Lab 06: Spark DataFrames and Datasets

Department of Computer Science and IT, La Trobe University

As usual, we have provided a skeleton scala file `lab06.scala` to enter your exercise solutions, and we recommend you use it for your own reference. A video can be found in ECHO360 to give you an introduction and some tips.

Task A: DataFrames and Datasets

In the previous lab, we primarily dealt with data stored in RDDs. One annoying thing that you might have noticed about RDDs is that accessing data fields is quite annoying and difficult to read—code like `_._2(13)` is pretty confusing. It would be much nicer to refer to fields by name instead. Luckily for us, the Spark team has developed DataFrames and Datasets to do this!

DataFrames and Datasets are very similar, and for the most part behave the same way. The key difference is that Datasets allow you to access rows as instances of a custom class, which can be more convenient to work with. In this task we will practise creating and inspecting DataFrames and Datasets.

1. The file `Input_data/TaskA/fridges.csv` contains efficiency data for many different refrigerators sold in Australia and New Zealand. Open the file in a text editor to get a feel for what the data looks like. Put this onto HDFS with the following commands:

```
$ hdfs dfs -mkdir lab_6
$ hdfs dfs -put Input_data/ lab_6/
```

2. Start a `spark-shell` in the same directory as this lab document.

3. While looking at the data you should have noticed that there are four columns: `brand`, `country`, `model`, and `efficiency`. We can define a `case class` for representing fridge records as follows:

```
scala> case class Fridge(brand: String, country: String, model: String,
                         efficiency: Double)
```

4. Now that we have a `Fridge` class which describes the fields of our data, we can use Spark to automatically generate a schema from it.

```
scala> import org.apache.spark.sql.Encoders
scala> val fridgeSchema = Encoders.product[Fridge].schema
```

5. Using our data schema, we can load the fridge data as a DataFrame.

```
scala> val fridgeDF = spark.read.option("header", "true").schema(fridgeSchema).
  csv("lab_6/Input_data/TaskA/fridges.csv")
```

6. Due to Scala's type system, accessing fields by name is a little bit clunky with DataFrames. Each element of `fridgeDF` is a `Row` object. Fields of a `Row` object can be retrieved using the `getAs` method.

```
scala> fridgeDF.map(f => f.getAs[String]("brand")).first
```

7. To make things a bit nicer, Spark provides an alternative to `DataFrame` called `Dataset`. You can convert a `DataFrame` into a `Dataset` using the `as` method.

```
scala> val fridgeDS = fridgeDF.as[Fridge]
```

   Each element of `fridgeDS` is a `Fridge` object, which makes the code quite readable.

```
scala> fridgeDS.map(f => f.brand).first
```

Exercise 1. Write a `case class` called Fruit which corresponds to the data in `lab_6/TaskA/fruit.csv`. Make sure that you select appropriate types for each of the three columns (hint: each column should have a different type).

Inspecting DataFrames and Datasets

Debugging is very difficult unless you can inspect the data being handled by your program. The `printSchema` and `show` methods are two very useful inspection tools at your disposal when working with Spark DataFrames and Datasets.

1. All DataFrames have schemas which describe the format of the data they contain. You can check the schema of a DataFrame using the `printSchema` method.

```
scala> fridgeDF.printSchema
```

   So, unlike RDDs, DataFrames must carry information about the names and types of the data fields. This allows Spark to serialise your data more efficiently to reduce network traffic.

2. Print the schema for the `fridgeDS` Dataset. The result should be the same as for the DataFrame.

3. When working with RDDs, we would often use the `take()` method to peek at the data they contain. That approach would return a Scala array, which is sometimes difficult to read on the screen. DataFrames/Datasets provide a nicer method called `show()` which displays data in a clearly formatted table instead.
   Use the following line of code to show 5 rows from the `fridgeDF` DataFrame.

