Chapter 5 - Basic Structured Operations

DataFrame

- series of records (like rows in a table)
- of type Row
- number of columns (like columns in a spreadsheet) - each column represents a computation expression that can be performed on every individual record in the Dataset

Schemas

- define the name as well as the type of data in each column
- StructType made up of a number of StructFields
- Can contain other structTypes(Spark's complex type)
- If the types in the data (at runtime) do not match the schema, Spark will throw an error

Partitioning

- defines the layout of the DataFrame
- the Dataset's physical distribution allocation across the cluster
- can be based on values in a certain column, or nondeterministically

Schema-on-read:

- letting the data source define schema for you
- Best for ad-hoc

When doing a production Extract, Transform, and Load (ETL), better to define your schemas manually - CSV and JSON use untyped data sources, so schema inference can vary

StructFields

- have a name, type, and a flag
- Boolean flag specifies if that column can contain missing or null values
- Users can optionally specify metadata about that column (Spark uses in machine learning library)

Columns

- select, manipulate, and remove columns from DataFrames
- these operations are represented as expressions
- columns are logical constructions that simply represent a value computed on a per-record basis by means of an expression
- to have a real value for a column, must have a row; and to have a row, we need to have a DataFrame
- cannot manipulate an individual column outside the context of a DataFrame
- must use Spark transformations within a DataFrame to modify the contents of a column.
- Columns are not resolved until compared with those maintained in the catalog
- Column and table resolution happens in the analyzer phase
- if you want to **programmatically access columns**, you can **use the columns property** to see all columns on a DataFrame

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

- use the col method on the specific DataFrame to refer to a specific DataFrame's column
- useful when you are performing a join and need to refer to a specific column in one DataFrame that might share a name with another column in the joined DataFrame
- Added benefit - Spark does not need to resolve this column itself anymore (during the analyzer phase) because we did it for Spark

Expressions

- a set of transformations on one or more values in a record in a DataFrame
- function that takes as input one or more column names, resolves them, and applies more expressions to create a single value for each record in the dataset
- "single value" can actually be a complex type like a Map or Array

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

expr("someCol - 5") is the same transformation as performing *col("someCol") - 5*, or even *expr("someCol") - 5,* because Spark compiles these to a logical tree specifying the order of operations

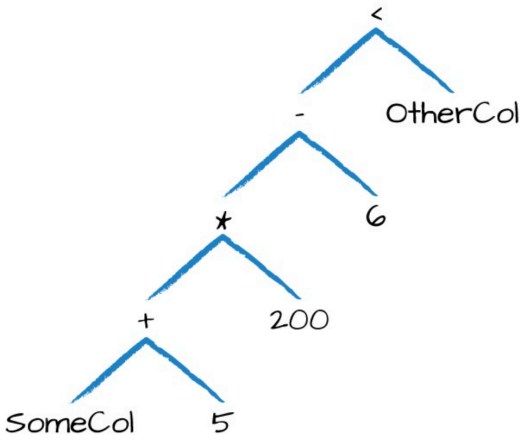

Figure 5-1. A logical tree

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

Is the same as:

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

The previous expression is actually valid SQL code, as well, just like you might put in a SELECT statement. This SQL expression and the previous DataFrame code compile to the same underlying logical tree prior to execution. You can write your expressions as DataFrame code or as SQL expressions and get the exact same performance characteristics.

Rows

- Each row in a DataFrame **is a single record**
- Spark represents this record as an object of type Row
- Spark manipulates Row objects using column expressions in order to produce usable values
- Row objects internally represent arrays of bytes -byte array interface is never shown to users because we only use column expressions to manipulate them
- Commands that return individual rows to the driver will always return one or moreRow types when we are working with DataFrames

