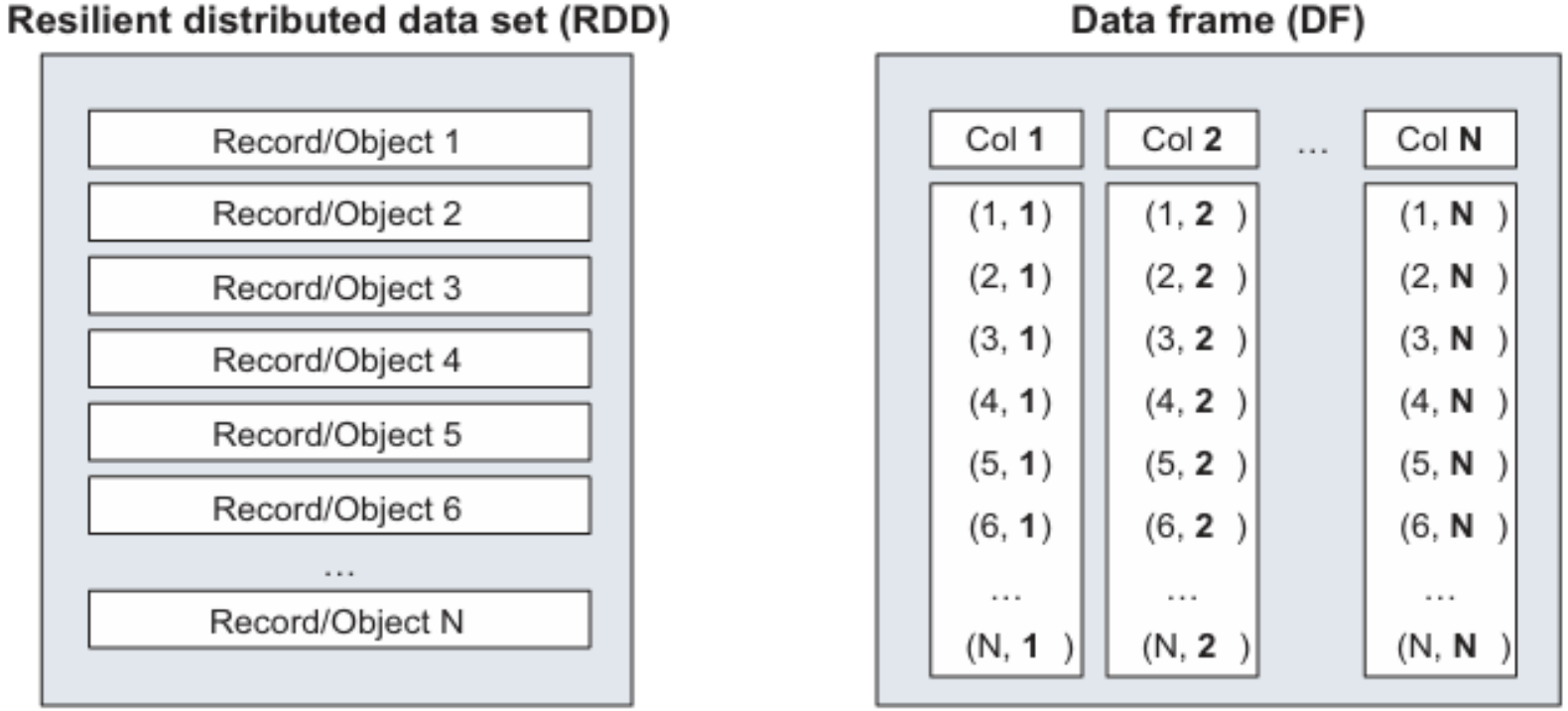
Big Data Advanced



In an RDD, we think of each record as being an independent object on which we perform functions to transform them. Think "collection," not "structure."

A data frame organizes the records in columns. We perform transformations either directly on those columns or on the data frame as a whole; we typically don't access records horizontally (record by record) as we do with the RDD.

An RDD versus a data frame. In the RDD, we think of each record as an independent entity. With the data frame, we mostly interact with columns, performing functions on them. We still can access the rows of a data frame, via RDD, if necessary.

- one stick of **128 GB** is more than the price of two sticks **of 64 GB**
- Instead **of buying thousands of dollars of RAM** to accommodate your data set, you'll rely on **multiple computers**, splitting the **job** between them.
- In a world where two modest computers are less costly than one large one, scaling out is **less expensive** than scaling up, which keeps more money in your pockets.

- As the data size grows, Spark can help control costs by s**caling the number of workers** and **executors** for a given job.
- Data **transformation** job on a modest data set (a few TB), you can limit yourself to a lower number—say**, five—machines**, scaling up to **60** when you want to do machine learning.
- Some vendors, such as **Databricks** , offer **auto-scaling**, meaning that they **increase and decrease** the number of machines during a job depending on the pressure on the **cluster**. The implementation of **auto-scaling/cost controlling is 100%** vendor-dependent.

- A **single** computer can **crash** or behave unpredictably at times. If instead of one you have one hundred, the chance that at least one of them goes down is now much higher.
- Spark therefore has a lot of hoops to manage, scale, and babysit so that you can focus on what you want, which is to work with data.
- Spark provides a powerful **API** (application programming interface, the set of **functions**, **classes**, and **variables** provided for you to interact with) that makes it look like you're working with a cohesive source of data while also working hard in the background to optimize your program to use all the power available.

- Spark itself is coded in **Scala**.
- **PySpark** provides access not only to the core Spark API but also to a set of bespoke functionality to scale out regular Python code, as well as pandas transformations.
- In Python's **data analysis** ecosystem**, pandas** is the de facto data frame library for memory bound data frames (**the entire data** frame needs to reside on a **single machine's memory**).

- **Python** code has to be translated to and from **JVM (Java Virtual Machine**, the **runtime** that powers Java and Scala code) instructions.

- **Spark** can run up to **100 times** faster than plain **Hadoop**. Because of the integration between the two frameworks, you can easily switch your Hadoop workflow to **Spark** and gain some performance boost without changing your hardware.
- **PySpark** isn't the right choice if you're dealing with rapid processing of (very) **small data sets**.
- Executing a program on **multiple machines** requires a level of coordination between the **nodes**, which comes with some **overhead**. If you're just using a **single node**, you're paying the price but aren't using the benefits. As an example, a **PySpark** shell will take a few seconds to launch; this is often more than enough time to process data that **fits within your RAM**.

- Databricks (see appendix B), allow for auto-scaling the number of machines at runtime.
- Some require more planning, especially if you run on the premises and own your hardware.
- The **workers** are called **executors** in Spark's literature: they perform the **actual work** on the **machines/nodes.**

- In our data factory, he’s the **manager** of the work floor. In Spark terms, we call this the **master**.
- The **master** here sits on one of the **workbenches/machines**, but it can also sit on a distinct machine (or even your computer!) depending on the cluster manager and deployment mode.
- The role of the **master** is crucial to the **efficient execution** of your program

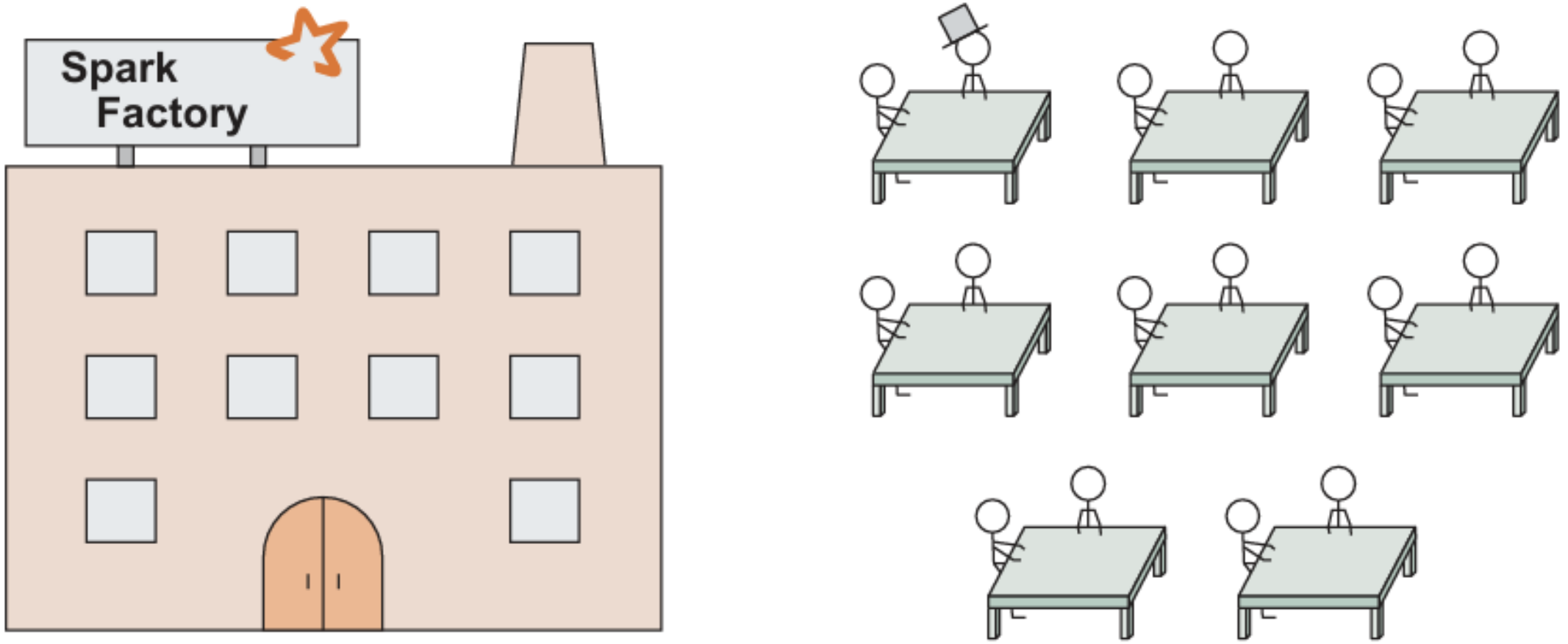


Figure 1.1 A totally relatable data factory, outside and in. Ninety percent of the time we care about the whole factory, but knowing how it's laid out helps when reflecting on our code performance.

- Upon reception of the **task**, which is called a **driver program** in the Spark world, the factory starts running.
- This **doesn’t mean that we get straight to processing**. Before that, the cluster needs to **plan** the capacity it will allocate for your program.
- The entity or program taking care of this is aptly called the **cluster manager.** In our factory, this cluster manager will look at the **workbenches with available space** and secure as many as necessary, and then start hiring workers to fill the capacity.
- In Spark, it will look at the machines with available computing resources and secure what’s necessary before launching the required **number of executors across them**

- Any directions about capacity (machines and executors) are encoded in a **Spark Context** representing the connection to our **Spark cluster**. If our instructions don’t mention any specific capacity, the cluster manager will allocate the **default capacity** prescribed by our **Spark installation**.

Listing 1.2 Content of the sample.csv` file

```
less data/list_of_numbers/sample.csv


old_column
1
4
4
5
7
7
7
10
14
1
4
8
```

case of computing the **average**, each **worker independently computes** the sum of the **values** and their counts before moving the result—not all the data!— over to a single worker (or the master directly, when the intermediate result is really small) that will process the aggregation into a single number, the average.

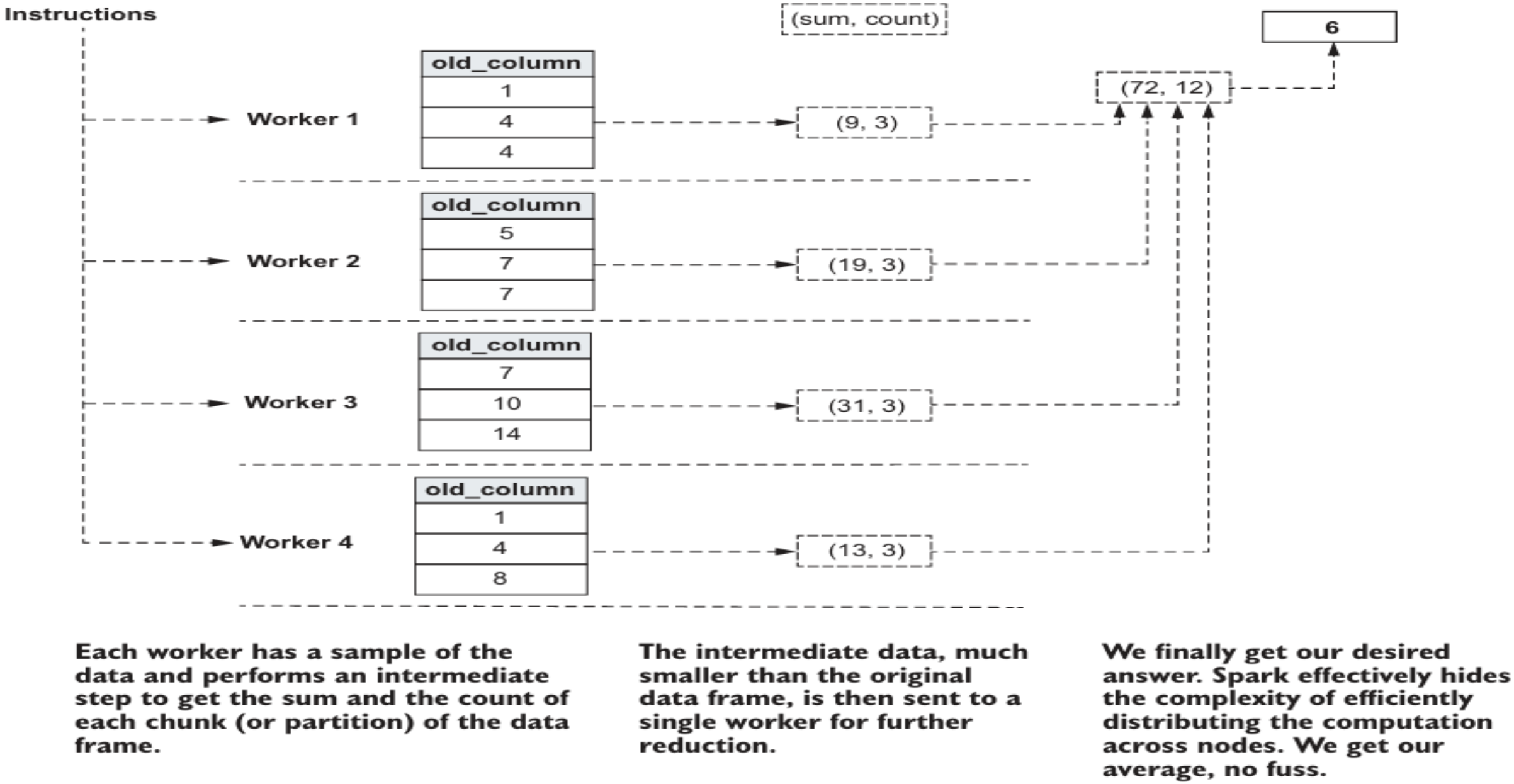


Figure 1.2 Computing the average of our small data frame, PySpark style: each worker works on a distinct piece of data. As necessary, the data gets moved/shuffled around to complete the instructions.

A **factory** made efficient through **a lazy leader**

- Just like in a large-scale factory, you don't go to each employee and give them a list of tasks. No, here, the **master/manager** is responsible for the workers. The **driver** is where the action happens.
- Think of a **driver** as a floor lead: you provide them **your list of steps** and let them deal with it.
- In Spark, the **driver/floor** lead takes your instructions (carefully written in **Python code**), translates them into **Spark steps**, and then processes them across the **worker.**
- The **driver** also manages which worker/table has which slice of the data, and makes sure **you don't lose some bits** in the process.
- The **executor/factory worker** sits atop the workers/tables and performs the actual work on the data.

Summary

- The **master** is like the **factory owner**, allocating resources as needed to complete the jobs.
- The **driver** is responsible for completing a **given** job. It requests resources from the master as needed.
- A **worker** is a set of computing/memory resources, like a **workbench** in our factory.
- **Executors** sit **atop a worker** and perform the work sent by the driver, like employees at a workbench.

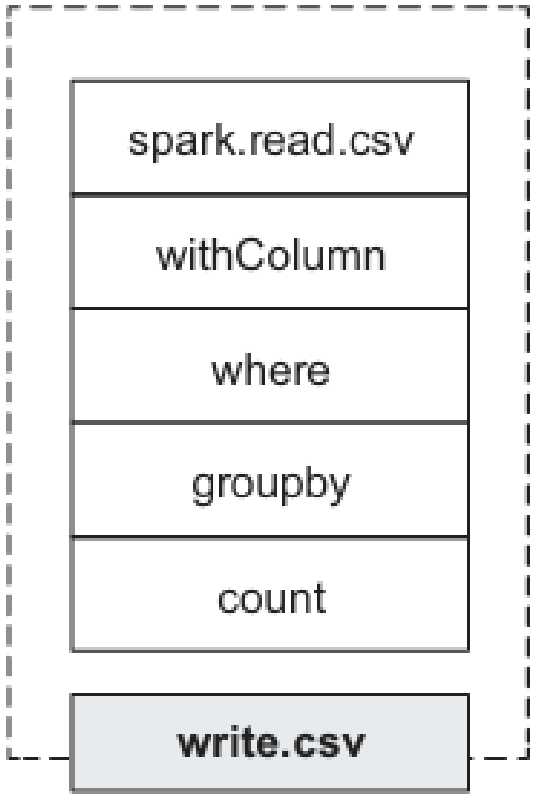
- Your floor lead/driver has all the qualities a good manager has: it's **smart, cautious, and lazy.**
- Every instruction you're providing in Spark can be classified into two categories: **transformations** and **actions.** Actions are what many programming languages would consider I/O.

The most typical actions are the following:

- Printing **information** on the screen
- **Writing data** to a hard drive or cloud bucket
- **Counting the number** of records

In Spark, we'll see those instructions most often via the **show(), write(), and count()** methods on a data frame.

PySpark does not evaluate all data transformations (including reading data). A variable containing a series of data frame transformations will return almost immediately, as no data work is being performed.



write.csv explicitly writes data to disk. An operation where PySpark actually writes or shows data is called an action and triggers the actual data work. No action, no visible result, no work. That's laziness!

Figure 1.3 Breaking down the data frame instructions as a series of transformations and one action. Each "job" Spark will perform consists of zero or more transformations and **one** action.

Transformations

- **Adding a column** to a table
- Performing an **aggregation** according to certain keys
- Computing **summary statistics**
- Training a **machine learning model**

- When thinking about computation over data, you, as the developer, are only concerned about the computation leading to an **action**.
- Spark, with **its lazy computation model**, will take this to the extreme and avoid performing data work until an **action triggers** the **computation chain.**
- Before that, the **driver will store your instructions**.
- **count()** is a transformation when applied as an aggregation function (where it counts the number of records of each group) but an action when applied on a data frame (where it counts the number of records in a data frame)

- First, **storing instructions** in **memory** takes much less space than **storing intermediate** data results.
- If you are performing many operations on a data set and are materializing the data each step of the way, you'll **blow your storage much faste**r, although you don't need the intermediate results.
- We can all agree that less waste is better.
- Second, by having the full list of tasks to be performed available, the **driver** can **optimize the work** between **executors** much more **efficiently.**
- It can use the **information available at run time,** such as the **node where specific parts of the data are located**. It can also reorder, eliminate useless transformations, combine **multiple operations**, and **rewrite some portion of the program** more effectively, if necessary.

