It is a group of much more than a programming paradigm but an
variety of packages, libraries, and systems built on top of the C

Spark Core consists of two APIs. The Unstructured and Structu
Unstructured API is Spark's lower level set of APIs including R
Distributed Datasets (RDDs), Accumulators, and Broadcast var
Structured API consists of DataFrames, Datasets, Spark SQL an
interface that most users should use. The difference between the
is optimized to work with structured data in a spreadsheet-like
the other is meant for manipulation of raw java objects. Outside
sit a variety of tools, libraries, and languages like MLlib for pe
machine learning, the GraphX module for performing graph pro
SparkR for working with Spark clusters from the R langauge.

We will cover all of these tools in due time however this chapte

processing. Single machines simply cannot have enough power
to perform computations on huge amounts of information (or the
have time to wait for the computation to finish). A *cluster*, or gr
machines, pools the resources of many machines together. Now
machines alone is not powerful, you need a framework to coord
across them. Spark is a tool for just that, managing and coordina

Spark Applications

Spark Applications consist of a *driver* process and a set of *exec* processes. The driver process, Figure 1-2, sits on the *driver no* responsible for three things: maintaining information about the S application, responding to a user's program, and analyzing, dist scheduling work across the executors. As suggested by figure 1- process is absolutely essential - it's the heart of a Spark Applic maintains all relevant information during the lifetime of the app

An executor is responsible for two things: executing code assig driver and reporting the state of the computation back to the dri

The last piece relevant piece for us is the cluster manager. The controls physical machines and allocates resources to Spark ap can be one of several core cluster managers: Spark's standalon manager, YARN, or Mesos. This means that there can be multip appliications running on a cluster at the same time.

NOTE:

Spark, in addition to its cluster mode, also has a *local mode*. Re
the driver and executors are processes? This means that Spark c
where these processes live. In local mode, these processes run
individual computer instead of a cluster. See figure 1-3 for a hig
diagram of this architecture. This is the easiest way to get starte
and what the demonstrations in this book should run on.

Scala

Spark is primarily written in Scala, making it Spark's "default"
book will include examples of Scala where ever there are code

Python

Python supports nearly everything that Scala supports. This boo
Python API examples wherever possible.

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

SQL

Spark supports user code written in ANSI 2003 Compliant SQL
easy for analysts and non-programmers to leverage the big data
Spark. This book will include numerous SQL examples.

R

Spark supports the execution of R code through a project called

will cover this in the Ecosystem section of the book along with
interesting projects that aim to do the same thing like Sparklyr.

architecture because at this point it's not necessary to get us clo
our own Spark code. The key points are that:

- Spark has some cluster manager that maintains an understa
resources available.
- The driver process is responsible for executing our driver
commands accross the executors in order to complete our
- There are two modes that you can use, cluster mode (on m
machines) and local mode (on a single machine).

Now in the previous chapter we talked about what you need to
with Spark by setting your Java, Scala, and Python versions. No
start Spark's local mode, this means running `./bin/spark-she`
start that you will see a console, into which you can enter comm
would like to work in Python you would run `./bin/pyspark`.

From the beginning of this chapter we know that we leverage a
to maintain our Spark Application. This driver process manifes
user as something called the SparkSession. The *SparkSession* i
entrance point to executing code in Spark, in any language, and
facing part of a Spark Application. In Scala and Python the vari
available as `spark` when you start up the Spark console. Let's g
look at the `SparkSession` in both Scala and/or Python.

```
%scala

spark
```

```
%python

spark
```

In Scala, you should see something like:

```
res0: org.apache.spark.sql.SparkSession = org.apac
```

In Python you'll see something like:

```
<pyspark.sql.session.SparkSession at 0x7efda4c1ccd
```

Now you need to understand how to submit commands to the Sp
Let's do that by performing one of the simplest tasks that we ca

range of numbers. This range of numbers is just like a named co
spreadsheet.

```
%scala

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

```
%python

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

You just ran your first Spark code! We created a DataFrame wit
containing 1000 rows with values from 0 to 999. This range of
represents a *distributed collection*. Running on a cluster, each
range of numbers would exist on a different executor. You'll no
value of *myRange* is a DataFrame, let's introduce DataFrames!