   ```
   scala> fridgeDF.show(5)
   ```

4. Use the `show` method to print 5 lines from the `fridgeDS` Dataset. Once again, the result should be the same as for the DataFrame.

Task B: Saving DataFrames to output files

Text-based CSV storage

CSV (comma separated value) files are a very simple, human-readable way to store data. Spark makes writing DataFrames to CSV files very easy. Run the following code to save `fridgeDF` in CSV format to the HDFS directory `lab_6/task-b/fridges.csv`.

```
scala> fridgeDF.write.mode("overwrite").csv("lab_6/task-b/fridges.csv")
```

The `.mode("overwrite")` part tells Spark to overwrite the output data directory if it already exists. This means that you can run the above code multiple times without crashing. Once this is done, open Hue and click the HDFS button on the top left, then find and view the contents of the CSV. This is demonstrated in the helper video in ECHO360 in case you're a little rusty.

If instead we want to write to local storage, we have to specify the entire path as below:

```
scala> fridgeDF.write.mode("overwrite").csv("file:///root/labfiles/lab_6/task-b/fridges.csv")
```

Binary data storage with Parquet

Although text-based formats like CSV are commonly used to exchange data, they do not make very efficient use of storage space. To achieve better performance, we would like to store data in a more efficient binary format. One such format is Parquet. An especially cool property of Parquet is that it stores data column-wise. This means that if you query a single column, then only the data for that column needs to be loaded!

1. Saving a DataFrame/Dataset in the Parquet format is very easy.

   ```
   scala> fridgeDF.write.mode("overwrite").parquet("lab_6/task-b/fridges.parquet")
   ```

All of the fridge data will be saved into the `lab6/task-b/fridges.parquet` HDFS directory. You may see a warning about the logger when you do this, but this can be ignored.

2. Open the `lab_6/task-b/` folder in Hue and compare the size of the `fridges.parquet` directory with the `fridges.csv` directory (Right Click > Summary in the file browser). Even though they both store the same data, you should see that the Parquet format uses much less disk space.

3. Parquet saves the schema with the data. This means that you do not need to specify the schema explicitly when reading Parquet data into a DataFrame.

```
scala> val fridgeParquetDF = spark.read.parquet("lab_6/task-b/fridges.parquet")
```

   Use `printSchema` for `fridgeParquetDF` to see for yourself that the schema was read in with the data.

Exercise 2. Convert `fridgeParquetDF` into a Dataset and show the first 10 rows.

Task C: DataFrame API basics

Working with DataFrames/Datasets is like a cross between RDDs and Hive. You can refer to column names like in Hive, but you can also perform very general computation like with RDDs. We will only cover some of the available functions in this lab, but full documentation for the API is available online.

For this task we will be using data about the countries of the world, available in the folder `Input_data/TaskC`. The data is stored in multiple JSON files, and is based on `http://country.io/data/`. Each file maps the ISO2 country code for each country to a particular value, depending on the file:

File	Value
`capital.json`	The full name of the country's capital city.
`continent.json`	The continent that the country belongs to.
`currency.json`	The official currency of the country.
`names.json`	The full name of the country.
`phone.json`	The country's phone code.

Top European currencies

In order to demonstrate Spark's DataFrame API, we will write code which ranks European currencies in order of the number of countries that use them. This will require filtering, joining, grouping, counting, and ordering the data.

Filtering and selecting

1. Read the continent data into a DataFrame.

```
scala> val continentDF = spark.read.json("lab_6/Input_data/TaskC/continent.json")
```

Spark is able to infer the schema automatically from the JSON, just like Parquet files.

2. Apply a filter to include only European countries.

```
scala> val filteredDF = continentDF.filter($"continent" === "EU")
```

3. Use show() to inspect the first 5 rows of the `filteredDF` DataFrame. You should see that each row has "EU" (Europe) as the continent.

