**YARN** – Yet another Resource Manager

Spark Overview

**Spark** Data Processing Engine for Large amounts of data. Spark Shell for Python

**Spark SQL –** For structured data

Dataframes, Datasets

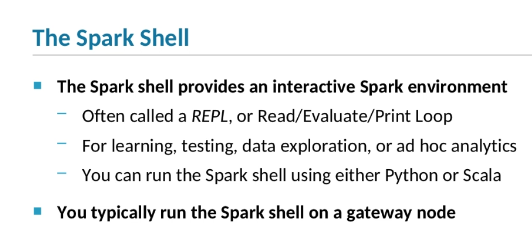
**Catalyst Optimizer** – Optimizes querys

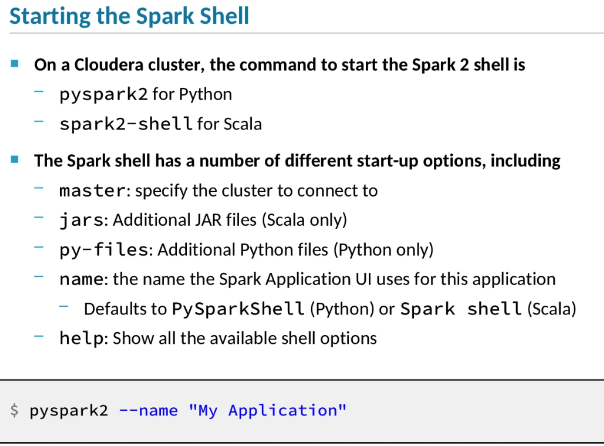
**Spark Streaming** - Streaming

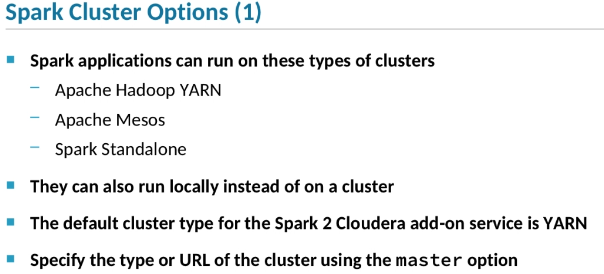
**Mlib –** Machine Learning

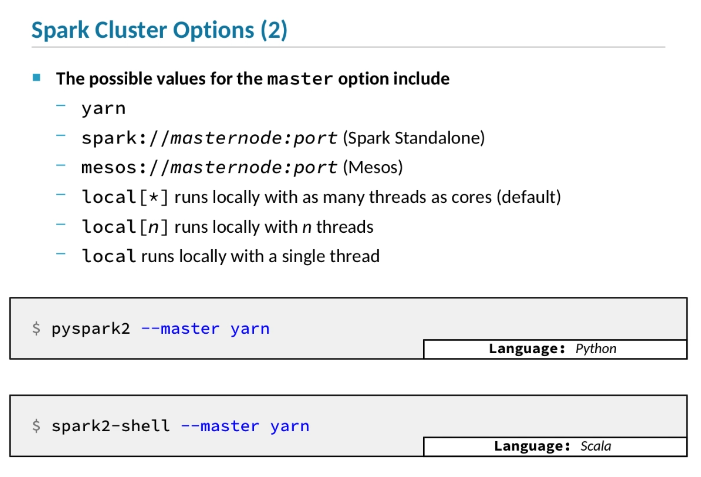
**GraphX** -Graphs

1. The Spark shell is an environment for running Spark code interactively.
2. The shell is a REPL—a read/evaluate/print loop—where you can type Spark code and get immediate results back.
3. Using the Spark shell is often an easier and more convenient way to use Spark,
4. instead of having to write a program, compile it, run it on the cluster, test, recompile, rerun, and so on.
5. It's especially useful for learning, testing, or doing ad hoc data exploration.
6. There are two versions of the Spark shell, one for Python and one for Scala.
7. Because the shell is an interactive tool, you would typically run it on a gateway node that’s part of your Hadoop cluster.
8. In order to start the Spark shell for Spark 2, use the command pyspark2 for the Python shell or spark2-shell for the Scala shell.
9. **The reason you need to include 2 in the commands is because Cloudera Manager allows you to install Spark 1 and Spark 2 on the same cluster,**
10. and with these commands you can specify which version you want to run.
11. Both the Python and the Scala Spark shell commands have a number of different options you can use at startup.
12. For example, the master option specifies which cluster to run Spark on.
13. If your code will be using additional non-Spark libraries,
14. you can make those available to the shell using the jars option for the Scala shell or the py-files option for the Python shell.
15. The name option sets a name for the Spark shell application when it runs on the cluster.
16. The YARN Resource Manager UI and the Spark Application UI both display this application name to make it easy to distinguish one running application from another.
17. The default name for the Python shell is PySparkShell; for the Scala shell it’s simply Spark shell.
18. The code example here shows how you’d start the Python shell, setting the application name to My Application.
19. These are just a few of the available options.
20. You can use the help option to get a list of all available options.
21. One of the key configuration options for any Spark application, including the Spark shell, is the cluster the application will run on.
22. Spark supports three different cluster platforms.
23. The one we will focus on here is YARN, which is part of Apache Hadoop.
24. Another supported cluster architecture is Apache Mesos.
25. Mesos is primarily intended to run non-Hadoop workloads, whereas YARN is geared mostly toward Hadoop.
26. That’s why Cloudera supports YARN, and why YARN is the focus of this course.
27. A third cluster option is Spark Standalone.
28. Spark Standalone is a basic, easy-to-install cluster framework included with Spark out of the box.
29. Note, however, that Spark Standalone is deprecated and is no longer supported by Cloudera.
30. Spark Standalone may be helpful for limited uses, like learning or testing,
31. but it lacks many of the features most organizations need for production clusters, such as support for security, and has limited configurability and scalability.
32. There’s also a fourth option, which is to not run on any cluster, but instead to run locally, right on the machine that the shell process is running on.
33. If you don’t specify where an application should run, it will use the default configured for your installation of Spark.
34. In the case of a cluster that’s been configured by Cloudera Manger, that default is YARN.
35. If you want to specify a cluster rather than use the default, use the master option when starting the shell or running the application.
36. The possible values for the master option correspond to the four available cluster modes.
37. To run on a YARN cluster, simply set the masteroption to yarn.
38. This will start the Spark shell or application on the YARN cluster that the gateway node you’re on is part of.
39. The example commands on this slide demonstrate how you’d run the Scala or Python Spark shell on YARN.
40. To run on Spark Standalone or Mesos, specify the URI of the master daemon for the cluster to run on.
41. If you want to run locally instead of on a cluster, set master to local with the number of threads you want to run in brackets.
42. When running on a cluster, Spark applications are distributed across various executors so that the operations run in parallel.
43. If you are developing and testing an application locally,
44. it’s helpful to run the application across multiple threads so that the operations can run in parallel, similar to how they would on a cluster.
45. Multiple threads can also improve your application's performance.
46. If you use an asterisk for the number of threads, Spark will run as many threads are there are CPU cores on the local machine, which generally provides the best performance.
47. Alternatively, you can use a specific number to start that many threads.
48. Or you can use local with no number specified to run in a single thread