DataFrames

A *DataFrame* is a table of data with rows and columns. We cal
columns and their types a *schema*. A simple analogy would be a
with named columns. The fundamental difference is that while a
sits on one computer in one specific location, a Spark DataFran
potentially thousands of computers. The reason for putting the d
than one computer should be intuitive: either the data is too larg
machine or it would simply take too long to perform that compu
machine.

The DataFrame concept is not unique to Spark. The R Language
concept as do certain libraries in the Python programming langu
Python/R DataFrames (with some exceptions) exist on one mach
multiple machines. This limits what you can do with a given Da
python and R to the resources that exist on that specific machine
since Spark has language interfaces for both Python and R, it's e
convert to Pandas (Python) DataFrames to Spark DataFrames a
DataFrames to Spark DataFrames (in R).

Note

Spark has several core abstractions: Datasets, DataFrames, SQ
Resilient Distributed Datasets (RDDs). These abstractions all r
distributed collections of data however they have different inter
working with that data. The easiest and most efficient are DataF
are available in all languages. We cover Datasets in Section II,
RDDs in depth in Section III Chapter 2 and 3. The following co
all of the core abstractions.

Partitions

In order to leverage the the resources of the machines in cluster, up the data into chunks, called partitions. A *partition* is a colle that sit on one physical machine in our cluster. A DataFrame co more partitions.

When we perform some computation, Spark will operate on eac parallel unless an operation calls for a *shuffle*, where multiple to share data. Think about it this way, if you need to run some e typically have to do those one by one, or serially. What if you c give one errand to a worker who would then complete that task back to you? In that scenario, the key is to break up errands effi you can get as much work done in as little time as possible. In t an "errand" is equivalent to computation + data and a "worker" to an executor.

Now with DataFrames, we do not manipulate partitions individ gives us the DataFrame interface for doing that. Now when we code, you'll notice there was no list of numbers, only a type sig because Spark organizes computation into two categories, trans actions. When we create a DataFrame, we perform a transforma

Transformations

In Spark, the core data structures are *immutable* meaning they c
changed once created. This might seem like a strange concept a
cannot change it, how are you supposed to use it? In order to "c
DataFrame you will have to instruct Spark how you would like
DataFrame you have into the one that you want. These instructio
transformations. Transformations are how you, as user, specify
would like to transform the DataFrame you currently have to the
that you want to have. Let's show an example. To computer whe
number is divisible by two, we use the modulo operation to see
left over from dividing one number by another.

results that do not result in zero. We can specify this filter using
clause.

```
%scala

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

```
%python

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

Note

DataFrame into a table.

These operations create a new DataFrame but do not execute ar
The reason for this is that DataFrame transformations do not trig
execute your code, they are lazily evaluated.

Lazy Evaluation

Lazy evaulation means that Spark will wait until the very last m
execute your transformations. In Spark, instead of modifying the
we build up a *plan* of the transformations that we would like to
by waiting for the last minute to execute your code, can try and
run as efficiently as possible across the cluster.

Actions

To trigger the computation, we run an *action*. An *action* instruct
compute a result from a series of *transformations*. The simples
`count` which gives us the total number of records in the DataFr

```
%scala

divisBy2.count()
```

```
%python

divisBy2.count()
```

We now see a result! There are 500 number divisible by two fr
(big surprise!). Now `count` is not the only action. There are thr
actions: actions to view data in the console, actions to collect d
objects in the respective language, and actions to write to outpu

Spark UI

During Spark's execution of the previous code block, users can
progress of their job through the Spark UI. The Spark UI is avai
4040 of the driver node. If you are running in local mode this w
`http://localhost:4040`. The Spark UI maintains information
our Spark jobs, environment, and cluster state. It's very useful,
tuning and debugging. In this case, we can see one Spark job wi
and nine tasks were executed.

In this chapter we will avoid the details of Spark jobs and the S
point you should understand that a Spark job represents a set of
triggered by an individual action. We talk in depth about the Spa
breakdown of a Spark job in Section IV.

be using some flight data from the United States Bureau of Tran
statistics.