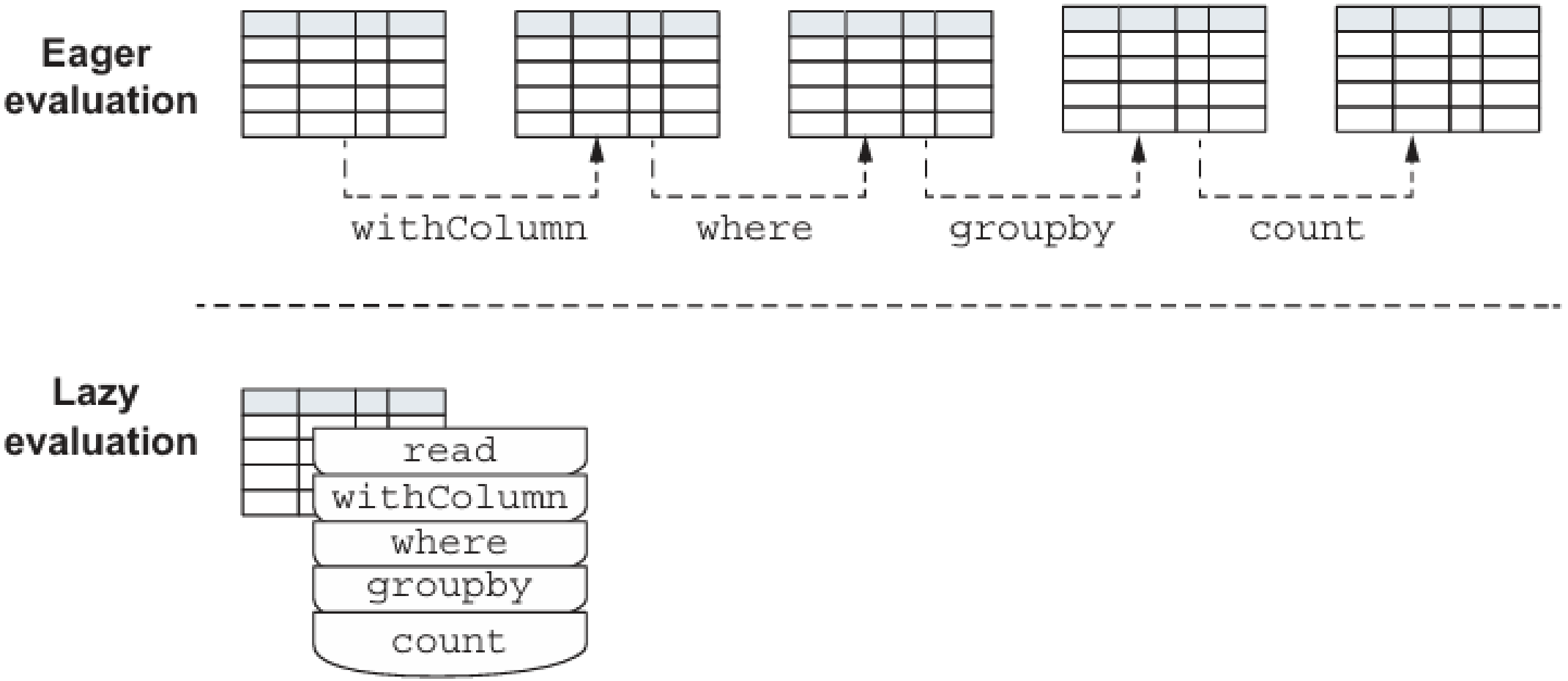


Figure 1.4 Eager versus lazy evaluation: storing (and computing on the fly) transformation saves memory by reducing the need for intermediate data frames. It also makes it easier to recreate the data frame if one of the nodes fails.

- Third, should one node fail during processing—computers fail!—Spark will be able to recreate the missing chunks of data since it has the instructions cached.
- It'll read the relevant chunk of data and process it up to where you are without the need for you to do anything.
- With this, you can focus on the data-processing aspect of your code,
- off loading the disaster and recovery part to Spark.
- Instead, you can iteratively build your chain of transformation, one at a time, and when you're ready to launch the computation, you can add an action and let Spark work its magic.
- **Lazy computation is a fundamental aspect of Spark's operating model and part of the reason it's so fast.**

What’s a manager without competent employees?

- Once the task, with its action, has been received, the driver starts allocating data to what Spark calls **executors**.
- **Executors** are processes that run computations and store data for the application.
- Those **executors** sit on what’s called a **worker node**, which is the **actual computer**.
- In our factory analogy, an **executor** is an employee performing the work, while the worker node is a workbench where many employees/executors can work.

- We first encode our instructions in **Python code**, forming a **driver** program.
- When submitting our program (or launching a PySpark shell), the **cluster manager** allocates resources for us to use. Those will mostly **stay constant** (with the exception of **auto-scaling**) for the duration of the program.
- The driver **ingests** your **code** and translates it into Spark **instructions**. Those instructions are either **transformations** or **actions**.
- Once the **driver** reaches an action, it **optimizes** the whole computation **chain** and **splits** the work between **executors**.
- **Executors** are processes performing the actual data work, and they reside on machines labeled **worker nodes**

PySpark is the Python API for Spark, a distributed framework for large-scale data analysis. It provides the expressiveness and dynamism of the Python pro gramming language to Spark.

Spark is fast: it owes its speed to a **judicious usage of the RAM** available and an **aggressive** and **lazy query optimizer.**

You can use Spark in **Python, Scala, Java, R,** and more. You can also use **SQL** for data manipulation.

Spark uses a driver that processes the instructions and orchestrates the work. The executors receive the instructions from the master and perform the work. All instructions in PySpark are either transformations or actions. Because Spark is lazy, only actions will trigger the computation of a chain of instructions.

Section -2

1 We start by loading or reading the data we wish to work with.

2 We transform the data, either via a few simple instructions or a very complex machine learning model.

3 We then export (or sink) the resulting data, either into a file or by summarizing our findings into a visualization.

Listing 2.2 Creating a `SparkSession` entry point from scratch

```
from pyspark.sql import SparkSession

spark = (SparkSession
         .builder
         .appName("Analyzing the vocabulary of Pride and Prejudice.")
         .getOrCreate())
```

The SparkSession entry point is located in the pyspark.sql package, providing the functionality for data transformation.

PySpark provides a builder pattern abstraction for constructing a SparkSession, where we chain the methods to configure the entry point.

Providing a relevant appName helps in identifying which programs run on your Spark cluster (see chapter 11).

NOTE By using the getOrCreate() method, your program will work in both interactive and batch mode by avoiding the creation of a new SparkSession if one already exists.

Note that if a session already exists, you won’t be able to change certain configuration settings (mostly related to JVM options).

If you need to change the configuration of your SparkSession, kill everything and start from scratch to avoid any confusion.

Reading older PySpark code

While this book shows modern PySpark programming, we are not living in a vacuum. Online you might face older PySpark code that uses the former `SparkContext/sqlContext` combo. You'll also see the `sc` variable mapped to the `SparkContext` entry point. With what we know about `SparkSession` and `SparkContext`, we can reason about old PySpark code by using the following variable assignments:

```
sc = spark.sparkContext
sqlContext = spark
```

You'll see traces of `SQLContext` in the API documentation for backward compatibility. I recommend avoiding using this, as the new `SparkSession` approach is cleaner, simpler, and more future-proof.

Deciding how chatty you want PySpark to be

- spark.sparkContext.setLogLevel("KEYWORD")

Table 2.1 Log-level keywords

Keyword	Signification
`OFF`	No logging at all (not recommended).
`FATAL`	Only fatal errors. A fatal error will crash your Spark cluster.
`ERROR`	Will show `FATAL`, as well as other recoverable errors.
`WARN`	Add warnings (and there are quite a lot of them).
`INFO`	Will give you runtime information, such as repartitioning and data recovery (see chapter 1).

Table 2.1 Log-level keywords *(continued)*

Keyword	Signification
`DEBUG`	Will provide debug information on your jobs.
`TRACE`	Will trace your jobs (more verbose debug logs). Can be quite informative but very annoying.
`ALL`	Everything that PySpark can spit, it will spit. As useful as `OFF`.

Mapping our program

- 1 **Read**—Read the input data (we're assuming a plain text file).
- 2 **Token**—Tokenize each word.
- 3 **Clean**—Remove any punctuation and/or tokens that aren't words. Lowercase each word.
- 4 **Count**—Count the frequency of each word present in the text.
- 5 Answer—Return the top 10 (or 20, 50, 100).

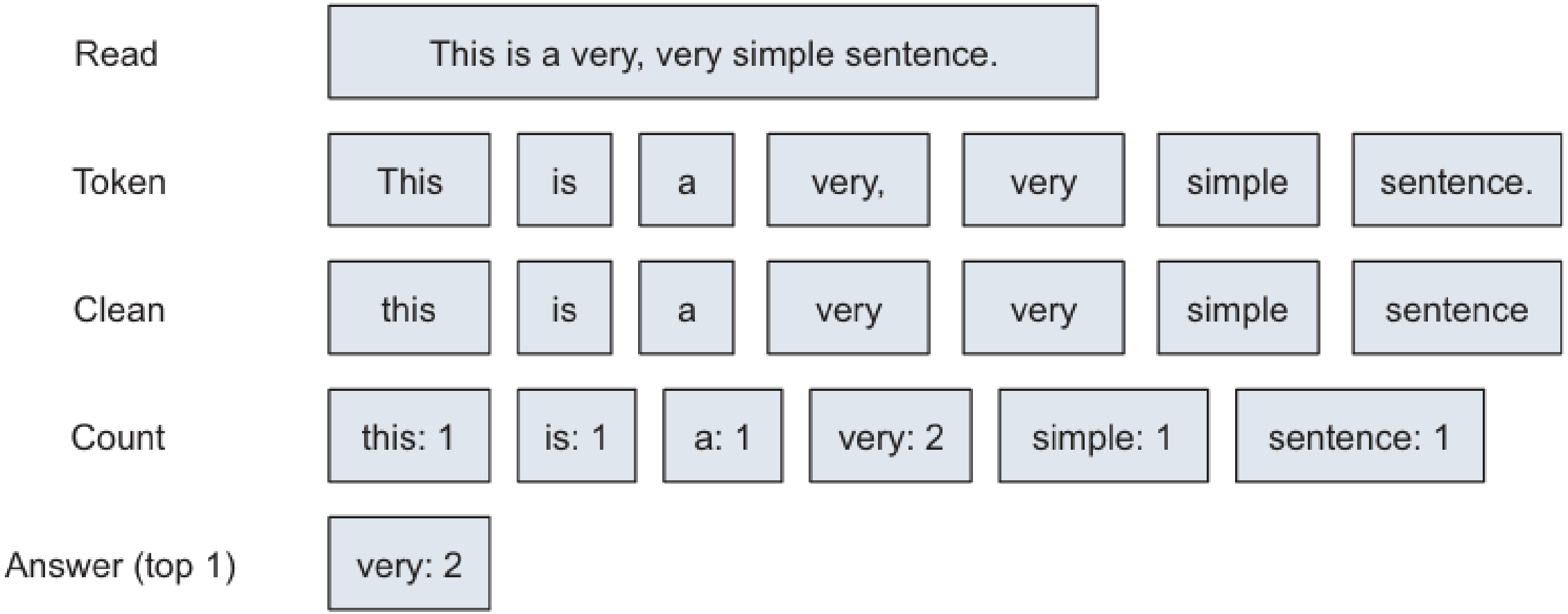


Figure 2.1 A simplified flow of our program, illustrating the five steps

Reading data into a data frame with spark.read

- The RDD
- The data frame

The RDD

- The RDD was the only structure for a long time.
- It looks like a distributed collection of objects (or rows).
- I visualize this as a bag that you give orders to.
- You pass orders to the RDD through regular Python functions over the items in the bag.

The data frame

- The **data frame** is a stricter version of the RDD.
- Conceptually, you can think of it like a table, where each cell can contain one value.
- The data frame makes heavy usage of the concept of columns, where you operate on columns instead of on records, like in the RDD.

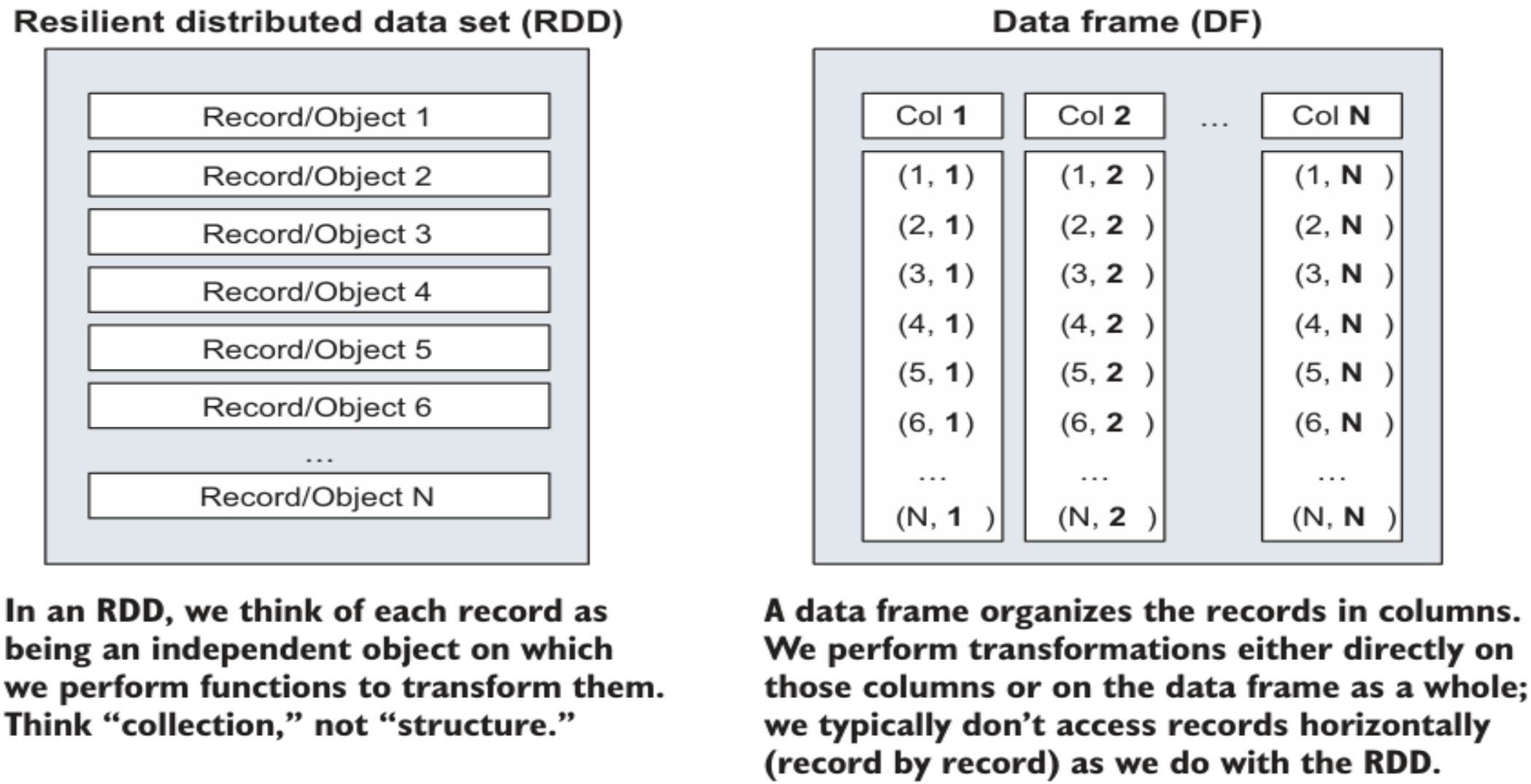


Figure 2.2 An RDD versus a data frame. In the RDD, we think of each record as an independent entity. With the data frame, we mostly interact with columns, performing functions on them. We still can access the rows of a data frame via RDD if necessary.

Listing 2.4 The `DataFrameReader` object

```
In [3]: spark.read
Out[3]: <pyspark.sql.readwriter.DataFrameReader at 0x115be1b00>

In [4]: dir(spark.read)
Out[4]: [<some content removed>, _spark', 'csv', 'format', 'jdbc', 'json',
'load', 'option', 'options', 'orc', 'parquet', 'schema', 'table', 'text']
```

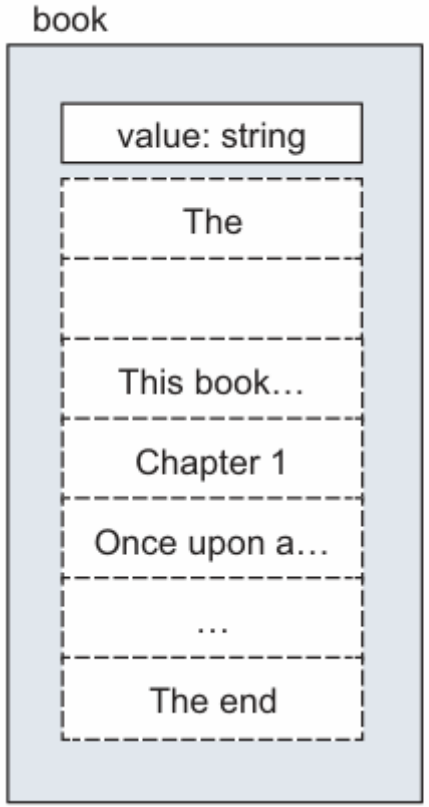


Figure 2.3 A high-level logical schema of our `book` data frame, containing a `value` string column. We can see the name of the column, its type, and a small snippet of the data.

- We use schemas to understand the data and its type (integer, string, date, etc.) for a given data frame.
- Spark displays the **logical schema** when we input the variable in the REPL: columns and types.
- In practice, your **data frame will be distributed across multiple nodes**, each one having a segment of the records.
- When performing data transformation and analysis, it is more conve nient to work with the logical schema.

Listing 2.6 Printing the schema of our data frame

```
book.printSchema()

# root
#  |-- value: string (nullable = true)

print(book.dtypes)

# [('value', 'string')]
```

Each data frame tree starts with a root, which the columns are attached to.

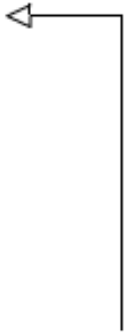
We have one column value, containing strings that can be null (or None in Python terms).

The same information is stored as a list of tuples under the data frame's dtypes attribute.