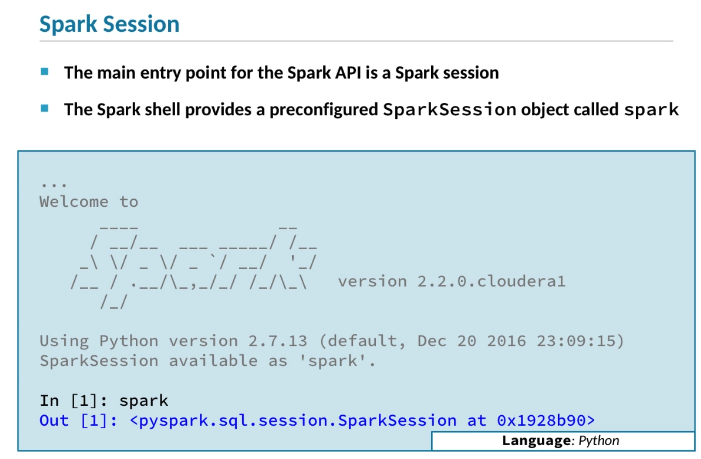


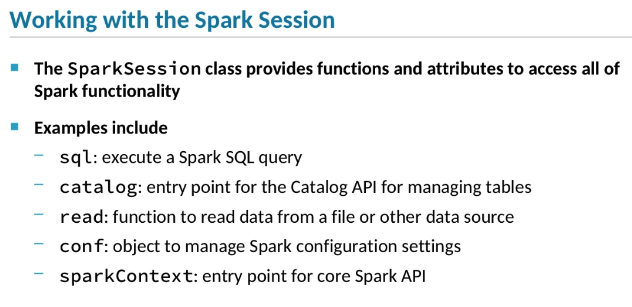


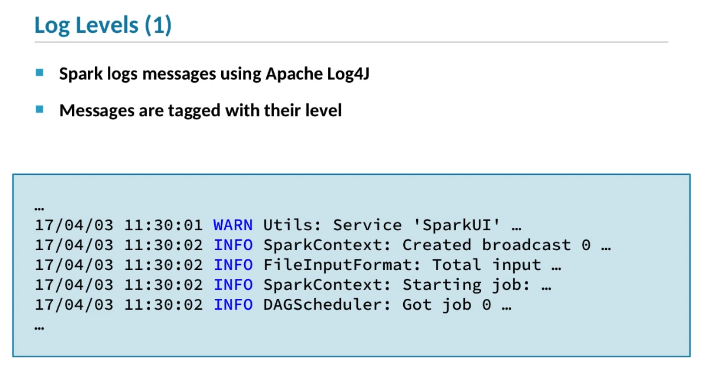




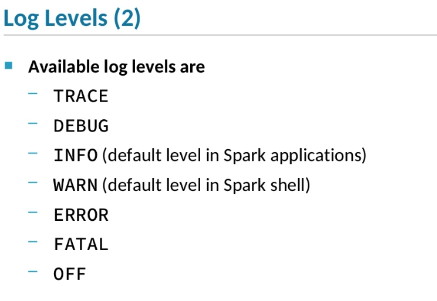
Spark Shell

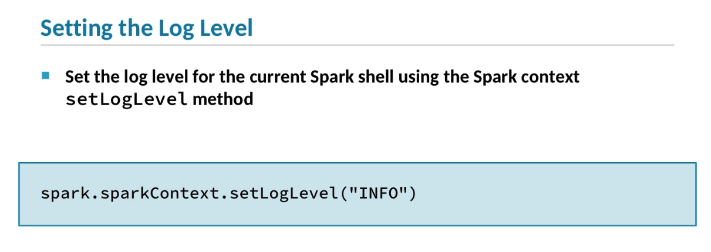






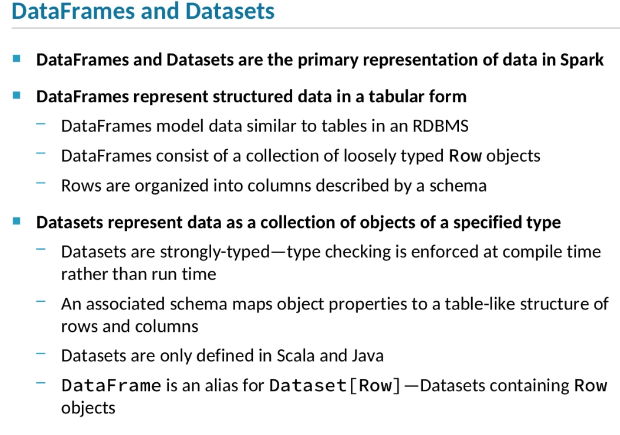
Log4j Log Levels

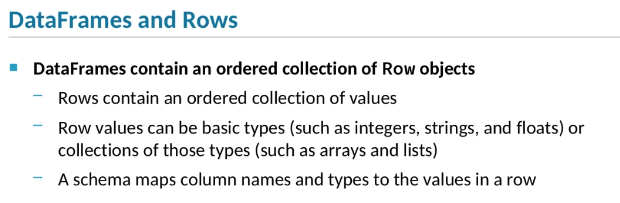


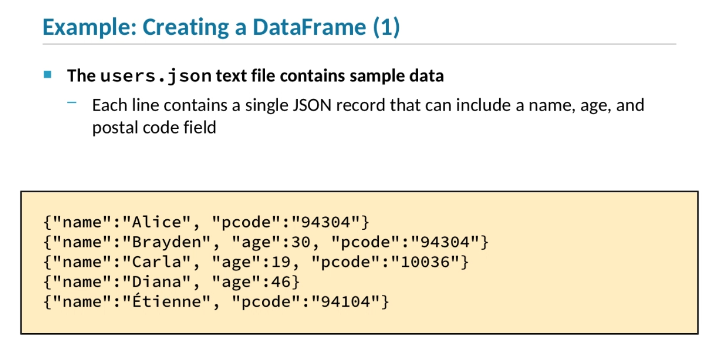


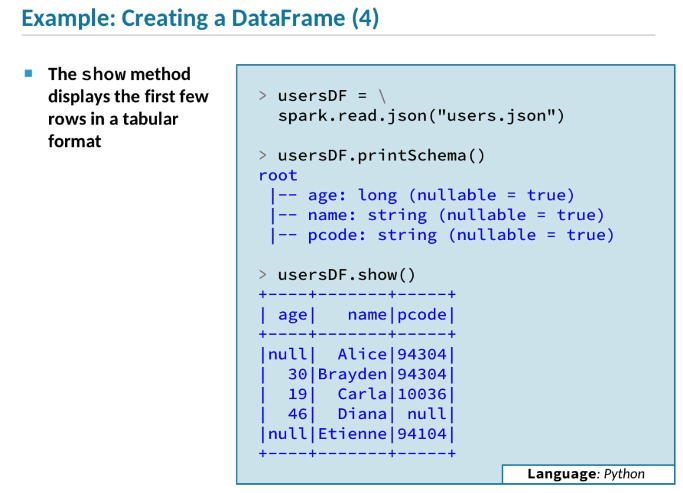
1. All Spark applications, including the Spark shell,
2. need to start with a SparkSession object.
3. SparkSession is the main entry point for the Spark API.
4. **One convenient feature of the Spark shell is that it will create a SparkSession object for you automatically.**
5. As you can see in the screenshot of the Spark shell startup messages,
6. the object is assigned to a variable named spark,
7. which is the starting point for your Spark code.
8. The SparkSession class has a number of functions and attributes
9. that provide access to the Spark API's functionality.
10. For example, you can use the sql function to execute a SQL query in Spark.
11. The catalog attribute points to a Catalog object,
12. which provides access to the Catalog API for managing tables.
13. To load data from a data source such as a file,
14. use the read function.
15. To manage Spark application configuration settings,
16. use the conf attribute.
17. SparkSession is the primary entry point specifically for Spark SQL
18. and the DataFrame and Dataset API.
19. To access core Spark functionality,
20. use the sparkContext attribute of SparkSession to return a SparkContext object.
21. Spark uses Apache Log4J for logging,
22. which you are probably familiar with if you have developed applications in Java or Scala.
23. Log4J
24. provides applications the ability to produce log messages at different levels.
25. This example shows some log messages at the INFOand WARN levels.
26. Each log output message is tagged with one of the levels shown here.
27. Messages at the ERROR level
28. are typically produced when Spark detects some sort of error condition.
29. WARN messages are generated when some unexpected situation occurs
30. but Spark will attempt to continue executing the operation.
31. And so on.
32. When you set a log level for the Spark context,
33. it will only generate messages at the specified level
34. and those at a higher level of severity.
35. For example, if you set the level to ERROR,
36. Spark will generate ERROR and FATAL messages,
37. but not WARN, INFO, DEBUG, or TRACE messages.
38. If you want more detailed logging,
39. you could set the level to INFO,
40. in which case you’d get INFO, WARN, ERROR, and FATAL messages,
41. but not the less severe types of messages.
42. Many developers run applications with the level set to WARN,
43. which is usually a good balance between providing the information you need
44. while not generating so much output that it’s hard to filter out what’s important.
45. Sometimes, though,
46. you might find you need more detailed logging to help you understand what Spark is doing
47. or to troubleshoot a problem.
48. The default level for log messages in the Spark shell is WARN.
49. You can use the Spark context setLogLevel function
50. to change the level Spark will produce,
51. as shown in this code snippet.
52. You could also use the setLogLevel function in an application,
53. but that is rarely a good idea,
54. because it hardcodes the setting.

Data Frames and Datasets



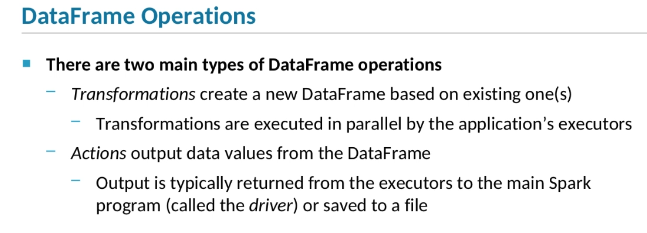


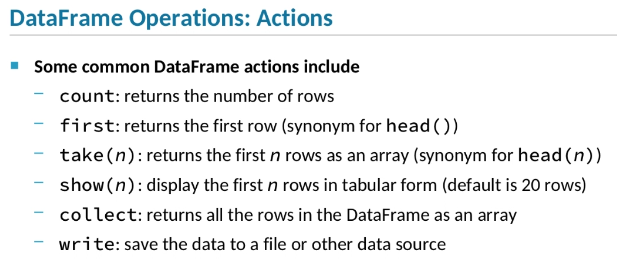




1. Let’s take a look at the Datasets and DataFrames API, which is part of the Spark SQL library.
2. DataFrames and Datasets are the primary abstractions for representing structured data in Spark.
3. Both consist of a collection of structured objects that together make up a set of data.
4. DataFrames represent tabular data.
5. That is, they organize data into rows and columns, much like a table in a relational database system does.
6. DataFrames are made up of Row objects, which can contain any number of values of various types.
7. DataFrames have schemas that map the row values to named columns of specific types.
8. Datasets are similar to DataFrames.
9. They also consist of a set of objects representing the underlying data.
10. The difference is that Datasets are strongly-typed.
11. Instead of Row objects, Datasets contain objects with properties whose types are known at compile time.
12. Because of this, Datasets enforce type consistency in a way that DataFrames can’t.
13. Datasets are only defined in Scala and Java.
14. Python doesn’t use Datasets, because Python is a loosely-typed language, so the type-checking feature of Datasets isn't applicable.
15. Other than typing, Datasets and DataFrames are quite similar—they both organize a set of structured data.
16. In fact, in Scala, DataFrame isn't actually its own class;
17. it’s simply an alias for Datasets containing Row objects.
18. DataFrames represent a set of data using an ordered collection of objects of the Row class.
19. Each Row object contains one or more values.
20. The values can be basic Scala, Java, or Python types, such as integers, strings, floating point numbers, and so on.
21. Or the values can be nested types—that is, collections of values, such as an array of strings or a list of numbers.
22. Every DataFrame has an associated schema that maps the values in the rows to column names and types.
23. Here, we’ll walk through a simple example of creating a DataFrame in the Spark shell by loading data from a JSON file called users.json.
24. The data file includes JSON records containing values for name, age, and postal code fields.
25. Note that the supported JSON format in Spark SQL is JSON Lines—that is, files in which each line in the file consists of a single JSON record, as shown here.
26. The first line in the file has a record where the namevalue is Alice and the pcode is 94304.
27. This record has no age value, which is allowed in the JSON format.
28. The second record is on its own line, and has the values Brayden, 30, and 94304.
29. And so on.
30. Now we’ll take a look at the Spark code to create a DataFrame using the users.json data file.
31. In this example, we are showing Python code in the PySpark shell, but the Scala code is nearly identical.
32. The only difference is the syntax for declaring a variable.
33. In the first line, we use the Spark session's readattribute to get a DataFrameReader object.
34. We then use DataFrameReader's json method to load the specified JSON data file.
35. This call returns a DataFrame object, which we assign to a variable called usersDF.
36. Remember that DataFrames always have an associated schema that structures the DataFrame’s rows into named columns.
37. The schema is created at the same time the DataFrame itself is.
38. The DataFrameReader object scans the data file and infers the schema of the DataFrame.
39. In the second line of our example, we call the DataFrame’s printSchema method, which displays the DataFrame's schema.
40. Here we see that the inferred schema defines three columns: age, of type long; name, which is a string; and pcode, which is also a string.
41. These column names correspond to the JSON record field names we just saw in the data file.
42. In the last line of the example, we call the DataFrame’s show method.
43. This displays the first few rows of data, nicely formatted as a table, with column headers as specified by the schema.
44. By default, the show method displays the first 20 rows,
45. but in this case, our data file only contains five records, so the DataFrame has only five rows.