We touched briefly on the SparkSession as the interface the entr
performing work on the Spark cluster. the SparkSession can do
simply parallelize an array it can create DataFrames directly fr
of files. In this case, we will create our DataFrames from a Jav
Notation (JSON) file that contains some summary flight informa
collected by the United States Bureau of Transport Statistics. In

```
%fs head /mnt/defg/chapter-1-data/json/2015-summar
```

What we'll do is start with one specific year and then work up
of data. Let's go ahead and create a DataFrame from 2015. To d
use the `DataFrameReader` (via `spark.read`) interface, specify
the path.

```
%scala

val flightData2015 = spark
  .read
  .json("/mnt/defg/chapter-1-data/json/2015-summar

%python

flightData2015 = spark\
```

You'll see that our two DataFrames (in Scala and Python) each
columns with an unspecified number of rows. Let's take a peek
a new action, `take`, which allows us to view the first couple of
DataFrame. Figure 1-7 illustrates the conceptual actions that we
process. We lazily create the DataFrame then call an action to ta
values.

```
%scala
flightData2015.take(2)
%python
flightData2015.take(2)
```

Remember how we talked about Spark building up a plan, this i
conceptual tool, this is actually what happens under the hood. W
actual plan built by Spark by running the `explain` method.

```
flightData2015.explain()
%python
flightData2015.explain()
```

just describes reading data from a certain location however as
you will start to notice patterns in the explain plans. Without go
much detail at this point, the explain plan represents the logical

transformations Spark will run on the cluster. We can use this to
our code is as optimized as possible. We will not cover that in
will touch on it in the optimization chapter.

Note

Remember, we cannot modify this DataFrame by specifying the
transformation, we can only create a new DataFrame by transfo
previous DataFrame. We can see that even though we're seemi
computation to be completed Spark doesn't yet execute this co
just building up a plan. The illustration in figure 1-8 represents
we see in the explain plan for that DataFrame.

```
%scala

val sortedFlightData2015 = flightData2015.sort("co
sortedFlightData2015.explain()
```

```
%python

sortedFlightData2015 = flightData2015.sort("count"
sortedFlightData2015.explain()
```

Now, just like we did before, we can specify an action in order
plan.

```
%scala

sortedFlightData2015.take(2)
```

```
%python

sortedFlightData2015.take(2)
```

The conceptual plan that we executed previously is illustrated i

Now this planning process is essentially defining *lineage* for th
that at any given point in time Spark knows how to recompute a
given DataFrame all the way back to a robust data source be it
database. Now that we performed this action, remember that we
to the Spark UI (port 4040) and see the information about this jo
tasks.

Now hopefully you have grasped the basics but let's just reinfo

its API is and ecosystem.

Let's go ahead and define our DataFrame just like we did befor
time we're going to specify an `option` for our DataFrameRead

allow you to control how you read in a given file format and tel
advantage of some of the structures or information available in
case we're going to use two popular options `inferSchema` and

```
%scala

val flightData2015 = spark.read
  .option("inferSchema", "true")
  .option("header", "true")
  .csv("/mnt/defg/chapter-1-data/csv/2015-summary.
flightData2015
```

```
%python

flightData2015 = spark.read\
  .option("inferSchema", "true")\
  .option("header", "true")\
  .csv("/mnt/defg/chapter-1-data/csv/2015-summary.
flightData2015
```

After running the code you should notice that we've basically a
same DataFrame that we had when we read in our data from jso
correct looking column names and types. However, we had to b
when it came to reading in the CSV file as opposed to json beca
provides a bit more structure than CSVs because JSON has a no

Looking at them, the `header` option should feel like it makes se
row in our csv file is the header (column names) and because C
guaranteed to have this information we must specify it manually
`inferSchema` option might feel a bit more unfamiliar. JSON ob
a bit more structure than csvs because JSON has a notion of typ
past this by infering the schema of the csv file we are reading in
do this magically, it must scan (read in) some of the data in orde
but this saves us from having to specify the types for each colum
the risk of Spark potentially making an errorneous guess as to w
a column should be.

