use  Spark's Structured APIs from Python or R, the majority of our manipulations will operate strictly on *Spark types*, not Python types

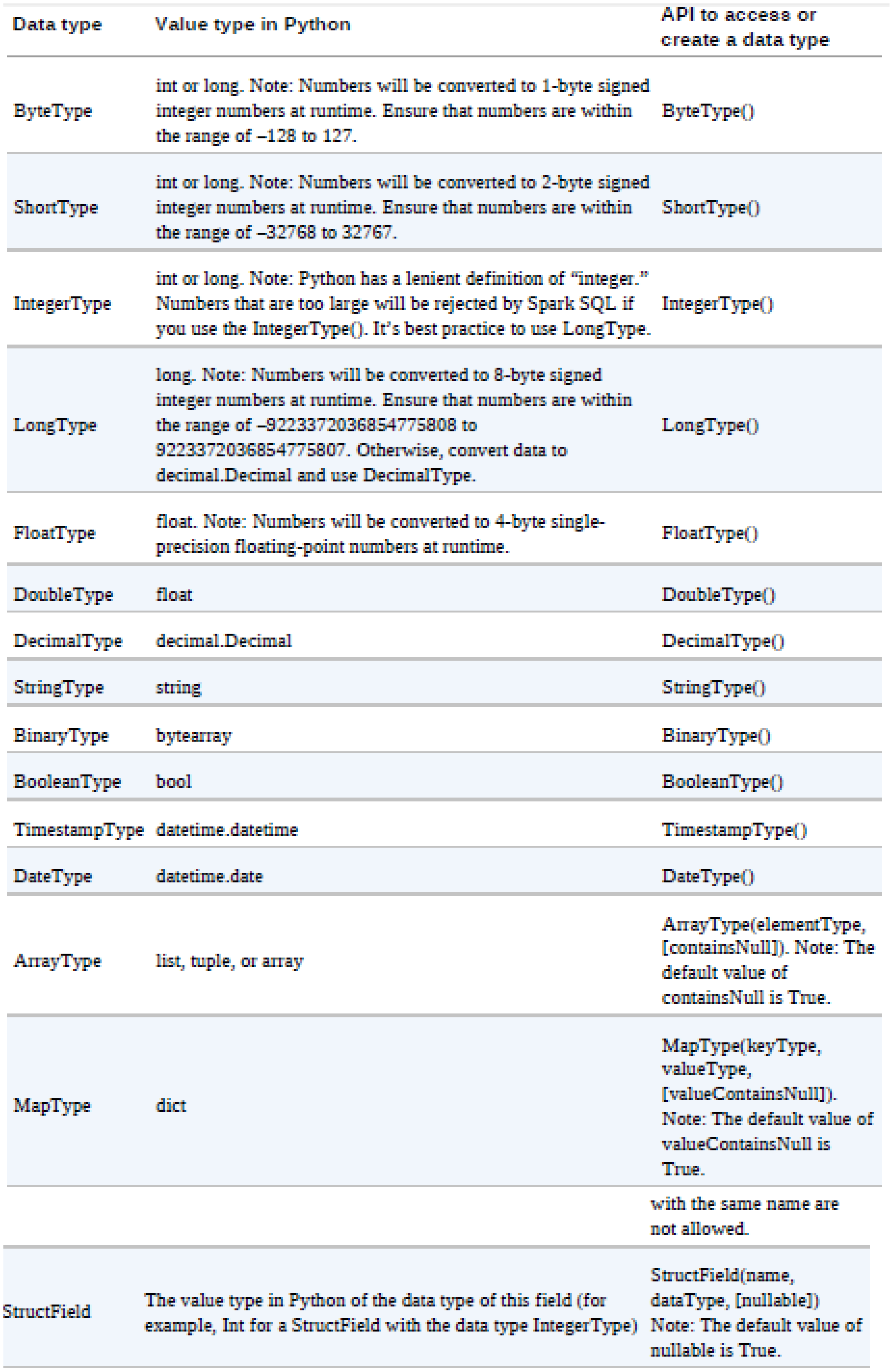
## Difference between DataFrames and DataSets

Within the Structured APIs, there are two more APIs, the "untyped" DataFrames and the "typed" Datasets. To say that DataFrames are untyped is aslightly inaccurate; they have types, but Spark maintains them completely and only checks whether those types line up to those specified in the schema at *runtime*. Datasets, on the other hand, check whether types conform to the specification at *compile time*. Datasets are only available to Java Virtual Machine (JVM)–based languages (Scala and Java) and we specify types with case classes or Java beans.

To Spark (in Scala), DataFrames are simply Datasets of Type Row. The "Row" type is Spark's internal representation of its optimized in-memory format for computation. This format makes for highly specialized and efficient computation because rather than using JVM types, which can cause high garbage-collection and object instantiation costs, Spark can operate on its own internal format without incurring any of those costs. To Spark (in Python or R), there is no such thing as a Dataset: everything is a DataFrame and therefore we always operate on that optimized format.

**Spark to Python Type**

Visit page 60 for other types

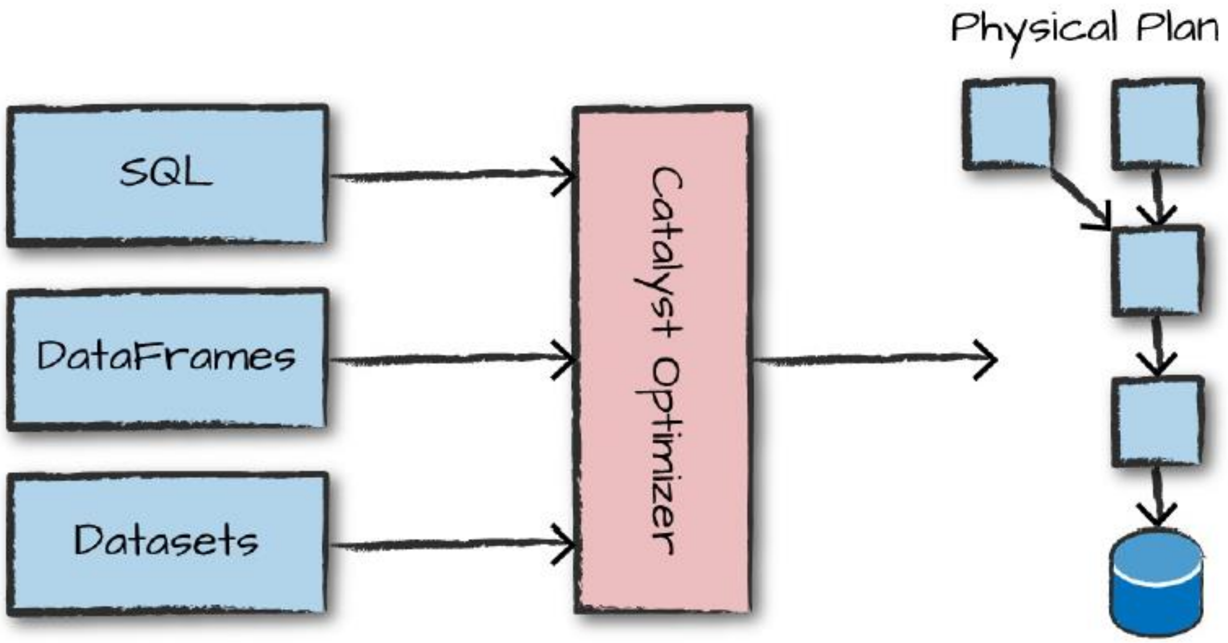


| Data type | Value type in Python | API to access or create a data type |
|---|---|---|
| ByteType | int or long. Note: Numbers will be converted to 1-byte signed integer numbers at runtime. Ensure that numbers are within the range of –128 to 127. | ByteType() |
| ShortType | int or long. Note: Numbers will be converted to 2-byte signed integer numbers at runtime. Ensure that numbers are within the range of –32768 to 32767. | ShortType() |
| IntegerType | int or long. Note: Python has a lenient definition of "integer." Numbers that are too large will be rejected by Spark SQL if you use the IntegerType(). It's best practice to use LongType. | IntegerType() |
| LongType | long. Note: Numbers will be converted to 8-byte signed integer numbers at runtime. Ensure that numbers are within the range of –9223372036854775808 to 9223372036854775807. Otherwise, convert data to decimal.Decimal and use DecimalType. | LongType() |
| FloatType | float. Note: Numbers will be converted to 4-byte single-precision floating-point numbers at runtime. | FloatType() |
| DoubleType | float | DoubleType() |
| DecimalType | decimal.Decimal | DecimalType() |
| StringType | string | StringType() |
| BinaryType | bytearray | BinaryType() |
| BooleanType | bool | BooleanType() |
| TimestampType | datetime.datetime | TimestampType() |
| DateType | datetime.date | DateType() |
| ArrayType | list, tuple, or array | ArrayType(elementType, [containsNull]). Note: The default value of containsNull is True. |
| MapType | dict | MapType(keyType, valueType, [valueContainsNull]). Note: The default value of valueContainsNull is True. |
| | | with the same name are not allowed. |
| StructField | The value type in Python of the data type of this field (for example, Int for a StructField with the data type IntegerType) | StructField(name, dataType, [nullable]) Note: The default value of nullable is True. |

## Structured API Execution

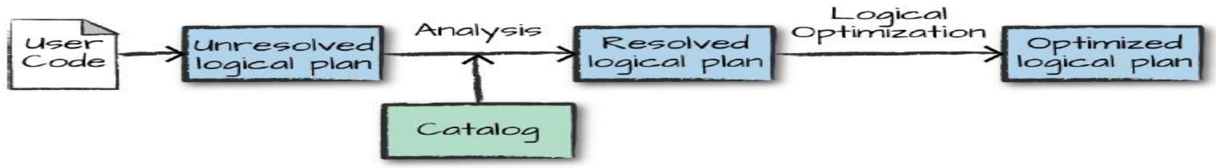
1. Write DataFrame/Dataset/SQL Code.
2. If valid code, Spark converts this to a *Logical Plan*.
3. Spark transforms this *Logical Plan* to a *Physical Plan*, checking for optimizations along the way.
4. Spark then executes this *Physical Plan* (RDD manipulations) on the cluster.