Dataframe Operations





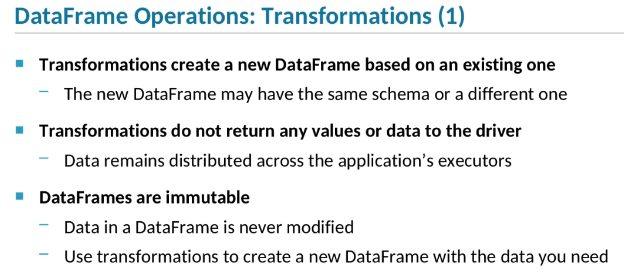
*You should be very cautious about using the collectfunction. (calls all rows)*

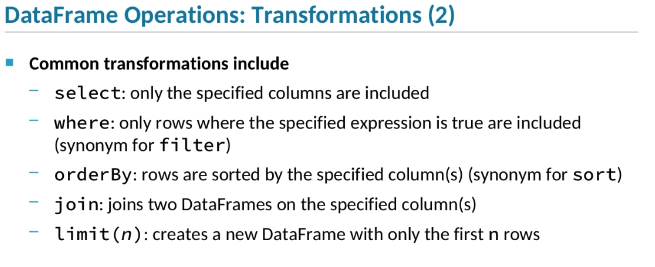
*Note there is Scala in addition to Python here*

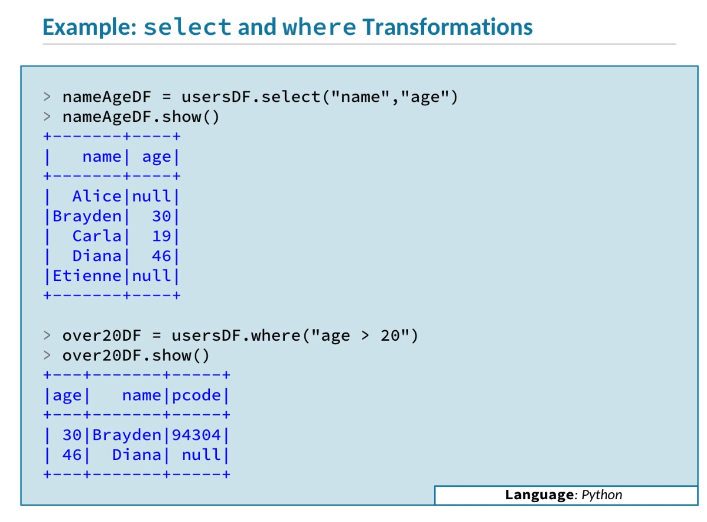


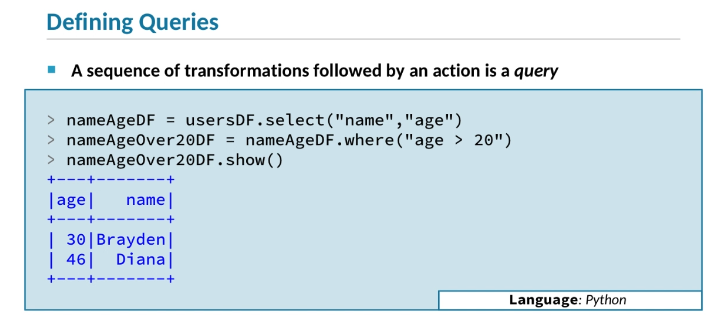
1. You process and analyze data in a DataFrame using a wide set of operations that are defined in the DataFrame API.
2. DataFrame operations can be broadly categorized as either *transformations* or *actions*.
3. Transformation operations create a new DataFrame as a result of performing some sort of transformation on the data in the original DataFrame.
4. When a Spark application is running on a cluster, transformations are executed by the application’s executors in parallel, in JVMs distributed on worker nodes in the cluster.
5. Actions are operations that generate output data from a DataFrame.
6. Output from actions is typically either saved to a file or returned to the application’s driver process.
7. When you are using the Spark shell, the driver is part of the shell itself.
8. There are many DataFrame actions defined in the API.
9. Here are a few of them.
10. The count operation returns the number of rows in the DataFrame.
11. The first operation returns a single Row object that represents the data in the first row of the DataFrame.
12. Calling the head operation with no parameters does the same thing.
13. The take operation is similar to first, except that you specify how many rows to return.
14. The return value will be an array containing the number of Row objects specified.
15. This is the same as calling the head function with a parameter specifying the number of rows.
16. The show action displays the specified number of rows; the default number is 20.
17. This is different than take, because show displays the requested data on the Spark shell’s console, rather than returning an array.
18. The collect action is similar to take or head, but returns all the rows instead of a specific number of rows.
19. You should be very cautious about using the collectfunction.
20. Spark is designed specifically to work with very large sets of data, which it manages by distributing the data across executors in a cluster.
21. The collect operation, like most actions, returns the requested data across the network from all executors handling the given DataFrame to a single process, the driver.
22. If the data is very large, the operation will use a lot of bandwidth, and the result may exceed available memory in the driver.
23. The collect operation is usually used with small sets of test data of known size.
24. All the actions we just discussed return data to the driver.
25. The write operation is different;
26. it tells Spark to save the data in the DataFrame to a file, a table, or another type of data source.
27. Here are Scala and Python code snippets that both do the same thing.
28. In the first line, we create a DataFrame called usersDF by reading a data file called users.json.
29. In the second line, we call the DataFrame’s takeaction, specifying that it should return an array of three rows.
30. We assign the returned array to a variable called usersDF.
31. Both the Scala and Python shells automatically display the type and some of the data when a new variable is defined.
32. The Python output looks a bit different from the Scala output,
33. but in both examples you can see that the new users variable is an array of Row objects containing the first three records from the users.json file.

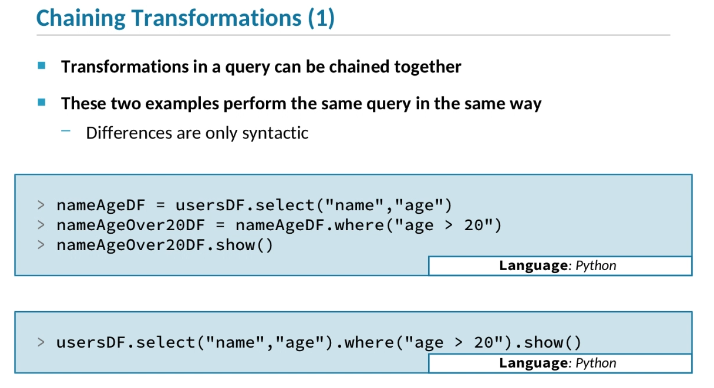
Data Frame Operations Part 2

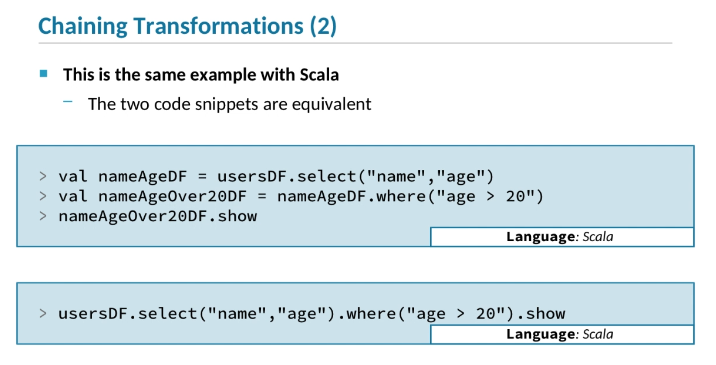












1. Transformations are operations that create a new DataFrame by transforming the data in the original DataFrame.
2. Depending on which transformation you execute, the new DataFrame may have the same schema as the first one or a different schema.
3. Unlike actions, transformations do not return any data to the Spark driver.
4. The data resulting from a transformation remains distributed across executors running on the cluster.
5. This is the key difference between actions and transformations.
6. DataFrames are immutable, meaning that the data in a DataFrame is never modified.
7. Transformations don’t modify the original DataFrame at all;
8. they generate a new DataFrame with transformed data.
9. Spark applications often contain series of transformations applied to a DataFrame
10. to create a final resulting DataFrame representing the data you need in the structure you need it.
11. There are many transformation operations available.
12. Here are just a few of them.
13. The select operation creates a new DataFrame containing only the specified columns from the base DataFrame.
14. The where operation creates a new DataFrame with only the rows from the base DataFrame that meet the specified criteria.
15. The where operation is also known as filter.
16. A DataFrame created by orderBy contains all the same data as the base DataFrame, but in a different order:
17. sorted by the values in the specified column or columns.
18. This is an alias for the sort function.
19. The join operation creates a DataFrame from two base DataFrames, joined together based on matching values in the given column or columns.
20. The limit operation results in a new DataFrame with the same structure as the original, but containing only the first n rows.
21. If you are familiar with SQL, the names and functionality of these transformations probably seem familiar.
22. DataFrame operations are often very similar to SQL commands, such as WHERE, SELECT, and JOIN.
23. Working with DataFrames should feel familiar to developers who know SQL.
24. Here’s a code example demonstrating two different transformations.
25. We start with the usersDF DataFrame, which is based on the users.json data file containing records with name, age, and pcode values.
26. The usersDF DataFrame has three columns, corresponding to the three values in the JSON records.
27. In the first line, we call usersDF.select, passing the name of two of the columns in the DataFrame: name and age.
28. We call the resulting DataFrame nameAgeDF.
29. Next, we call show on the result,
30. and we see that the new DataFrame has a different schema than the original—two columns instead of three—
31. and only the data contained in the first two named columns.
32. Next, we call the where transformation on usersDF.
33. We pass a string as the where criterion, so that only those records in which the age value is greater than 20 are included in the result.
34. We assign the result to over20DF, and call show.
35. As you can see, the resulting DataFrame has the same schema as the original, but only a subset of the data; those rows that meet the where criteria.
36. In the DataFrames and Datasets API,
37. the series of transformations on a starting DataFrame, followed by an action to return, save, or display the result, is known as a *query*.
38. The example here starts with the usersDFDataFrame,
39. calling select to create a new DataFrame called nameAgeDF, which contains only the selected columns: name and age.
40. The next line performs a second transformation to create a third DataFrame, containing only rows where age is greater than 20.
41. Finally, we complete the query by calling show on the final DataFrame in the series.
42. We can see that the query results in a DataFrame with the two selected columns and the two rows from the original DataFrame that match the specified where criteria.
43. Because a DataFrame transformation returns another DataFrame, a series of transformations can be chained together into a single line of code.
44. The first code snippet here shows three lines of Python code that define a query consisting of a select, a where, and a show.
45. The second snippet does exactly the same thing, but chains the operations into a single command.
46. In the first snippet, we assign intermediate DataFrames to different variables.
47. In the second, the same DataFrames are created, but not assigned to variables.
48. The sequence of operations performed is identical;
49. the difference is only syntactic.
50. This slide shows two Scala snippets, executing the same query that the Python snippets on the previous slide did.
51. In the first snippet, each transformation result is assigned to a separate variable.
52. In the second, the transformations are chained together, but the query operations are identical.

