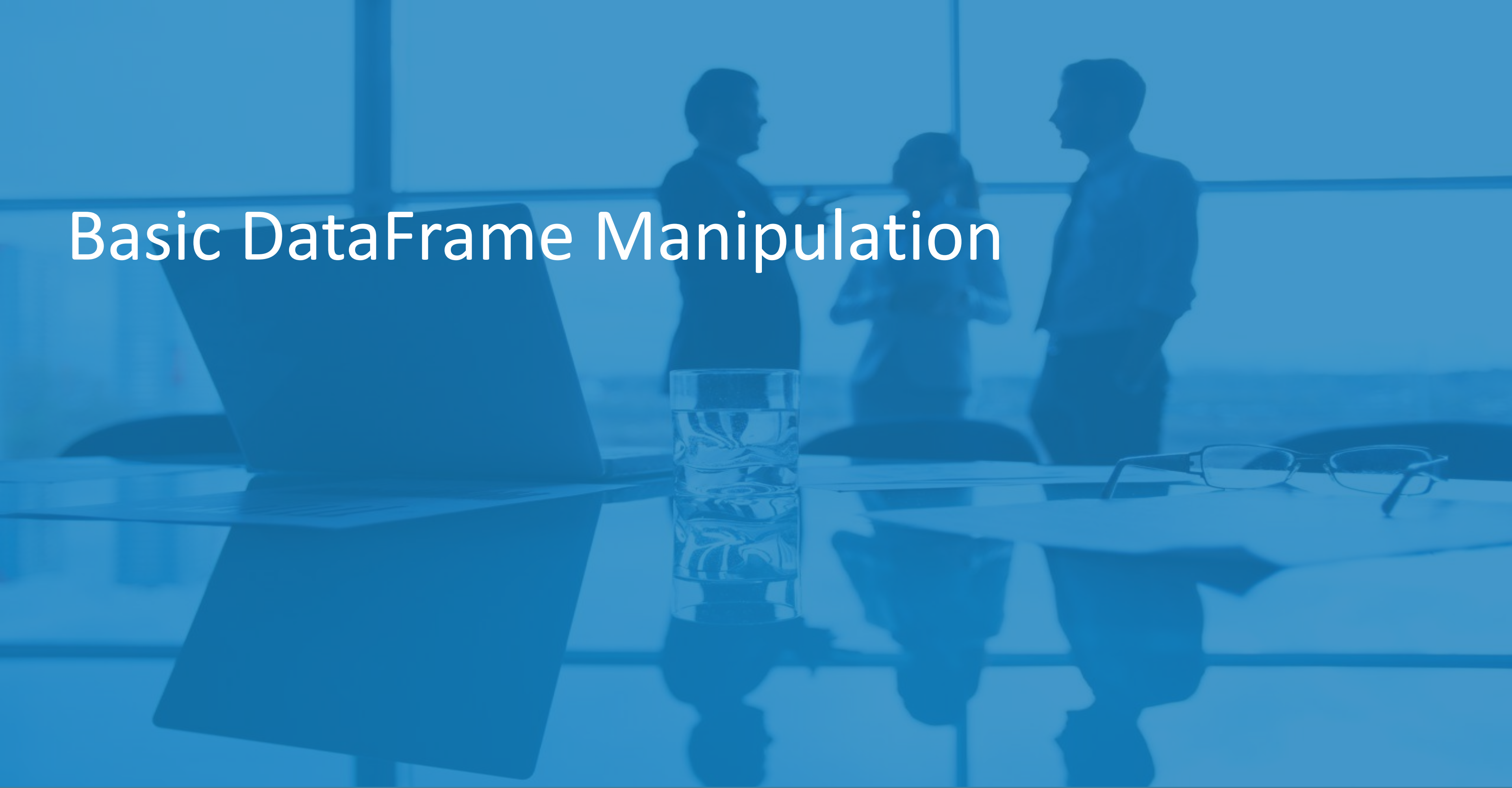
Antra

People · Process · Performance

Agenda

- Working with Databricks using Python

Basic DataFrame Manipulation



- A few most frequently used methods

df.count()	Count rows in a dataframe
display(df)	Display a dataframe
df.limit()	Used to display a small set of rows from a dataframe
df.select()	Select a subset of columns from a dataframe
df.distinct()/df.dropDuplicates (["column_name"])	Returns a new Dataset that contains only the unique rows from this Dataset
df.drop("column_name")	Remove columns from a dataframe
print(type(df))	Python method to get the datatypes,calls repr() underneath

Antra

Using cache() and persist() to speed up operations

- These two methods are equivalent(only without parameters)
- Without caching, every action requires Spark to read data from its source
- Caching moves the data into the memory of the workers for much faster access
- You can manually remove a cache by calling **unpersist()** on the dataframe

```
(df
  .cache()
  .count()
)
```

How to use the documentation

- Go to spark.apache.org
- Click “Documentation” and find the version you are looking for
- Hover on “API Docs” and select the language you are using
- Search using the search box in the left panel

Antra

show() vs display()

- show() and display() can both be used to print a dataframe

df.show(n=20, truncate=True)	display(df)
Part of core spark	Part of databricks notebooks
Parameters to truncate both rows and columns	No such options
Works only for dataframe/datasets	Works for some additional types
Prints result to the console in text format	• Download result as CSV • Databricks visualization • See up to 1000 records at a time