4. The `continent` column is no longer useful to us, so we can remove it using `select`. This should remind you of `SELECT` in SQL/Hive.

```
scala> val europeDF = filteredDF.select($"countryCode")
```

5. Use show() to inspect the first 5 rows of the `europeDF` DataFrame. Confirm that the `continent` column is not present.

Joining

1. Read currency data from HDFS into another DataFrame called `currencyDF`.

2. We will now join the data from `europeDF` and `currencyDF` into a single DataFrame, using `countryCode` as the join key.

```
scala> val europeCurrDF = europeDF.join(currencyDF, "countryCode")
```

3. Use show() to inspect the first 5 rows of the `europeCurrDF` DataFrame. You should see that the DataFrame contains the currency type for each country in Europe.

Grouping

1. Group countries together by currency.

```
scala> val grouped = europeCurrDF.groupBy($"currency")
```

2. `grouped` is not a DataFrame, it is something called a RelationalGroupedDataset. This means that you can't use methods like `show` or `printSchema` on it. Try for yourself.

3. In order to get a DataFrame from `grouped`, we must aggregate the values in each group. In this case we will simply count the number of countries which use each currency.

```
scala> val currCountDF = grouped.count()
```

4. `currCountDF` is a DataFrame. Use show to inspect the first 5 rows. You should see that there is no longer a `countryCode` column, but a `count` column has been added by the aggregation.

When using RDDs, doing a `groupByKey` followed by a count would be a big "no-no", since all of the data would literally be grouped before being counted. That means a lot of data shuffle! When using RDDs you should use `reduceByKey` instead to do this type of counting. However, when using structured data with Spark SQL (DataFrames/Datasets), `groupBy(...).count()` is optimised efficiently and can be used without losing performance.

Ordering

1. Order the European currencies in `currCountDF` from highest number of utilising countries to lowest.

```
scala> val currSortedDF = currCountDF.orderBy($"count".desc)
```

2. Show the first 5 rows of `currSortedDF`. What are the two currencies used by the most European countries?

Putting everything together

Once you get used to working with Spark's DataFrame API you can simply chain the calls together into one big query. Here's what the combined code for finding the top 5 European currencies looks like.

```
scala> val continentDF = spark.read.json("lab_6/Input_data/TaskC/continent.json")
scala> val currencyDF = spark.read.json("lab_6/Input_data/TaskC/currency.json")
scala> continentDF.
  filter($"continent" === "EU").
  select($"countryCode").
  join(currencyDF, "countryCode").
  groupBy($"currency").
  count().
  orderBy($"count".desc).
  show(5)
```

Exercise 3. Using DataFrames, find the full name of every country in Oceania (continent "OC"). Show the first 10 country names in ascending alphabetical order.

Exercise 4. Using the `fridgeDF` DataFrame as input, calculate the average refrigerator efficiency for each brand. Order the results in descending order of average efficiency and show the first 5 rows. Hint: RelationalGroupedDataset has a method called `avg()` for calculating per-group averages. It works similarly to `count()`, except you must pass `avg()` the names of the columns to be averaged.

Task D: Using SQL with DataFrames

Spark also comes with a full SQL query parser. This means that you can write code similar to Hive within Spark, but you get the result as a DataFrame.

1. In order to use SQL in Spark, you must first register DataFrames that you want to use as tables. This is done by creating SQL temporary views with the `createOrReplaceTempView` function.

   ```
   scala> continentDF.createOrReplaceTempView("continents")
   ```

   Now you can reference `continentDF` from SQL code as `continents`.

2. Let's run an extremely simple query to select all unique continents.

   ```
   scala> val resultDF = spark.sql("SELECT DISTINCT continent from continents")
   ```

   The `spark.sql` function takes an SQL query string as input and returns a DataFrame which contains the result of the query.

3. Since the result is a DataFrame, you can use any method from the DataFrame API on it. For example, you can use the `show` method to look at the data.

   ```
   scala> resultDF.show
   ```

   You should see codes for the 7 continents printed out.