How to Spark Data Frames and Datasets

<http://spark.apache.org/docs/2.2.0/sql-programming-guide.html>

Soark API for Python <http://spark.apache.org/docs/2.2.0/api/python/index.html>

**Hands-On Exercise: Exploring DataFrames Using the Apache Spark Shell**

 Bookmark this page

**Open the Exercise Environment Portal**

Top of Form

Open

*Don't forget to shut down your VMs when you are done using them!*

*If you are having issues accessing your exercise environment, please contact*[*ondemand-feedback@cloudera.com*](mailto:ondemand-feedback@cloudera.com)*.*

Bottom of Form

**Hands-On Exercise: Exploring DataFrames Using the Apache Spark Shell**

**Table of Contents**

[View the Spark Documentation](https://ondemand.cloudera.com/courses/course-v1:Cloudera+DevSH+180515/courseware/72d2e7075fd44e0682ce290cd1e488ad/330364e232f745ee908433d74ca26f9b/?child=first#d0e1627)

[Start the Spark Shell](https://ondemand.cloudera.com/courses/course-v1:Cloudera+DevSH+180515/courseware/72d2e7075fd44e0682ce290cd1e488ad/330364e232f745ee908433d74ca26f9b/?child=first#d0e1672)

[Read and Display a JSON File](https://ondemand.cloudera.com/courses/course-v1:Cloudera+DevSH+180515/courseware/72d2e7075fd44e0682ce290cd1e488ad/330364e232f745ee908433d74ca26f9b/?child=first#d0e1819)

[Query a DataFrame](https://ondemand.cloudera.com/courses/course-v1:Cloudera+DevSH+180515/courseware/72d2e7075fd44e0682ce290cd1e488ad/330364e232f745ee908433d74ca26f9b/?child=first#d0e1935)

| **Files and Data Used in This Exercise** | |
| --- | --- |
| Exercise directory | $DEVSH/exercises/spark-shell |
| Data files (local) | $DEVDATA/devices.json |

**In this exercise, you will use the Spark shell to work with DataFrames.**

You will start by viewing and bookmarking the Spark documentation in your browser. Then you will start the Spark shell and read a simple JSON file into a DataFrame.

**Important:** This exercise depends on a previous exercise: “Access HDFS with Command Line and Hue.” If you did not complete that exercise, run the course catch-up script and advance to the current exercise:

$ **$DEVSH/scripts/catchup.sh**

**View the Spark Documentation**

1. View the Spark documentation in your web browser by visiting the URI <http://spark.apache.org/docs/2.2.0/>.
2. From the **Programming Guides** menu, select the **DataFrames, Datasets and SQL**. Briefly review the guide and bookmark the page for later review.
3. From the **API Docs** menu, select either **Scala** or **Python**, depending on your language preference. Bookmark the API page for use during class. Later exercises will refer to this documentation.
4. If you are viewing the Scala API, notice that the package names are displayed on the left. Use the search box or scroll down to find the org.apache.spark.sql package. This package contains most of the classes and objects you will be working with in this course. In particular, note the Dataset class. Although this exercise focuses on DataFrames, remember that DataFrames are simply an alias for Datasets of Row objects. So all the DataFrame operations you will practice using in this exercise are documented on the Dataset class.
5. If you are viewing the Python API, locate the pyspark.sql module. This module contains most of the classes you will be working with in this course. At the top are some of the key classes in the module. View the API for the DataFrame class; these are the operations you will practice using in this exercise.

**Start the Spark Shell**

You may choose to do the remaining steps in this exercise using either Scala or Python.

**Note on Spark Shell Prompt**

To help you keep track of whether a Spark command is Python or Scala, the prompt will be shown here as either pyspark> or scala>. Some commands are the same for both Scala and Python. These will be shown with a > undesignated prompt. The actual prompt displayed in the shell will vary depending on which version of Python or Scala you are using and which command number you are on.

1. If you do not have an open terminal window, start one now.
2. In the terminal window, start the Spark 2 shell. Start either the Python shell or the Scala shell, not both.

To start the Python shell, use the pyspark2 command.

$ **pyspark2**

To start the Scala shell, use the spark2-shell command.

$ **spark2-shell**

You may get several WARN messages, which you can disregard.

1. Spark creates a SparkSession object for you called spark. Make sure the object exists. Use the first command below if you are using Python, and the second one if you are using Scala. (You only need to complete the exercises in Python *or*Scala.)

pyspark> spark

scala> spark

Python will display information about the spark object such as:

<pyspark.sql.session.SparkSession at *address*>

Scala will display similar information in a different format:

org.apache.spark.sql.SparkSession = org.apache.spark.sql.SparkSession@*address*

**Note:**In subsequentinstructions, both Python and Scala commands will be shown but not noted explicitly; Python shell commands are in blue and preceded with pyspark>, and Scala shell commands are in red and preceded with scala>.

1. Using command completion, you can see all the available Spark session methods: type spark. (spark followed by a dot) and then the **TAB** key.

**Note:** You can exit the Scala shell by typing sys.exit. To exit the Python shell, press **Ctrl**+**D** or type exit. However, stay in the shell for now to complete the remainder of this exercise.

**Read and Display a JSON File**

1. Open a new terminal window (in addition to the terminal running the Spark shell).
2. Review the simple text file you will be using: $DEVDATA/devices.json. You can view the file either in gedit, or by starting a new terminal window then using the **less** command. (Do not modify the file.) This file contains records for each of Loudacre’s supported devices. For example:
3. {"devnum":1,"release\_dt":"2008-10-21T00:00:00.000-07:00",

"make":"Sorrento","model":"F00L","dev\_type":"phone"}

Notice the field names and types of values in the first few records.

1. Upload the data file to the /loudacre directory in HDFS:

$ **hdfs dfs -put $DEVDATA/devices.json /loudacre/**

1. In the Spark shell, create a new DataFrame based on the devices.json file in HDFS.
2. pyspark> devDF = spark.read. \

json("/loudacre/devices.json")

scala> val devDF = spark.read.

json("/loudacre/devices.json")

1. Spark has not yet read the data in the file, but it has scanned the file to infer the schema. View the schema, and note that the column names match the record field names in the JSON file.

pyspark> devDF.printSchema()

scala> devDF.printSchema

1. Display the data in the DataFrame using the show function. If you don’t pass an argument to show, Spark will display the first 20 rows in the DataFrame. For this step, display the first five rows. Note that the data is displayed in tabular form, using the column names defined in the schema.

> devDF.show(5)

**Note:**Like many Spark queries,this command is the same whether you are using Scala or Python.

1. The show and printSchema operations are actions—that is, they return a value from the distributed DataFrame to the Spark driver. Both functions display the data in a nicely formatted table. These functions are intended for interactive use in the shell, but do not allow you actually work with the data that is returned. Try using the take action instead, which returns an array (Scala) or list (Python) of Row objects. You can display the data by iterating through the collection.
2. pyspark> rows = devDF.take(5)

pyspark> for row in rows: print row

scala> val rows = devDF.take(5)

scala> rows.foreach(println)

**Query a DataFrame**

1. Use the count action to return the number of items in the DataFrame.

> devDF.count()

1. DataFrame transformations typically return another DataFrame. Try using a **select** transformation to return a DataFrame withonly the make and modelcolumns, then display its schema. Note that only theselected columns are in the schema.
2. pyspark> makeModelDF = devDF.select("make","model")

pyspark> makeModelDF.printSchema()

scala> val makeModelDF = devDF.select("make","model")

scala> makeModelDF.printSchema

1. A query is a series of one or more transformations followed by anaction. Spark does not execute the query until you call the action operation. Display the first 20 lines of the final DataFrame in the series using the show action.

pyspark> makeModelDF.show()

scala> makeModelDF.show

1. Transformations in a query can be chained together. Execute a single command to show the results of a query using select and where. The resulting DataFrame will contain only the columns devnum, make, and model, and only the rows where the make is Ronin.
2. pyspark> devDF.select("devnum","make","model"). \
3. where("make = 'Ronin'"). \

show()

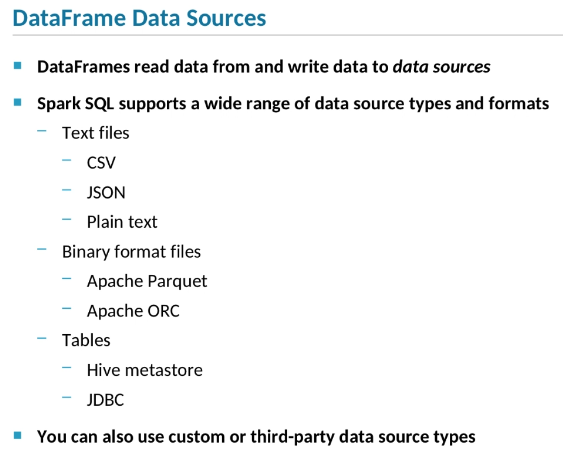
scala> devDF.select("devnum","make","model").

where("make = 'Ronin'").

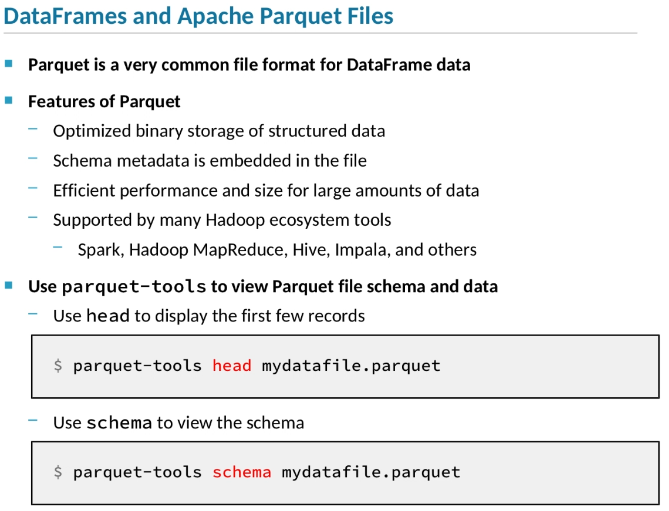
show

**This is the end of the exercise.**

Creating Data frames from Sources



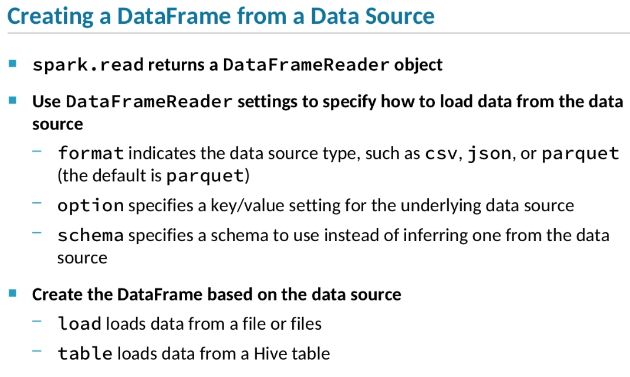
Take care when using a relational database as a data source, though, because repeated, distributed reads from the database can trigger a Distributed Denial of Service, or DDOS, situation.