A discerning reader might notice that the schema returned by ou

```
  .option("header", "true")
  .load("/mnt/defg/chapter-1-data/csv/2015-summary
  .schema

val jsonSchema = spark
  .read.format("json")
  .load("/mnt/defg/chapter-1-data/json/2015-summar
  .schema

println(csvSchema)
println(jsonSchema)
```

```
%python
```

```
print(csvSchema)
print(jsonSchema)
```

The csv schema:

```
StructType(StructField(DEST_COUNTRY_NAME,StringTyp
StructField(ORIGIN_COUNTRY_NAME,StringType,true),
StructField(count,IntegerType,true))
```

The JSON schema:

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

For our purposes the difference between a `LongType` and an `In` of little consequence however this may be of greater significanc scenarios. Naturally we can always explicitly set a schema (rat inferring it) when we read in data as well. These are just a few we have when we read in data, to learn more about these optio chapter on reading and writing data.

```
%scala

val flightData2015 = spark.read
  .schema(jsonSchema)
  .option("header", "true")
  .csv("/mnt/defg/chapter-1-data/csv/2015-summary.
```

```
%python

flightData2015 = spark.read\
  .schema(jsonSchema)\
  .option("header", "true")\
  .csv("/mnt/defg/chapter-1-data/csv/2015-summary.
```

DataFrames and SQL

Spark provides another way to query and operate on our DataFr
that's with SQL! Spark SQL allows you as a user to register any
a table or view (a temporary table) and query it using pure SQL
performance difference between writing SQL queries or writing
code, they both "compile" to the same underlying plan that we s
DataFrame code.

Any DataFrame can be made into a table or view with one simp

```
%scala

flightData2015.createOrReplaceTempView("flight_dat
```

```
%python

flightData2015.createOrReplaceTempView("flight_dat
```

Now we can query our data in SQL. To execute a SQL query, w
`spark.sql` function (remember `spark` is our SparkSession vari
conveniently, returns a new DataFrame. While this may seem a
logic - that a SQL query against a DataFrame returns another Da
actually quite powerful. As a user, you can specify transformati
manner most convenient to you at any given point in time and no

two explain plans.

```
Vi%scala

val sqlWay = spark.sql("""
SELECT DEST_COUNTRY_NAME, count(1)
FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
""")

val dataFrameWay = flightData2015
  .groupBy('DEST_COUNTRY_NAME)
  .count()
```

```
sqlWay.explain
dataFrameWay.explain

%python

sqlWay = spark.sql("""
```

transformation, as we are effectively filtering down to one row. that looks like.

```
// scala or python
spark.sql("SELECT max(count) from flight_data_2015
```

```
%scala

import org.apache.spark.sql.functions.max

flightData2015.select(max("count")).take(1)
```

```
%python

from pyspark.sql.functions import max

flightData2015.select(max("count")).take(1)
```

Let's move onto something a bit more complicated. What are th

query so we'll take it step by step. We will start with a fairly st
SQL aggregation.

```
%scala

val maxSql = spark.sql("""
SELECT DEST_COUNTRY_NAME, sum(count) as destinatio
  FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
ORDER BY sum(count) DESC
LIMIT 5""")

maxSql.collect()
```

```
%python

maxSql = spark.sql("""
SELECT DEST_COUNTRY_NAME, sum(count) as destinatio
  FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
ORDER BY sum(count) DESC
LIMIT 5""")

maxSql.collect()
```

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

```
%scala
import org.apache.spark.sql.functions.desc

flightData2015
  .groupBy("DEST_COUNTRY_NAME")
  .sum("count")
  .withColumnRenamed("sum(count)", "destination_to
  .sort(desc("destination_total"))
  .limit(5)
  .collect()
```

```
%python
from pyspark.sql.functions import desc
```

```
.limit(5)\
.collect()
```

Now there are 7 steps that take us all the way back to the source
Illustrated below are the set of steps that we perform in "code"
execution plan (the one visible in explain) will differ from wha
below because of optimizations in physical execution, however
is as good of a starting point as any. With Spark, we are always
directed acyclic graph of transformations resulting in immutable
we can subsequently call an action on to see a result.

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

The second step is our grouping, technically when we call "gro
up with a RelationalGroupedDataset which is a fancy name for
that has a grouping specified but needs a user to specify an aggr
it can be queried further. We can see this by trying to perform a
(which will not work). We still haven't performed any computa

of the data.

Therefore the third step is to specify the aggregation. Let's use t
aggregation method. This takes as input a column expression or
column name. The result of the sum method call is a new dataFr
that it has a new schema but that it does know the type of each c
important to reinforce (again!) that no computation has been per
simply another transformation that we've expressed and Spark i
to trace the type information we have supplied.

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

The fifth step sorts the data such that if we were to take results

destination_total column.