The implementation of SQL bundled with Spark is quite comprehensive, and fully documented at `https://spark.apache.org/docs/2.3.0/api/sql/`. Here's what "top European currencies" query from Task C looks like using SQL code in Spark:

```
scala> currencyDF.createOrReplaceTempView("currencies")
scala> spark.sql("""SELECT currency, COUNT(1) AS count FROM
  continents INNER JOIN currencies
  ON continents.countryCode = currencies.countryCode
  WHERE continent = 'EU'
  GROUP BY currency
  ORDER BY count DESC""").show(5)
```

The triple quotes (`"""`) allow us to write a string that goes over multiple lines.

A reason *not* to use SQL strings in Spark

A big downside to using SQL like this (as opposed to the DataFrame API) is that typos can't be caught at compile time. This means that bugs can be harder to find and fix.

1. Enter the following function containing a bad SQL query into the Spark shell. Type it exactly, including the deliberately misspelt word "CELECT".

```
scala> def function1() {
  spark.sql("CELECT continent from continents").show(5)
}
```

Scala will accept the code without complaining. However, if you try to call `function1()` you should get an `ParseException`, which is a runtime exception indicating that the SQL contains a syntax error.

2. Now try entering similarly bad code, but this time using the DataFrame API.

```
scala> def function2() {
  continentDF.celect($"continent").show(5)
}
```

This time you should get an error immediately, and `function2` will never be created. This is because Scala's type system allows it to catch the error as it tries to compile the code.

In general, avoid writing long SQL strings in Spark and try to use the DataFrame API instead.

Exercise 5. Redo Exercise 3, but this time using `spark.sql` instead of the DataFrames API. You can still use `show()` to print the results.

Task E: Manipulating columns

Adding a new column

1. Suppose that we want to calculate the fraction of the world's countries which belong to each continent. As a first step, we can count the number of countries in each continent by using the groupBy method.

```
scala> val countDF = continentDF.groupBy($"continent").count()
```

2. At this point, it seems as if we're stuck. That's because we have not yet learnt how to add new columns to a DataFrame yet. Luckily for us, the `withColumn` function makes it really easy to add new columns! The `withColumn` function takes two arguments—the name that you want to give the new column, and an expression describing how to calculate the values of the new column.
Here's how we can use `withColumn` to add a column called `fraction` which contains the fraction of countries which belong to each continent.

```
scala> val fractionDF = countDF.withColumn("fraction", $"count"/continentDF.count)
```

3. Use `show()` to print the 7 rows in `fractionDF`. You should see the fraction of countries belonging to each continent, represented as decimal values.

Renaming columns

Firstly, let's look at a scenario where not renaming columns causes problems. To demonstrate this, we will find all pairs of countries that belong to the same continent.

1. We can find pairs of countries that belong to the same continent by performing a "self-join" (joining a DataFrame with itself).

```
scala> val countryPairDF = continentDF.as("df1").join(continentDF.as("df2"),
 $"df1.continent" === $"df2.continent" and $"df1.countryCode" < $"df2.countryCode")
```

2. Print the schema for `countryPairDF`. You should see something like the following:

```
scala> countryPairDF.printSchema
root
 |-- continent: string (nullable = true)
 |-- countryCode: string (nullable = true)
 |-- continent: string (nullable = true)
 |-- countryCode: string (nullable = true)
```

   Yikes, the DataFrame contains duplicate column names! **This is because the DataFrames used in the join had columns with the same name as each other.**

3. Duplicate column names are problematic since Spark won't know which column we mean when writing queries. For example, the following code will not work:

```
scala> countryPairDF.where($"continent" === "OC").show(5)
```

   The error message tells us exactly why it doesn't work: "Reference 'continent' is ambiguous". Spark doesn't know which `continent` column that we're talking about!

Now let's redo the the code properly, this time using column renaming to avoid the problem of ambiguous column names.