DataFrame Transformations

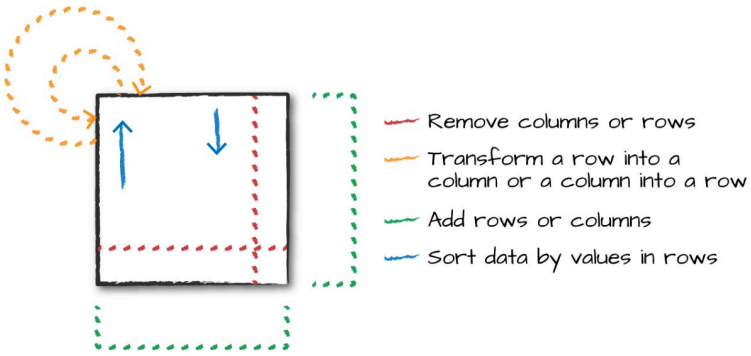

Figure 5-2. Different kinds of transformations

CreateOrReplaceTempView - Creates a new temporary view using a SparkDataFrame in the Spark Session. If a temporary view with the same name already exists, replaces it.

Create DataFrames on the fly by taking a set of rows and converting them to a DataFrame

Useful methods:

- **select** method when you're working with columns or expressions
- **selectExpr** method when you're working with expressions in strings
- Naturally some transformations are not methods on columns; the group of functions found in the org.apache.spark.sql.functions package

expr is the most flexible reference

- can refer to a plain column or a string manipulation of a column

Change the column name, and then change it back by using the AS keyword and then the alias method on the column:

```
df.select(expr("DEST_COUNTRY_NAME AS destination")).show(2)
```

Because select followed by a series of expr is such a common pattern, Spark has a shorthand - **selectExpr**

SelectExpr

- is a simple way to build up complex expressions that create new DataFrames
- Can add any valid non-aggregatingSQL statement, and as long as the columns resolve, it will be valid

Example adds a new column within Country to our DataFrame that specifies whether the destination and origin are the same:

```
df.selectExpr("*",# all original columns"(DEST_COUNTRY_NAME =
ORIGIN_COUNTRY_NAME) as withinCountry")\.show(2)
```

Converting to Spark Types (Literals)
Literals - pass explicit values into Spark that are just a value (rather than a new column) - a constant value, or something needed for later comparisons

withColumn

- Function takes two arguments: the column name,
- and the expression that will create the value for that given row in the DataFrame
- can also rename a column this way

withColumnRenamed - rename the column with the name of the string in the first argument to the string in the second argument

Must set case sensitivity by setting config

Filter rows

- must create an expression that evaluates to true or false
- filter out the rows with an expression that is equal to false
- most common way to do this with DataFrames is to create either an expression as a String or build an expression by using a set of column manipulations
- There are two methods to perform this operation: you can use where or filter and they both will perform the same operation and accept the same argument types when used with DataFrames
- Book uses where because SQL; however, filter is valid as well.

Random Splits:

- Method is designed to be randomized
- we will also specify a seed (just replace seed with a number of your choosing in the code block)
- It's important to note that if you don't specify a proportion for each DataFrame that adds up to one, they will be normalized so that they do

Concatenating and Appending Rows (Union):
DataFrames are immutable - users cannot append to DataFrames because that would be changing the DataFrame
To append to a DataFrame, you must union the original DataFrame along with the new DataFrame.
To union two DataFrames, you must be sure that they have the same schema and number of columns; otherwise, the union will fail

Sorting Rows:
Sort with either the largest or smallest values at the top of a DataFrame. Two equivalent operations to do this - **sort** and **orderBy** - that work the exact same way. They accept both column expressions and strings as well as multiple columns. The default is to sort in ascending order

Repartition and Coalesce:

- important optimization opportunity is to partition the data according to some frequently filtered columns, which control the physical layout of data across the cluster including the partitioning scheme and the number of partitions
- filtering by a certain column often, it can be worth repartitioning based on that column
- Coalesce will not incur a full shuffle, and will try to combine partitions. This operation will shuffle your data into five partitions based on the destination country name, and then coalesce them (without a full shuffle).

Spark maintains the state of the cluster in the driver. There are times when you'll want to collect some of your data to the driver in order to manipulate it on your local machine. *Collect* gets all data from the entire DataFrame, *take* selects the first N rows, and *show* prints out a number of rows.

The method toLocalIterator collects partitions to the driver as an iterator. Thismethod allows you to iterate over the entire dataset partition-by-partition in a serial manner.