SJK006 - Master in Intelligent Systems

Spark - Data Frames

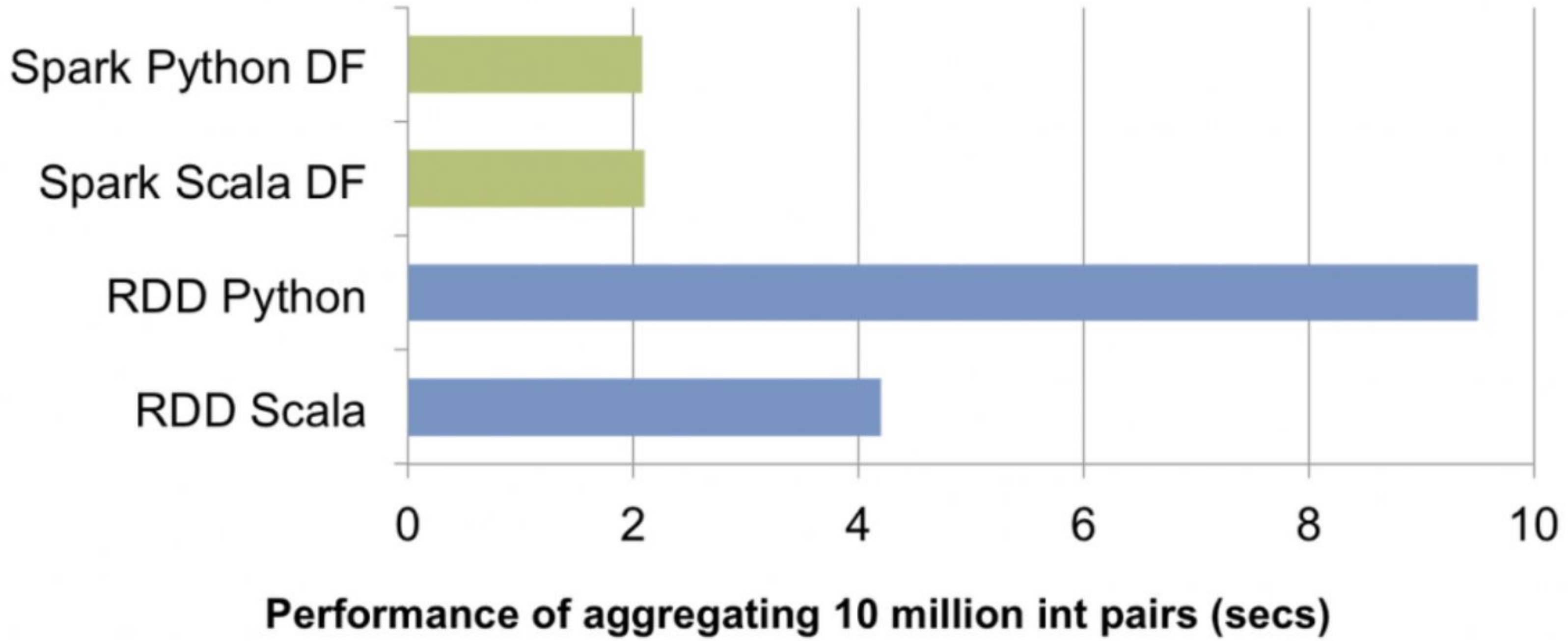
Analysing dynamic data

RDD or DataFrames?

There are two main APIs in Spark for working with data

- RDDs (Spark 1)
  - Oriented to unstructured data (text)
  - Low-level functional transformations (map, reduce, ...)
- DataFrames (Spark 2 onwards)
  - Semi-structured and structured data (with schema)
  - High-level transformations (Pandas and SQL style)
  - They allow more optimizations (more speed, less space)

RDD or DataFrames?



Source: https://towardsdatascience.com/deep-learning-with-apache-spark-part-1-6d397c16abd

RDD or DataFrames?

More than 90% of Spark applications use DataFrames and/or Spark SQL

Working with DataFrames

- Load data: from a file, database….
- Transform and get results
- Further processing: Apply machine learning/MLP algorithms, …
- Save the result

Before starting…

- Create a session:

```
from pyspark.sql import SparkSession
spark = SparkSession \
    .builder \
    .getOrCreate()
```

Load data from a file

- Read a CSV with a header:

```
df = spark.read.format('csv') \
    .options(header='true', inferschema='true') \
    .load('file.csv')
```

- To see the DataFrame schema:

```
df.printSchema()
```

Load data from a file

- Read a CSV without a header:

```
df = spark.read.format('csv') \
    .options(header='false', inferschema='true') \
    .load(file.csv') \
    .toDF('column1', 'column2', ...)
```

Read from other sources

- Many formats available
- See documentation at https://spark.apache.org/docs/latest/sql-data-sources.html