This code is then submitted to Spark either through the console or via a submitted job. This code then passes through the Catalyst Optimizer, which decides how the code should be executed and lays out a plan for doing so before, finally, the code is run and the result is returned to the user



## Logical Panning

This logical plan only represents a set of abstract transformations that do not refer to executors or driver, it's purely to convert the user's set of expressions into the most optimized version. It does this by converting user code into an *unresolved logical plan*. This plan is unresolved because although your code might be valid, the tables or columns that it refers to might or might not exist. Spark uses the *catalog*, a repository of all table and DataFrame information, to *resolve* columns and tables in the *analyzer*. The analyzer might reject the unresolved logical plan if the required table or column name does not exist in the catalog. If the analyzer can resolve it, the result is passed through the Catalyst Optimizer, a collection of rules that attempt to optimize the logical plan by pushing down predicates or selections. Packages can extend the Catalyst to include their own rules for domain-specific optimizations

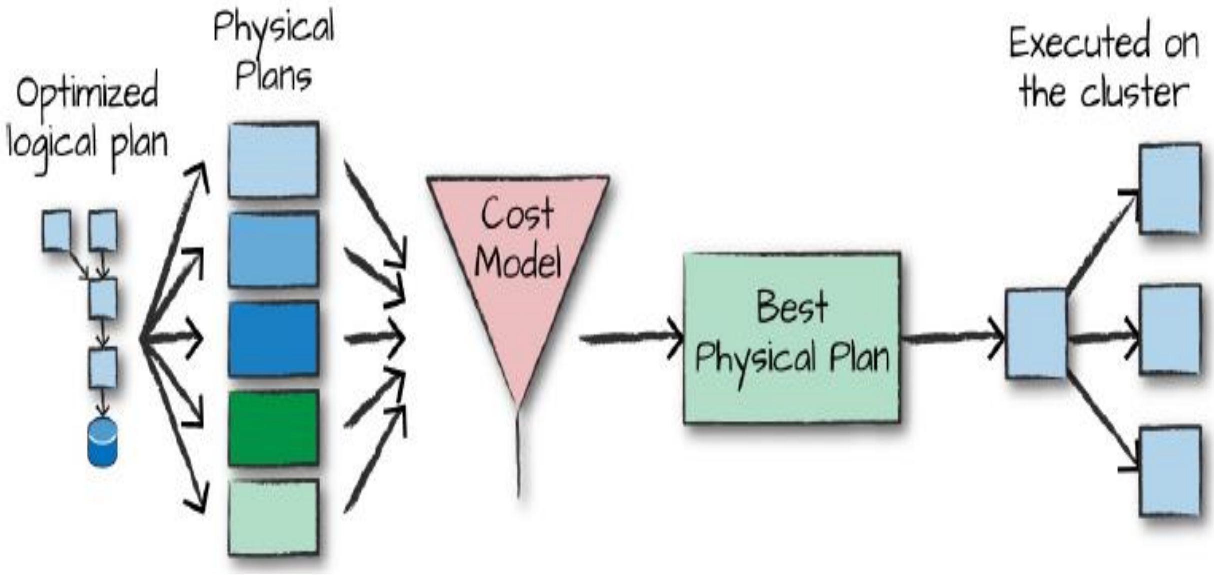


## Physical Planning

After successfully creating an optimized logical plan, Spark then begins the physical planning process. The *physical plan*, often called a Spark plan, specifies how the logical plan will execute on the cluster by generating different physical execution strategies and comparing them through a cost model, as depicted in Figure 4-3. An example of the cost comparison might be choosing how to perform a given join by looking at the physical attributes of a given table (how big the table is or how big its partitions are). Physical planning results in a series of RDDs and transformations. This result is why you might have heard Spark referred to as a compiler—it takes queries in DataFrames, Datasets, and SQL and compiles them into RDD transformations for you.

## Execution

Upon selecting a physical plan, Spark runs all of this code over RDDs, the lower-level programming interface of Spark (which we cover in Part III). Spark performs further optimizations at runtime, generating native Java bytecode that can remove entire tasks or stages during execution. Finally the result is returned to the user.



## Schema

A schema is a StructType made up of a number of fields, StructFields, that have a name, type, a Boolean flag which specifies whether that column can contain missing or null values, and, finally, users can optionally specify associated metadata with that column. The metadata is a way of storing information about this column (Spark uses this in its machine learning library).

## Example:

```
StructType(List(StructField(DEST_COUNTRY_NAME,StringType,true),StructField
(ORIGIN_COUNTRY_NAME,StringType,true),StructField(count,LongType,true)))
```

## Expressions

We mentioned earlier that columns are expressions, but what is an expression? An *expression* is a set of transformations on one or more values in a record in a DataFrame. Think of it like a function that takes as input one or more column names, resolves them, and then potentially applies more expressions to create a single value for each record in the dataset. Importantly, this "single value" can actually be a complex type like a Map or Array. In the simplest case, an expression, created via the expr function, is just a DataFrame column reference. In the simplest case, expr("someCol") is equivalent to col("someCol").

Columns provide a subset of expression functionality. If you use **col()** and want to perform transformations on that column, you must perform those on that column reference. When using an expression, the **expr** function can actually parse transformations and column references from a string and can subsequently be passed into further transformations.

```
(((col("someCol") + 5) * 200) - 6) < col("otherCol")
```

```
from pyspark.sql.functions import expr
expr("(((someCol + 5) * 200) - 6) < otherCol")
```

## Rows

In Spark, each row in a DataFrame is a single record. Spark represents this record as an object of type Row. Row objects internally represent arrays of bytes. The byte array interface is never shown to users because we only use column expressions to manipulate them.

## select and selectExpr

select and selectExpr allow you to do the DataFrame equivalent of SQL queries on a table of data:

```
-- in SQL
SELECT * FROM dataFrameTable
SELECT columnName FROM dataFrameTable
SELECT columnName * 10, otherColumn, someOtherCol as c FROM dataFrameTable
```

## Converting to Spark Types (Literals)

Sometimes, we need to pass explicit values into Spark that are just a value (rather than a new column). This might be a constant value or something we'll need to compare to later on. The way we do this is through *literals*. This is basically a translation from a given programming language's literal value to one that Spark understands. Literals are expressions and you can use them in the same way:

```
# in Python
from pyspark.sql.functions import lit
df.select(expr("*"), lit(1).alias("One")).show(2)
```

In SQL, literals are just the specific value:

```
-- in SQL
SELECT *, 1 as One FROM dfTable LIMIT 2
```

Giving an output of:

```
+-----------------+------------------+-----+---+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|One|
+-----------------+------------------+-----+---+
| United States| Romania| 15| 1|
| United States| Croatia| 1| 1|
+-----------------+------------------+-----+---+
```

## Case Sensitivity

By default Spark is case insensitive; however, you can make Spark case sensitive by setting the configuration:

```
-- in SQL
set spark.sql.caseSensitive true
```

Instinctually, you might want to put multiple filters into the same expression. Although this is possible, it is not always useful, because Spark automatically performs all filtering operations at the same time regardless of the filter ordering. This means that if you want to specify multiple AND filters, just chain them sequentially and let Spark handle the rest

## NULLS

An advanced tip is to use asc_nulls_first, desc_nulls_first, asc_nulls_last, or desc_nulls_last to specify where you would like your null values to appear in an ordered
DataFrame.

## Repartition and Coalesce

Another important optimization opportunity is to partition the data according to some frequently filtered columns, which control the physical layout of data across the cluster including the partitioning scheme and the number of partitions.

Repartition will incur a full shuffle of the data, regardless of whether one is necessary. This means that you should typically only repartition when the future number of partitions is greater than your current number of partitions or when you are looking to partition by a set of columnsWhat is Coalesce?

The coalesce method **reduces** the **number of partitions** in a DataFrame. Coalesce **avoids full shuffle**, instead of creating new partitions, it shuffles the data using Hash Partitioner (Default), and **adjusts into existing partitions**, this means it can **only decrease** the number of partitions.

## What is Repartitioning?

The repartition method can be used to either **increase** or **decrease** the **number of partitions** in a DataFrame. Repartition is a **full Shuffle** operation, whole data is taken out from existing partitions and **equally distributed** into **newly formed partitions**.

Link : https://blog.knoldus.com/apache-spark-repartitioning-v-s-coalesce/

## Collecting Rows to the Driver

As discussed in previous chapters, Spark maintains the state of the cluster in the driver. There are times when you'll want to collect some of your data to the driver in order to manipulate it on your local machine.

Any collection of data to the driver can be a very expensive operation! If you have a large dataset and call collect, you can crash the driver. If you use toLocalIterator and have very large partitions, you can easily crash the driver node and lose the state of your application. This is also expensive because we can operate on a one-by-one basis, instead of running computation in parallel.

## monotonically_increasing_id

As a last note, we can also add a unique ID to each row by using the function monotonically_increasing_id. This function generates a unique value for each row, starting with 0:

```
# in Python
from pyspark.sql.functions import monotonically_increasing_id
df.select(monotonically_increasing_id()).show(2)
```

## Working with Date and Times

There are a lot of caveats, unfortunately, when working with dates and timestamps, especially when it comes to timezone handling. In version 2.1 and before, Spark parsed according to the machine's timezone if timezones are not explicitly specified in the value that you are parsing. You can set a session local timezone if necessary by setting spark.conf.sessionLocalTimeZone in the SQL configurations. This should be set according to the Java TimeZone format.

Another common "gotcha" is that Spark's **TimestampType** class supports only second-level precision, which means that if you're going to be working with milliseconds or microseconds, you'll need to work around this problem by potentially operating on them as longs. Any more precision when coercing to a **TimestampType** will be removed.