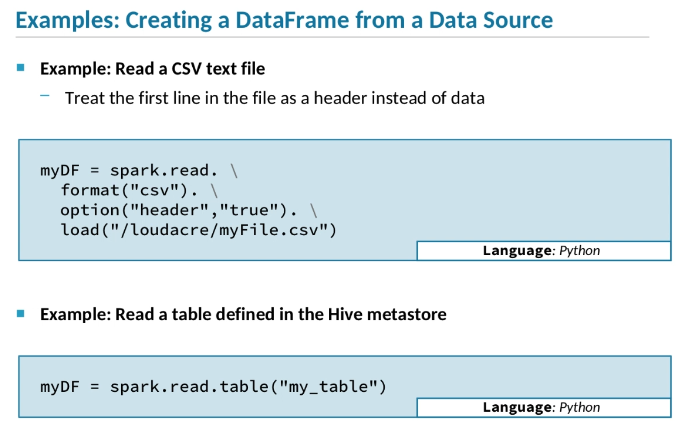


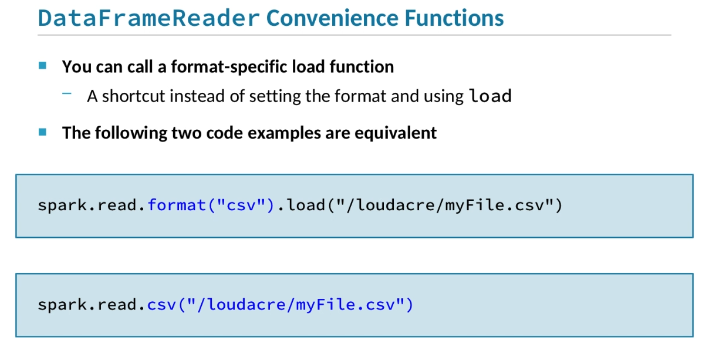
Parquet is the most efficient of the columnar file formats.

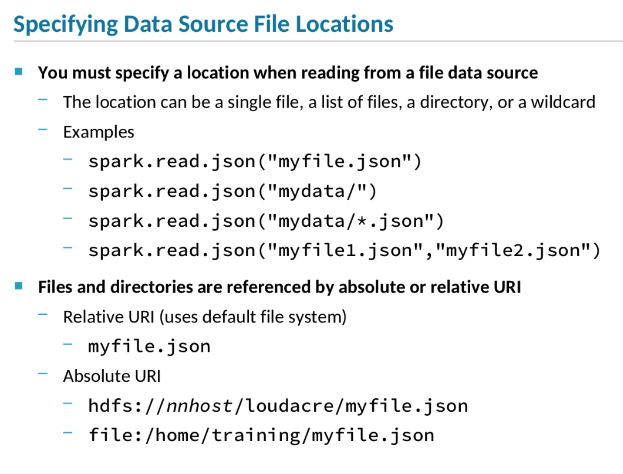
1. In Spark SQL, data is loaded from and saved to a *data source*.
2. In order to create and populate a DataFrame, you need to associate it with a source of data, such as a file or a table.
3. Similarly, in order to save the data from a DataFrame, you need to specify a destination, which is also typically a file or table.
4. Spark SQL includes support for a number of different types of data sources.
5. For instance, you can read from or write to text files using a variety of formats.
6. Two common types are CSV files—that is, files with fields delimited by a comma or another character—and JSON files.
7. Both CSV and JSON files are considered to be semi-structured data.
8. The data is not fully structured as it would be in a database table, for instance, but it is structured into records and fields.
9. DataFrames can also read and write plain text files.
10. This is less common, because plain text is unstructured.
11. When Spark loads a plain text file into a DataFrame, each line is treated as a single string.
12. So the resulting DataFrame has only a single string column with one row per line in the file.
13. Spark SQL also supports binary file formats that contain fully structured data.
14. The most common such format is Apache Parquet, an efficient, compressed, columnar format.
15. ORC, which stands for Optimized Row Columnar format, is also supported.
16. ORC is similar to Parquet;
17. it’s also a self-describing columnar format.
18. In addition to reading from files, DataFrames can also use Hive tables,
19. or they can use JDBC to access a table in a relational database like Oracle or MySQL.
20. Take care when using a relational database as a data source, though,
21. because repeated, distributed reads from the database can trigger a Distributed Denial of Service, or DDOS, situation.
22. You can also use a third-party data source integration.
23. For instance, Databricks provides the spark-avro library to support the Apache Avro data format.
24. If none of the supported or third-party data source types meets your needs, you can also create a custom data source type yourself.
25. Spark's default data source format is Parquet.
26. Parquet stores structured data in an efficient binary file format.
27. A key feature of Parquet is that metadata describing the data's structure is embedded directly in the files.
28. When Spark creates a DataFrame based on a Parquet file, it can define the DataFrame’s schema using the embedded metadata.
29. Parquet's columnar architecture minimizes the disk space required to store large amounts of data,
30. and optimizes the performance of queries that work with just a few of the data columns.
31. Parquet is the most efficient of the columnar file formats.
32. It was originally developed by Cloudera and Twitter, but it’s now an Apache project.
33. Parquet is supported by most of the key Hadoop ecosystem tools, like Spark, MapReduce, Hive, and Impala.
34. Parquet's binary format provides efficient storage, but is not human-readable, making it hard for developers to work with directly.
35. The parquet-tools application lets you view the schema and data for exploration, testing, and debugging.
36. The parquet-tools head command outputs the first few records in a Parquet file in plain text.
37. The parquet-tools schema command displays the schema of the data.

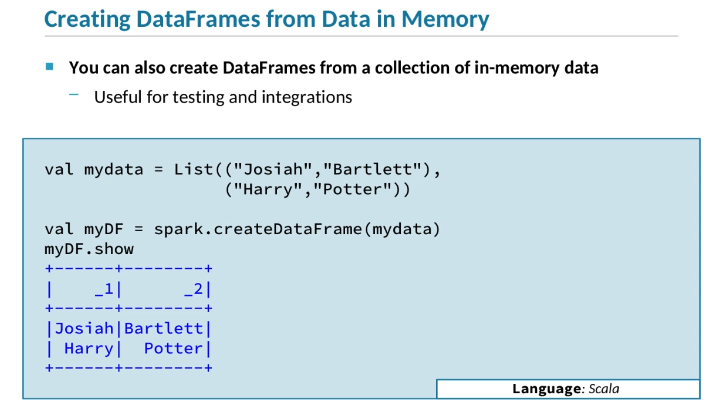
Creating Data frames from Sources Part 2