1. Firstly, let's print a row from `continentDF`. We do this to work out what order the existing columns are in.

```
scala> continentDF.show(1)
+---------+-----------+
|continent|countryCode|
+---------+-----------+
|       AF|         BW|
+---------+-----------+
```

   So the order is (continent, countryCode).

2. Now we can create DataFrames with the same data as `continentDF`, but with new column names using the `toDF` function. The order of the column names will correspond to the column order we found in the previous step. Note that `toDF` won't copy the data or anything like that, so it doesn't impact performance.

```
scala> val df1 = continentDF.toDF("continent1", "countryCode1")
scala> val df2 = continentDF.toDF("continent2", "countryCode2")
```

3. Join time!

```
scala> val countryPairDF = df1.join(df2,
  $"continent1" === $"continent2" and $"countryCode1" < $"countryCode2")
```

4. If you print out the schema for `countryPairDF` now, you should see that all of the columns have unique names. This means that we can perform further queries without issue since there's no longer any ambiguity.

```
scala> countryPairDF.where($"continent1" === "OC").show(5)
```

Removing columns

1. Earlier in the lab we saw how to remove columns with `select`. When using `select` we need to specify all of the column names that we want to *keep*. For example, if we want to keep all columns from `countryPairDF` except `continent2` we could write:

```
scala> val selectedDF = countryPairDF.select($"continent1", $"countryCode1",
  $"countryCode2")
```

2. Using `select` can be annoying when we have a lot of columns and only want to remove one of them. In this situation you can use `drop` instead.

```
scala> val droppedDF = countryPairDF.drop($"continent2")
```

3. Print the schemas for `selectedDF` and `droppedDF`. You should see that they are the same, and that neither of them have the `continent2` column.

Task F: User defined functions

A user defined function (UDF) is a piece of custom code which can be used to transform data in a DataFrame. You implement the behaviour of a UDF by writing a normal Scala function, and then use the `spark.udf.register` higher order function to register it as a UDF that Spark can use.

1. Let's implement a function which formats a number (of type `Double`) as a percentage (of type `String`).

```
scala> def toPercentage(x: Double): String = {
  "%.2f%%".format(x * 100)
}
```

2. Call the `toPercentage` function on the number 0.12345. The result should be "12.35%".

3. Create a UDF from the `toPercentage` function.

```
scala> val toPercentageUdf =
  spark.udf.register[String, Double]("toPercentage", toPercentage)
```

   Notice that we need to tell the `spark.udf.register` higher order function what type of value the function returns (in this case, String) and the type of any arguments (in this case, Double). The "toPercentage" specifies a name to associate with the UDF—this name can be used to call the function from SQL code.

4. Now we shall use our UDF to format the decimal fractions in `fractionDF` as nice percentage values.

```
scala> fractionDF.select($"continent", toPercentageUdf($"fraction")).show()
```

5. We can also use the UDF in SQL code using the name "toPercentage". Here's what the previous step looks like if we use SQL instead of the DataFrame API.

```
scala> fractionDF.createOrReplaceTempView("fractions")
scala> spark.sql("SELECT continent, toPercentage(fraction) FROM fractions").show()
```

Exercise 6. Using `toPercentageUdf`, add a new column to `fractionDF` called "percentage" containing the fraction as a formatted percentage string. Drop the original `fraction` column. Call the resulting DataFrame `percentageDF`. If you solve this exercise correctly, `percentageDF` should look like this (order is not important):

```
scala> val percentageDF = fractionDF.**TODO**
scala> percentageDF.show()
+---------+-----+----------+
|continent|count|percentage|
+---------+-----+----------+
|       NA|   41|    16.40%|
|       SA|   14|     5.60%|
|       AS|   52|    20.80%|
|       AN|    5|     2.00%|
```

```
|        OC|   27|    10.80%|
|        EU|   53|    21.20%|
|        AF|   58|    23.20%|
+---------+-----+----------+
```