Spark is working with Java dates and timestamps and therefore conforms to those standards.

Spark will not throw an error if it cannot parse the date; rather, it will just return null. This can be a bit tricky in larger pipelines because you might be expecting your data in one format and getting it in another

Implicit type casting is an easy way to shoot yourself in the foot, especially when dealing with null values or dates in different timezones or formats. We recommend that you parse them explicitly instead of relying on implicit conversions.

## ifnull, nullIf, nvl, and nvl2

There are several other SQL functions that you can use to achieve similar things. ifnull allows you to select the second value if the first is null, and defaults to the first. Alternatively, you could use nullif, which returns null if the two values are equal or else returns the second if they are not. nvl returns the second value if the first is null, but defaults to the first. Finally, nvl2 returns the second value if the first is not null; otherwise, it will return the last specified value (else_value in the following example):

```
-- in SQL
SELECT
ifnull(null, 'return_value'),
nullif('value', 'value'),
nvl(null, 'return_value'),
nvl2('not_null', 'return_value', "else_value")
FROM dfTable LIMIT 1
+------------+----+------------+------------+
| a| b| c| d|
+------------+----+------------+------------+
|return_value|null|return_value|return_value|
+------------+----+------------+------------+
```

## Ordering

As we discussed in Chapter 5, you can use asc_nulls_first, desc_nulls_first, asc_nulls_last, or desc_nulls_last to specify where you would like your null values to appear in an ordered DataFrame.

## Working with Complex Types

There are three kinds of complex types:

1- structs
2- arrays
3- maps

## Structs

You can think of structs as DataFrames within DataFrames. A worked example will illustrate this more clearly. We can create a struct by wrapping a set of columns in parenthesis in a query:

```
df.selectExpr("(Description, InvoiceNo) as complex", "*")
df.selectExpr("struct(Description, InvoiceNo) as complex", "*")
```

## Arrays

To define arrays, let's work through a use case. With our current data, our objective is to take every single word in our Description column and convert that into a row in our DataFrame. The first task is to turn our Description column into a complex type, an array.

## split

We do this by using the split function and specify the delimiter:

```
// in Scala
import org.apache.spark.sql.functions.split
df.select(split(col("Description"), " ")).show(2)
# in Python
from pyspark.sql.functions import split
df.select(split(col("Description"), " ")).show(2)
-- in SQL
SELECT split(Description, ' ') FROM dfTable
+---------------------+
|split(Description, )|
+---------------------+
| [WHITE, HANGING, ...|
| [WHITE, METAL, LA...|
+---------------------+
```

This is quite powerful because Spark allows us to manipulate this complex type as another column. We can also query the values of the array using Python-like syntax:

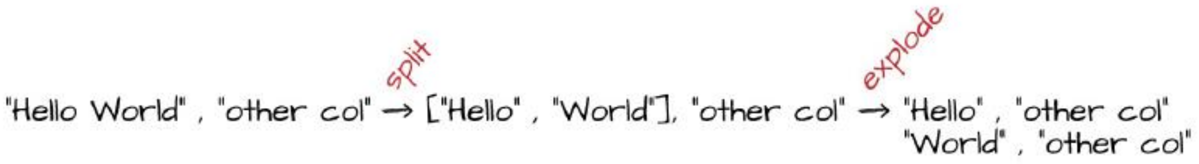
```
// in Scala
df.select(split(col("Description"), " ").alias("array_col"))
.selectExpr("array_col[0]").show(2)
# in Python
df.select(split(col("Description"), " ").alias("array_col"))\
.selectExpr("array_col[0]").show(2)
-- in SQL
SELECT split(Description, ' ')[0] FROM dfTable
```

This gives us the following result:

```
+------------+
|array_col[0]|
+------------+
| WHITE|
| WHITE|
+------------+
```

## Explode

The explode function takes a column that consists of arrays and creates one row (with the rest of the values duplicated) per value in the array



## Maps

Maps are created by using the map function and key-value pairs of columns. You then can select them just like you might select from an array:

```
# in Python
df.select(map(col("Description"), col("InvoiceNo")).alias("complex_map"))\
.selectExpr("complex_map['WHITE METAL LANTERN']").show(2)
```

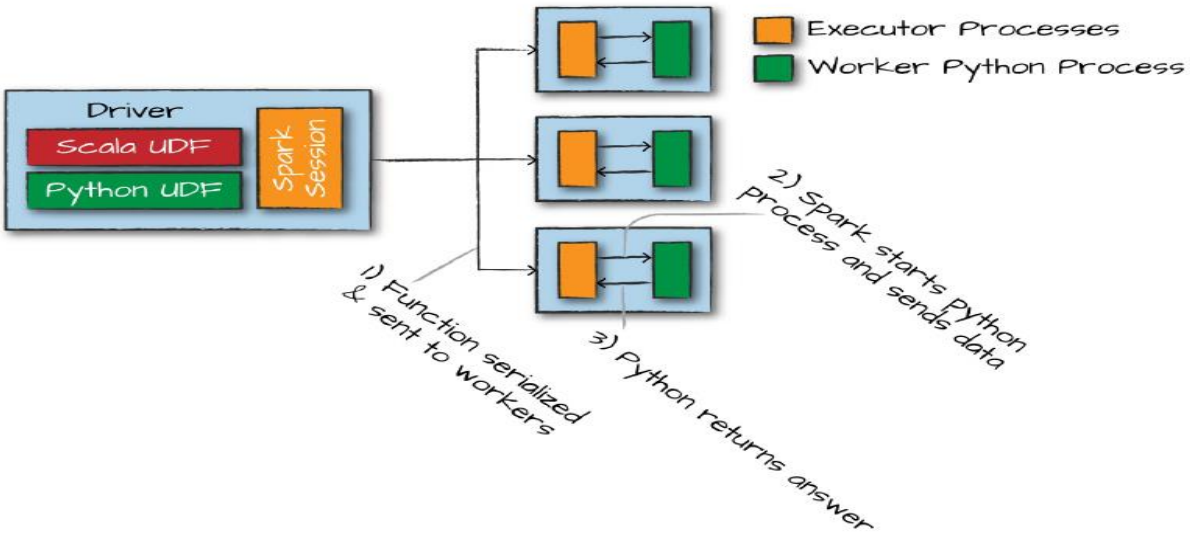
This gives us the following result:

```
+--------------------------------+
|complex_map[WHITE METAL LANTERN]|
+--------------------------------+
| null|
| 536365|
+--------------------------------+
```

## User-Defined Functions

Now that we've created these functions and tested them, we need to register them with Spark so that we can use them on all of our worker machines. Spark will serialize the function on the driver and transfer it over the network to all executor processes. This happens regardless of language.

When you use the function, there are essentially two different things that occur. If the function is written in Scala or Java, you can use it within the Java Virtual Machine (JVM). This means that there will be little performance penalty aside from the fact that you can't take advantage of code generation capabilities that Spark has for built-in functions. There can be performance issues if you create or use a lot of objects; we cover that in the section on optimization in Chapter 19.
If the function is written in Python, something quite different happens. Spark starts a Python process on the worker, serializes all of the data to a format that Python can understand (remember, it was in the JVM earlier), executes the function row by row on that data in the Python process, and then finally returns the results of the row operations to the JVM and Spark.



Starting this Python process is expensive, but the real cost is in serializing the data to Python. This is costly for two reasons: it is an expensive computation, but also, after the data enters Python, Spark cannot manage the memory of the worker. This means that you could potentially cause a worker to fail if it becomes resource constrained (because both the JVM and Python are competing for memory on the same machine). We recommend that you write your UDFs in Scala or Java—the small amount of time it should take you to write the function in Scala will always yield significant speed ups, and on top of that, you can still use the function from Python!

**NOTE:**

we can also register this UDF as a Spark SQL function. This is valuable because it makes it simple to use this function within SQL as well as across languages.

Because this function is registered with Spark SQL—and we've learned that any Spark SQL function or expression is valid to use as an expression when working with DataFrames—we can turn around and use the UDF that we wrote in Scala, in Python. However, rather than using it as a DataFrame function, we use it as a SQL expression:

```
# in Python
udfExampleDF.selectExpr("power3(num)").show(2)
# registered in Scala
```

We can also register our Python function to be available as a SQL function and use that in any language, as well. One thing we can also do to ensure that our functions are working correctly is specify a return type. As we saw in the beginning of this section, Spark manages its own type information, which does not align exactly with Python's types. Therefore, it's a best practice to define the return type for your function when you define it. It is important to note that specifying the return type is not necessary, but it is a best practice. If you specify the type that doesn't align with the actual type returned by the function, Spark will not throw an error but will just return null to designate a failure.

## AGGREGATION:

The simplest grouping is to just summarize a complete DataFrame by performing an aggregation in a select statement.

A “group by” allows you to specify one or more keys as well as one or more aggregation functions to transform the value columns.

A “window” gives you the ability to specify one or more keys as well as one or more aggregation functions to transform the value columns. However, the rows input to the function are somehow related to the current row.

A “grouping set,” which you can use to aggregate at multiple different levels. Grouping sets are available as a primitive in SQL and via rollups and cubes in DataFrames.

A “rollup” makes it possible for you to specify one or more keys as well as one or more aggregation functions to transform the value columns, which will be summarized hierarchically.

A “cube” allows you to specify one or more keys as well as one or more aggregation functions to transform the value columns, which will be summarized across all combinations of columns.

Each grouping returns a **RelationalGroupedDataset** on which we specify our aggregations.