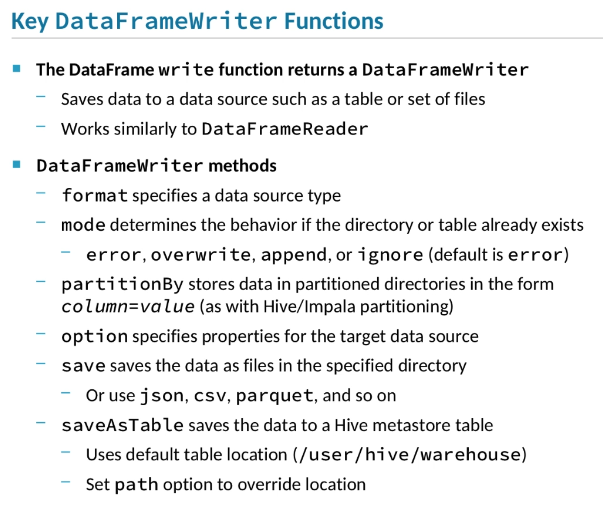


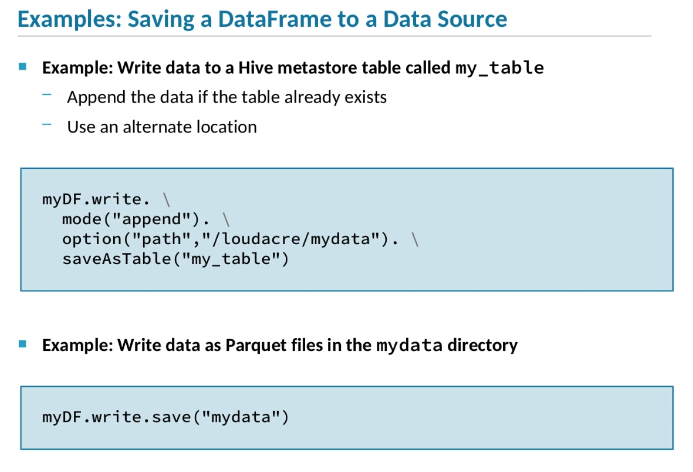


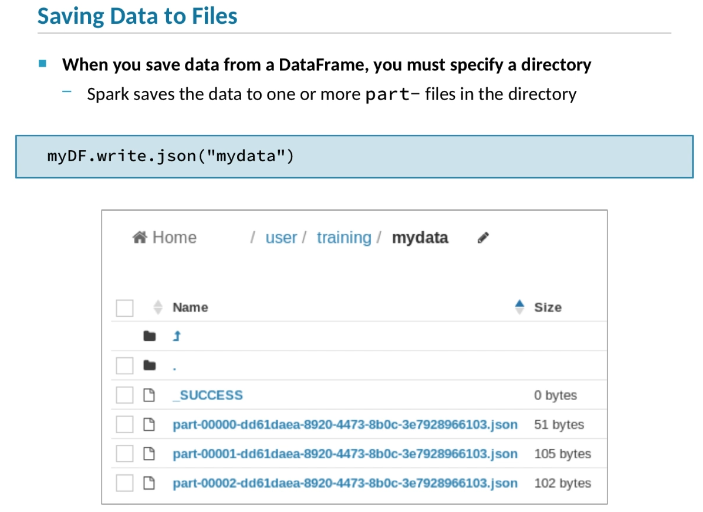


1. To create a DataFrame from a file or table, use the DataFrameReader API included in Spark SQL.
2. The entry point for this API is the read attribute of the SparkSession object.
3. This points to a DataFrameReader object that provides a number of functions to load data using different formats and options.
4. The format method sets the file format, such as csv, json, parquet, orc, and so on.
5. By default, Spark SQL will assume the format is parquet if you don’t explicitly set a different one, although this behavior is configurable.
6. The option method takes two strings as a key-value pair to configure options related to the data source.
7. Each type of data source defines its own options.
8. For instance, with CSV you can indicate whether the first line of the file should be treated as a header,
9. whereas with JDBC, you would specify the URI and login information needed to access your database.
10. With most data source types, you have the choice to use a manually defined schema instead of letting Spark infer it automatically from the source.
11. Use the DataFrameReader object's schema method to set an alternate schema.
12. After you’ve configured the DataFrameReaderobject for the specific data source you are using,
13. call the load method to read a file or the tablemethod to read a Hive table.
14. Let’s look at a couple of examples of how to create a DataFrame based on a data source.
15. The first code snippet creates a DataFrame based on a CSV file.
16. All the methods that configure and read the data are chained together.
17. First we call spark.read to get a DataFrameReaderobject.
18. Then we call format on that object to set the format to csv.
19. Next we set the CSV header option to true.
20. This means that the first line of the data file is a header with column names, rather than actual data.
21. The DataFrameReader will use that header line to infer the column names for the new schema.
22. Finally, we call the load function, passing the location of the file to read—in this case, myFile.csv.
23. At this point, the DataFrameReader will retrieve the header from the file, define the schema, and create a new DataFrame object, which we assign to the myData variable.
24. The code in the next snippet reads from a Hive table instead of a file.
25. Again, we use spark.read to retrieve a DataFrameReader object.
26. Then we call table to specify the name of the table to read.
27. This call retrieves the table’s metadata so that it can correctly define a new schema.
28. Although the canonical way to read a data file is to specify the format using the format function before calling load,
29. Spark also provides convenience functions to read any supported file type directly, without needing to call format.
30. For example, call the csv function to read a CSV file, or the json function to read a JSON file.
31. Here we show two code snippets that do exactly the same thing.
32. The first calls format to specify CSV, and then loadto read the file.
33. The second one uses the csv convenience function, making a call to format unnecessary.
34. When you create a DataFrame from a file data source, you need to specify which file or files to read.
35. If your data is in a single file, just specify the path to that file.
36. Often, however, a set of data is stored in a collection of files rather than a single file.
37. In that case, you can specify a directory path instead of a specific file name, and Spark will read all the data files in that directory.
38. You can also specify a wildcard to match a set of files, or provide multiple individual files.
39. The examples shown here specify the file or directory relative to the default directory on the default file system.
40. On a cluster configured by Cloudera Manager, the default file system is generally HDFS, and the default directory is the current user’s home directory.
41. So when you specify just the file name, myfile.json,
42. the json function will look for that file in your HDFS home directory, which is usually a directory named for your username under /home.
43. Alternatively, you can specify the full URI of a file or directory instead of a relative path.
44. The URI can include the file system, which is usually a distributed file system like HDFS or S3.
45. If you don’t specify a file system, Spark assumes the cluster’s default file system.
46. It’s also possible to reference a file on the local system, meaning the Linux OS file system on which the application is running.
47. This is uncommon and usually only used for testing purposes when running the Spark shell or your Spark application in local mode.
48. If you are running your application or shell on a cluster, and you reference a file on the local file system,
49. the path needs to exist in the same location on all worker nodes in the cluster,
50. because it’s the distributed application executors that actually read the file or files.
51. Usually DataFrames load data from files or tables, but they can also use data that’s in memory.
52. This is useful when testing code, and can also be used to integrate with other applications or systems that generate in-memory data.
53. In-memory data for a DataFrame needs to be a collection of tuples containing values of supported data types.
54. In this example, we assign mydata to a List of two-item tuples, each containing a first and last name.
55. Then we call the spark.createDataFrame method, passing in that data.
56. When we call show, we can see that the DataFrame’s schema consists of two columns corresponding to the two items in each tuple in the list of data.
57. Note that the column names are \_1 and \_2.
58. This naming pattern is the default when you don’t provide a schema definition explicitly.

Saving Data Frames to Data Sources

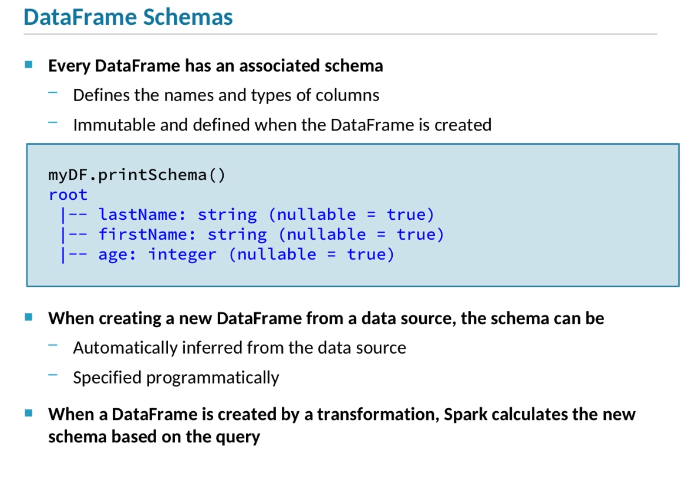


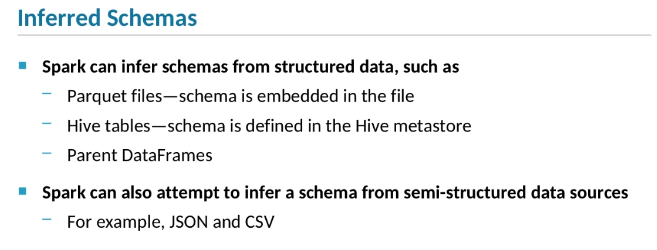


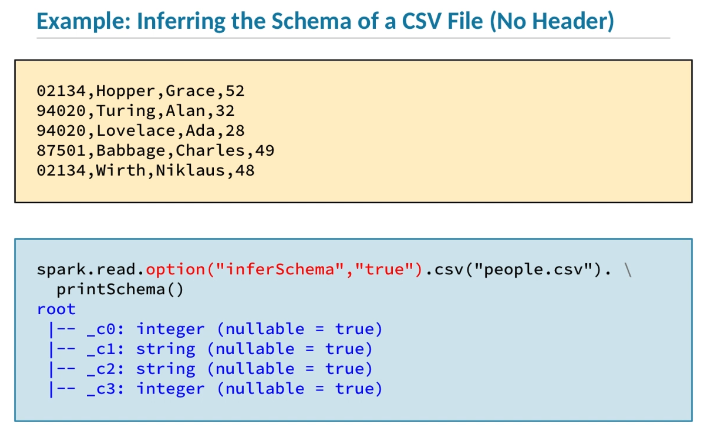


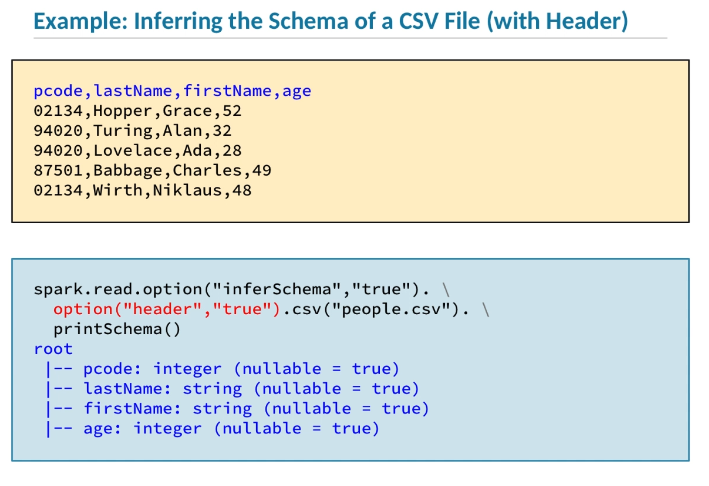
1. Just as DataFrame data is loaded from a data source, when you save a DataFrame, the data is saved to a data source such as a table or set of files.
2. DataFrames are saved by a DataFrameWriter object associated with the DataFrame.
3. The DataFrameWriter class is very similar to the DataFrameReader class used to load data.
4. Here are some of the key DataFrameWriterfunctions.
5. Use the format function to specify the type of data source to write to, such as CSV, JSON, or Parquet.
6. **Unless you configure a different default for your Spark session, the default format is Parquet.**
7. The mode function lets you indicate what should happen if the directory or table you are saving to already exists.
8. The error mode, which is the default, means that Spark will throw an error if the source already exists.
9. The other options are to overwrite the existing data, append the new data to the existing data, or ignorethe save command completely, without an error.
10. Hive tables and Spark data source files can be partitioned based on values for a specified column or columns.
11. This means, for example, if you partition customer data by a country,
12. data for customers in each country would be stored in separate directories—
13. United States customers in one directory, customers in France in another, and so on.
14. This makes queries limited by country much more efficient, because it only reads data from the relevant directories.
15. To use this feature, call the partitionBy function with the columns to partition by.
16. The option function of DataFrameWriter works just like it does with DataFrameReader, allowing you to set data source options.
17. For instance, you could set the sep option for a CSV data source to tell Spark to save the file using an alternate field separator instead of a comma, which is the default.
18. The generic function for saving data in files is save.
19. Alternatively, you can use convenience functions like json, csv, parquet, and so on, so that you don’t need to specify a format separately.
20. To save data to a Hive table, use saveAsTable.
21. This defines the table in the Hive metastore and saves the data itself to Hive files.
22. By default, when Spark creates a new Hive table, it will save the table data in Hive’s default location, which is usually /user/hive/warehouse.
23. If you want to set a different location for the table, use the option function to set the path option to the alternate location.
24. Here are a couple examples of how to save a DataFrame.
25. In both these examples, we assume we already have a DataFrame called myData.
26. The first code snippet shows how to save a DataFrame as a Hive table.
27. We use myData.write to retrieve the DataFrame’s DataFrameWriter.
28. Then we set the mode to append this data to any data that might already exist in the target table.
29. We also set the path option to specify an alternate table data location, instead of using Hive’s default location.
30. Note that if the table already exists, the path option is disregarded, and the data will be saved to the location of the existing data.
31. To change the location of an existing table's data, you need to drop and recreate the table.
32. Then we call saveAsTable, specifying the name of the Hive table as my\_table.
33. At this point, the writer will create the table in the Hive metastore if it isn’t already defined,
34. and then write the data to the table’s directory, which in this example is /loudacre/mydata.
35. The next example shows how we would save the data to a directory called mydata, located under the default directory in the default file system.
36. Because no format is specified, the resulting file will be in Parquet format.
37. Note that both save and saveAsTable are action operations.
38. Unlike loading files, when you save the data in a DataFrame, you specify a directory rather than a file or set of files.
39. This is because Spark typically saves data into multiple files, not a single file.
40. The specified directory is where all the files will be stored.
41. Each generated file name starts with part-, and then a number that reflects the order of the files:
42. part-00000 is the first part of the data, part-00001 is the second, and so on.
43. Each file contains one block of the whole set of data.
44. This is because of the way Spark tasks are distributed across a cluster.
45. Data in a DataFrame is partitioned across multiple executors, with each executor loading, transforming, and saving its own data partition.
46. When an executor saves its partition, it saves it to a separate file.
47. This is why a DataFrame is saved as multiple part- files.