Listing 2.8 Showing a little data using the `.show()` method

```
book.show()
```

```
# +--------------------+
# |               value|
# +--------------------+
# |The Project Guten...|
# |                    |
# |This eBook is for...|
# |almost no restric...|
# |re-use it under t...|
# |with this eBook o...|
# |                    |
# |                    |
# |Title: Pride and ...|
# |                    |
# |  [... more records]|
# |Character set enc...|
# |                    |
# +--------------------+
# only showing top 20 rows
```



Spark displays the data from the data frame in an ASCII art-like table, limiting the length of each cell to 20 characters. If the contents spill over the limit, an ellipsis is added at the end.

- The **show()** method takes three optional parameters:
- **n** can be set to any positive integer and will display that number of rows.
- **truncate**, if set to true, will truncate the columns to display only 20 characters. Set to False, it will display the whole length, or any positive integer to truncate to a specific number of characters.
- **vertical** takes a Boolean value and, when set to True, will display each record as a small table. If you need to check records in detail, this is a very useful option

Show()

- we want to avoid accidentally triggering the chain of computations, so the Spark developers made show() explicit.
- When building a complicated chain of transformations, triggering its execution is a lot more annoying and time-consuming than having to type the show() method when you're ready.
- This **transformation**-versus-**action** distinction also leads to more opportunities for the **Spark optimizer** to generate a more **efficient program**

For eager evaluation not lazy

- from pyspark.sql import SparkSession spark = (SparkSession.builder .config("spark.sql.repl.eagerEval.enabled", "True") .getOrCreate())

Simple column transformations: Moving from a sentence to a list of words

Listing 2.10 Splitting our lines of text into arrays or words

```
from pyspark.sql.functions import split

lines = book.select(split(book.value, " ").alias("line"))

lines.show(5)

# +--------------------+
# |                line|
# +--------------------+
# |[The, Project, Gu...|
# |                  []|
# |[This, eBook, is,...|
# |[almost, no, rest...|
# |[re-use, it, unde...|
# +--------------------+
# only showing top 5 rows
```

- The **select()** method and its canonical usage, which is selecting data
- The **alias()** method to rename transformed columns
- Importing column functions from pyspark.sql.functions and using them

Listing 2.12 Selecting the `value` column from the `book` data frame

```
from pyspark.sql.functions import col

book.select(book.value)
book.select(book["value"])
book.select(col("value"))
book.select("value")
```

Listing 2.14 Our data frame before and after the aliasing

```
book.select(split(col("value"), " ")).printSchema()
# root
#  |-- split(value,  , -1): array (nullable = true)
#  |    |-- element: string (containsNull = true)

book.select(split(col("value"), " ").alias("line")).printSchema()

# root
#  |-- line: array (nullable = true)
#  |    |-- element: string (containsNull = true)
```

Our new column is called split(value, , -1), which isn't really pretty.

We aliased our column to the name line. Much better!

When you're using a method where you're specifying which columns you want to appear, like the select() method, use alias().

If you just want to rename a column without changing the rest of the data frame, use **.withColumnRenamed.** Note that, should the column not exist, PySpark will treat this method as a no-op and not perform anything.

Explode function

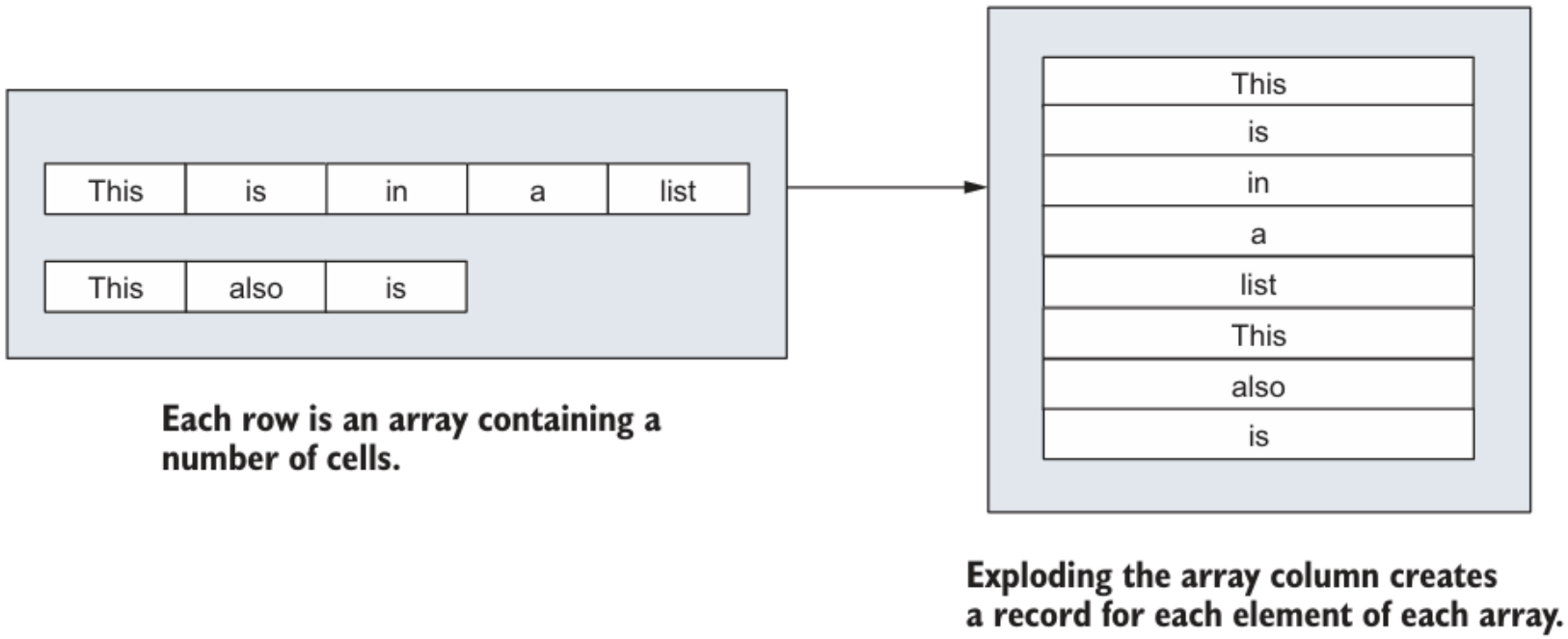


Figure 2.4 Exploding a data frame of `array[String]` into a data frame of `String`. Each element of each array becomes its own record.

Listing 2.18 Using `regexp_extract` to keep what looks like a word

```
from pyspark.sql.functions import regexp_extract
words_clean = words_lower.select(
    regexp_extract(col("word_lower"), "[a-z]+", 0).alias("word")
)

words_clean.show()

# +----------+
# |      word|
# +----------+
# |       the|
# |   project|
# |gutenberg|
# |     ebook|
# |        of|
# |     pride|
# |       and|
# |prejudice|
# |        by|
# |      jane|
# |    austen|
# |          |
# |      this|
# |     ebook|
# |        is|
```

We only match for multiple lowercase characters (between a and z). The plus sign (+) will match for one or more occurrences.

Regular expressions for the rest of us

PySpark uses regular expressions in two functions we have used so far: `regexp_extract()` and `split()`. You do not have to be a regexp expert to work with PySpark (I certainly am not). Throughout the book, each time I use a nontrivial regular expression, I'll provide a plain English definition so you can follow along.

Grouping records: Counting word frequencies

`words_nonull: DataFrame`

Word
online
some
online
some
some
still
…
cautious

`groups = words_nonull.groupby("word"): GroupedData`

Word	
online	
some	
…	…
cautious	

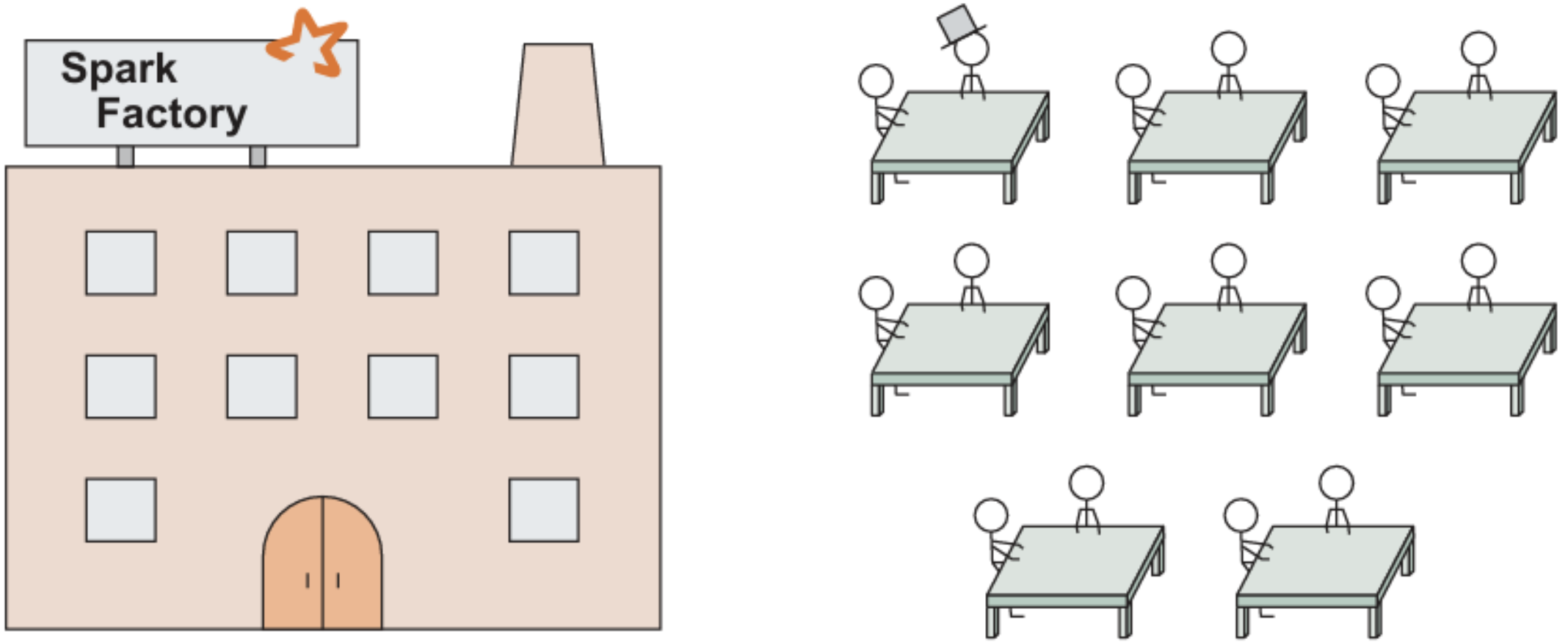


Figure 1.1 A totally relatable data factory, outside and in. Ninety percent of the time we care about the whole factory, but knowing how it's laid out helps when reflecting on our code performance.

Word	Count
same	2
cautious	1
all	6

Word	Count
same	3
none	10

Word	Count
cautious	3
same	1
all	2

Step 1: The data is unordered in each partition, after the groupby operation.

Word	Count
none	10
all	8
same	7

Step 2: Each worker sends enough summarized data to the master node for display.

Figure 3.2 A distributed group by on our `words_nonull` data frame. The work is performed in a distributed fashion until we need to assemble the results in a cohesive display via `show()`.

coalesce()

- By default, PySpark will give you one file per partition.
- This means that our program, as run on my machine, yields **200 partitions** at the end.
- This isn't the best for **portability**. To reduce the number of partitions, we apply the **coalesce()** method with the desired number of partitions.
- The next listing shows the difference when using coalesce(1) on our data frame before writing to disk. We still get a directory, but there is a single CSV file inside of it.

NOTE

- You might have realized that we're not ordering the file before writing it. Since our data here is pretty small, we could have written the words by decreasing order of frequency. If you have a large data set, this operation will be quite expensive. Furthermore, since reading is a potentially distributed operation, what guarantees that it'll get read the same way? Never assume that your data frame will keep the same ordering of records unless you explicitly ask via **orderBy()** right before the showing step.

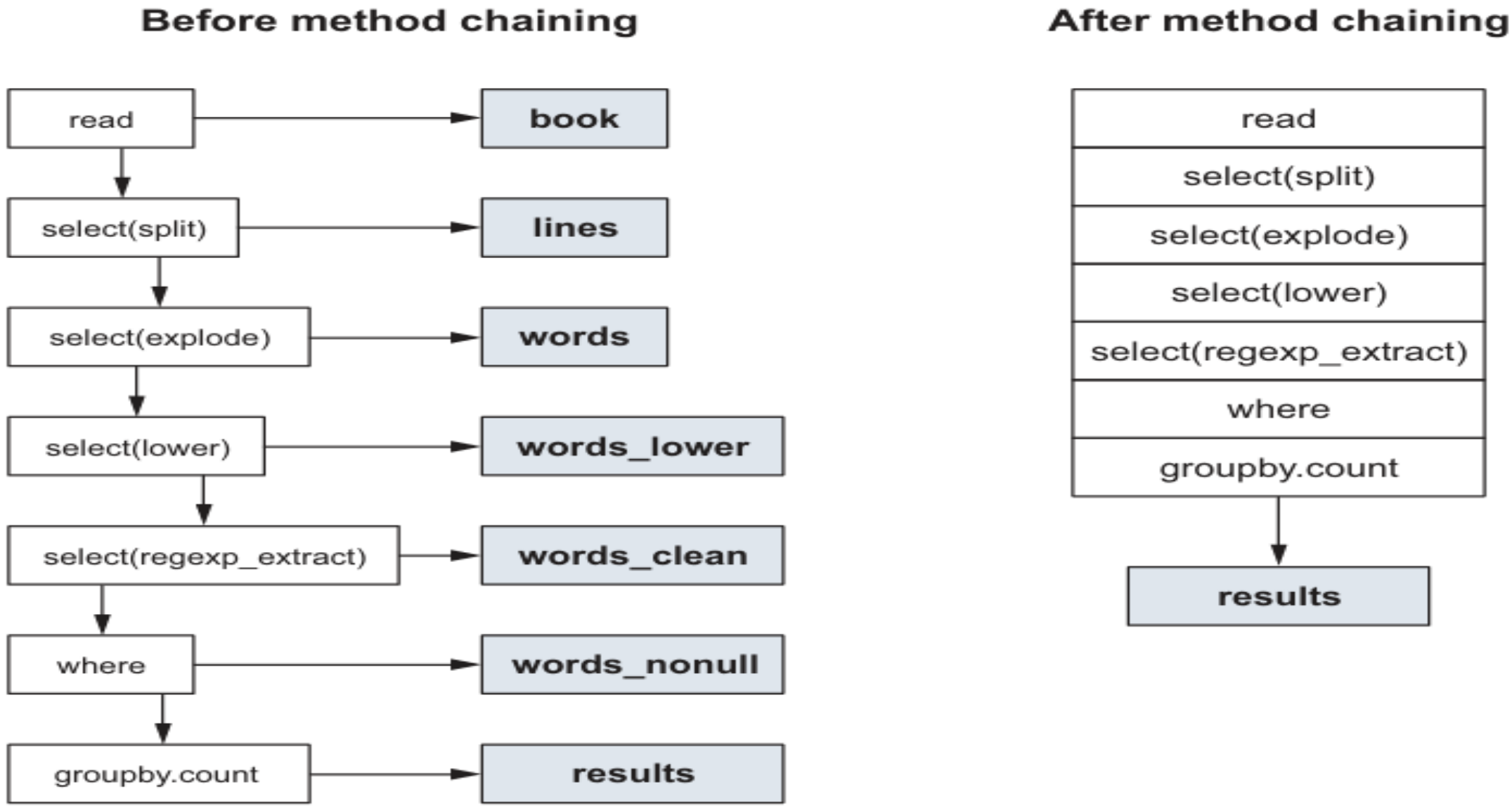


Figure 3.3 Method chaining eliminates the need for intermediate variables.

Listing 3.8 Chaining for writing over the same variable

```
df = spark.read.text("./data/gutenberg_books/1342-0.txt")
df = df.select(F.split(F.col("value"), " ").alias("line"))
```

Instead of doing this . . .

```
df = (
      spark.read.text("./data/gutenberg_books/1342-0.txt")
      .select(F.split(F.col("value"), " ").alias("line"))
    )
```

. . . you can do this—no variable repetition!

Make your life easier by using Python's parentheses

If you look at the "after" code in listing 3.7, you'll notice that I start the right side of the equal sign with an opening parenthesis (`spark = ( [...]`). This is a trick I use when I need to chain methods in Python. If you don't wrap your result into a pair of parentheses, you'll need to add a `\` character at the end of each line, which adds visual noise to your program. PySpark code is especially prone to line breaks when you use method chaining:

```
results = spark\
          .read.text('./data/ch02/1342-0.txt')\
          ...
```

As a lazy alternative, I am a big fan of using Black as a Python code formatting tool (https://black.readthedocs.io/). It removes a lot of the guesswork involved in having your code logically laid out and consistent. Since we read code more than we write it, readability matters.

Listing 3.9 Submitting our job in batch mode

```
$ spark-submit ./code/Ch03/word_count_submit.py

# [...]
# +----+-----+
# |word|count|
# +----+-----+
# | the| 4480|
# |  to| 4218|
# |  of| 3711|
# | and| 3504|
# | her| 2199|
# |   a| 1982|
# |  in| 1909|
# | was| 1838|
# |   i| 1749|
# | she| 1668|
# +----+-----+
# only showing top 10 rows
# [...]
```

TIP If you get a deluge of `INFO` messages, don't forget that you have control over this: use `spark.sparkContext.setLogLevel("WARN")` right after your `spark` definition. If your local configuration has `INFO` as a default, you'll still get a slew of messages until it catches this line, but it won't obscure your results.

Listing 3.10 Scaling our word count program using the glob pattern

```
# Before
results = spark.read.text('./data/gutenberg_books/1342-0.txt')

# After
results = spark.read.text('./data/gutenberg_books/*.txt')
```

Here we have a single file passed as a parameter . . .

. . . and here the star (or glob) picks all the text files within the directory.

Analyzing tabular data with pyspark.sql

Section -4

Listing 4.2 Creating a data frame out of our grocery list

```
my_grocery_list = [
    ["Banana", 2, 1.74],
    ["Apple", 4, 2.04],
    ["Carrot", 1, 1.09],
    ["Cake", 1, 10.99],
]

df_grocery_list = spark.createDataFrame(
    my_grocery_list, ["Item", "Quantity", "Price"]
)

df_grocery_list.printSchema()
# root
#  |-- Item: string (nullable = true)
#  |-- Quantity: long (nullable = true)
#  |-- Price: double (nullable = true)
```

My grocery list is encoded in a list of lists.

PySpark automatically inferred the type of each field from the information Python had about each value.