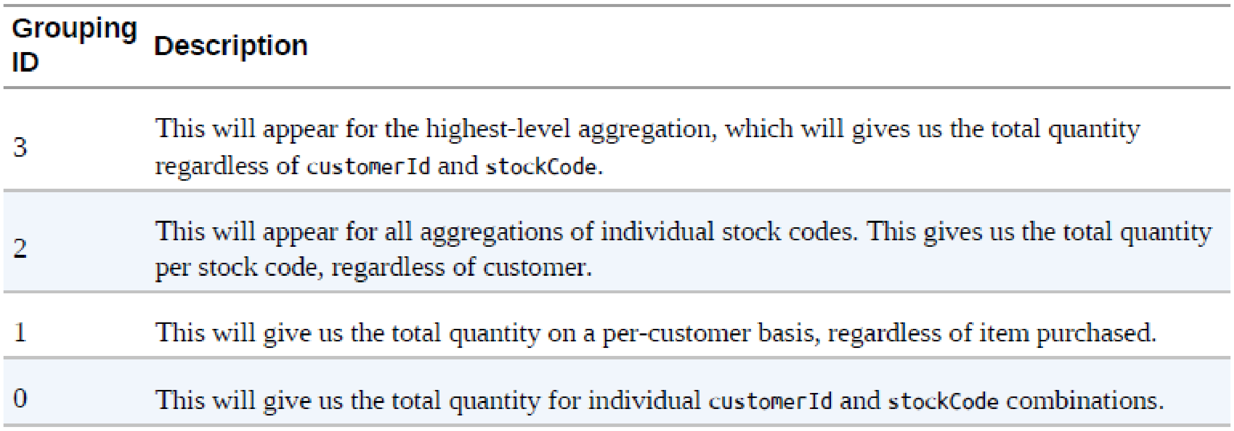
There are a number of gotchas when it comes to null values and counting. For instance, when performing a count(*), Spark will count null values (including rows containing all nulls). However, when counting an individual column, Spark will not count the null values.

## Aggregating to Complex Types

In Spark, you can perform aggregations not just of numerical values using formulas, you can also perform them on complex types. For example, we can collect a list of values present in a given column or only the unique values by collecting to a set.

## Grouping Metadata

Sometimes when using cubes and rollups, you want to be able to query the aggregation levels so that you can easily filter them down accordingly. We can do this by using the grouping_id, which gives us a column specifying the level of aggregation that we have in our result set. The query in the example that follows returns four distinct grouping IDs:



| Grouping ID | Description |
|---|---|
| 3 | This will appear for the highest-level aggregation, which will gives us the total quantity regardless of customerId and stockCode. |
| 2 | This will appear for all aggregations of individual stock codes. This gives us the total quantity per stock code, regardless of customer. |
| 1 | This will give us the total quantity on a per-customer basis, regardless of item purchased. |
| 0 | This will give us the total quantity for individual customerId and stockCode combinations. |

## User-Defined Aggregation Functions

User-defined aggregation functions (UDAFs) are a way for users to define their own aggregation functions based on custom formulae or business rules. You can use UDAFs to compute custom calculations over groups of input data (as opposed to single rows). Spark maintains a single AggregationBuffer to store intermediate results for every group of input data.

To create a UDAF, you must inherit from the UserDefinedAggregateFunction base class and implement the following methods:

- inputSchema represents input arguments as a StructType
- bufferSchema represents intermediate UDAF results as a StructType
- dataType represents the return DataType
- deterministic is a Boolean value that specifies whether this UDAF will return the same result for a given input
- initialize allows you to initialize values of an aggregation buffer
- update describes how you should update the internal buffer based on a given row
- merge describes how two aggregation buffers should be merged
- evaluate will generate the final result of the aggregation

## Join Types

Whereas the join expression determines whether two rows *should* join, the join type determines *what* should be in the result set. There are a variety of different join types available in Spark for you to use:

- Inner joins (keep rows with keys that exist in the left and right datasets)
- Outer joins (keep rows with keys in either the left or right datasets)
- Left outer joins (keep rows with keys in the left dataset)
- Right outer joins (keep rows with keys in the right dataset)
- Left semi joins (keep the rows in the left, and only the left, dataset where the key appears in the right dataset)
- Left anti joins (keep the rows in the left, and only the left, dataset where they do not appear in the right dataset)
- Natural joins (perform a join by implicitly matching the columns between the two datasets with the same names)
- Cross (or Cartesian) joins (match every row in the left dataset with every row in the right dataset)

## Left Semi Joins

Semi joins are a bit of a departure from the other joins. They do not actually include any values from the right DataFrame. They only compare values to see if the value exists in the second DataFrame. If the value does exist, those rows will be kept in the result, even if there are duplicate keys in the left DataFrame. Think of left semi joins as filters on a DataFrame, as opposed to the function of a conventional join

## Left Anti Joins

Left anti joins are the opposite of left semi joins. Like left semi joins, they do not actually include any values from the right DataFrame. They only compare values to see if the value exists in the second DataFrame. However, rather than keeping the values that exist in the second DataFrame, they keep only the values that *do not* have a corresponding key in the second DataFrame. Think of anti joins as a NOT IN SQL-style filter:

## Natural Joins

Natural joins make implicit guesses at the columns on which you would like to join. It finds matching columns and returns the results. Left, right, and outer natural joins are all supported.

**WARNING**

Implicit is always dangerous! The following query will give us incorrect results because the two DataFrames/tables share a column name (id), but it means different things in the datasets. You should always use this join with caution.

```
-- in SQL
SELECT * FROM graduateProgram NATURAL JOIN person
```

## Joins on Complex Types

Even though this might seem like a challenge, it's actually not. Any expression is a valid join expression, assuming that it returns a Boolean:

## Handling Duplicate Column Names

One of the tricky things that come up in joins is dealing with duplicate column names in your results DataFrame. In a DataFrame, each column has a unique ID within Spark's SQL Engine, Catalyst. This unique ID is purely internal and not something that you can directly reference. This makes it quite difficult to refer to a specific column when you have a DataFrame with duplicate column names.

This can occur in two distinct situations:

The join expression that you specify does not remove one key from one of the input DataFrames and the keys have the same column name

Two columns on which you are not performing the join have the same name

Let's create a problem dataset that we can use to illustrate these problems:

```
val gradProgramDupe = graduateProgram.withColumnRenamed("id", "graduate_program")
val joinExpr = gradProgramDupe.col("graduate_program") === person.col(
"graduate_program")
```

Note that there are now two graduate_program columns, even though we joined on that key:

```
person.join(gradProgramDupe, joinExpr).show()
```

The challenge arises when we refer to one of these columns:

```
person.join(gradProgramDupe, joinExpr).select("graduate_program").show()
```

Given the previous code snippet, we will receive an error. In this particular example, Spark generates this message:

```
org.apache.spark.sql.AnalysisException: Reference 'graduate_program' is
ambiguous, could be: graduate_program#40, graduate_program#1079.;
```

## Approach 1: Different join expression

When you have two keys that have the same name, probably the easiest fix is to change the join expression from a Boolean expression to a string or sequence. This automatically removes one of the columns for you during the join:

```
person.join(gradProgramDupe,"graduate_program").select("graduate_program").show()
```

## Approach 2: Dropping the column after the join

Another approach is to drop the offending column after the join. When doing this, we need to refer to the column via the original source DataFrame. We can do this if the join uses the same key names or if the source DataFrames have columns that simply have the same name:

```
person.join(gradProgramDupe, joinExpr).drop(person.col("graduate_program"))
.select("graduate_program").show()
val joinExpr = person.col("graduate_program") === graduateProgram.col("id")
person.join(graduateProgram, joinExpr).drop(graduateProgram.col("id")).show()
```

This is an artifact of Spark's SQL analysis process in which an explicitly referenced column will pass analysis because Spark has no need to resolve the column. Notice how the column uses the .col method instead of a column function. That allows us to implicitly specify that column by its specific ID.

## Approach 3: Renaming a column before the join

We can avoid this issue altogether if we rename one of our columns before the join:

```
val gradProgram3 = graduateProgram.withColumnRenamed("id", "grad_id")
val joinExpr = person.col("graduate_program") === gradProgram3.col("grad_id")
person.join(gradProgram3, joinExpr).show()
```

## How Spark Performs Joins

To understand how Spark performs joins, you need to understand the two core resources at play: the *node-to-node communication strategy* and *per node computation strategy*. These internals are likely irrelevant to your business problem. However, comprehending how Spark performs joins can mean the difference between a job that completes quickly and one that never completes at all.

## Basics of Reading Data

The foundation for reading data in Spark is the **DataFrameReader**. We access this through the SparkSession via the read attribute:

```
spark.read
```

After we have a DataFrame reader, we specify several values:

The *format*
The *schema*
The *read mode*
A series of *options*

The format, options, and schema each return a **DataFrameReader** that can undergo further transformations and are all optional, except for one option.

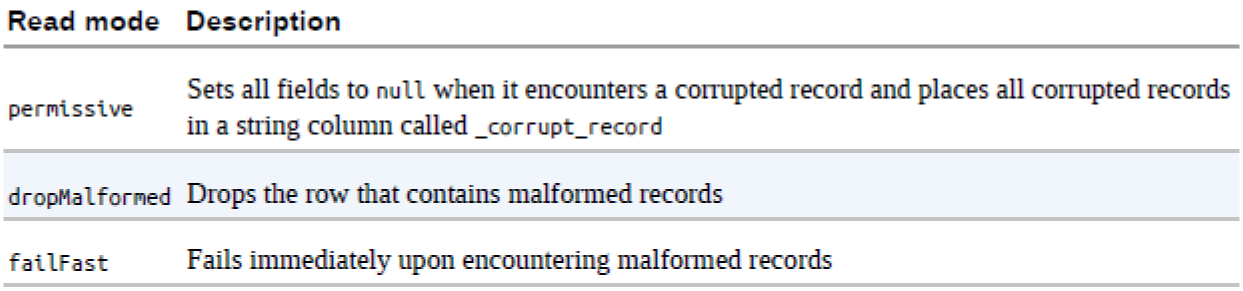
Each data source has a specific set of options that determine how the data is read into Spark (we cover these options shortly). At a minimum, you must supply the **DataFrameReader** a path to from which to read.