You likely noticed that we had to import a function to do this, th
function. You might also notice that desc does not return a strin
In general, many DataFrame methods will accept Strings (as co
Column types or expressions. Columns and expressions are actu
same thing

The final step is just a limit. This just specifies that we only wa
This is just like a filter except that it filters by position (lazily)
value. It's safe to say that it basically just specifies a DataFram
size.

The last step is our action! Now we actually begin the process
results of our DataFrame above and Spark will give us back a l
the language that we're executing. Now to reinforce all of this,
explain plan for the above query.

```
flightData2015
  .groupBy("DEST_COUNTRY_NAME")
  .sum("count")
```

```
.withColumnRenamed("sum(count)", "destination_to
.sort(desc("destination_total"))
.limit(5)
.explain()
```

```
== Physical Plan ==
TakeOrderedAndProject(limit=5, orderBy=[destinatio
+- *HashAggregate(keys=[DEST_COUNTRY_NAME#7323], f
   +- Exchange hashpartitioning(DEST_COUNTRY_NAME#
      +- *HashAggregate(keys=[DEST_COUNTRY_NAME#73
         +- InMemoryTableScan [DEST_COUNTRY_NAME#7
               +- InMemoryRelation [DEST_COUNTRY_N
                     +- *Scan csv [DEST_COUNTRY_NA
```

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

You are now equipped with the Spark knowledge to writing you
code. In the next chapter we will explore some of Spark's more

Spark's Structured APIs

For our purposes there is a spectrum of types of data. The two e
spectrum are *structured* and *unstructured*. Structured and semi-
refer to to data that have structure that a computer can understan
easily. Unstructured data, like a poem or prose, is much harder
to understand. Spark's Structured APIs allow for transformation
on structured and semi-structured data.

The Structured APIs specifically refer to operations on DataFra
and in Spark SQL and were created as a high level interface for
manipulate big data. This section will cover all the principles o
APIs. Although distinct in the book, the vast majority of these us
operations apply to both *batch* as well as *streaming* computatio

BOX

Before proceeding, let's review the fundamental concepts
that we covered in the previous section. Spark is a distribu
programming model where the user specifies *transformati*
build up a directed-acyclic-graph of instructions, and *acti*
begin the process of executing that graph of instructions, as
by breaking it down into stages and tasks to execute across
The way we store data on which to perform transformation
are DataFrames and Datasets. To create a new DataFrame
call a transformation. To start computation or convert to na
types, you call an action.

In Section I, we talked all about DataFrames. Spark has two not
"structured" data structures: DataFrames and Datasets. We will
(nuanced) differences shortly but let's define what they both rep

To the user, DataFrames and Datasets are (distributed) tables w
columns. Each column must have the same number of rows as al
columns (although you can use null to specify the lack of a valu
have type information that tells the user what exists in each colu

To Spark, DataFrames and Datasets represent by immutable, laz
plans that specify how to perform a series of transformations to
correct output. When we perform an action on a DataFrame we
to perform the actual transformations that represent that DataFra
represent plans of how to manipulate rows and columns to com
desired result. Let's go over rows and column to more precisely
concepts.

Schemas

One core concept that differentiates the Structured APIs from th
APIs is the concept of a *schema*. A schema defines the column n
of a DataFrame. Users can define schemas manually or users ca
schema from a data source (often called *schema on read*). Now
what defines DataFrames and Datasets and how they get their st
Schema, let's see an overview of all of the types.

Spark is effectively a programming language of its own. When y
operations with Spark, it maintains its own type information thr
process. This allows it to perform a wide variety of optimizatio
execution process. These types correspond to the types that Spa
in each of Scala, Java, Python, SQL, and R. Even if we use Spa
APIs from Python, the majority of our manipulations will opera
Spark types, not Python types. For example, the below code do
addition in Scala or Python, it actually performs addition *purel*

```
%scala

val df = spark.range(500).toDF("number")
df.select(df.col("number") + 10)
// org.apache.spark.sql.DataFrame = [(number + 10)
```

```
%python

df = spark.range(500).toDF("number")
df.select(df["number"] + 10)
# DataFrame[(number + 10): bigint]
```

This is because, as mentioned, Spark maintains its own type inf
stored as a *schema*, and through some magic in each languages
convert an expression in one language to Spark's representation