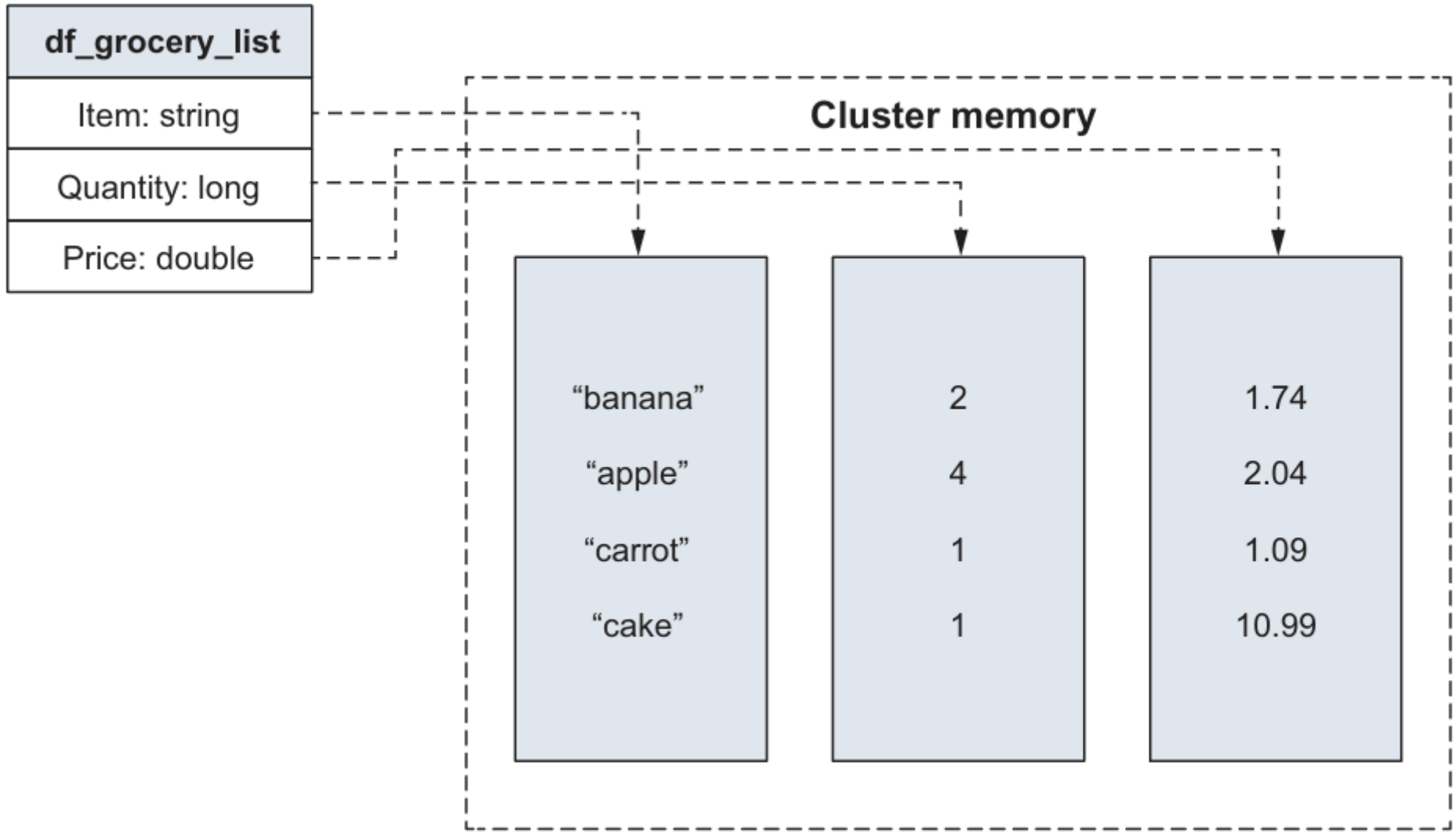


Figure 4.2 Each column of our data frame maps to some place on our worker nodes.

- PySpark doesn’t provide any charting capabilities and doesn’t play with other charting libraries (like Matplotlib, seaborn, Altair, or plot.ly), and this makes a lot of sense: PySpark distributes your data over many computers.
- It doesn’t make much sense to distribute a chart creation. The usual solution will be to transform your data using PySpark, use the toPandas() method to transform your PySpark data frame into a pandas data frame, and then use your favorite charting library.
- When using toPandas(), remember that you lose the advantages of working with multiple machines, as the data will accumulate on the driver.
- Reserve this operation for an aggregated or manageable data set. While this is a crude formula, I usually take the number of rows times the number of columns;
- if this number is over 100,000 (for a 16 GB driver), I try to reduce it further. This simple trick helps me get a sense of the size of the data I am dealing with, as well as what’s possible given my driver size.

- You do not want to move your data between a pandas and a PySpark data frame all the time.
- Reserve toPandas() for either discrete operations or for moving your data into a pandas data frame once and for all.
- Moving back and forth will yield a ton of unnecessary work in distributing and collecting the data for nothing.

Listing 4.3 Reading our broadcasting information

```
import os

DIRECTORY = "./data/broadcast_logs"
logs = spark.read.csv(
    os.path.join(DIRECTORY, "BroadcastLogs_2018_Q3_M8.CSV"),
    sep="|",
    header=True,
    inferSchema=True,
    timestampFormat="yyyy-MM-dd",
)
```

We specify the file path where our data resides first.

Our file uses a vertical bar as delimiter/separator, so we pass | as a parameter to sep.

header takes a Boolean. When true, the first row of your file is parsed as the column names.

inferSchema takes a Boolean as well. When true, it'll pre-parse the data to infer the type of the column.

timestampFormat is used to inform the parser of the format (year, month, day, hour, minutes, seconds, microseconds) of the timestamp fields (see section 4.4.3).

TIP Inferring the schema can be very expensive if you have a lot of data. In chapter 6, I cover how to work with (and extract) schema information; if you read a data source multiple times, it's a good idea to keep the schema information once inferred! You can also take a small representative data set to infer the schema, followed by reading the large data set.

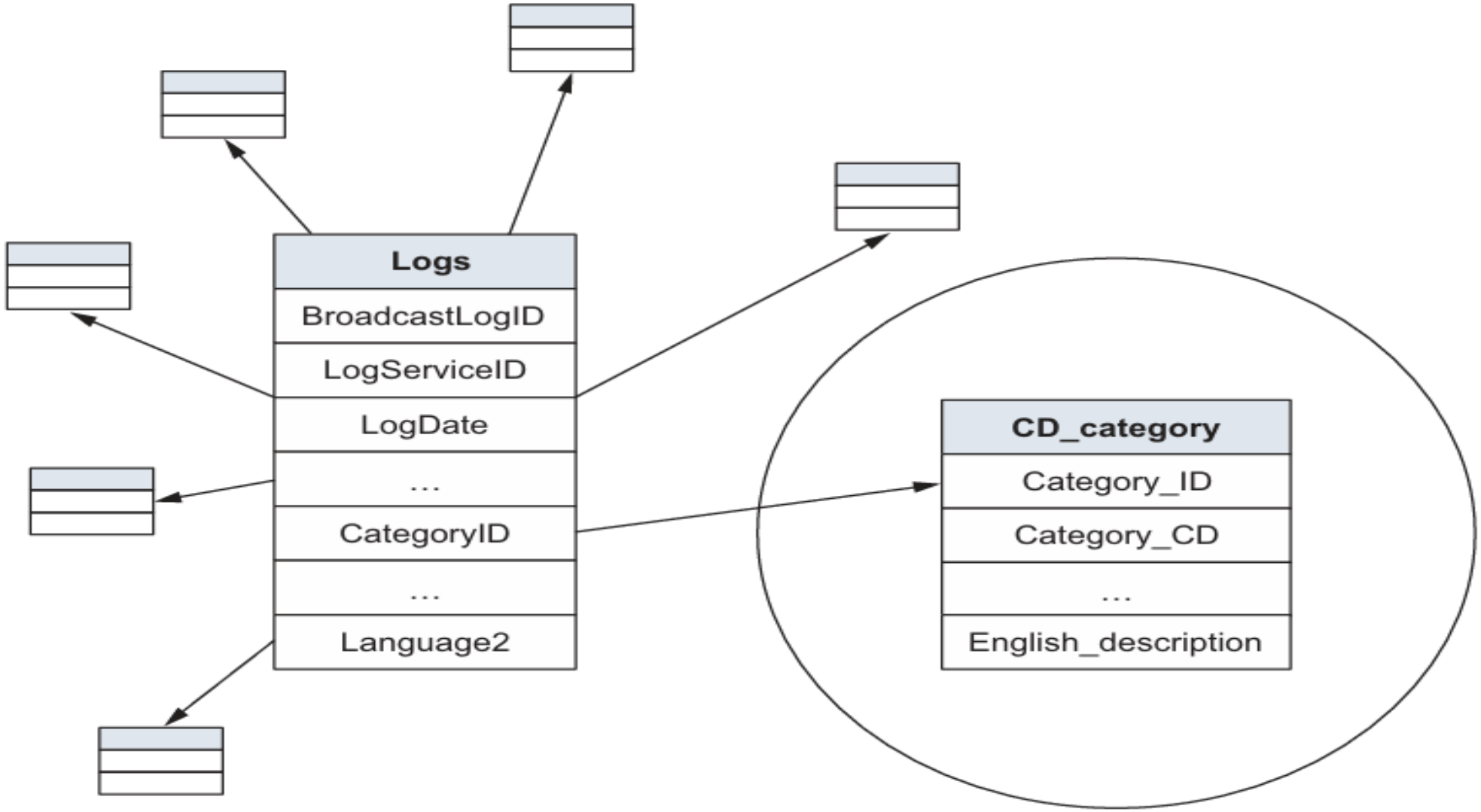


Figure 4.4 The `logs` table `ID` columns map to other tables, like the `CD_category` table, which links the `Category_ID` field.

Listing 4.11 Selecting and displaying the `Duration` column

```
logs.select(F.col("Duration")).show(5)

# +-----------------+
# |         Duration|
# +-----------------+
# |02:00:00.0000000|
# |00:00:30.0000000|
# |00:00:15.0000000|
# |00:00:15.0000000|
# |00:00:15.0000000|
# +-----------------+
# only showing top 5 rows

print(logs.select(F.col("Duration")).dtypes)

# [('Duration', 'string')]
```

The dtypes attribute of a data frame contains the name of the column and its type, wrapped in a tuple.

Listing 4.12 Extracting the hours, minutes, and seconds from the `Duration` column

```
logs.select(
    F.col("Duration"),
    F.col("Duration").substr(1, 2).cast("int").alias("dur_hours"),
    F.col("Duration").substr(4, 2).cast("int").alias("dur_minutes"),
    F.col("Duration").substr(7, 2).cast("int").alias("dur_seconds"),
).distinct().show(
    5
)
```

The original column, for sanity.

To avoid seeing identical rows, I've added a distinct() to the results.

The seventh and eighth characters are the seconds.

The fourth and fifth characters are the minutes.

The first two characters are the hours.

```
# +----------------+---------+-----------+-----------+
# |        Duration|dur_hours|dur_minutes|dur_seconds|
# +----------------+---------+-----------+-----------+
# |00:10:06.0000000|        0|         10|          6|
# |00:10:37.0000000|        0|         10|         37|
# |00:04:52.0000000|        0|          4|         52|
# |00:26:41.0000000|        0|         26|         41|
# |00:08:18.0000000|        0|          8|         18|
# +----------------+---------+-----------+-----------+
# only showing top 5 rows
```

logs

Duration	...
00:10:30.0000000	
00:25:52.0000000	
00:28:08.0000000	
06:00:00.0000000	
00:32:08.0000000	

```
logs.select(
"Duration",
[…].alias("Duration_seconds"))
```

Duration	Duration_seconds
00:10:30.0000000	630
00:25:52.0000000	1552
00:28:08.0000000	1688
06:00:00.0000000	21600
00:32:08.0000000	1928

```
logs.withColumn("Duration_seconds",[…])
```

Duration	...	Duration_seconds
00:10:30.0000000		630
00:25:52.0000000		1552
00:28:08.0000000		1688
06:00:00.0000000		21600
00:32:08.0000000		1928

Figure 4.5 `select()` versus `withColumn()`, visually. `withColumn()` keeps all the preexisting columns without the need the specify them explicitly.

Tidying our data frame: Renaming and reordering columns

Section -5 joining and grouping

Listing 5.2 A bare-bone recipe for a join in PySpark

```
[LEFT].join(
    [RIGHT],
    on=[PREDICATES]
    how=[METHOD]
)
```

Listing 5.3 A bare-bone join in PySpark, with left and right tables filled in

```
logs.join(
    log_identifier,
    on=[PREDICATES]
    how=[METHOD]
)
```

logs is the left table . . .

. . . and log_identifier is the right table.

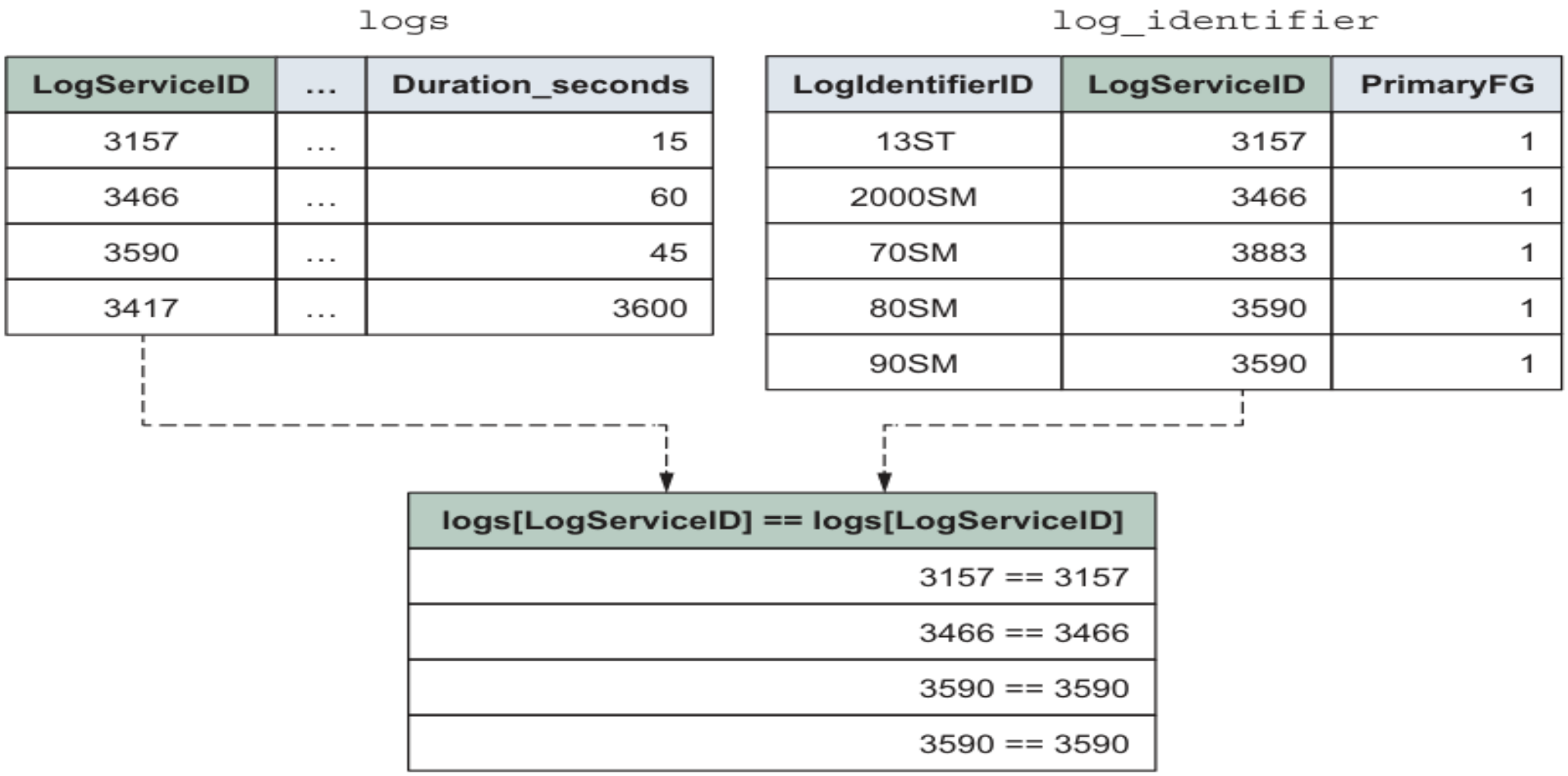


Figure 5.1 A simple join predicate resolution between `logs` and `log_identifier` using `LogServiceID` in both tables and equality testing in the predicate. I show only the four successes in the result table. Our predicate is applied to a sample of our two tables: `3590` in the left table resolves the predicate twice, while `3417` on the left and `3883` on the right have no matches.

logs (4 records)

LogServiceID	...	Duration_seconds
3157	...	15
3466	...	60
3590	...	45
3417	...	3600

log_identifier (5 records)

LogIdentifierID	LogServiceID	PrimaryFG
13ST	3157	1
2000SM	3466	1
70SM	3883	1
80SM	3590	1
90SM	3590	1

logs[LogServiceID]	logs[...]	log_identifier[LogServiceID]	log_identifier[...]
3157	...	3157	...
3466	...	3466	...
3590	...	3590	...
3590	...	3590	...

Figure 5.2 An inner join. Each successful predicate creates a joined record.

logs (4 records)

LogServiceID	...	Duration_seconds
3157	...	15
3466	...	60
3590	...	45
3417	...	3600

log_identifier (5 records)

LogIdentifierID	LogServiceID	PrimaryFG
13ST	3157	1
2000SM	3466	1
70SM	3883	1
80SM	3590	1
90SM	3590	1

left_join

logs[LogServiceID]	logs[...]	log_identifier[LogServiceID]	log_identifier[...]
3157	...	3157	...
3466	...	3466	...
3590	...	3590	...
3590	...	3590	...
3417	...	**null**	**null**

right_join

logs[LogServiceID]	logs[...]	log_identifier[LogServiceID]	log_identifier[...]
3157	...	3157	...
3466	...	3466	...
null	**null**	3883	...
3590	...	3590	...
3590	...	3590	...

Figure 5.3 A left and right joined table. All the records of the direction table are present in the resulting table.

logs (4 records)

LogServiceID	...	Duration_seconds
3157	...	15
3466	...	60
3590	...	45
3417	...	3600

log_identifier (5 records)

LogIdentifierID	LogServiceID	PrimaryFG
13ST	3157	1
2000SM	3466	1
70SM	3883	1
80SM	3590	1
90SM	3590	1

outer_join

logs[LogServiceID]	logs[...]	log_identifier[LogServiceID]	log_identifier[...]
3157	...	3157	...
3466	...	3466	...
null	**null**	3883	...
3590	...	3590	...
3590	...	3590	...
3417	...	**null**	**null**

Figure 5.4 A left and right joined table. We can see all the records from both tables.

LEFT SEMI-JOIN AND LEFT ANTI-JOIN

- A left semi-join (how="left_semi") is the same as an inner join, but keeps the col umns in the left table. It also won’t duplicate the records in the left table if they fulfill the predicate with more than one record in the right table. Its main purpose is to fil ter records from a table based on a predicate that is depending on another table.
- A left anti-join (how="left_anti") is the opposite of an inner join. It will keep only the records from the left table that do not match the predicate with any record in the right table. If a record from the left table matches a record from the right table, it gets dropped from the join operation.