Here’s an example of the overall layout:

```
spark.read.format("csv")
.option("mode", "FAILFAST")
.option("inferSchema", "true")
.option("path", "path/to/file(s)")
.schema(someSchema)
.load()
```

## Read modes

Reading data from an external source naturally entails encountering malformed data, especially when working with only semi-structured data sources. Read modes specify what will happen when Spark does come across malformed records.



| Read mode | Description |
|---|---|
| `permissive` | Sets all fields to `null` when it encounters a corrupted record and places all corrupted records in a string column called `_corrupt_record` |
| `dropMalformed` | Drops the row that contains malformed records |
| `failFast` | Fails immediately upon encountering malformed records |

The default is permissive.

## Write API Structure

The core structure for writing data is as follows:

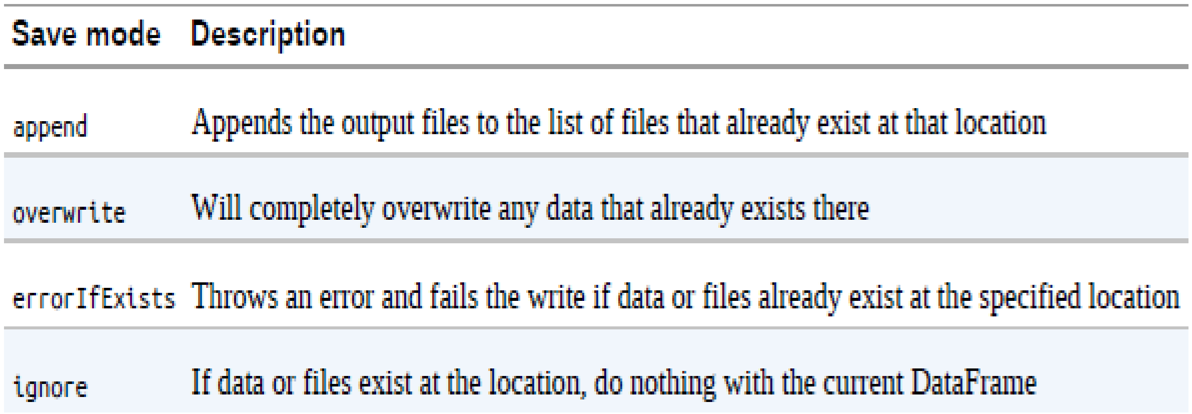
```
DataFrameWriter.format(...).option(...).partitionBy(...).bucketBy(...).sortBy(
...).save()
```

We will use this format to write to all of our data sources. format is optional because by default, Spark will use the arquet format. option, again, allows us to configure how to write out our given data. **PartitionBy**, **bucketBy**, and **sortBy** work only for file-based data sources; you can use them to control the specific layout of files at the destination.

After we have a **DataFrameWriter**, we specify three values: the **format**, a series of **options**, and the **save** mode. At a minimum, you must supply a path.

### Save modes

Save modes specify what will happen if Spark finds data at the specified location (assuming all else equal).



| Save mode | Description |
|---|---|
| append | Appends the output files to the list of files that already exist at that location |
| overwrite | Will completely overwrite any data that already exists there |
| errorIfExists | Throws an error and fails the write if data or files already exist at the specified location |
| ignore | If data or files exist at the location, do nothing with the current DataFrame |

The default is **errorIfExists**. This means that if Spark finds data at the location to which you're writing, it will fail the write immediately.

## JSON Files

Those coming from the world of JavaScript are likely familiar with JavaScript Object Notation, or JSON, as it's commonly called. There are some catches when working with this kind of data that are worth considering before we jump in. In Spark, when we refer to JSON files, we refer to ***line-delimited*** JSON files. This contrasts with files that have a large JSON object or array per file.

The line-delimited versus multiline trade-off is controlled by a single option: multiLine. When you set this option to true, you can read an entire file as one json object and Spark will go through the work of parsing that into a DataFrame. Line-delimited JSON is actually a much more stable format because it allows you to append to a file with a new record (rather than having to read in an entire file and then write it out), which is what we recommend that you use. Another key reason for the popularity of line-delimited JSON is because JSON objects have structure, and JavaScript (on which JSON is based) has at least basic types. This makes it easier to work with because Spark can make more assumptions on our behalf about the data.

## Parquet

Even though there are only two options, you can still encounter problems if you're working with incompatible Parquet files. Be careful when you write out Parquet files with different versions of Spark (especially older ones) because this can cause significant headache.

## ORC Files

ORC is a self-describing, type-aware columnar file format designed for Hadoop workloads. It is optimized for large streaming reads, but with integrated support for finding required rows quickly. ORC actually has no options for reading in data because Spark understands the file format quite well. An often-asked question is: What is the difference between ORC and Parquet? For the most part, they're quite similar; the fundamental difference is that Parquet is further optimized for use with Spark, whereas ORC is further optimized for Hive.

## Reading from databases in parallel

All throughout this book, we have talked about partitioning and its importance in data processing. Spark has an underlying algorithm that can read multiple files into one partition, or conversely, read multiple partitions out of one file, depending on the file size and the “splitability” of the file type and compression. The same flexibility that exists with files, also exists with SQL databases except that you must configure it a bit more manually. What you can configure, as seen in the previous options, is the ability to specify a maximum number of partitions to allow you to limit how much you are reading and writing in parallel:

```
// in Scala
val dbDataFrame = spark.read.format("jdbc")
.option("url", url).option("dbtable", tablename).option("driver", driver)
.option("numPartitions", 10).load()
# in Python
dbDataFrame = spark.read.format("jdbc")\
.option("url", url).option("dbtable", tablename).option("driver", driver)\
.option("numPartitions", 10).load()
```

In this case, this will still remain as one partition because there is not too much data. However, this configuration can help you ensure that you do not overwhelm the database when reading and writing data:

```
dbDataFrame.select("DEST_COUNTRY_NAME").distinct().show()
```

There are several other optimizations that unfortunately only seem to be under another API set. You can explicitly push predicates down into SQL databases through the connection itself. This optimization allows you to control the physical location of certain data in certain partitions by specifying predicates. That’s a mouthful, so let’s look at a simple example. We only need data from two countries in our data: Anguilla and Sweden. We could filter these down and have them pushed into the database, but we can also go further by having them arrive in their own partitions in Spark. We do that by specifying a list of predicates when we create the data source:

```
// in Scala
val props = new java.util.Properties
props.setProperty("driver", "org.sqlite.JDBC")
val predicates = Array(
"DEST_COUNTRY_NAME = 'Sweden' OR ORIGIN_COUNTRY_NAME = 'Sweden'",
"DEST_COUNTRY_NAME = 'Anguilla' OR ORIGIN_COUNTRY_NAME = 'Anguilla'")
spark.read.jdbc(url, tablename, predicates, props).show()
spark.read.jdbc(url, tablename, predicates, props).rdd.getNumPartitions // 2
# in Python
props = {"driver":"org.sqlite.JDBC"}
predicates = [
"DEST_COUNTRY_NAME = 'Sweden' OR ORIGIN_COUNTRY_NAME = 'Sweden'",
"DEST_COUNTRY_NAME = 'Anguilla' OR ORIGIN_COUNTRY_NAME = 'Anguilla'"]
spark.read.jdbc(url, tablename, predicates=predicates, properties=props).show()
spark.read.jdbc(url,tablename,predicates=predicates,properties=props)\
.rdd.getNumPartitions() # 2
```

```
+-----------------+-------------------+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----------------+-------------------+-----+
| Sweden| United States| 65|
| United States| Sweden| 73|
| Anguilla| United States| 21|
| United States| Anguilla| 20|
+-----------------+-------------------+-----+
```

If you specify predicates that are not disjoint, you can end up with lots of duplicate rows. Here’s an example set of predicates that will result in duplicate rows:

```
// in Scala
val props = new java.util.Properties
props.setProperty("driver", "org.sqlite.JDBC")
val predicates = Array(
"DEST_COUNTRY_NAME != 'Sweden' OR ORIGIN_COUNTRY_NAME != 'Sweden'",
"DEST_COUNTRY_NAME != 'Anguilla' OR ORIGIN_COUNTRY_NAME != 'Anguilla'")
spark.read.jdbc(url, tablename, predicates, props).count() // 510
# in Python
props = {"driver":"org.sqlite.JDBC"}
predicates = [
"DEST_COUNTRY_NAME != 'Sweden' OR ORIGIN_COUNTRY_NAME != 'Sweden'",
"DEST_COUNTRY_NAME != 'Anguilla' OR ORIGIN_COUNTRY_NAME != 'Anguilla'"]
spark.read.jdbc(url, tablename, predicates=predicates, properties=props).count()
```

## Writing Text Files

When you write a text file, you need to be sure to have only one string column; otherwise, the write will fail. If you perform some partitioning when performing your write (we’ll discuss partitioning in the next couple of pages), you can write more columns. However, those columns will manifest as directories in the folder to which you’re writing out to, instead of columns on every single file.

## Advanced I/O Concepts

We saw previously that we can control the parallelism of files that we write by controlling the partitions prior to writing. We can also control specific data layout by controlling two things: *bucketing* and *partitioning* (discussed momentarily).

## Splittable File Types and Compression

Certain file formats are fundamentally “splittable.” This can improve speed because it makes it possible for Spark to avoid reading an entire file, and access only the parts of the file necessary to satisfy your query. Additionally if you’re using something like Hadoop Distributed File System (HDFS), splitting a file can provide further optimization if that file spans multiple blocks. In conjunction with this is a need to manage compression. Not all compression schemes are splittable. How you store your data is of immense consequence when it comes to making your Spark jobs run smoothly. We recommend Parquet with gzip compression

## Reading Data in Parallel

Multiple executors cannot read from the same file at the same time necessarily, but they can read different files at the same time. In general, this means that when you read from a folder with multiple files in it, each one of those files will become a partition in your DataFrame and be read in by available executors in parallel (with the remaining queueing up behind the others).

