Spark Basics

Jim Harner

10/4/2020

The `sparklyr` package provides an R frontend to Spark based on a `dplyr` interface to Spark SQL. Once these required packages are loaded, a Spark connection is established.

```
library(dplyr, warn.conflicts = FALSE)
library(sparklyr)

# start the sparklyr session locally, on the master container, or on AWS
master <- "local"
# master <- "spark://master:7077"
# master <- "yarn"
sc <- spark_connect(master = master)
# Location of SPARK_HOME using an R funtions
Sys.getenv("SPARK_HOME")
```

```
## [1] "/opt/spark"
```

Spark's home is located in `/opt/spark` on Unbuntu 18.04.

Spark is a general-purpose cluster computing system, which:

- has high-level APIs in Java, Scala, Python and R;

- supports multi-step data pipelines structured as directed acyclic graphs (DAGs);
- supports in-memory data sharing across DAGs allowing different jobs to work with the same data.

We can use `bash` to determine the Spark version:

```
# echo $SPARK_HOME
# The following command displays the Spark version installed:
spark-submit --version 2>&1 | grep -v "SLF4J:"
```

```
## Welcome to
##       ____              __
##      / __/__  ___ _____/ /__
##     _\ \/ _ \/ _ `/ __/  '_/
##    /___/ .__/\_,_/_/ /_/\_\   version 3.0.0
##       /_/
##
## Using Scala version 2.12.10, OpenJDK 64-Bit Server VM, 1.8.0_265
## Branch HEAD
## Compiled by user ubuntu on 2020-06-06T11:32:25Z
## Revision 3fdfce3120f307147244e5eaf46d61419a723d50
## Url https://gitbox.apache.org/repos/asf/spark.git
## Type --help for more information.
```

Spark provides a unified framework to manage big data processing with a variety of data sets that are diverse in nature, e.g., text data, graph data, etc., as well as the source of data (batch vs. real-time streaming data).

Spark supports a rich set of higher-level tools including:

- *Spark SQL* for running SQL-like queries on Spark data using the JDBC API or the Spark SQL CLI. Spark SQL allows users to extract data from different formats, (e.g., JSON, Parquet, or Hive), transform it, and load it for *ad-hoc* querying, i.e., ETL.
- *MLlib* for machine learning, including classification, regression, clustering, collaborative filtering, dimensionality reduction, and the underlying optimization algorithms. MLlib uses the DataFrame API, which is built on the Spark SQL engine.
- *Structured Streaming* for real-time data processing. Spark streaming uses a fault-tolerant stream processing engine built on the Spark SQL engine. Thus, you can express your streaming computation the same way you would express a batch computation on static data. Using the DataFrame API, the Spark SQL engine will take care of running the analysis incrementally and it continuously update the final result as streaming data continues to arrive.

The SparkR package is another R frontend to Spark, which is officially supported as part of the Spark distribution. However, we will focus on the sparklyr since it ties into the larger tidyverse suite of packages.

5.1 Sparklyr Basics

The `sparklyr` package is being developed by RStudio. It is undergoing rapid expansion. See RStudio's sparklyr for information.

The `sparklyr` R package provides a `dplyr` backend to Spark. Using `sparklyr`, you can:

- filter and aggregate Spark DataFrames and bring them into R for analysis and visualization;
- develop workflows using `dplyr` and compatible R packages;
- write R code to access Spark's machine learning library, MLlib;
- create Spark extensions.

Using `sparklyr`, connections can be made to local instances or to remote Spark clusters. In our default case the connection is to a local connection bundled in the `rstudio` container. Optionally, a master and 3 worker nodes are available in `rspark`.

The `sparklyr` library is loaded in the setup above and a Spark connection is established. The Spark connection `sc` provides the connector to Spark.

5.1.1 dplyr

The `dpyr` verbs, e.g., `mutate`, `filter`, can be used on Spark DataFrames. A more complete discussion is given in Section 5.2.

We will use the `flights` data in the `nycflights13` package as an example. If its size becomes an issue, execute each chunk in sequence in notebook mode.

```
library(nycflights13)
str(flights)
```

```
## tibble [336,776 x 19] (S3: tbl_df/tbl/data.frame)
##  $ year          : int [1:336776] 2013 2013 2013 2013 2013 2013 2013 2013 2013 2013 ...
##  $ month         : int [1:336776] 1 1 1 1 1 1 1 1 1 1 ...
##  $ day           : int [1:336776] 1 1 1 1 1 1 1 1 1 1 ...
##  $ dep_time      : int [1:336776] 517 533 542 544 554 554 555 557 557 558 ...
```

```
##  $ sched_dep_time: int [1:336776] 515 529 540 545 600 558 600 600 600 600 ...
##  $ dep_delay     : num [1:336776] 2 4 2 -1 -6 -4 -5 -3 -3 -2 ...
##  $ arr_time      : int [1:336776] 830 850 923 1004 812 740 913 709 838 753 ...
##  $ sched_arr_time: int [1:336776] 819 830 850 1022 837 728 854 723 846 745 ...
##  $ arr_delay     : num [1:336776] 11 20 33 -18 -25 12 19 -14 -8 8 ...
##  $ carrier       : chr [1:336776] "UA" "UA" "AA" "B6" ...
##  $ flight        : int [1:336776] 1545 1714 1141 725 461 1696 507 5708 79 301 ...
##  $ tailnum       : chr [1:336776] "N14228" "N24211" "N619AA" "N804JB" ...
##  $ origin        : chr [1:336776] "EWR" "LGA" "JFK" "JFK" ...
##  $ dest          : chr [1:336776] "IAH" "IAH" "MIA" "BQN" ...
##  $ air_time      : num [1:336776] 227 227 160 183 116 150 158 53 140 138 ...
##  $ distance      : num [1:336776] 1400 1416 1089 1576 762 ...
##  $ hour          : num [1:336776] 5 5 5 5 6 5 6 6 6 6 ...
##  $ minute        : num [1:336776] 15 29 40 45 0 58 0 0 0 0 ...
##  $ time_hour     : POSIXct[1:336776], format: "2013-01-01 05:00:00" "2013-01-01 05:00:00" ...
```

The `flights` R data frame is a tibble, which allows large data to be displayed. This data frame has the date of departure, the actual departure time, etc. See the package documentation for variable definitions.

The `copy_to` function copies an R `data.frame` or `tibble` to Spark as a Spark SQL table. The resulting object is a `tbl_spark`, which is a `dplyr`-compatible interface to the Spark DataFrame.

```
flights_tbl <- copy_to(sc, nycflights13::flights, "flights", overwrite = TRUE)
flights_tbl
```

```
## # Source: spark<flights> [?? x 19]
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      517            515         2      830            819
##  2  2013     1     1      533            529         4      850            830
##  3  2013     1     1      542            540         2      923            850
##  4  2013     1     1      544            545        -1     1004           1022
##  5  2013     1     1      554            600        -6      812            837
##  6  2013     1     1      554            558        -4      740            728
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
##  9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
## # ... with more rows, and 11 more variables: arr_delay <dbl>, carrier <chr>,
## #   flight <int>, tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>,
## #   distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

```
src_tbls(sc)
```

```
## [1] "flights"
```

By default, the `flights` Spark table is cached in memory (`memory = TRUE`), which speeds up computations, but by default the table is not partitioned (`repartition = 0L`) since we are not running an actual cluster by default. See the `copy_to` function in the `sparklyr` package for more details.

The Spark connection should be disconnected at the end of a task.

```
spark_disconnect(sc)
```