Data Fram Schemas Part 1









1. Every DataFrame is associated with a schema that maps the values in the Row objects in the DataFrame to columns.
2. The schema defines the names and types of the columns.
3. A DataFrame’s schema is defined when the DataFrame is first created.
4. Once a DataFrame’s schema is set, it never changes;
5. it’s immutable, just like the data in a DataFrame.

im·mu·ta·ble

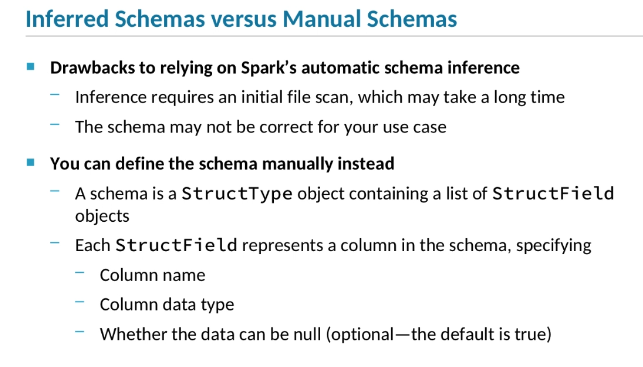
i(m)ˈmyo͞odəb(ə)l/

*adjective*

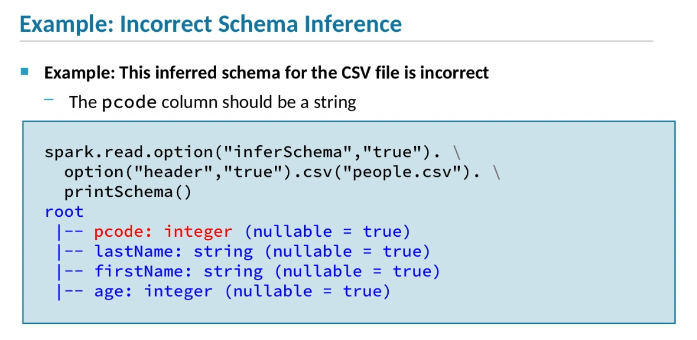
unchanging over time or unable to be changed.

1. Here’s an example of a DataFrame’s printSchemamethod displaying a formatted version of the DataFrame’s schema object.
2. The myDF DataFrame has three columns: lastNameand firstName, which are both strings, and age, which is an integer.
3. The columns are also noted as being nullable, meaning that a row may not contain a value in that column.
4. When a new DataFrame is created by reading data from a data source, the schema is determined in one of two ways.
5. It can either be inferred from the data itself, or you can define a schema manually and set that schema when creating the DataFrame.
6. When the new DataFrame is returned as the result of a transformation on a base DataFrame, Spark calculates a new schema by analyzing the transformation.
7. For instance, if the transformation is a select that specifies just two of the base DataFrame's columns,
8. the schema of the new DataFrame will include only those two columns.
9. When you create a DataFrame based on a structured data source such as a Parquet file or Hive table,
10. the schema can be inferred from the structure of that data.
11. For example, Parquet files embed the schema of their data right in the file.
12. Spark reads that schema from the file and uses it to create the DataFrame’s schema.
13. Hive table schemas are part of the table’s metadata, which is stored in the Hive metastore.
14. So Spark will retrieve the table’s schema from the metastore and create the DataFrame schema based on that.
15. Spark can also attempt to infer a schema from semi-structured data in a text file such as a JSON or CSV file.
16. In that case, it will scan the actual data stored in the file,
17. make its best guess at the structure of that data, and create a schema that reflects that structure.
18. Let’s look at an example of how Spark would infer a schema from a CSV file.
19. Here we have a data file called people.csv.
20. Note that, unlike some CSV files, this file does not have a header line that defines the names of the fields.
21. Each line contains four fields, separated by commas.
22. In the code example, we read the CSV file.
23. We set the inferSchema option to true, indicating that the reader should scan the file and attempt to infer the types of the fields from the data.
24. In this case, the first and last fields of each line contain integers,
25. so Spark will infer that the first and last column of the schema should be integers as well.
26. The other columns are treated as strings.
27. The default value for inferSchema is false.
28. If you don’t set the value to true, or if you explicitly set it to false, all columns are assumed to be strings.
29. Note that the column names are based on the order of the fields in the file, following the pattern \_c0, \_c1, and so on.
30. This example is very similar to the last, with one difference—
31. the first line of the CSV file is not data, but rather a header that associates names with each field in the remaining lines of the file.
32. So when we read the file, we set the header option to true.
33. When header is true, the DataFrame’s schema will use the field names as column names in the schema, as shown.
34. The header names will be used regardless of whether the inferSchema option is true.
35. In this example, as in the last, inferSchema is true,
36. so the schema will also reflect the inferred data types instead of defining all the columns as strings.

Data Frame Schemas Part 2

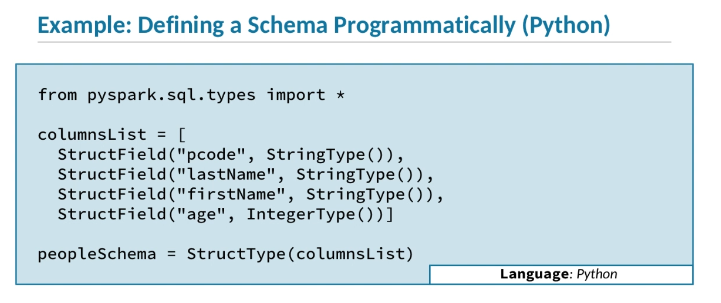


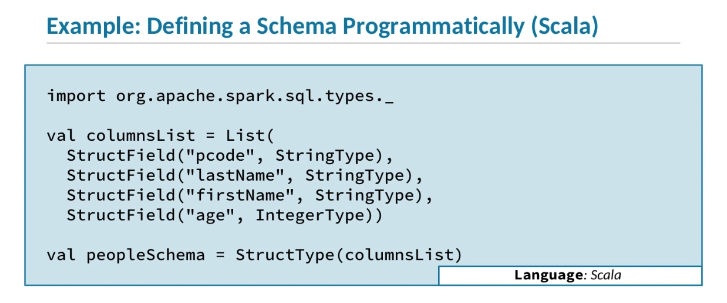
If nullable is false, that does not imply that Spark will enforce the rule that values in that column can’t be null. When a DataFrame column is noted as nullable, it simply means that Spark can assume that values in that column will never be null which helps Spark optimize queries that use that column.

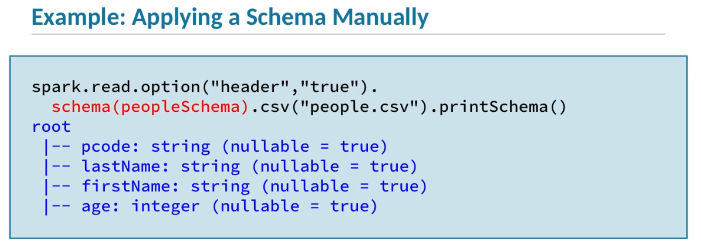


Postal codes must always be treated as strings. Although in this bit of sample data, all of them happen to be integers, they don’t have to be. Canadian postal codes, for example, can contain letters.

Also, it doesn’t make sense to perform arithmetic operations on a postal code; the sum of two postal codes would not be a meaningful value.