Listing 5.5 Our join in PySpark, with all the parameters filled in

```
logs_and_channels = logs.join(
    log_identifier,
    on="LogServiceID",
    how="inner"
)
```

I could have omitted the how parameter outright, since inner join is the default.

logs (4 records)

LogServiceID	...	Duration_seconds
3157	...	15
3466	...	60
3590	...	45
3417	...	3600

log_identifier (5 records)

LogIdentifierID	LogServiceID	PrimaryFG
13ST	3157	1
2000SM	3466	1
70SM	3883	1
80SM	3590	1
90SM	3590	1

4 x 5 = 20 records

logs[LogServiceID]	logs[...]	log_identifier[LogServiceID]	log_identifier[...]
3157	...	3157	...
3157	...	3466	...
3157	...	3883	...
3157	...	3590	...
3157	...	3590	...
...	...	...	...
3417	...	3157	...
3417	...	3466	...
3417	...	3883	...
3417	...	3590	...
3417	...	3590	...

Figure 5.5 A visual example of a cross join. Each record on the left is matched to every record on the right.

Listing 5.6 A join that generates two seemingly identically named columns

```
logs_and_channels_verbose = logs.join(
    log_identifier, logs["LogServiceID"] == log_identifier["LogServiceID"]
)

logs_and_channels_verbose.printSchema()

# root
#   |-- LogServiceID: integer (nullable = true)
#   |-- LogDate: timestamp (nullable = true)
#   |-- AudienceTargetAgeID: integer (nullable = true)
#   |-- AudienceTargetEthnicID: integer (nullable = true)
#   [...]
#   |-- duration_seconds: integer (nullable = true)
#   |-- LogIdentifierID: string (nullable = true)
#   |-- LogServiceID: integer (nullable = true)
#   |-- PrimaryFG: integer (nullable = true)

try:
    logs_and_channels_verbose.select("LogServiceID")
except AnalysisException as err:
    print(err)

# "Reference 'LogServiceID' is ambiguous, could be: LogServiceID, LogServiceID.;"
```

This is one LogServiceID column . . .

. . . and this is another.

PySpark doesn't know which column we mean: is it LogServiceID or LogServiceID?

Listing 5.8 Using the origin name of the column for unambiguous selection

```
logs_and_channels_verbose = logs.join(
    log_identifier, logs["LogServiceID"] == log_identifier["LogServiceID"]
)

logs_and_channels.drop(log_identifier["LogServiceID"]).select(
    "LogServiceID")

# DataFrame[LogServiceID: int]
```

By dropping one of the two duplicated columns, we can then use the name for the other without any problem.

Listing 5.9 Aliasing our tables to resolve the origin

```
logs_and_channels_verbose = logs.alias("left").join(
    log_identifier.alias("right"),
    logs["LogServiceID"] == log_identifier["LogServiceID"],
)

logs_and_channels_verbose.drop(F.col("right.LogServiceID")).select(
    "LogServiceID"
)

# DataFrame[LogServiceID: int]
```

Our logs table gets aliased as left.

Our log_identifier gets aliased as right.

F.col() will resolve left and right as a prefix for the column names.

logs: DataFrame

ProgramClassCD	ProgramClass_Description	…	Duration_seconds
PGR	PROGRAM	…	15
PGR	PROGRAM	…	60
COM	COMMERCIAL MESSAGE	…	45
COM	COMMERCIAL MESSAGE	…	3600
PGR	PROGRAM	…	30
PFS	PROGRAM FIRST SEGMENT	…	60
…	…	…	540
COM	COMMERCIAL MESSAGE	…	60

Figure 5.6 The original data frame, with the focus on the columns we are grouping by

`logs.groupby("ProgramClassCD", "ProgramClass_Description"): GroupedData`

ProgramClassCD	ProgramClass_Description	[Group cell]
PGR	PROGRAM	… | Duration_seconds … | 15 … | 60 … | 30
COM	COMMERCIAL MESSAGE	… | Duration_seconds … | 45 … | 3600 … | 60
PFS	PROGRAM FIRST SEGMENT	… | Duration_seconds … | 60

There is one record per key set (in this case, each ProgramClassCD, ProgramClass_Description is unique).

All the nonkey columns are here, split between the key columns value.

Figure 5.7 The `GroupedData` object resulting from grouping

- agg() is not the only player in town You can also use groupby(), with the apply() (Spark 2.3+) and applyInPandas() (Spark 3.0+) method, in the creatively named split-apply-combine pattern. We explore this powerful tool in chapter 9. Other less-used (but still useful) methods are also available.

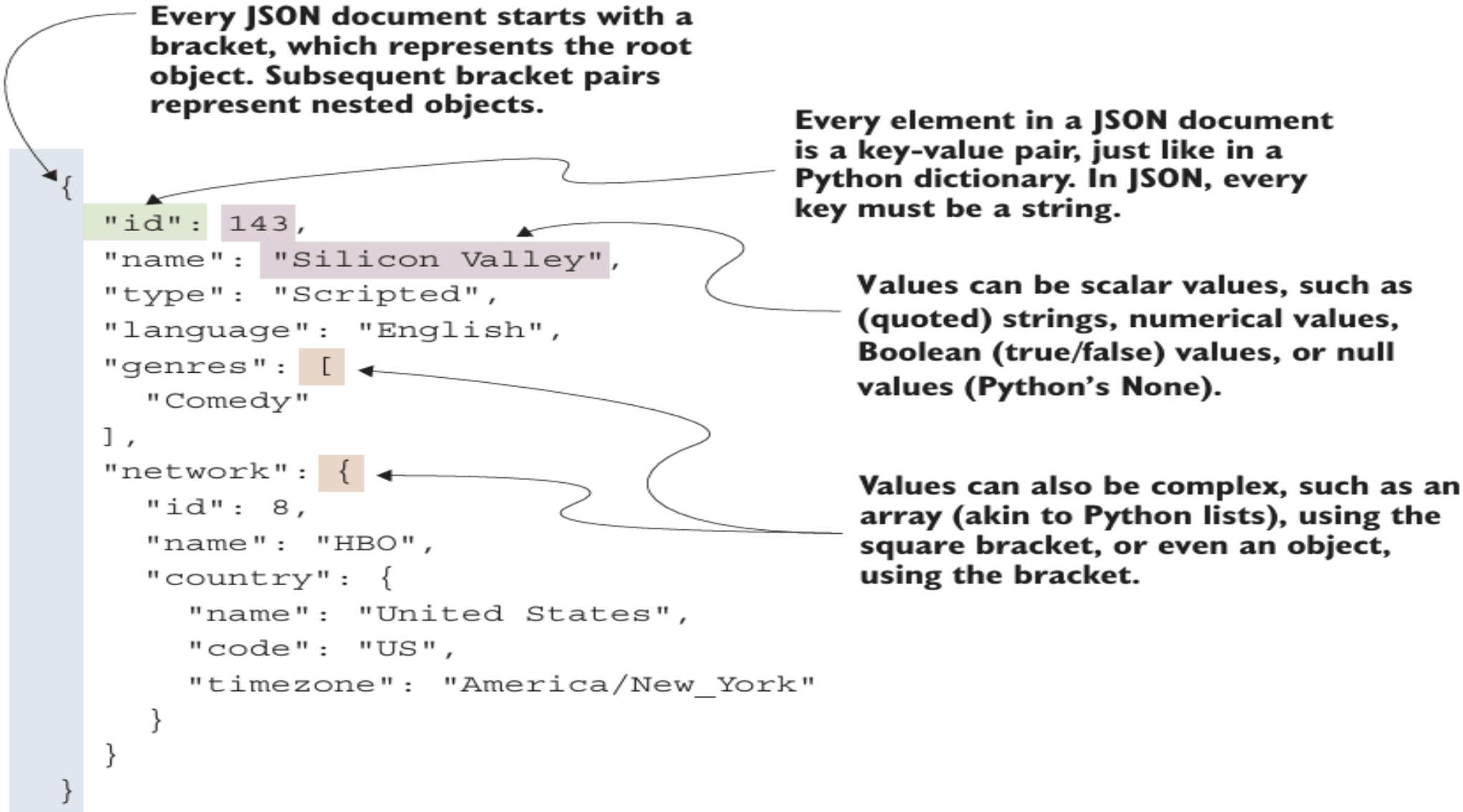


Figure 6.1 A simple JSON object, illustrating its main components: the root object, the keys, and the values. Objects use bracket delimiters and arrays/lists use square bracket

Listing 6.2 Reading a simple JSON document as a Python dictionary

```
import json

sample_json = """{
   "id": 143,
   "name": "Silicon Valley",
   "type": "Scripted",
   "language": "English",
   "genres": [
      "Comedy"
   ],
   "network": {
      "id": 8,
      "name": "HBO",
      "country": {
         "name": "United States",
         "code": "US",
         "timezone": "America/New_York"
      }
   }
}"""

document = json.loads(sample_json)
print(document)
# {'id': 143,
#  'name': 'Silicon Valley',
#  'type': 'Scripted',
#  'language': 'English',
#  'genres': ['Comedy'],
#  'network': {'id': 8,
#   'name': 'HBO',
#   'country': {'name': 'United States',
#    'code': 'US',
#    'timezone': 'America/New_York'}}}

type(document)
# dict
```

I import the json module, available in the Python standard library.

Our loaded document looks like a Python dictionary with string keys. Python recognized that 143 was an integer and parsed the number as such.

Our loaded document is of type dict.

SQL in spark

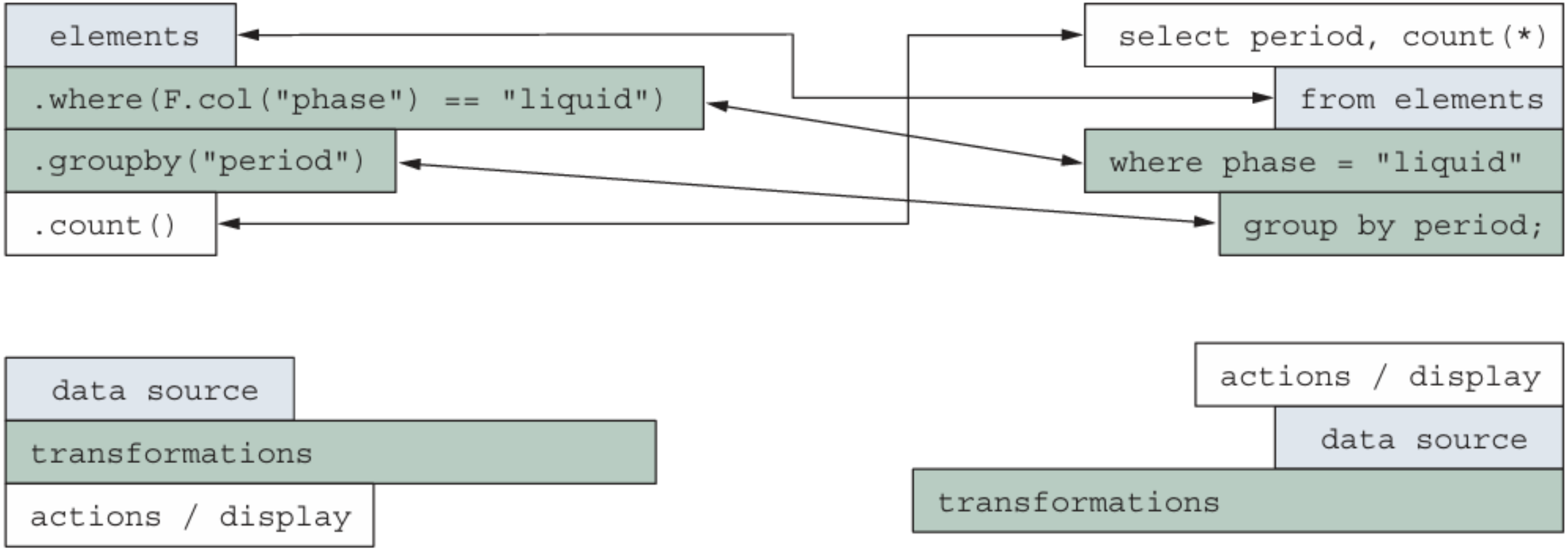


Figure 7.1 PySpark and SQL share the same keywords, but the order of operations differs. PySpark looks like an ordered list of operations (take this table, do these transformations, then finally show the result), while SQL has a more descriptive approach (show me the results from that table after performing these transformations).

```
elements.createOrReplaceTempView("elements")

spark.sql(
    "select period, count(*) from elements where phase='liq' group by period"
).show(5)

# +------+--------+
# |period|count(1)|
# +------+--------+
# |     6|       1|
# |     4|       1|
# +------+--------+
```

We register our table using the createOrReplaceTempView() method on the element data frame.

The same query works once Spark is able to de-reference the SQL view name.

Now we have a view registered. In the case of a low number of views to manage, it's pretty easy to keep the name in memory. What about if you have dozens of views or you need to delete some? Enter the catalog, Spark's way of managing its SQL namespace.

Advanced-ish topic: Spark SQL views and persistence

PySpark has four methods to create temporary views, and they look quite similar at first glance:

- `createGlobalTempView()`
- `createOrReplaceGlobalTempView()`
- `createOrReplaceTempView()`
- `createTempView()`

We can see that there is a two-by-two matrix of possibilities:

- Do I want to replace an existing view (`OrReplace`)?
- Do I want to create a global view (`Global`)?

The first one is relatively easy to answer: if you use `createTempView` with a name already being used for another table, the method will fail. On the other hand, if you use `createOrReplaceTempView()`, Spark will replace the old table with a new one. In SQL, it is equivalent to using `CREATE VIEW` versus `CREATE OR REPLACE VIEW`. I personally always use the latter, as it mimics Python's way of doing things: when reassigning a variable, you comply.

What about `Global`? The difference between a local view and a global view has to do with how long it will last in memory. A local table is tied to your `SparkSession`, while a global table is tied to the Spark application. The differences at this time are not significant, as we are not using multiple `SparkSessions` that need to share data. In the context of data analysis with Spark, you won't deal with multiple `SparkSessions` at once, so I usually don't use the `Global` methods.

Listing 7.4 Using the catalog to display our registered view and then drop it

```
spark.catalog
```

The catalog is reached through the catalog property of our SparkSession.

```
#  <pyspark.sql.catalog.Catalog at 0x117ef0c18>

spark.catalog.listTables()
```

The listTables method gives us a list of Table objects that contain the information we want.

```
#  [Table(name='elements', database=None, description=None,
#         tableType='TEMPORARY', isTemporary=True)]

spark.catalog.dropTempView("elements")

spark.catalog.listTables()

# []
```

To delete a view, we use the method dropTempView() and pass the name of the view as a parameter.

Our catalog now has no table for us to query.

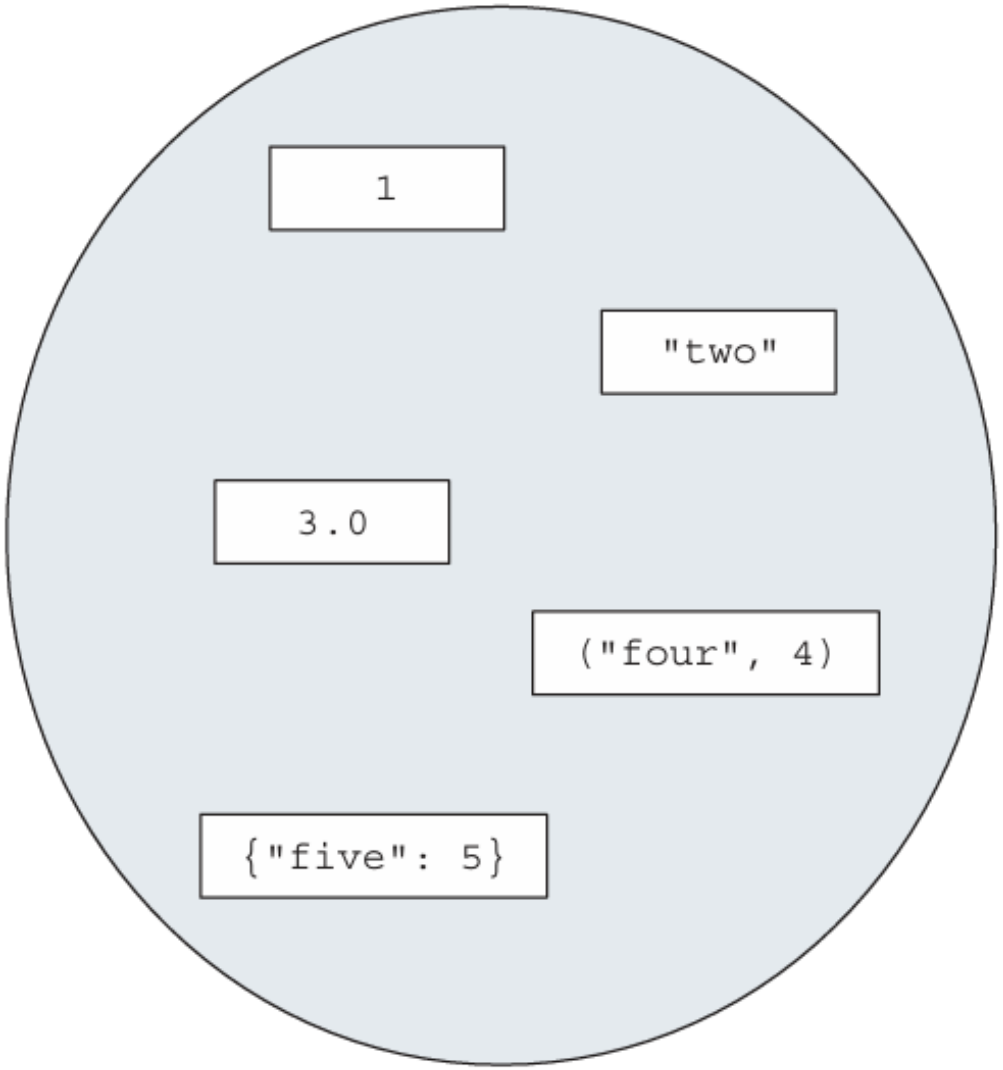


Figure 8.1 The `collection_rdd`. Each object is independent from the others in the container—no column, no structure, no schema.

Pyspark UI

Listing 11.1 Our end-to-end program counting the occurrence of words

```
from pyspark.sql import SparkSession
import pyspark.sql.functions as F

spark = SparkSession.builder.appName(
    "Counting word occurences from a book, one more time."
).getOrCreate()
```

Like with any modern PySpark program, ours starts with creating a SparkSession and connecting to our Spark instance.

```
results = (
    spark.read.text("./data/gutenberg_books/*.txt")
    .select(F.split(F.col("value"), " ").alias("line"))
    .select(F.explode(F.col("line")).alias("word"))
    .select(F.lower(F.col("word")).alias("word"))
    .select(F.regexp_extract(F.col("word"), "[a-z']+", 0).alias("word"))
    .where(F.col("word") != "")
    .groupby(F.col("word"))
    .count()
)

results.orderBy(F.col("count").desc()).show(10)
```

results maps to a data frame from a data source plus a series of transformations.

By show()-ing the results, we trigger the chain of transformations and display the top 10 most frequent words.