## Writing Data in Parallel

The number of files or data written is dependent on the number of partitions the DataFrame has at the time you write out the data. By default, one file is written per partition of the data. This means that although we specify a "file," it's actually a number of files within a folder, with the name of the specified file, with one file per each partition that is written

## Bucketing

Bucketing is another file organization approach with which you can control the data that is specifically written to each file. This can help avoid shuffles later when you go to read the data because data with the same bucket ID will all be grouped together into one physical partition. This means that the data is prepartitioned according to how you expect to use that data later on, meaning you can avoid expensive shuffles when joining or aggregating.
Rather than partitioning on a specific column (which might write out a ton of directories), it's probably worthwhile to explore bucketing the data instead. This will create a certain number of files and organize our data into those "buckets"

**NOTE**: CSV files do not support complex types, whereas Parquet and ORC do.

## Managing File Size

Managing file sizes is an important factor not so much for writing data but reading it later on. When you're writing lots of small files, there's a significant metadata overhead that you incur managing all of those files. Spark especially does not do well with small files, although many file systems (like HDFS) don't handle lots of small files well, either. You might hear this referred to as the "small file problem." The opposite is also true: you don't want files that are too large either, because it becomes inefficient to have to read entire blocks of data when you need only a few rows.
Spark 2.2 introduced a new method for controlling file sizes in a more automatic way. We saw previously that the number of output files is a derivative of the number of partitions we had at write time (and the partitioning columns we selected). Now, you can take advantage of another tool in order to limit output file sizes so that you can target an optimum file size. You can use the **maxRecordsPerFile** option and specify a number of your choosing. This allows you to better control file sizes by controlling the number of records that are written to each file. For example, if you set an option for a writer as df.write.option("maxRecordsPerFile", 5000), Spark will ensure that files will contain at most 5,000 records.

## Catalog

The Catalog is an abstraction for the storage of metadata about the data stored in your tables as well as other helpful things like databases, tables, functions, and views. The catalog is available in the **org.apache.spark.sql.catalog.Catalog** package and contains a number of helpful functions for doing things like listing tables, databases, and functions.

## Difference between DataFrame and Table

The core difference between tables and DataFrames is this: you define DataFrames in the scope of a programming language, whereas you define tables within a database.

An important thing to note is that in Spark 2.X, tables *always contain data*. There is no notion of a temporary table, only a view, which does not contain data. This is important because if you go to drop a table, you can risk losing the data when doing so.

## Spark-Managed Tables

One important note is the concept of *managed* versus *unmanaged* tables. Tables store two important pieces of information. The data within the tables as well as the data about the tables; that is, the *metadata*. You can have Spark manage the metadata for a set of files as well as for the data. When you define a table from files on disk, you are defining an **unmanaged table**. When you use **saveAsTable** on a DataFrame, you are creating a **managed table** for which Spark will track of all of the relevant information.

## USING AND STORED AS

The specification of the USING syntax in the previous example is of significant importance. If you do not specify the format, Spark will default to a Hive SerDe configuration. This has performance implications for future readers and writers because Hive SerDes are much slower than Spark's native serialization. Hive users can also use the STORED AS syntax to specify that this should be a Hive table.

## Creating External Tables

As we mentioned in the beginning of this chapter, Hive was one of the first big data SQL systems, and Spark SQL is completely compatible with Hive SQL (HiveQL) statements. One of the use cases that you might encounter is to port your legacy Hive statements to Spark SQL. Luckily, you can, for the most part, just copy and paste your Hive statements directly into Spark SQL. For example, in the example that follows, we create an *unmanaged table*. Spark will manage the table's metadata; however, the files are not managed by Spark at all. You create this table by using the CREATE EXTERNAL TABLE statement. You can view any files that have already been defined by running the following command:

```
CREATE EXTERNAL TABLE hive_flights (
DEST_COUNTRY_NAME STRING, ORIGIN_COUNTRY_NAME STRING, count LONG)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' LOCATION '/data/flight-data-hive/'
```

You can also create an external table from a select clause:

```
CREATE EXTERNAL TABLE hive_flights_2
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
LOCATION '/data/flight-data-hive/' AS SELECT * FROM flights
```

## Encoder:

To efficiently support domain-specific objects, a special concept called an “Encoder” is required. The encoder maps the domain-specific type T to Spark’s internal type system.

For example, given a class Person with two fields, name (string) and age (int), an encoder directs Spark to generate code at runtime to serialize the Person object into a binary structure. When using DataFrames or the “standard” Structured APIs, this binary structure will be a Row. When we want to create our own domain-specific objects, we specify a case class in Scala or a JavaBean in Java. Spark will allow us to manipulate this object (in place of a Row) in a distributed manner.

Spark converts the Spark Row format to the object you specified (a case class or Java class). This conversion slows down your operations but can provide more flexibility. You will notice a hit in performance but this is a far different order of magnitude from what you might see from something like a user-defined function (UDF) in Python, because the performance costs are not as extreme as switching programming languages, but it is an important thing to keep in mind.

## When to Use Datasets

- When the operation(s) you would like to perform cannot be expressed using DataFrame manipulations
- When you want or need type-safety, and you’re willing to accept the cost of performance to achieve it

## Case Classes

To create Datasets in Scala, you define a Scala case class. A case class is a regular class that has the following characteristics:

- Immutable
- Decomposable through pattern matching
- Allows for comparison based on structure instead of reference
- Easy to use and manipulate

These traits make it rather valuable for data analysis because it is quite easy to reason about a case class. Probably the most important feature is that case classes are immutable and allow for comparison by structure instead of value

## When to Use the Low-Level APIs?

You should generally use the lower-level APIs in three situations:

- You need some functionality that you cannot find in the higher-level APIs; for example,
- If you need very tight control over physical data placement across the cluster.
- You need to maintain some legacy codebase written using RDDs.

- You need to do some custom shared variable manipulation. We will discuss shared variables

## SPARK CONTEXT

A **SparkContext** is the entry point for low-level API functionality. You access it through the **SparkSession**, which is the tool you use to perform computation across a Spark cluster

## RDD

RDD represents an immutable, partitioned collection of records that can be operated on in parallel. Unlike DataFrames though, where each record is a structured row containing fields with a known schema, in RDDs the records are just Java, Scala, or Python objects of the programmer's choosing.

## Types of RDDs

1. generic RDD
2. a key-value RDD

Both just represent a collection of objects, but key-value RDDs have special operations as well as a concept of custom partitioning by key.

Let's formally define RDDs. Internally, each RDD is characterized by five main properties:

1. A list of partitions
2. A function for computing each split
3. A list of dependencies on other RDDs
4. Optionally, a Partitioner for key-value RDDs (e.g., to say that the RDD is hashpartitioned)
5. Optionally, a list of preferred locations on which to compute each split (e.g., block locations for a Hadoop Distributed File System [HDFS] file)

RDDs give you complete control because every record in an RDD is a just a Java or Python object.

Spark's Structured APIs automatically store data in an optimized , compressed binary format

RDDs follow the exact same Spark programming paradigms that we saw in earlier chapters. They provide *transformations*, which **evaluate lazily**, and *actions*, which **evaluate eagerly**, to manipulate data in a distributed fashion

## When to Use RDDs?

The most likely reason for why you'll want to use RDDs is because you need fine-grained control over the physical distribution of data (custom partitioning of data).

## StorageLevels

### There are different PySpark StorageLevels to decide the storage of RDD, such as:

- **DISK_ONLY:** StorageLevel(True, False, False, False, 1)
- **DISK_ONLY_2:** StorageLevel(True, False, False, False, 2)
- **MEMORY_AND_DISK:** StorageLevel(True, True, False, False, 1)
- **MEMORY_AND_DISK_2:** StorageLevel(True, True, False, False, 2)
- **MEMORY_AND_DISK_SER:** StorageLevel(True, True, False, False, 1)
- **MEMORY_AND_DISK_SER_2:** StorageLevel(True, True, False, False, 2)
- **MEMORY_ONLY:** StorageLevel(False, True, False, False, 1)
- **MEMORY_ONLY_2:** StorageLevel(False, True, False, False, 2)
- **MEMORY_ONLY_SER:** StorageLevel(False, True, False, False, 1)
- **MEMORY_ONLY_SER_2:** StorageLevel(False, True, False, False, 2)
- **OFF_HEAP:** StorageLevel(True, True, True, False, 1)

## Spark Jobs Not Starting

This issue can arise frequently, especially when you're just getting started with a fresh deployment or environment.

### Signs and symptoms

- Spark jobs don't start.
- The Spark UI doesn't show any nodes on the cluster except the driver.
- The Spark UI seems to be reporting incorrect information

### Potential treatments

This mostly occurs when your cluster or your application's resource demands are not configured properly. Spark, in a distributed setting, does make some assumptions about networks, file systems, and other resources. During the process of setting up the cluster, you likely configured something incorrectly, and now the node that runs the driver cannot talk to the executors. This might be because you didn't specify what IP and port is open or didn't open the correct one. This is most likely a cluster level, machine, or configuration issue. Another option is that your application requested more resources per executor than your cluster manager currently has free, in which case the driver will be waiting forever for executors to be launched.

- Ensure that machines can communicate with one another on the ports that you expect. Ideally, you should open up all ports between the worker nodes unless you have more stringent security constraints.
- Ensure that your Spark resource configurations are correct and that your cluster manager is properly set up for Spark. Try running a simple application first to see if that works. One common issue may be that you requested more memory per executor than the cluster

manager has free to allocate, so check how much it is reporting free (in its UI) and your spark-submit memory configuration.

## Errors before Execution

This can happen when you're developing a new application and have previously run code on this cluster, but now some new code won't work.

## Signs and symptoms

- Commands don't run at all and output large error messages.
- You check the Spark UI and no jobs, stages, or tasks seem to run.

## Potential treatments

After checking and confirming that the Spark UI environment tab shows the correct information for your application, it's worth double-checking your code. Many times, there might be a simple typo or incorrect column name that is preventing the Spark job from compiling into its underlying Spark plan (when using the DataFrame API).