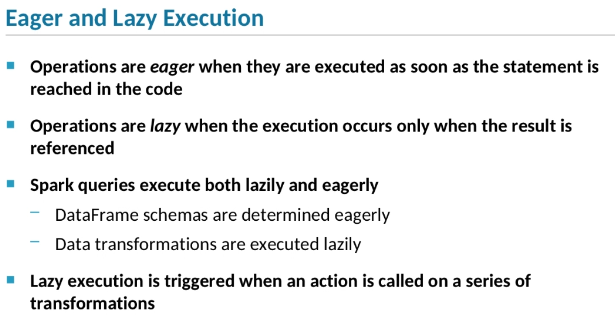


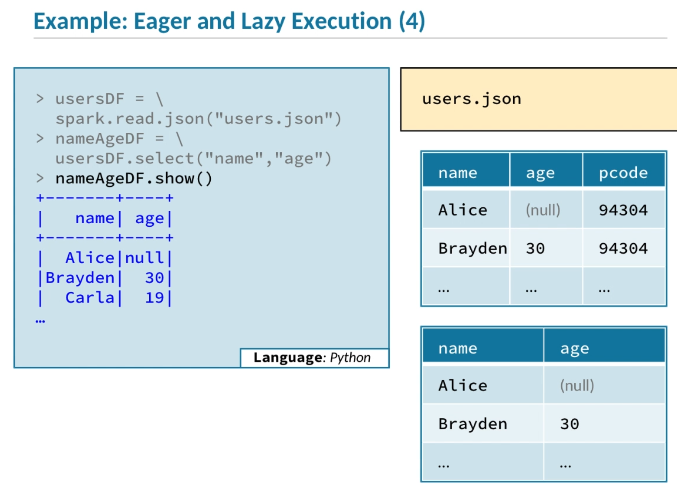




1. Although Spark’s ability to infer a DataFrame’s schema from its data source is convenient, there are a couple of drawbacks to using it.
2. One is that inferring the schema requires that Spark read every line of every file in a set of files.
3. If you are working with a large amount of data, that will take a lot of time, hurting performance.
4. The other issue is that, although Spark will do its best to guess at the correct schema, the result may not match the schema you expect or need the data to have.
5. You can avoid both these problems by defining a schema programmatically rather than relying on Spark’s automatic schema inference.
6. A DataFrame’s schema is represented by an object of the StructType class.
7. That object contains of a collection of StructFieldobjects.
8. Each StructField object represents a column.
9. For each column, the StructField object indicates the name and data type of the column, and, optionally, whether that column is nullable—
10. that is, whether every row must have a valid value for that column, or whether that value can simply be null.
11. The nullable setting of StructField may be confusing.
12. In a relational database, defining a column as non-nullable means that the database system will enforce that no rows have a null value in that column,
13. and will fail with an error if you try to insert a row with a null value.
14. This is not the case with Spark SQL.
15. If nullable is false, that does not imply that Spark will enforce the rule that values in that column can’t be null.
16. When a DataFrame column is noted as nullable, it simply means that Spark can assume that values in that column will never be null,
17. which helps Spark optimize queries that use that column.
18. For data sources like files that do not enforce non-nullability, such as JSON and CSV files,
19. all the columns will always be nullable, even if you attempt to set nullable to false.
20. Let’s look at an example of where the inferred schema is wrong.
21. We start by letting Spark infer the schema from a CSV file.
22. The values of the first field, representing a postal code, are five-digit numbers in all the lines in the file, so the resulting schema defines the pcodecolumn as an integer.
23. Unfortunately, for this column, that doesn’t make sense.
24. Postal codes must always be treated as strings.
25. Although in this bit of sample data, all of them happen to be integers, they don’t have to be.
26. Canadian postal codes, for example, can contain letters.
27. Also, it doesn’t make sense to perform arithmetic operations on a postal code;
28. the sum of two postal codes would not be a meaningful value.
29. So, we need to make sure the schema correctly uses a string type for the pcode column.
30. To make sure that pcode is a string column, we need to specify the schema programmatically instead of using an inferred schema.
31. The first step is to define the schema, which is an object of the StructType class.
32. We’ll look how to create a schema object in Python and Scala separately, because the code is slightly different.
33. Here we show the Python example.
34. First, we need to import the column type definitions from the pyspark.sql.types module.
35. Next, we create a list containing a series of StructField instances.
36. Each StructField is created with a column name and a type.
37. StructField objects can also be defined with an optional nullable setting, but in this example we are working with CSV files.
38. CSV files can’t enforce non-nullability, so the columns in this schema will always be nullable, no matter what we specify here, so we leave that setting out.
39. Having created a list of StructField objects defining the columns in the schema,
40. we next create the schema itself, which is a StructType object based on the StructFields we defined above.
41. We call this StructType object peopleSchema.
42. Here we show Scala code that does the exact same thing as the Python code we just looked at.
43. The only difference is that we import a different library containing the column type definitions.
44. In both cases, the result is a StructType object called peopleSchema.
45. The last step in this example is to apply the schema we defined when we created the DataFrame, rather than using an inferred schema as we did earlier.
46. We do so by calling the schema function and passing in the peopleSchema object we just created.
47. When we view the resulting schema, we see that the pcode column is now typed correctly as a string.
48. Note that although the CSV file in this example includes a header line,
49. Spark disregards the column names in the header and uses those specified in the schema instead.
50. You might be wondering what would happen if you define a column in a schema to use one type, but the data file contained a value of a different type.
51. For example, what if one of the age values is a string like foo?
52. In that case, Spark would throw a NumberFormatException,
53. because it wouldn’t be able to convert foo to an integer, and the age column requires an integer value.

Eager and Lazy Execution





1. Spark SQL queries involve both *eager* and *lazy*execution.
2. When an operation is called eager, it means the operation is executed as soon as the function is called.
3. In contrast, a lazy operation doesn’t execute right away;
4. instead, the execution occurs when the resulting data is actually needed.
5. **In Spark, DataFrame schemas are determined eagerly; data is transformed lazily.**
6. Transformations in a query are only executed when the final action of the query is called.
7. Lazy transformation is one of the key features of Spark.
8. It allows a series of transformations to be pipelined together for optimal performance.
9. Only the data actually returned by the action is ever read or processed.
10. Let’s look at an example of how a query executes.
11. On the first line, we call the read.json method to load a data file into a DataFrame called usersDF.
12. When we call the json function, Spark will scan the JSON records in the data file so that it can infer the schema of the resulting DataFrame.
13. But it doesn't actually read the data into the DataFrame at this point.
14. Instead, it simply creates an empty DataFrame with the correct schema,
15. which in this case includes three columns corresponding to the three fields in the JSON records in the data file.
16. This is because schema definition is an eager operation, whereas data loading is a lazy operation.
17. On the next line, we call select on the first DataFrame to create a new DataFrame called nameAgeDF, containing only the name and agecolumns of usersDF.
18. Spark calculates the schema of the new DataFrame by analyzing the transformation.
19. The select transformation limits the columns to those selected, so the new schema has only nameand age columns.
20. The schema calculation occurs immediately, but the users.json data file at the base of this query has not been loaded.
21. The final operation in the query is a show action.
22. The action triggers the execution of all the transformations in the query.
23. So when we call show, the data from users.json is loaded, transformed, and displayed all at once.
24. Once all the transformations are executed, the show method displays the first few lines of the resulting DataFrame.
25. Lazy execution of transformations allows the Catalyst optimizer to analyze and optimize the whole query before any data is processed, for maximum performance.

Bottom of Form

### Hands-On Exercise: Working with DataFrames and Schemas

**Table of Contents**

[Create a DataFrame Based on a Hive Table](https://ondemand.cloudera.com/courses/course-v1:Cloudera+DevSH+180515/courseware/d20fe742a1164ebc934da86e197eac8d/5fe8b9830cc8476ca93ef6119f138fc8/?child=first#d0e2075)

[Define a Schema for a DataFrame](https://ondemand.cloudera.com/courses/course-v1:Cloudera+DevSH+180515/courseware/d20fe742a1164ebc934da86e197eac8d/5fe8b9830cc8476ca93ef6119f138fc8/?child=first#d0e2162)

| **Files and Data Used in This Exercise:** | |
| --- | --- |
| Exercise directory | $DEVSH/exercises/dataframes |
| Data files (HDFS) | /loudacre/devices.json |
| Hive Tables | accounts |

**In this exercise, you will work with structured account and mobile device data using DataFrames.**

You will practice creating and saving DataFrames using different types of data sources, and inferring and defining schemas.

**Important:** This exercise depends on a previous exercise: “Exploring DataFrames Using the Spark Shell.” If you did not complete that exercise, run the course catch-up script and advance to the current exercise:

$ **$DEVSH/scripts/catchup.sh**

#### Create a DataFrame Based on a Hive Table

1. This exercise uses a DataFrame based on the accounts Hive table. Before you start working in Spark, visit Hue in the web browser on your remote desktop and use the Impala Query Editor to review the schema and data of the accounts table in the default database.
2. If you do not have one already, open a terminal, and start the Spark 2 shell (either Scala or Python, as you prefer).
3. Create a new DataFrame using the Hive accounts table.

pyspark> accountsDF = spark.read.table("accounts")

scala> val accountsDF = spark.read.table("accounts")

1. Print the schema and the first few rows of the DataFrame, and note that the schema and data are the same as the Hive table.
2. Create a new DataFrame with rows from the accounts data where the zip code is 94913, and save the result to CSV files in the /loudacre/accounts\_zip94913 HDFS directory. You can do this in a single command, as shown below, or with multiple commands.
3. pyspark> accountsDF.where("zipcode = 94913"). \
4. write.option("header","true"). \

csv("/loudacre/accounts\_zip94913")

scala> accountsDF.where("zipcode = '94913'").

write.option("header","true").

csv("/loudacre/accounts\_zip94913")

1. Use Hue or the command line (in a separate terminal window) to view the /loudacre/accounts\_zip94913 directory in HDFS and the data in one of the saved files. Confirm that the CSV file includes a header line, and that only records for the selected zip code are included.
2. Optional: Try creating a new DataFrame based on the CSV files you created above. Compare the schema of the original accountsDF and the new DataFrame. What’s different? Try again, this time setting the inferSchema option to true and compare again.

#### Define a Schema for a DataFrame

1. If you have not done so yet, review the data in the HDFS file /loudacre/devices.json.
2. Create a new DataFrame based on the devices.json file. (This command could take several seconds while it infers the schema.)
3. pyspark> devDF = spark.read. \

json("/loudacre/devices.json")

scala> val devDF = spark.read.

json("/loudacre/devices.json")

1. View the schema of the devDF DataFrame. Note the column names and types that Spark inferred from the JSON file. In particular, note that the release\_dt column is of type string, whereas the data in the column actually represents a timestamp.
2. Define a schema that correctly specifies the column types for this DataFrame. Start by importing the package with the definitions of necessary classes and types.

pyspark> from pyspark.sql.types import \*

scala> import org.apache.spark.sql.types.\_

1. Next, create a collection of StructField objects, which represent column definitions. The release\_dt column should be a timestamp.
2. pyspark> devColumns = [
3. StructField("devnum",LongType()),
4. StructField("make",StringType()),
5. StructField("model",StringType()),
6. StructField("release\_dt",TimestampType()),

StructField("dev\_type",StringType())]

scala> val devColumns = List(

StructField("devnum",LongType),

StructField("make",StringType),

StructField("model",StringType),

StructField("release\_dt",TimestampType),

StructField("dev\_type",StringType))

1. Create a schema (a StructType object) using the column definition list.

pyspark> devSchema = StructType(devColumns)

scala> val devSchema = StructType(devColumns)

1. Recreate the devDF DataFrame, this time using the new schema.
2. pyspark> devDF = spark.read. \

schema(devSchema).json("/loudacre/devices.json")

scala> val devDF = spark.read.

schema(devSchema).json("/loudacre/devices.json")

1. View the schema and data of the new DataFrame, and confirm that the release\_dt column type is now timestamp.
2. Now that the device data uses the correct schema, write the data in Parquet format, which automatically embeds the schema. Save the Parquet data files into an HDFS directorycalled /loudacre/devices\_parquet.
3. Optional: In a separate terminal window, use parquet-tools to view the schema of the saved files.
4. $ **parquet-tools schema \**

**hdfs://master-1/loudacre/devices\_parquet/**

Note that the type of the release\_dt column is noted as int96; this is how Spark denotes a timestamp type in Parquet.

For more information about parquet-tools, run parquet-tools --help.

1. Create a new DataFrame using the Parquet files you saved in devices\_parquet and view its schema. Note that Spark is able to correctly infer the timestamp type of the release\_dt column from Parquet’s embedded schema.

**This is the end of the exercise.**