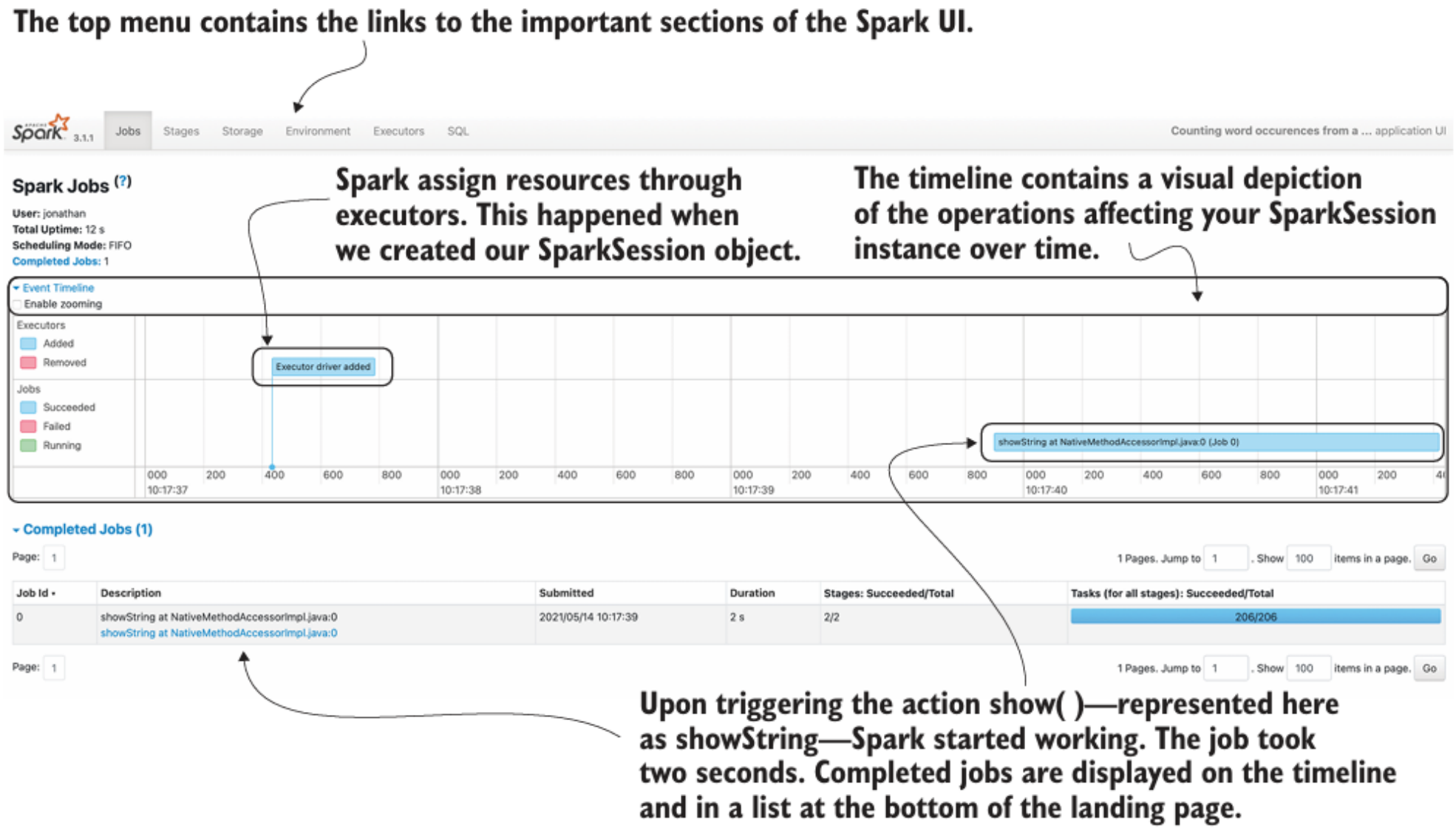


Figure 11.1 The landing page (Jobs) for the Spark UI. We see an empty timeline because the only event that has happened is the launch of the PySpark shell.

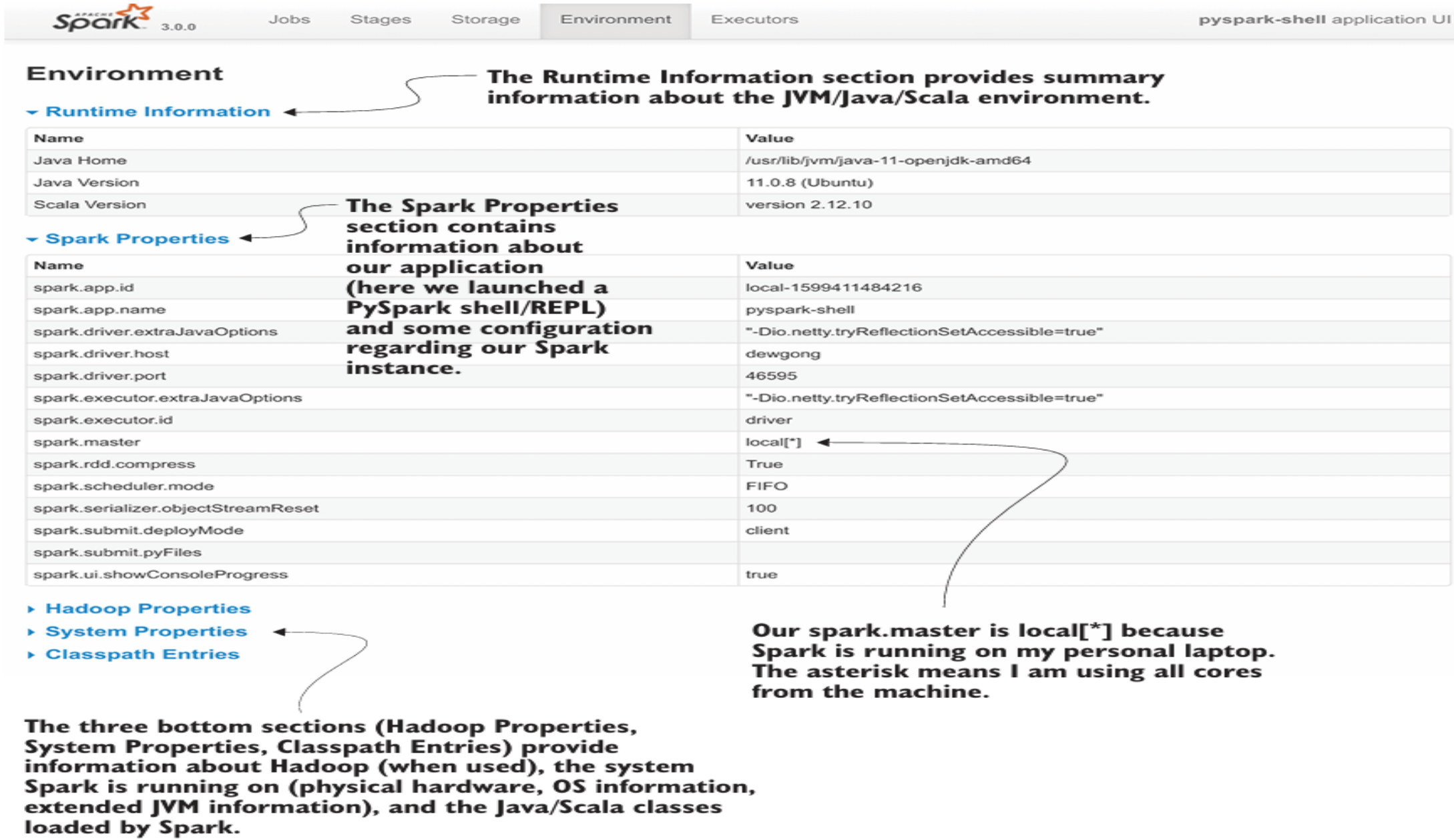


Figure 11.2 The Environment tab contains information about the hardware, OS, and libraries/software versions Spark is sitting on top of.

Executors tab

- Spark uses RAM for three main purposes, as illustrated in figure A portion of the RAM is reserved for Spark internal processing, such as user data structures, internal metadata, and safeguarding against potential out-of memory errors when dealing with large records.
- The second portion of the RAM is used for operations (operational memory). This is the RAM used during data transformation.
- The last portion of the RAM is used for the storage (storage memory) of data. RAM access is a lot faster than reading and writing data from and to disk, so Spark will try to put as much data in memory as possible. If operational memory needs grow beyond what's available, Spark will spill some of the data from RAM to disk

Jobs Stages Storage Environment Executors

Executors

▸ Show Additional Metrics

Summary

RAM Disk CPU

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks
Active(1)	0	0.0 B / 434.4 MiB	0.0 B	12	0
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0
Total(1)	0	0.0 B / 434.4 MiB	0.0 B	12	0

Executors

Show 20 entries

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores
driver	dewgong:33727	Active	0	0.0 B / 434.4 MiB	0.0 B	12

Showing 1 to 1 of 1 entries

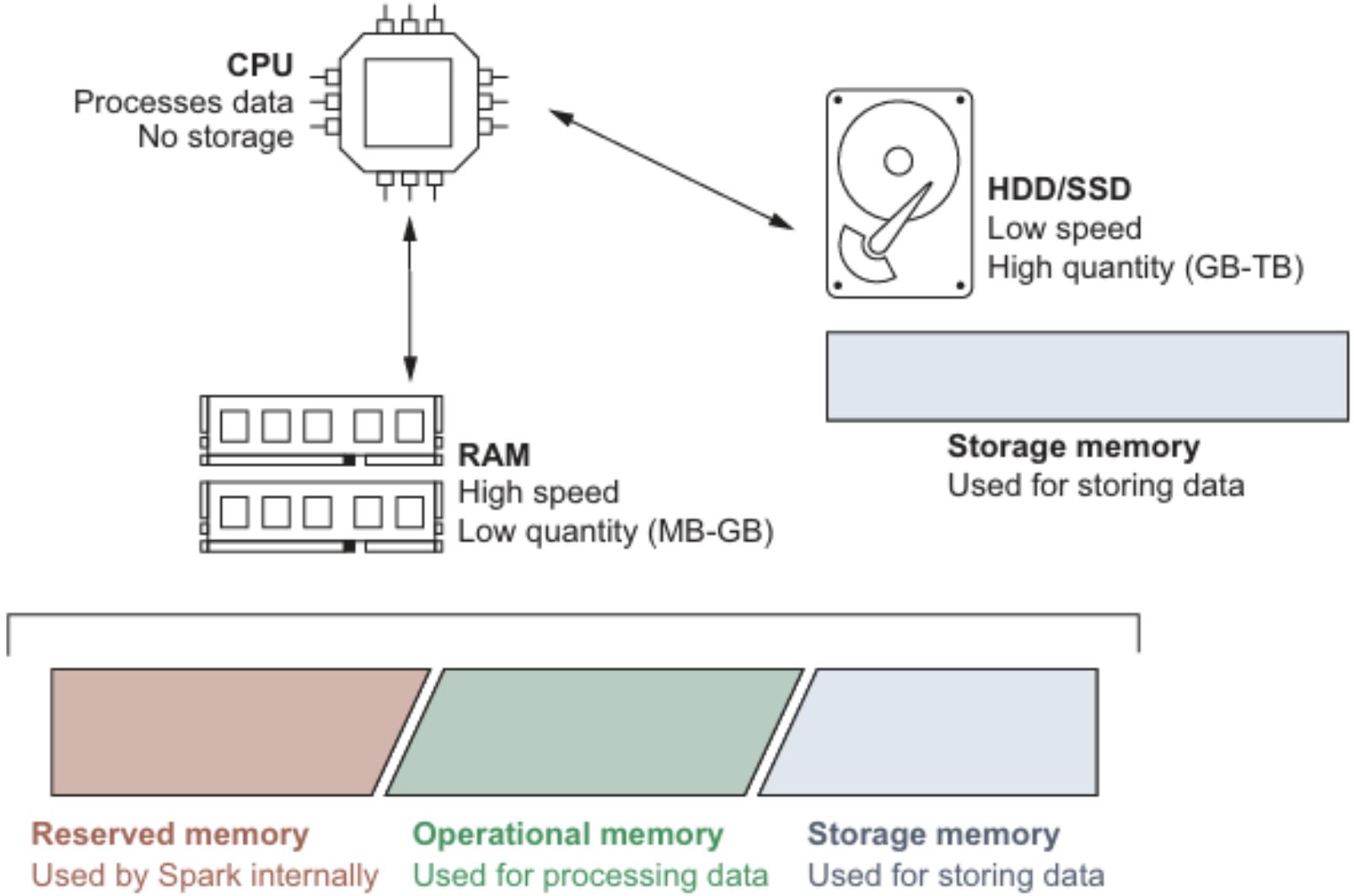


Figure 11.4 A simplified layout, or the resources Spark uses by default. Spark uses RAM as much as it can, resorting to disk when RAM is not enough (via spilling).

- Spark provides a few configuration flags to change the memory and number of CPU cores available. We have access to two identical sets of parameters to define the resources our drivers and executors will have access to.
- When creating the SparkSession, you can set the master() method to connect to a specific cluster manager1 (in cluster mode) when working locally and specify the resources/number of cores to allocate from your computer.

- I decide to go from 12 cores to only 8 by passing master("local[8]") in my SparkSession builder object. When working locally, our (single) machine will host the driver and perform the work on the data. In the case of a Spark cluster, you'll have a driver node coordinating the work, a cluster manager, and a series of worker nodes hosting executors performing the work

Memory allocation is done through configuration flags; the most important when working locally is `spark.driver.memory`. This flag takes size as an attribute and is set via the `config()` method of the `SparkSession` builder object. The different abbreviations are listed in table 11.1: Spark will not accept decimal numbers, so you need to pass integer values.

Table 11.1 The different value types Spark will accept for size. You can change the `1` to another integer value.

Abbreviation	Definition
`1b`	1 byte
`1k` or `1kb`	1 kibibyte = 1,024 bytes
`1m` or `1mb`	1 mebibyte = 1,024 kibibytes
`1g` or `1gb`	1 gibibyte = 1,024 mebibytes
`1t` or `1tb`	1 tebibyte = 1,024 gibibytes
`1p` or `1pb`	1 pebibyte = 1,024 tebibytes

WARNING Spark uses power-of-two numbers (there are 1,024 bytes in a kibibyte, whereas there are only 1,000 bytes in a kilobytes), whereas RAM memory is usually shown in power-of-ten units.

Listing 11.2 Relaunching PySpark to change the number of cores/RAM available

```
from pyspark.sql import SparkSession

spark = (
    SparkSession.builder.appName("Launching PySpark with custom options")
    .master("local[8]")
    .config("spark.driver.memory", "16g")
).getOrCreate()

# [... Run the program here ...]
```

local[8] means that we use only eight cores for the master.

The driver will use 16 g instead of the default of 1 g.

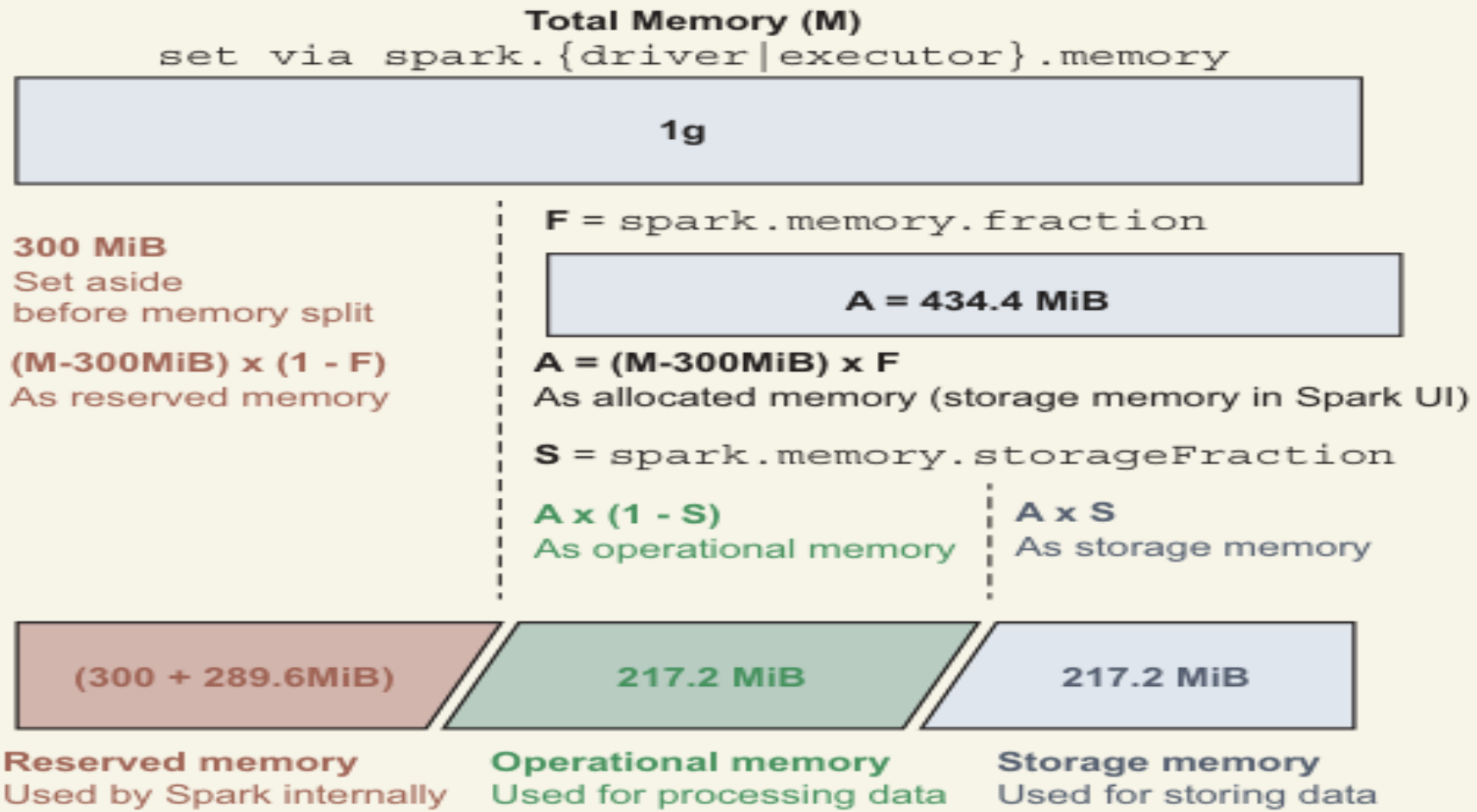
Math time! How to go from 1 GiB to 434.4 MiB

As mentioned earlier in the chapter, Spark allocates 1 GiB of memory by default to the driver program. What's the deal with 434.4 MiB? How do we go from 1 GiB of allocated memory to 434.4 MiB of usable memory?