- You should take a look at the error returned by Spark to confirm that there isn't an issue in your code, such as providing the wrong input file path or field name.
- Double-check to verify that the cluster has the network connectivity that you expect between your driver, your workers, and the storage system you are using.
- There might be issues with libraries or classpaths that are causing the wrong version of a library to be loaded for accessing storage. Try simplifying your application until you get a smaller version that reproduces the issue (e.g., just reading one dataset).

## Errors during Execution

This kind of issue occurs when you already are working on a cluster or parts of your Spark Application run before you encounter an error. This can be a part of a scheduled job that runs at some interval or a part of some interactive exploration that seems to fail after some time.

### Signs and symptoms

- One Spark job runs successfully on the entire cluster but the next one fails.
- A step in a multistep query fails.
- A scheduled job that ran yesterday is failing today.
- Difficult to parse error message.

## Potential treatments

- Check to see if your data exists or is in the format that you expect. This can change over time or some upstream change may have had unintended consequences on your application.
- If an error quickly pops up when you run a query (i.e., before tasks are launched), it is most likely an *analysis error* while planning the query. This means that you likely

misspelled a column name referenced in the query or that a column, view, or table you referenced does not exist.

- Read through the stack trace to try to find clues about what components are involved (e.g., what operator and stage it was running in).
- Try to isolate the issue by progressively double-checking input data and ensuring the data conforms to your expectations. Also try removing logic until you can isolate the problem in a smaller version of your application.
- If a job runs tasks for some time and then fails, it could be due to a problem with the input data itself, wherein the schema might be specified incorrectly or a particular row does not conform to the expected schema. For instance, sometimes your schema might specify that the data contains no nulls but your data does actually contain nulls, which can cause certain transformations to fail.
- It’s also possible that your own code for processing the data is crashing, in which case Spark will show you the exception thrown by your code. In this case, you will see a task marked as “failed” on the Spark UI, and you can also view the logs on that machine to understand what it was doing when it failed. Try adding more logs inside your code to figure out which data record was being processed

## Slow Tasks or Stragglers

This issue is quite common when optimizing applications, and can occur either due to work not being evenly distributed across your machines (“skew”), or due to one of your machines being slower than the others (e.g., due to a hardware problem).

### Signs and symptoms

Any of the following are appropriate symptoms of the issue:

- Spark stages seem to execute until there are only a handful of tasks left. Those tasks then take a long time.
- These slow tasks show up in the Spark UI and occur consistently on the same dataset(s).
- These occur in stages, one after the other.
- Scaling up the number of machines given to the Spark Application doesn’t really help—some tasks still take much longer than others.
- In the Spark metrics, certain executors are reading and writing much more data than others.

## Potential treatments

Slow tasks are often called “stragglers.” There are many reasons they may occur, but most often the source of this issue is that your data is partitioned unevenly into DataFrame or RDD partitions. When this happens, some executors might need to work on much larger amounts of work than others. One particularly common case is that you use a group-by-key operation and one of the keys just has more data than others. In this case, when you look at the Spark UI, you might see that the shuffle data for some nodes is much larger than for others.

- Try increasing the number of partitions to have less data per partition.
- Try repartitioning by another combination of columns. For example, stragglers can come up when you partition by a skewed ID column, or a column where many values are null. In the latter case, it might make sense to first filter out the null values.

- Try increasing the memory allocated to your executors if possible. Monitor the executor that is having trouble and see if it is the same machine across jobs; you might also have an unhealthy executor or machine in your cluster—for example, one whose disk is nearly full.
- Check whether your user-defined functions (UDFs) are wasteful in their object allocation or business logic. Try to convert them to DataFrame code if possible.
- Ensure that your UDFs or User-Defined Aggregate Functions (UDAFs) are running on a small enough batch of data. Oftentimes an aggregation can pull a lot of data into memory for a common key, leading to that executor having to do a lot more work than others.
- Another common issue can arise when you're working with Datasets. Because Datasets perform a lot of object instantiation to convert records to Java objects for UDFs, they can cause a lot of garbage collection. If you're using Datasets, look at the garbage collection metrics in the Spark UI to see if they're consistent with the slow tasks.

## Slow Aggregations

If you have a slow aggregation, start by reviewing the issues in the "Slow Tasks" section before proceeding. Having tried those, you might continue to see the same problem.

### Signs and symptoms

- Slow tasks during a groupBy call.
- Jobs after the aggregation are slow, as well.

### Potential treatments

Unfortunately, this issue can't always be solved. Sometimes, the data in your job just has some skewed keys, and the operation you want to run on them needs to be slow.

- Increasing the number of partitions, prior to an aggregation, might help by reducing the number of different keys processed in each task.
- Increasing executor memory can help alleviate this issue, as well. If a single key has lots of data, this will allow its executor to spill to disk less often and finish faster, although it may still be much slower than executors processing other keys.
- If you find that tasks after the aggregation are also slow, this means that your dataset might have remained unbalanced after the aggregation. Try inserting a repartition call to partition it randomly.
- Ensuring that all filters and SELECT statements that can be are above the aggregation can help to ensure that you're working only on the data that you need to be working on and nothing else. Spark's query optimizer will automatically do this for the structured APIs.
- Ensure null values are represented correctly (using Spark's concept of null) and not as some default value like " " or "EMPTY". Spark often optimizes for skipping nulls early in the job when possible, but it can't do so for your own placeholder values.
- Some aggregation functions are also just inherently slower than others. For instance, collect_list and collect_set are very slow aggregation functions because they *must* return all the matching objects to the driver, and should be avoided in performance-critical code.

## Slow Joins

Joins and aggregations are both shuffles, so they share some of the same general symptoms as well as treatments.

### Signs and symptoms

- A join stage seems to be taking a long time. This can be one task or many tasks.
- Stages before and after the join seem to be operating normally.

### Potential treatments

- Many joins can be optimized (manually or automatically) to other types of joins.
- Experimenting with different join orderings can really help speed up jobs, especially if some of those joins filter out a large amount of data; do those first.
- Partitioning a dataset prior to joining can be very helpful for reducing data movement across the cluster, especially if the same dataset will be used in multiple join operations. It's worth experimenting with different prejoin partitioning. Keep in mind, again, that this isn't "free" and does come at the cost of a shuffle.
- Slow joins can also be caused by data skew. There's not always a lot you can do here, but sizing up the Spark application and/or increasing the size of executors can help, as described in earlier sections.
- Ensuring that all filters and select statements that can be are above the join can help to ensure that you're working only on the data that you need for the join.
- Ensure that null values are handled correctly (that you're using null) and not some default value like " " or "EMPTY", as with aggregations.
- Sometimes Spark can't properly plan for a broadcast join if it doesn't know any statistics about the input DataFrame or table. If you know that one of the tables that you are joining is small, you can try to force a broadcast, or use Spark's statistics collection commands to let it analyze the table.

## Driver OutOfMemoryError or Driver Unresponsive

This is usually a pretty serious issue because it will crash your Spark Application. It often happens due to collecting too much data back to the driver, making it run out of memory.

### Signs and symptoms

- Spark Application is unresponsive or crashed.
- OutOfMemoryErrors or garbage collection messages in the driver logs.
- Commands take a very long time to run or don't run at all.
- Interactivity is very low or non-existent.
- Memory usage is high for the driver JVM.

### Potential treatments

There are a variety of potential reasons for this happening, and diagnosis is not always straightforward.

- Your code might have tried to collect an overly large dataset to the driver node using operations such as collect.
- You might be using a broadcast join where the data to be broadcast is too big. Use Spark's maximum broadcast join configuration to better control the size it will broadcast.
- A long-running application generated a large number of objects on the driver and is unable to release them. Java's *jmap* tool can be useful to see what objects are filling most of the memory of your driver JVM by printing a histogram of the heap. However, take note that *jmap* will pause that JVM while running.
- Increase the driver's memory allocation if possible to let it work with more data.

- Issues with JVMs running out of memory can happen if you are using another language binding, such as Python, due to data conversion between the two requiring too much memory in the JVM. Try to see whether your issue is specific to your chosen language and bring back less data to the driver node, or write it to a file instead of bringing it back as in-memory objects.
- If you are sharing a SparkContext with other users (e.g., through the SQL JDBC server and some notebook environments), ensure that people aren’t trying to do something that might be causing large amounts of memory allocation in the driver

## Executor OutOfMemoryError or Executor Unresponsive

Spark applications can sometimes recover from this automatically, depending on the true underlying issue.

### Signs and symptoms

- OutOfMemoryErrors or garbage collection messages in the executor logs. You can find these in the Spark UI.
- Executors that crash or become unresponsive.
- Slow tasks on certain nodes that never seem to recover.

### Potential treatments

- Try increasing the memory available to executors and the number of executors.
- Try increasing PySpark worker size via the relevant Python configurations.
- Look for garbage collection error messages in the executor logs. Some of the tasks that are running, especially if you’re using UDFs, can be creating lots of objects that need to be garbage collected. Repartition your data to increase parallelism, reduce the amount of records per task, and ensure that all executors are getting the same amount of work.
- Ensure that null values are handled correctly (that you’re using null) and not some default value like " " or "EMPTY", as we discussed earlier.
- This is more likely to happen with RDDs or with Datasets because of object instantiations. Try using fewer UDFs and more of Spark’s structured operations when possible.
- Use Java monitoring tools such as *jmap* to get a histogram of heap memory usage on your executors, and see which classes are taking up the most space.
- If executors are being placed on nodes that also have other workloads running on them, such as a key-value store, try to isolate your Spark jobs from other jobs.