NOTE

> There are two distinct APIs within the In the Structured A
> API that goes across languages, more commonly referred t
> DataFrame or "untyped API". THe second API is the "type
> "Dataset API", that is only available to JVM based langua
> Java). This is a bit of a misnomer because the "untyped A
> types but it only operates on *Spark* types at *run time*. The
> allows you to define your own types to represent each rec

dataset with "case classes or Java Beans" and types are ch
compile time. Each record (or row) in the "untyped API" (
`Row` object that are available across languages and still hav
only Spark types, not native types. The "typed API" is cov
Datasets Chapter at the end of Section II. The majority of S
cover the "untyped API" however all of this still applies to

Notice how the following code produces a `Dataset` of type `Lon`
an internal Spark type (bigint).

```
%scala

spark.range(500)
```

Notice how the following code produces a `DataFrame` with an
type (bigint).

```
%python
```

Columns

For now, all you need to understand about columns is that they c
simple type like an integer or string, a *complex types* like an ar
null value. Spark tracks all of this type information to you and h
ways that you can transform columns. Columns are discussed ex
next chapter but for the most part you can think about Spark `Col`
columns in a table.

Rows

There are two ways of getting data into Spark, through Rows an
Row objects are the most general way of getting data into, and c
and are available in all languages. Each record in a DataFrame
type as we can see when we collect the following DataFrames.

```
%scala

spark.range(2).toDF().collect()
```

```
%python

spark.range(2).collect()
```

corresponding language specific types. Caveats and details are
reader as well to make it easier to reference.

To work with the correct Scala types:

```
import org.apache.spark.sql.types._
val b = ByteType()
```

To work with the correct Java types you should use the factory
following package:

```
import org.apache.spark.sql.types.DataTypes;
ByteType x = DataTypes.ByteType();
```

Python types at time have certain requirements (like the listed r
`ByteType` below.To work with the correct Python types:

```
from pyspark.sql.types import *
b = byteType()
```

Spark Type	Scala Value Type	Scala

Encoders

Using Spark from Scala and Java allows you to define your ow
use in place of Rows that consist of the above data types. To do
Encoder. Encoders are only available in Scala, with case class
with JavaBeans. For some types, like Long, Spark already inclu
Encoder. For instance we can collect a `Dataset` of type `Long` a
Scala types back. We will cover encoders in the Datasets chapt

```
spark.range(2).collect()
```

structured API query from user code to executed code. As an ov
steps are:

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*
4. Spark then executes this *Physical Plan* on the cluster

To execute code, we have to write code. This code is then subm
either through the console or via a submitted job. This code the
through the Catalyst Optimizer which decides how the code sho
and lays out a plan for doing so, before finally the code is run a
returned to the user.

This logical plan only represents a set of abstract transformatio
refer to executors or drivers, it's purely to convert the user's se
into the most optimized version. It does this by converting user
unresolved logical plan. This unresolved because while your c
valid, the tables or columns that it refers to may or may not exis
the *catalog*, a repository of all table and DataFrame informatio
resolve columns and tables in the *analyzer*. The analyzer may re
unresolved logical plan if it the required table or column name
in the catalog. If it can resolve it, this result is passed through th
collection of rules, which attempts to optimize the logical plan
down predicates or selections.

Physical Planning

After successfully creating an optimized logical plan, Spark the
physical planning process. The *physical plan*, often called a Sp
specifies how the logical plan will execute on the cluster by ge
different physical execution strategies and comparing them thro
model. An example of the cost comparison might be choosing h
given join by looking at the physical attributes of a given table
table is or how big its partitions are.)

Physical

Physical planning results in a series of RDDs and transformatio
is why you may have heard Spark referred to as a compiler, it ta
DataFrames, Datasets, and SQL and compiles them into RDD tr
for you.

Execution

Upon selecting a physical plan, Spark runs all of this code over
lower-level programming interface of Spark covered in Part III
performs further optimizations by, at runtime, generating native
that can remove whole tasks or stages during execution. Finally
returned to the user.

in depth later in this section.

Definitionally, a DataFrame consists of a series of *records* (like table), that are of type `Row`, and a number of *columns* (like colu spreadsheet) that represent an computation expression that can each individual record in the dataset. The *schema* defines the na the type of data in each column. The *partitioning* of the DataFra layout of the DataFrame or Dataset's physical distribution acros The *partitioning scheme* defines how that is broken up, this can based on values in a certain column or non-deterministically.