DataFrame transformations

- SQL-like primitives are included:
  - `select`
  - `filter`
  - `groupBy`
  - `agg`
  - `orderBy`
- Or you can use simply SQL directly on DataFrames
- Or even the Pandas API

DataFrame transformations

- To use Spark SQL, you must create a *view*: a name for the DataFrame that allows it to be referenced in queries
- There are two types of views
    - Local temporary views: associated with the session
    - Global temporary views: visible in all sessions

select()

- To select specific columns:

```
df.select("column1")
```

- It also allows to execute functions on columns:

```
from pyspark.sql.functions import round

df.select(round("column1"))
```

- Be careful: The functions are not the "normal" Python-native ones, they work only with DataFrames columns
- The pyspark.sql.functions package contains the predefined functions

`select()`

- In general it is convenient to give simple names to columns (aliases)

```
df.select(round("column1")
           .alias("rounded"))
```

- `alias` can be applied to any column expression (in the example above, to a function)

In Spark SQL

- The equivalent to the previous code would be:

```
# Create a temporary view
df.createTempView("view")


# Run the query
spark.sql("select round(columna) as rounded
from view"))
```

`withColumn()`

- To transform a column or create a new one, use `withColumn`()

```
df.withColumn("rounded", round("column1"))
df.withColumn("doubled", df.column1 * 2))
```

- These will return the entire DataFrame with the modified or added column

Spark SQL functions

With Spark DataFrames it is **not possible to use standard Python functions**, only those contained in the package `pyspark.sql.functions`

Later we will see how to create our own functions (UDFs)

The most common mathematical and statistical functions, and other useful functions, are implemented.

Spark SQL functions

- Example: `when().otherwise()`

```
from pyspark.sql.functions import when

withColumn("conditional",

   when(df.column2 > 0, df.column1 * 3)\

      .otherwise(df.column1 / 3))
```

- In SQL this is not a function:

```
SELECT CASE WHEN column2 > 0 THEN df.column1 * 3 ELSE df.columna1
/ 3 END AS conditional
```

filter()

- Removes rows that do not fulfill a condition

```
df.filter(df.value > 7)

df.filter("value > 7")
```

- In Spark SQL:

  ```
  spark.sql("select * from vista where valor > 7"))
  ```

groupby()

- Groups rows whose values match the specified columns
- The main use is to aggregate the data of each group

```
df.groupBy("columna1")

df.groupBy(["columna1", "columna2", ...])
```

Aggregation

- Typical aggregation functions can be used directly on a column or a group

```
df.column1.sum()

df.groupBy("column1").count()

df.groupBy("column1").avg()
```

agg()

- General aggregation function
- Allows you to calculate an aggregate value (`sum`, `avg`, `count`, …) on a column or the groups created by `groupBy`
- Aggregation functions must be Spark functions

```
df.groupBy("column1").agg(mean(df.column2))

df.groupBy("column1")

  .agg({"column2": "mean"})


df.groupBy("column1")

  .agg(mean(df.column2), max(df.column3))

df.groupBy("column1")

  .agg({"column2": "mean", "column3": "max"})
```

In Spark SQL

The equivalent to the last example would be

```
spark.sql("select avg(column2), max(column3)
from view group by column1")
```

orderBy()

- Sorts by one or more columns, ascending or descending

```
df.orderBy("column1")

df.orderBy("column1", ascending=False)

df.orderBy("column1", "column2")

df.orderBy("column1", "column2",
                  ascending=[False, True])
```

In Spark SQL

The equivalent to the last example would be

```
spark.sql("select * from view order by column1
desc, column2 asc")
```

User-defined functions (UDFs)

- The functions that can be applied on DataFrames are those defined in `pyspark.sql.functions`
- In order to use a 'normal' function on a DataFrame, it is necessary to encapsulate it as a **UDF**
- You must specify the type of data that the function returns. Data types are available in the `pyspark.sql.types package`
- Note that performance will be worse than using native Spark functions

User-defined functions (UDFs)

```
def to_km(n):
  return str(n/1000) + " km"

from pyspark.sql.functions import udf
from pyspark.sql.types import StringType

to_km_udf = udf(a_km, StringType())
df.withColumn("km_distance",
              to_km_udf(df.column1))
```

In Spark SQL

You must register the function in the SQL context, giving it a name to be able to use it in queries:

```
spark.udf.register("to_km_udf", a_km_udf)
```

The equivalent to the last example would be

```
spark.sql("select to_km_udf(column1) as
distancia_km from vista")
```

Interoperability with Pandas

- Pandas is the standard API in Python for using data frames
- Pandas data frames work only in memory → Not suitable for big data
- The API is different from the Spark API
- It is possible to get a Pandas DataFrame from a Spark DataFrame using `toPandas()`.
  - Be careful with the size of the data!
- Since PySpark 3.2, a direct implementation of the Pandas API ('pandas on Spark') has been included.
  - https://spark.apache.org/docs/latest/api/python/user_guide/pandas_on_spark/index.html

Questions?