## Performance Tuning

There are a variety of different parts of Spark jobs that you might want to optimize, and it’s valuable to be specific. Following are some of the areas:

- Code-level design choices (e.g., RDDs versus DataFrames)
- Data at rest
- Joins
- Aggregations
- Data in flight
- Individual application properties

- Inside of the Java Virtual Machine (JVM) of an executor
- Worker nodes
- Cluster and deployment properties

We can either do so ***indirectly*** by setting configuration values or changing the runtime environment. These should improve things across Spark Applications or across Spark jobs. Alternatively, we can try to ***directly*** change execution characteristic or design choices at the individual Spark job, stage, or task level. These kinds of fixes are very specific to that one area of our application and therefore have limited overall impact.
One of the best things you can do to figure out how to improve performance is to implement good monitoring and job history tracking. Without this information, it can be difficult to know whether you're really improving job performance

## UDF

Things do get a bit more complicated when you need to include custom transformations that cannot be created in the Structured APIs. These might manifest themselves as RDD transformations or user-defined functions (UDFs). If you're going to do this, R and Python are not necessarily the best choice simply because of how this is actually executed. It's also more difficult to provide stricter guarantees of types and manipulations when you're defining functions that jump across languages. We find that using Python for the majority of the application, and porting some of it to Scala or writing specific UDFs in Scala as your application evolves, is a powerful technique—it allows for a nice balance between overall usability, maintainability, and performance.

## Dynamic allocation

Spark provides a mechanism to dynamically adjust the resources your application occupies based on the workload. This means that your application can give resources back to the cluster if they are no longer used, and request them again later when there is demand. This feature is particularly useful if multiple applications share resources in your Spark cluster. This feature is disabled by default and available on all coarse-grained cluster managers; that is, standalone mode, YARN mode, and Mesos coarse-grained mode. If you'd like to enable this feature, you should set **spark.dynamicAllocation.enabled** to true. The Spark documentation presents a number of individual parameters that you can tune.

## Splittable file types and compression

Whatever file format you choose, you should make sure it is "splittable", which means that different tasks can read different parts of the file in parallel. When we read in the file, all cores were able to do part of the work. That's because the file was splittable. If we didn't use a splittable file type—say something like a malformed JSON file—we're going to need to read in the entire file on a single machine, greatly reducing parallelism.
The main place splittability comes in is compression formats. A ZIP file or TAR archive cannot be split, which means that even if we have 10 files in a ZIP file and 10 cores, only one core can

read in that data because we cannot parallelize access to the ZIP file. This is a poor use of resources. In contrast, files compressed using gzip, bzip2, or lz4 are generally splittable if they were written by a parallel processing framework like Hadoop or Spark. For your own input data, the simplest way to make it splittable is to upload it as separate files, ideally each no larger than a few hundred megabytes.

## Table partitioning

We discussed table partitioning in Chapter 9, and will only use this section as a reminder. Table partitioning refers to storing files in separate directories based on a key, such as the date field in the data. Storage managers like Apache Hive support this concept, as do many of Spark's built-in data sources. Partitioning your data correctly allows Spark to skip many irrelevant files when it only requires data with a specific range of keys. For instance, if users frequently filter by "date" or "customerId" in their queries, partition your data by those columns. This will greatly reduce the amount of data that end users must read by most queries, and therefore dramatically increase speed.

The one downside of partitioning, however, is that if you partition at too fine a granularity, it can result in many small files, and a great deal of overhead trying to list all the files in the storage system.

## The number of files

In addition to organizing your data into buckets and partitions, you'll also want to consider the number of files and the size of files that you're storing. If there are lots of small files, you're going to pay a price listing and fetching each of those individual files. For instance, if you're reading a data from Hadoop Distributed File System (HDFS), this data is managed in blocks that are up to 128 MB in size (by default). This means if you have 30 files, of 5 MB each, you're going to have to potentially request 30 blocks, even though the same data could have fit into 2 blocks (150 MB total).

Although there is not necessarily a panacea for how you want to store your data, the trade-off can be summarized as such. Having lots of small files is going to make the scheduler work much harder to locate the data and launch all of the read tasks. This can increase the network and scheduling overhead of the job. Having fewer large files eases the pain off the scheduler but it will also make tasks run longer. In this case, though, you can always launch more tasks than there are input files if you want more parallelism—Spark will split each file across multiple tasks assuming you are using a splittable format. In general, we recommend sizing your files so that they each contain at least a few tens of megatbytes of data.

One way of controlling data partitioning when you write your data is through a write option introduced in Spark 2.2. To control how many records go into each file, you can specify the maxRecordsPerFile option to the write operation.

## Data Locality

Data locality in simple terms means doing computation on the node where data resides.

Just to elaborate:

Spark is cluster computing system. It is not a storage system like HDFS or NOSQL. Spark is used to process the data stored in such distributed system.

Typical installation is like spark is installed on same nodes as that of HDFS/NOSQL.

In case there is a spark application which is processing data stored HDFS. Spark tries to place computation tasks alongside HDFS blocks.

With HDFS the Spark driver contacts NameNode about the DataNodes (ideally local) containing the various blocks of a file or directory as well as their locations (represented as InputSplits), and then schedules the work to the SparkWorkers.

Note : Spark's compute nodes / workers should be running on storage nodes.

This is how data locality achieved in Spark.

The advantage is performance gain as less data is transferred over the network.

## Memory Pressure and Garbage Collection

During the course of running Spark jobs, the executor or driver machines may struggle to complete their tasks because of a lack of sufficient memory or "memory pressure." This may occur when an application takes up too much memory during execution or when garbage collection runs too frequently or is slow to run as large numbers of objects are created in the JVM and subsequently garbage collected as they are no longer used. One strategy for easing this issue is to ensure that you're using the Structured APIs as much as possible. These will not only increase the efficiency with which your Spark jobs will execute, but it will also greatly reduce memory pressure because JVM objects are never realized and Spark SQL simply performs the computation on its internal format.

## Direct Performance Enhancements

## Parallelism

The first thing you should do whenever trying to speed up a specific stage is to increase the degree of parallelism. In general, we recommend having at least two or three tasks per CPU core in your cluster if the stage processes a large amount of data. You can set this via the **spark.default.parallelism** property as well as tuning the **spark.sql.shuffle.partitions** according to the number of cores in your cluster.

## Improved Filtering

Another frequent source of performance enhancements is moving filters to the earliest part of your Spark job that you can. Sometimes, these filters can be pushed into the data sources themselves and this means that you can avoid reading and working with data that is irrelevant to your end result. Enabling partitioning and bucketing also helps achieve this. Always look to be filtering as much data as you can early on, and you'll find that your Spark jobs will almost always run faster.

## Repartitioning and Coalescing

Repartition calls can incur a shuffle. However, doing some can optimize the overall execution of a job by balancing data across the cluster, so they can be worth it. In general, you should try to shuffle the least amount of data possible. For this reason, if you're reducing the number of overall partitions in a DataFrame or RDD, first try **coalesce** method, which will not perform a shuffle but rather merge partitions on the same node into one partition. The slower repartition method will also shuffle data across the network to achieve even load balancing. Repartitions can be particularly helpful when performing joins or prior to a cache call. Remember that repartitioning is not free, but it can improve overall application performance and parallelism of your jobs.

## User-Defined Functions (UDFs)

In general, avoiding UDFs is a good optimization opportunity. UDFs are expensive because they force representing data as objects in the JVM and sometimes do this multiple times per record in a query. You should try to use the Structured APIs as much as possible to perform your manipulations simply because they are going to perform the transformations in a much more efficient manner than you can do in a high-level language.

## Temporary Data Storage (Caching)

In applications that reuse the same datasets over and over, one of the most useful optimizations is caching. Caching will place a DataFrame, table, or RDD into temporary storage (either memory or disk) across the executors in your cluster, and make subsequent reads faster. Although caching might sound like something we should do all the time, it's not always a good thing to do. That's because caching data incurs a serialization, deserialization, and storage cost. For example, if you are only going to process a dataset *once* (in a later transformation), caching it will only slow you down.

The use case for caching is simple: as you work with data in Spark, either within an interactive session or a standalone application, you will often want to reuse a certain dataset (e.g., a DataFrame or RDD). For example, in an interactive data science session, you might load and clean your data and then reuse it to try multiple statistical models. Or in a standalone application, you might run an iterative algorithm that reuses the same dataset. You can tell Spark to cache a dataset using the cache method on DataFrames or RDDs.

Caching is a lazy operation, meaning that things will be cached only as they are accessed. The RDD API and the Structured API differ in how they actually perform caching, so let's review the gory details before going over the storage levels. When we cache an RDD, we cache the actual, physical data (i.e., the bits). The bits. When this data is accessed again, Spark returns the proper data. This is done through the RDD reference. However, in the Structured API, caching is done based on the *physical plan*. This means that we effectively store the **physical plan** as our key (as opposed to the object reference) and perform a lookup prior to the execution of a Structured job. This can cause confusion because sometimes you might be expecting to access raw data but because someone else already cached the data, you're actually accessing their cached version. Keep that in mind when using this feature.

The cache command in Spark always places data in memory by default, caching only part of the dataset if the cluster's total memory is full. For more control, there is also a persist method that takes a StorageLevel object to specify where to cache the data: in memory, on disk, or both.

## Joins

Additionally, **equi-joins** are the easiest for Spark to optimize at this point and therefore should be preferred wherever possible. Beyond that, simple things like trying to use the filtering ability of inner joins by changing join ordering can yield large speedups. Additionally, using **broadcast** join hints can help Spark make intelligent planning decisions when it comes to creating query plans. Avoiding Cartesian joins or even full outer joins is often low-hanging fruit for stability and optimizations because these can often be optimized into different filtering style joins when you look at the entire data flow instead of just that one particular job area. Additionally, bucketing your data appropriately can also help Spark avoid large shuffles when joins are performed.

## Broadcast Variables

The basic premise is that if some large piece of data will be used across multiple UDF calls in your program, you can broadcast it to save just a single read-only copy on each node and avoid re-sending this data with each job. For example, broadcast variables may be useful to save a lookup table or a machine learning model. You can also broadcast arbitrary objects by creating broadcast variables using your SparkContext, and then simply refer to those variables in your tasks