Let's define a DataFrame to work with.

```
%scala

val df = spark.read.format("json")
  .load("/mnt/defg/flight-data/json/2015-summary.j
```

```
%python

df = spark.read.format("json")\
  .load("/mnt/defg/flight-data/json/2015-summary.j
```

We discussed that a DataFame will have columns, and we use a view all of those. We can run the following command in Scala

```
df.printSchema()
```

The schema ties the logical pieces together and is the starting po understand DataFrames.

Schemas

A schema defines the column names and types of a DataFrame. let a data source define the schema (called *schema on read*) or it explicitly ourselves.

NOTE

Deciding whether or not you need to define a schema prior your data depends your use case. Often times for ad hoc an on read works just fine (although at times it can be a bit sl text file formats like csv or json). However, this can also l precision issues like a long type incorrectly set as an integ reading in a file. When using Spark for production ETL, it idea to define your schemas manually, especially when wo untyped data sources like csv and json because schema inf depending on the type of data that you read in.

Let's start with a simple file we saw in the previous chapter and structured nature of line-delimited JSON define the structure. T data from the United States Bureau of Transportation statistics.

```
%scala

spark.read.format("json")
  .load("/mnt/defg/flight-data/json/2015-summary.j
  .schema
```

Scala will return:

```
org.apache.spark.sql.types.StructType = ...
StructType(StructField(DEST_COUNTRY_NAME,StringTyp
StructField(ORIGIN_COUNTRY_NAME,StringType,true),
StructField(count,LongType,true))
```

```
%python
```

```
StructField(ORIGIN_COUNTRY_NAME,StringType,true),
StructField(count,LongType,true)))
```

A schema is a `StructType` made up of a number of fields, `Stru`
have a name, type, and a boolean flag which specifies whether
column can contain missing or `null` values. Schemas can also c
`StructType` (Spark's complex types). We will see this in the ne
when we discuss working with complex types.

types in the data (at runtime), do not match the schema. Spark w
error.

```
%scala

import org.apache.spark.sql.types.{StructField, St

val myManualSchema = new StructType(Array(
  new StructField("DEST_COUNTRY_NAME", StringType,
  new StructField("ORIGIN_COUNTRY_NAME", StringTyp
  new StructField("count", LongType, false) // jus
))

val df = spark.read.format("json")
  .schema(myManualSchema)
  .load("/mnt/defg/flight-data/json/2015-summary.j
```

Here's how to do the same in Python.

```
    .schema(myManualSchema)\
    .load("/mnt/defg/flight-data/json/2015-summary.j
```

As discussed in the previous chapter, we cannot simply set type
language types because Spark maintains its own type informatio
discuss what schemas define, columns.

Columns and Expressions

To users, columns in Spark are similar to columns in a spreadsh
dataframe, pandas DataFrame. We can select, manipulate, and r
from DataFrames and these operations are represented as *expre*

To Spark, columns are logical constructions that simply represe
computed on a per-record basis by means of an *expression*. Thi
order to have a real value for a column, we need to have a row,
have a row we need to have a DataFrame. This means that we c
manipulate an actual column outside of a DataFrame, we can or
logical column's expressions then perform that expression withi
a DataFrame.

Columns

There are a lot of different ways to construct and or refer to col
two simplest ways are with the `col` or `column` functions. To use
functions, we pass in a column name.

```
%scala

import org.apache.spark.sql.functions.{col, column

col("someColumnName")
column("someColumnName")
```

```
%python

from pyspark.sql.functions import col, column

col("someColumnName")
column("someColumnName")
```

We will stick to using `col` throughout this book. As mentioned,
may or may not exist in our of our DataFrames. This is because,
the previous chapter, columns are not *resolved* until we compar
names with those we are maintaining in the *catalog*. Column an
resolution happens in the *analyzer* phase as discussed in the firs
this section.