In figure 11.4, I explained that Spark segments the memory on a node into three sections: the reserved, operational, and storage memory. The 434.4 MiB represents both the operational and storage memory. A few configuration flags are responsible for the exact memory split:

- `spark.{driver|executor}.memory` determines the total memory envelope available to the Spark driver or an executor, which I'll call `M` (by default `1g`). You can have different memory requirements for your driver versus your executor, but I usually see both values being the same.
- `spark.memory.fraction`, which I'll call `F`, sets the fraction of memory available to Spark (operational plus storage; by default `0.6`).
- `spark.memory.storageFraction`, which I'll call `S`, is the fraction of the memory available to Spark (`M` × `F`) and will be used predominantly for storage (by default `0.5`).

In the case of 1 GiB of RAM being provided, Spark will start by putting 300 MiB aside. The rest will be split between reserved and allocated (operational plus storage) using the `spark.memory.fraction` value: `(1 GiB - 300 MiB) * 0.6 = 434.4MiB`. This is the value shown in the Spark UI. Internally, Spark will manage the operational and storage memory using the `spark.memory.storageFraction` ratio. In our case, since the ratio is at `0.5`, the memory will be split evenly between operational and storage.



- In practice, the storage can outgrow its allotted place: the spark.memory.storage Fraction defines the zone where Spark will protect the data from being spilled to disk (e.g., during memory-intensive computations), but if we have more data than what fits into the storage memory, Spark will borrow from the operational memory section.
- For the vast majority of your programs, it is not recommended to play with those values. While it may seem counterintuitive to use too much memory for storing data, remember that reading data from RAM makes Spark much faster than when it needs to rely on the hard drive

Listing 11.3 The chain of transformation applied to our text files

```
from pyspark.sql import SparkSession
import pyspark.sql.functions as F

spark = (
    SparkSession.builder.appName(
        "Counting word occurences from a book, one more time."
    )
    .master("local[4]")
    .config("spark.driver.memory", "8g")
    .getOrCreate()
)

results = (
    spark.read.text("./data/gutenberg_books/*.txt")
    .select(F.split(F.col("value"), " ").alias("line"))
    .select(F.explode(F.col("line")).alias("word"))
    .select(F.lower(F.col("word")).alias("word"))
    .select(F.regexp_extract(F.col("word"), "[a-z']+", 0).alias("word"))
    .where(F.col("word") != "")
    .groupby(F.col("word"))



        .count()
    )

    results.orderBy(F.col("count").desc()).show(10)
```

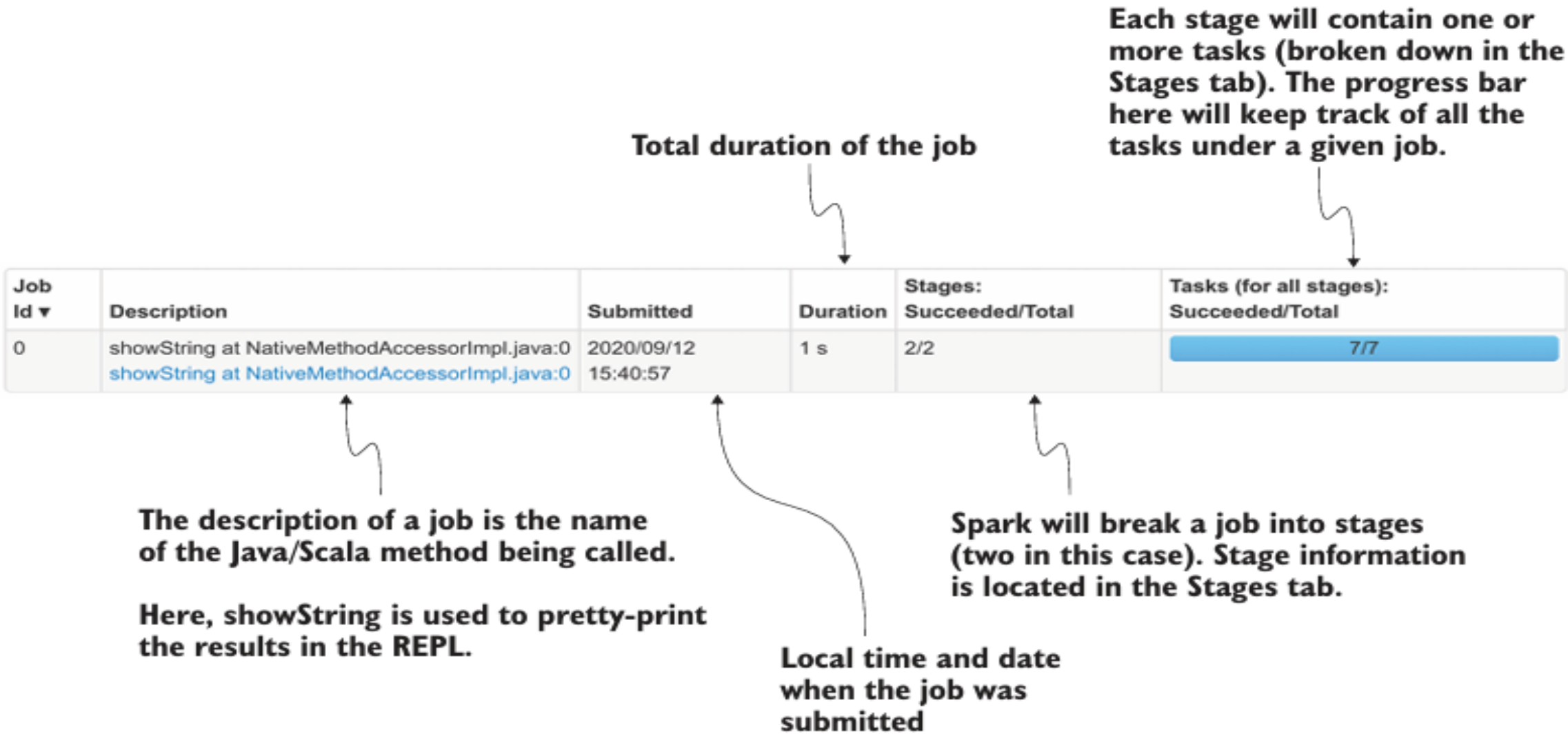


Figure 11.6 The Completed Jobs table on the Jobs tab of the Spark UI with our one completed job. Our word count, with a single action, is listed as one job.

- Each job is internally broken down into stages, which are units of work being per formed on the data. What goes into a stage depends on how the query optimizer decides to split the work. Our simple program has three steps:
- Stage 0 reads the data from all (six) text files present in the directory and per forms all the transformations (split, explode, lowercase, extract the regular expression, filter). It then groups by and counts the word frequency for each partition independently.
- 2 Spark then exchanges (or shuffles) the data across each node to prepare for the next stage. Because the data is very small once grouped by (and we only need 10 records to show), all the data gets back to one node in a single partition.
- Finally, during stage 1, we compute the total word count for the 10 selected records and display the records in table form.

- In the Completed Stages table (we've now moved to the Stages tab of the Spark UI), Spark provides four main metrics related to the memory consumption:
- Input is the amount of data read from source. Our program reads 4.1 MiB of data. This seems like an inevitable cost: we need to read the data in order to perform work. If you have control over the format and organization of your input data, you can achieve a significant performance boost.
- Output is the counterpoint to input: it represents the data our program outputs as the result of the action. Since we print to terminal, we have no value at the end of stage 1.

- Shuffle read and shuffle write are part of the shuffling (or exchange; see fig ure 11.7) operation. Shuffling rearranges the memory on a Spark worker to prepare for the next stage. In our case, we needed to write 965.6 KiB at the end of stage 0 to prepare for stage 1. In stage 1, we only read 4.8 KiB since we asked for just 10 records. Since Spark lazily optimized the whole job, it knows right from the start that we need only the count for 10 words; at exchange time (between stage 0 and 1), the driver only kept the relevant 5 words for each file, dropping the required data to answer our action by 99.5% (from 965.6 to 4.8 KiB). When working with massive files and show()-ing the content (by default 20 records), this results in significant speed increases

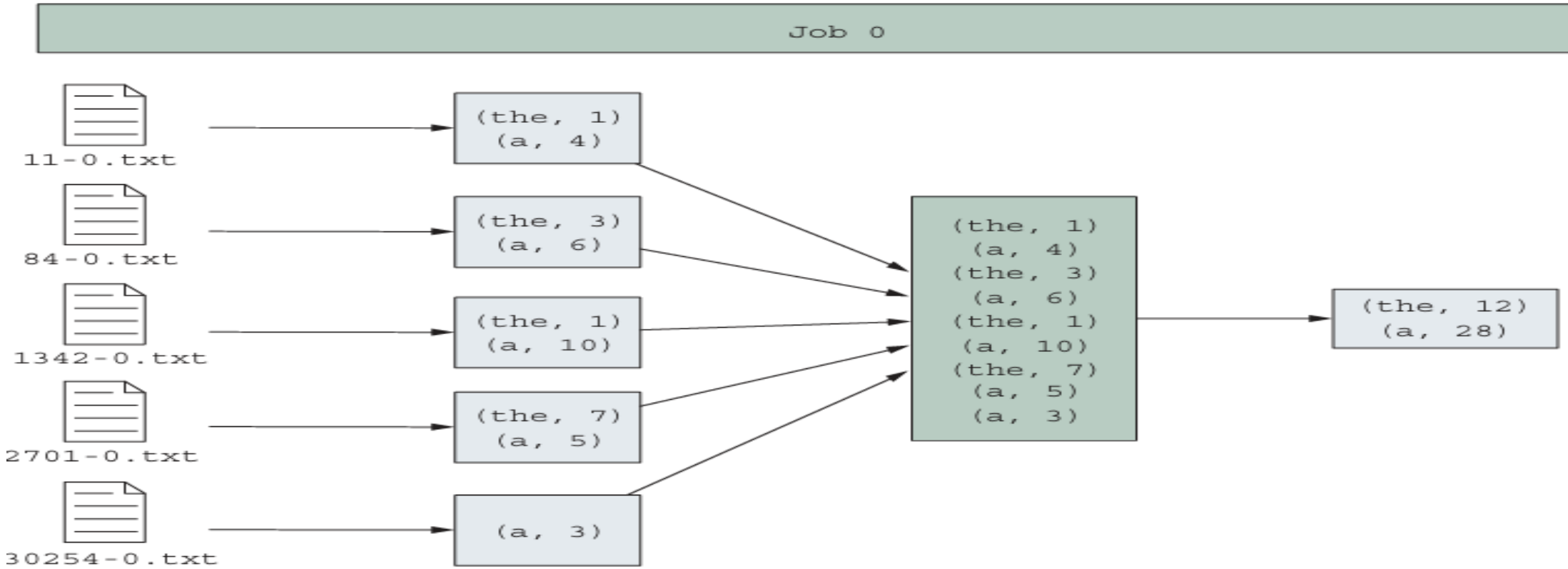


Figure 11.7 The two stages of job 0, with the summary statistics for each stage. The size of the data ingested is listed in *input*, the data outputted as *output*, and we see the intermediate data movement as *shuffle*. We can use these values to infer how large of a data set are we processing.

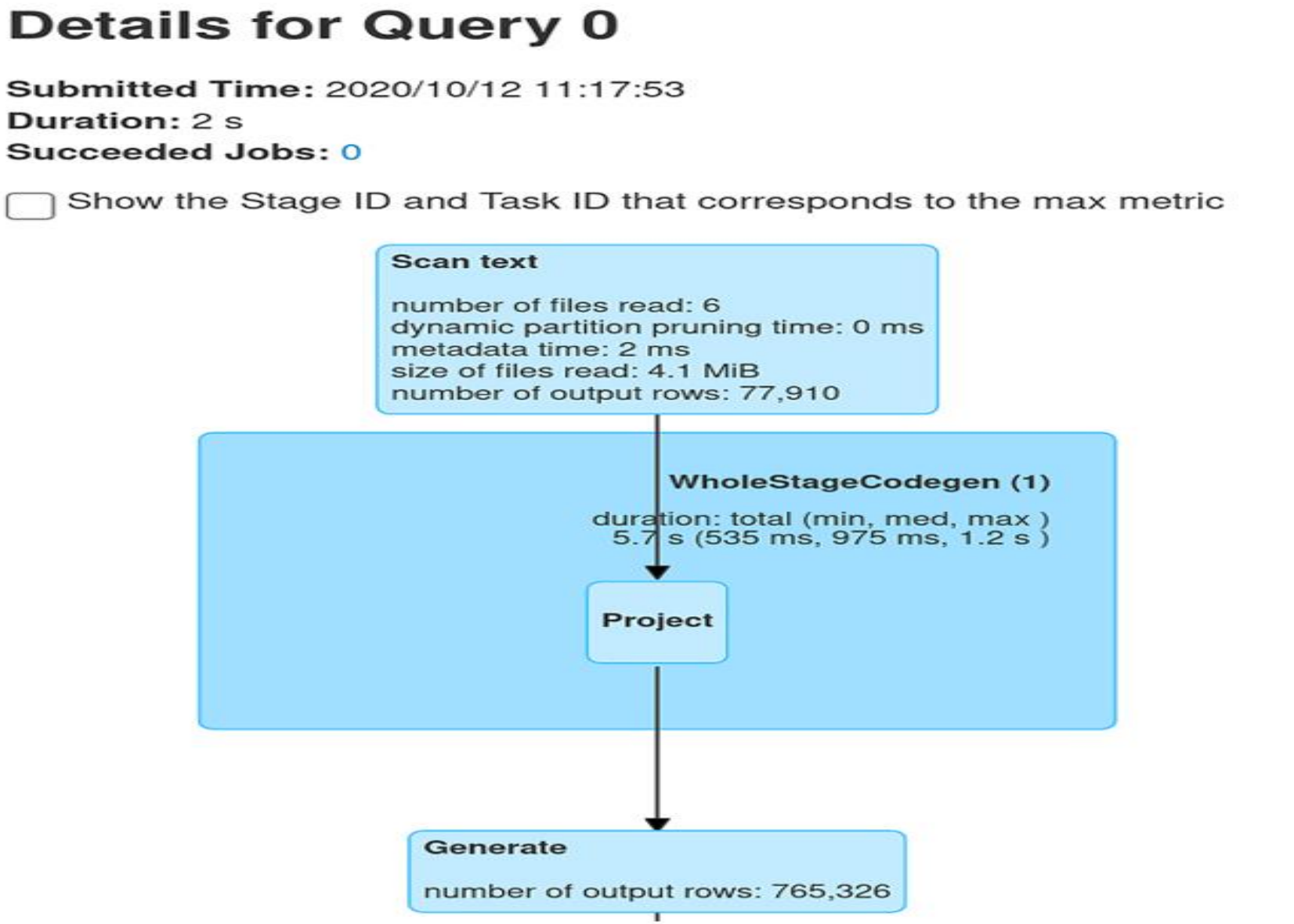


Figure 11.8 The chain of transformations on our data frame, encoded and optimized by Spark. Our code instructions on the `results` data frame are represented in stages that are illustrated and described when you hover over each box.

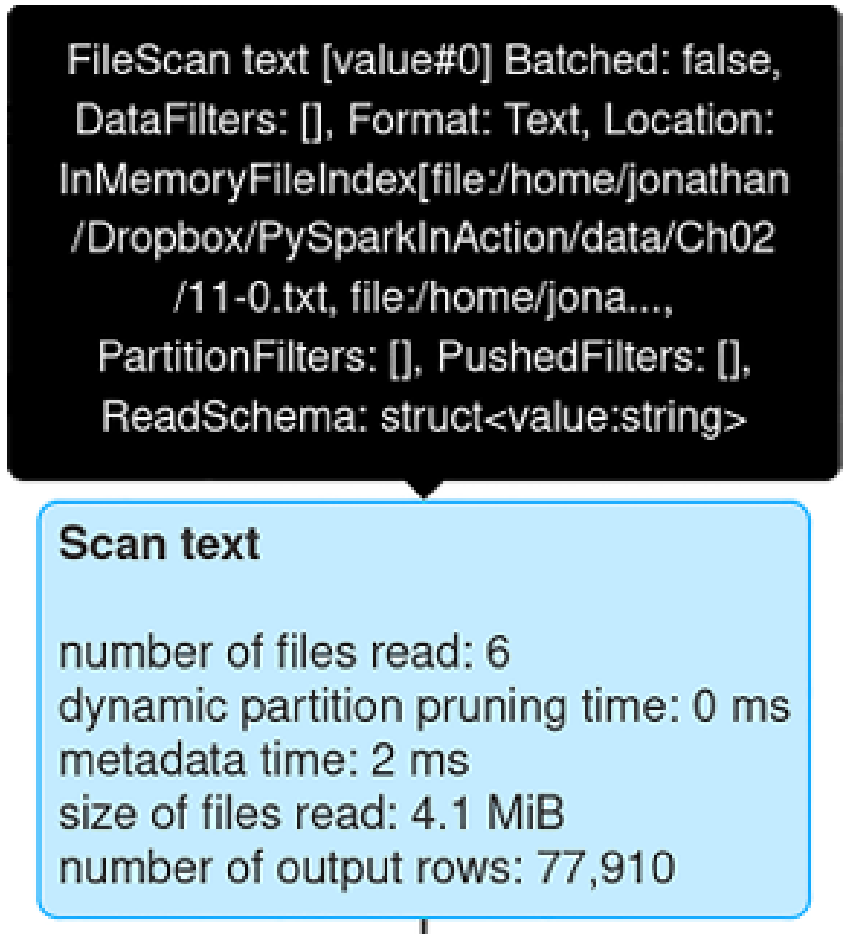


Figure 11.9 When hovering over one of the `Scan Text`, `Project`, or `Generate` boxes, a black overlay appears containing a textual representation of the transformation undertaken (called a *plan*) during that step. Most of the time we only see the beginning and end of the plan because of space constraints.

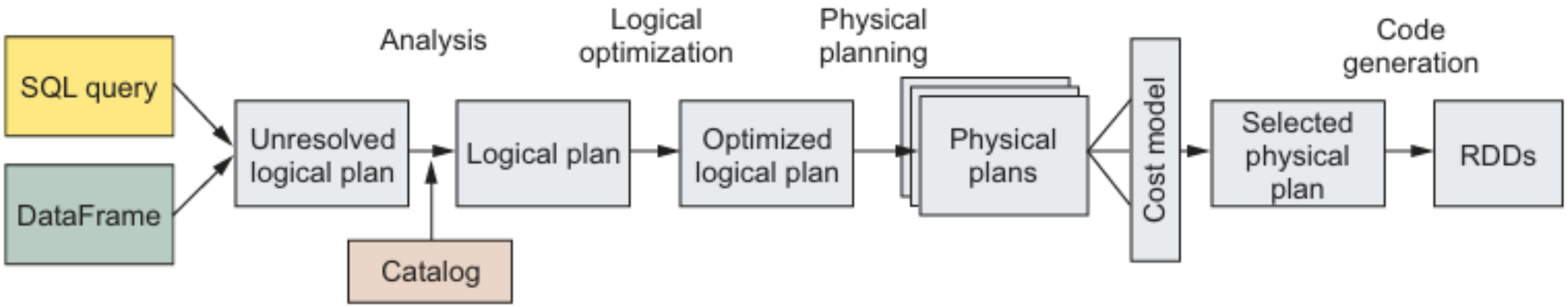


Figure 11.10 Spark optimizes jobs using a multitiered approach: unresolved logical plan, logical plan, optimized logical plan, and physical plan. The (selected) physical plan is the one applied to the data.

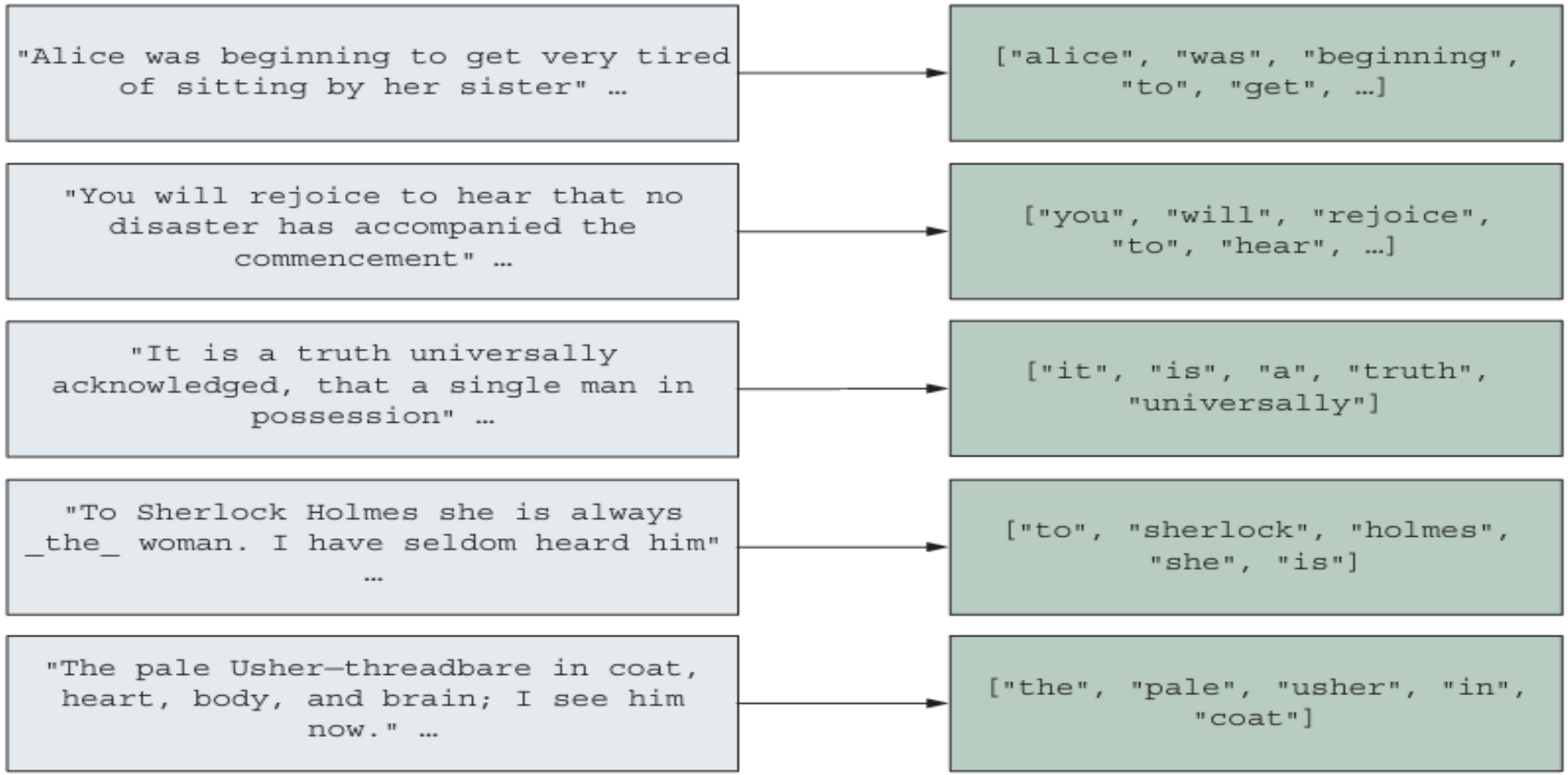


Figure 11.11 A narrow transformation will apply without requiring the records to move across nodes. Spark can parallelize the operations across each nodes.

- Narrow transformations are very convenient when working in a distributed setting: they do not require any exchanging (or shuffling) of the records. Because of this, Spark will often club many sequential narrow transformations together into a single step.
- Since the data is sitting in RAM (or on the hard drive), reading the data only once (by performing multiple operations on each record) usually yields better perfor mance than reading the same data multiple times and performing one operation each time.
- PySpark (versions 2.0 and above) can also leverage specialized CPU (and since Spark 3.0, GPU) instructions that speed up data transformation

Wide Transformation

- Unlike their narrow counterpart, wide transformations need the data to be laid in a certain way between the multiple nodes. For this, Spark uses an exchange step to move the data for the operation to complete.

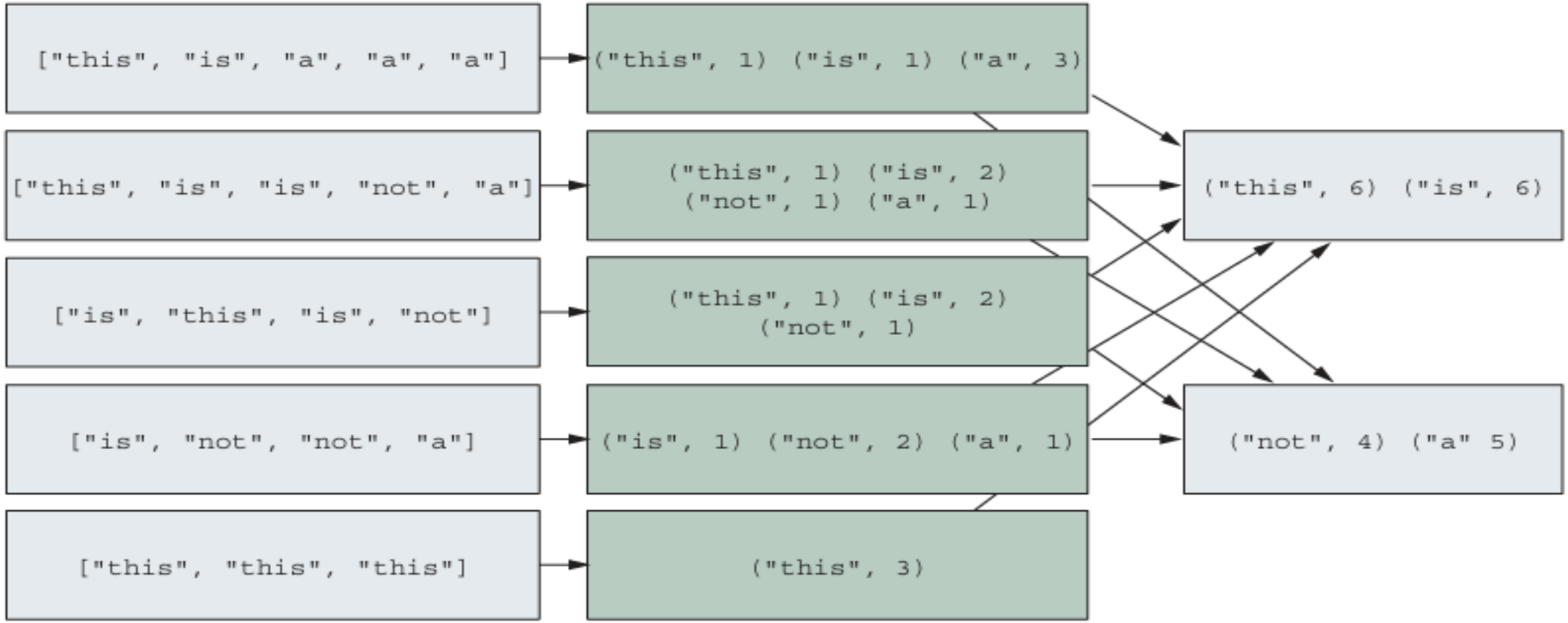


Figure 11.12 A wide transformation can happen in two stages because Spark needs to exchange data across nodes. Spark calls those necessary exchange operations *shuffles*.

```
# Your results will vary.

%timeit results.show(5, False)
920 ms ± 46.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

%timeit results_bit.show(5, False)
427 ms ± 4.14 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Caching a data frame: Powerful, but often deadly

- For instance, if we show() the results data frame of our word count example five times, Spark will read the data from source and transform the data frame five times. While this seems like a highly inefficient way of working, bear in mind that data pipelines most often “flow” the data from one transformation to the next (hence the pipeline analogy); keeping intermediate states is useless and wasteful.
- . A cached data frame will be serialized to the storage mem ory, which means that retrieving it will be speedy. The trade-off is that you take up RAM space on your cluster. In the case of very large data frames, that means that some data might spill to disk (leading to a slower retrieval) and that your cluster might run slower if you are using memory-heavy processing

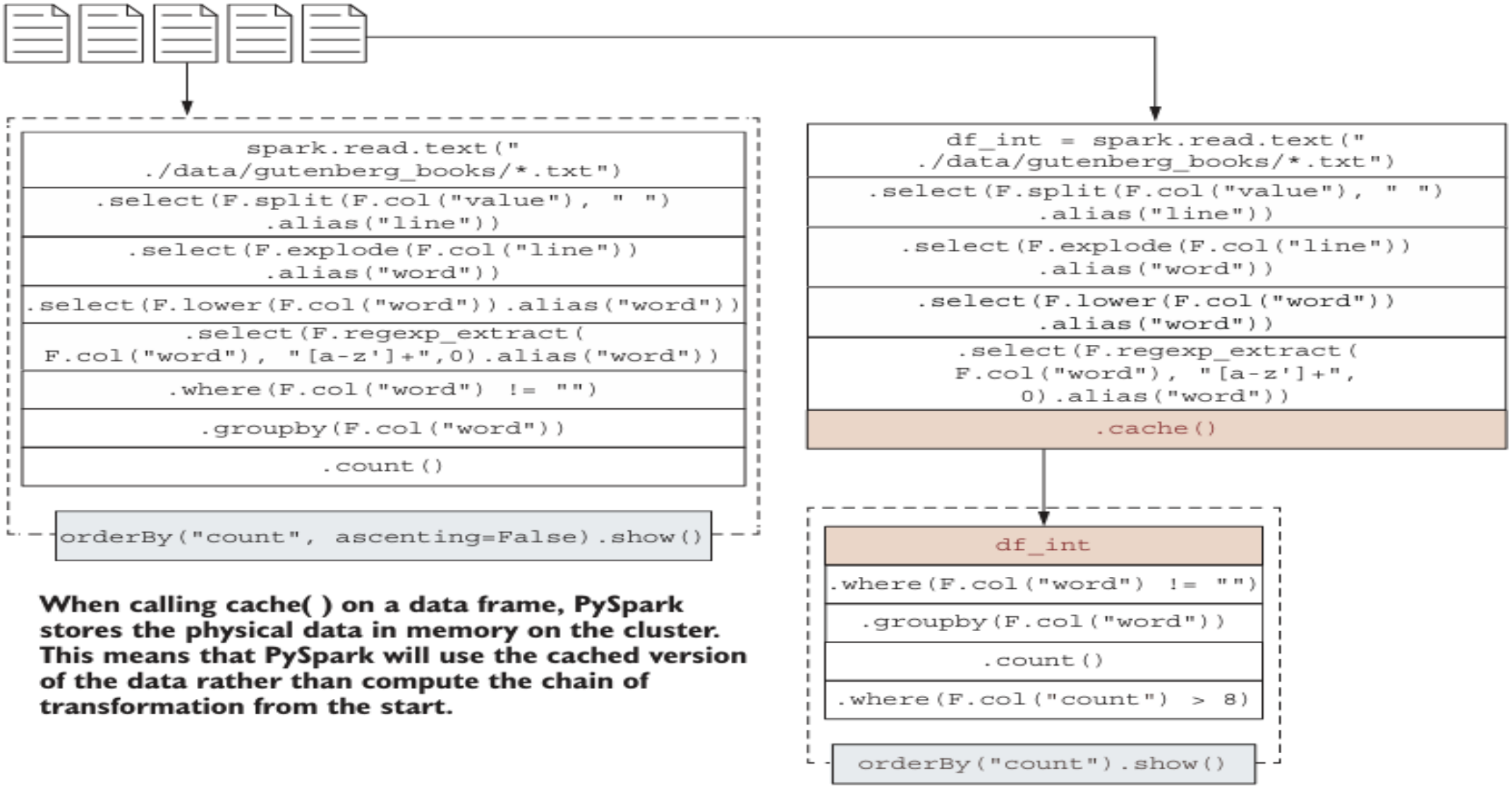


Figure 11.13 Caching a data frame in our word count program. In this case, the second action does not compute the chain of transformations from the `spark.read` operation and leverages the cached `df_int` data frame instead.

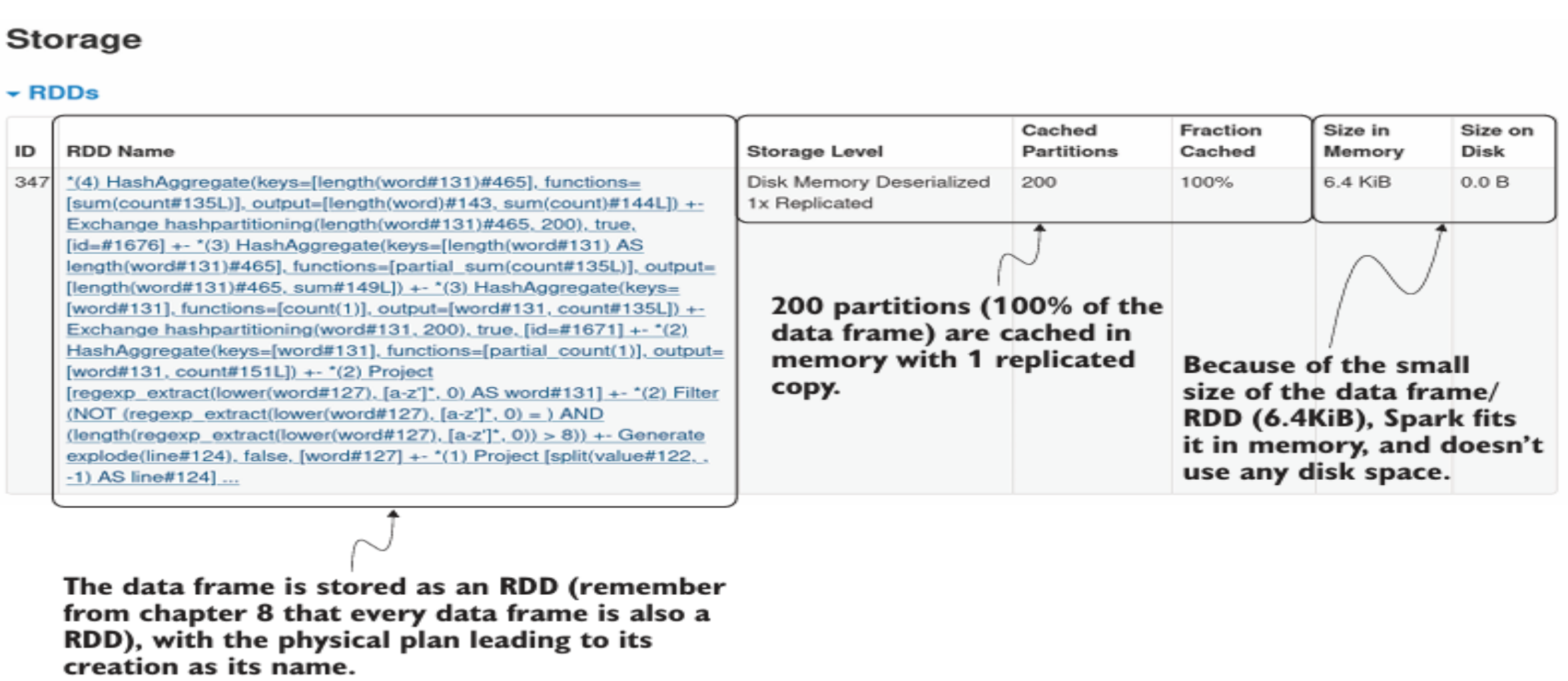


Figure 11.14 The `results` data frame, successfully cached. Because of the size, everything fits into the RAM.

In the Executors tab, you can also check how much memory storage is being used. An uncached data frame will take a little space (because each executor keeps the instructions to recompute a data frame on the fly), but nowhere as much as caching the data frame.

Persisting: Caching, but with more control

By default, a data frame will be cached using the `MEMORY_AND_DISK` policy, which means that the storage RAM will be used as a priority, falling back to disk if we run out of memory. An RDD will use the `MEMORY_ONLY` policy, which means that it won't use the disk at all for storage. If we don't have enough storage RAM, Spark will recompute the RDD from scratch (negating the effects of caching).

If you want more control over how your data is cached, you can use the `persist()` method, passing the level (as a string) as a parameter. Beyond `MEMORY_ONLY` and `MEMORY_AND_DISK`, you can also opt for `DISK_ONLY`, which foregoes RAM to go straight to disk. You can also add a `_2` suffix (e.g., `MEMORY_ONLY_2`), which will use the same heuristic but duplicate each partition over two nodes.

If you can afford the RAM, I suggest using it as much as possible. RAM access is orders of magnitude faster than disk. The actual decision will depend on your Spark instance configuration.

Caching looks like a very useful functionality

- it provides an insurance policy, so you don’t have to recompute a data frame from scratch if you want to backtrack.
- In practice, this scenario happens very rarely
- Caching takes computing and memory resources that are not available for gen eral processing.
- Computing a data frame can sometimes be faster than retrieving it from cache.
- In a non interactive program, you seldom need to reuse a data frame more than a few times: caching brings no value if you don’t reuse the exact data frame more than once.

- When writing a PySpark program, make it work first, then make it clean, and then make it fast, using the Spark UI to help you along the way.
- Data pipelines, whether they are for ETL or ML, gain a lot from being easy to reason about. Shedding a few minutes of a program is not worth it if it takes you a day to decipher what the program does

Summary

- Spark uses RAM (or memory) to store data (storage memory), as well as for processing (operational memory) data. Providing enough memory is paramount in the fast processing of Spark jobs and can be configured in the `SparkSession` initialization.
- The Spark UI provides useful information about cluster configuration. This includes memory, CPU, libraries, and OS information.
- A Spark job consists of a series of transformations and one action. Job progress is available in the Job tab of the Spark UI when processing.
- A job is split into stages, which are the logical units of work on a cluster. Stages are split by exchange operations, which are when the data moves around worker nodes. We can look at the stages and steps via the SQL tab in the Spark UI and via the `explain()` method over the resulting data frame.
- A stage consists of narrow operations that are optimized as a unit. Wide operations may require a shuffle/exchange if the necessary data is not local to a node.
- Caching moves data from the source to the storage memory (with an option to spill to disk if there isn't enough memory available). Caching interferes with Spark's ability to optimize and is usually not needed in a pipeline-like program. It is appropriate when reusing a data frame multiple times, such as during ML training.