NOTE

thing as what we have already, namely creating a column, a
performance improvement.

```
%scala

$"myColumn"
'myColumn
```

The `$` allows us to designate a string as a special string that sho
expression. The tick mark `'` is a special thing called a symbol, t
specific construct of referring to some identifier. They both perf
thing and are shorthand ways of referring to columns by name. Y
all the above references when you read different people's spark
leave it to the reader for you to use whatever is most comfortab

Explicit Column References

If you need to refer to a specific DataFrame's column, you can u
method on the specific DataFrame. This can be useful when you
a join and need to refer to a specific column in one DataFrame t
name with another column in the joined DataFrame. We will see
joins chapter. As an added benefit, Spark does not need to resol
itself (during the *analyzer* phase) because we did that for Spark

```
df.col("count")
```

Now we mentioned that columns are expressions, so what is an
An *expression* is a set of transformations on one or more values
a DataFrame. Think of it like a function that takes as input one o
names, resolves them and then potentially applies more express
single value for each record in the dataset. Importantly, this "sin
actually be a complex type like a Map type or Array type.

In the simplest case, an expression, created via the `expr` functio
DataFrame column reference.

```
import org.apache.spark.sql.functions.{expr, col}
```

In this simple instance, `expr("someCol")` is equivalent to `col`

Columns as Expressions

Columns provide a subset of expression functionality. If you use
wish to perform transformations on that column, you must perfo
column reference. When using an expression, the `expr` function
parse transformations and column references from a string and
subsequently be passed into further transformations. Let's look
examples.

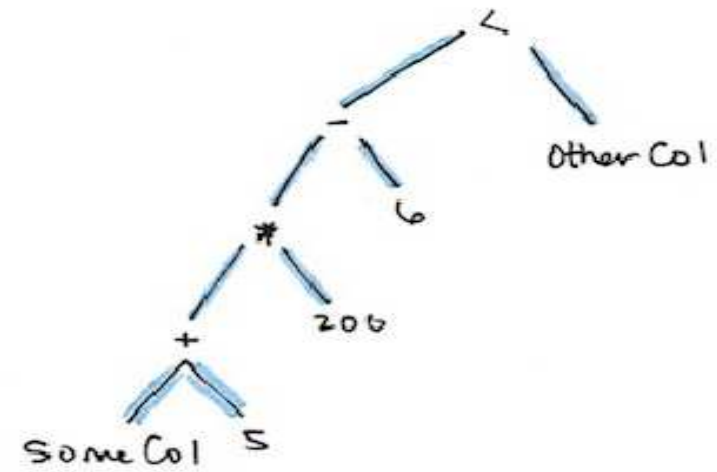
`expr("someCol - 5")` is the same transformation as performi
`col("someCol") - 5` or even `expr("someCol") - 5`. That's
compiles these to a logical tree specifying the order of operatio
be a bit confusing at first, but remember a couple of key points.

1. Columns are just expressions.
2. Columns and transformations of those column compile to t
   plan as parsed expressions.

Let's ground this with an example.

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

Figure 1 shows an illustration of that logical tree.



This might look familiar because it's a directed acyclic graph. represented equivalently with the following code.

```
%scala

import org.apache.spark.sql.functions.expr

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

```
%python

from pyspark.sql.functions import expr

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

This is an extremely important point to reinforce. Notice how th expression is actually valid SQL code as well, just like you mig `SELECT` statement? That's because this SQL expression and the DataFrame code compile to the same underlying logical tree pri execution. This means you can write your expressions as DataF as SQL expressions and get the exact same benefits. You likely in the first chapters of the book and we covered this more exten

```
myRow[0]
.  lit(1).alias("One")
).show(2)
```

In SQL, literals are just the specific value.

```
%sql

SELECT
  *,
  1 as One
FROM
  dfTable
LIMIT 2
```

This will come up when you might need to check if a date is gre
constant or some value

```
  expr("*"),
  lit(1).as("something")
).show(2)
```

```
%python

from pyspark.sql.functions import expr, col, colum

df.select("DEST_COUNTRY_NAME").show(2)
  .load("/mnt/defg/flight-data/json/2015-summary.j

df.createOrReplaceTempView("dfTable")
```

We can also create DataFrames on the fly by taking a set of row
converting them to a DataFrame.

```
%scala

import org.apache.spark.sql.Row
import org.apache.spark.sql.types.{StructField, St
                                   StringType, Lon

val myManualSchema = new StructType(Array(
  new StructField("some", StringType, true),
  new StructField("col", StringType, true),
  new StructField("names", LongType, false) // jus
))
```

```
))

val myRows = Seq(Row("Hello", null, 1L))
val myRDD = spark.sparkContext.parallelize(myRows)

val myDf = spark.createDataFrame(myRDD, myManualSc
myDf.show()
```