Creating temp views and query with SQL

- We can use **df.createOrReplaceTempView(“view_name”)** to create a temp view from the DF
- We can then run SQL queries on this table

Cmd 54

```
df.createOrReplaceTempView("tempview")
```

Cmd 55

```
%sql

SELECT * FROM tempview
```

- We can also run queries on DF directly with spark.sql()

Cmd 55

```
resultDF = spark.sql("SELECT * FROM df")
```


Antra

Fundamentals of Catalyst Optimizer

Shuffles

- Shuffle is triggered when data needs to move between executors.
- To perform a shuffle, spark
  - Convert the data to the **UnsafeRow**, commonly referred to as **Tungsten Binary Format**.
  - Write that data to disk on the local node
  - Send that data across the wire to another executor
    - The Driver decides which executor gets which piece of data.
    - Then the executor pulls the data it needs from the other executor's shuffle files.
  - Copy the data back into RAM on the new executor
- Spark can operate directly out of Tungsten, thus the shuffling is highly optimized and much faster than using JVM objects

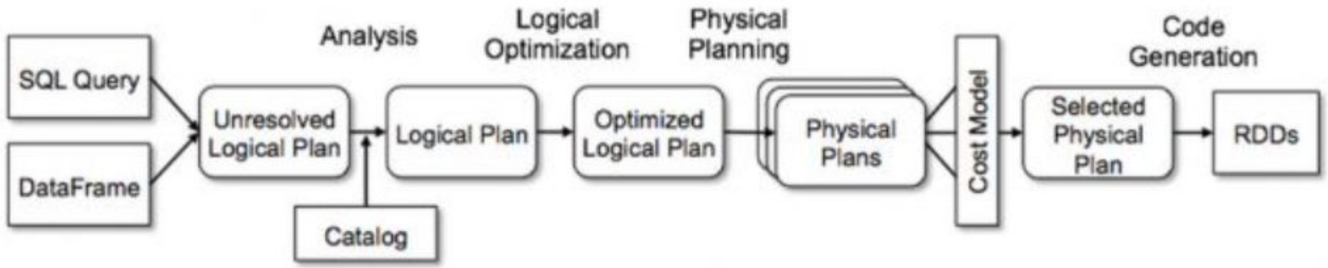
Pipelining

- The idea of executing as many operations as possible on a **single partition of data**.
- Once a single partition of data is read into RAM, Spark will combine as many **narrow transformations** as it can into a single **Task.**
- When a Shuffle becomes necessary, the **stage** is concluded and a pipeline is ended.

- During planning, Spark works backwards and check the dependency of each operation.
- Sometimes the shuffle files can be reused and thus allowing some transformations to be skipped.
- On top of this, we can also manually cache() the results of some operations

Query optimization

- Query optimization in Spark is done in four steps
  - Analyzing a logical plan to resolve references -> rule based
  - Logical plan generation and optimization -> rule based
  - Physical planning -> model based
    - Optimizer may generate multiple plans and compare them bases on cost
  - Code generation, which compiles parts of the query to Java Bytecode -> rule based



Logical plan

- Generated by the SparkContext.
- An abstract of all transformation steps that needs to be performed.
- **Unresolved Logical Plan**
  - This is the first step in creating a Logic Plan, no checks for column name, table names, etc.
  - Our code might be valid, but maybe the column name or table name is wrong.
- **Resolved Logical Plan**
  - Generated after the “Analyzer” has resolved/verified the unresolved logical plan by cross-checking from the “Catalog”(a repository where all the information about Spark table, DataFrame, DataSet will be present.)
- **Optimized Logical Plan**
  - Generate from the resolved logical plan by **Catalyst Optimizer**
  - Transformations are grouped together if possible
  - Order of joins are optimized
  - Move filter clause before project clause
  - ...

Physical Plan

- Physical plan specifies how our logical plan is going to be executed on the cluster.
- Different execution strategies are generated and compared using the "cost model"
  - Execution time and resource consumption are estimated and compared for each strategy
- After a physical plan is chosen, Spark's **Tungsten Execution Engine** will generate the code for the query which will be executed in a cluster.

Working with Columns & Rows

Creating a Column Object

- The **Column** class is an object that contains metadata of the column and transformations available to the column.
- Using PySpark, we can use indexing to create a column from a df easily
  - **column = df[“column_name”]**
- If we import sql.functions, we will have some additional options
  - **from** pyspark.sql.functions **import ***
    - **column = col(“requests”)** <- recommended
    - column = expr(“requests”)
    - column = lit(“requests”)

Renaming Columns

- There are multiple ways to rename columns in a DF
- The recommended way is to use
  - **df.withColumnRenamed("original", "new")**

Usage of Columns

- Many transformations can take a column object as input
- Suppose we want to sort a DF according to column in descending order, we can do
  - Sorted_df = df.orderBy(col(“column_name”).desc())
  - Sorted_df = df.orderBy(df[“column_name”].desc())

- Q: Why can’t we do df.orderBy(“column_name”).desc()?
  - orderBy() is a transformation which returns a new df, and there is no .desc() on a df, just like how you have to provide a column name when doing order by in SQL.

filter() & where()

- These functions are aliases of each other and are used to filter rows based on the given condition
  - **df.filter(col(“column_name”) == “apple”)**
    - Only return rows where the value of “column_name” column is string “apple”
- We can chain multiple filter() or where() together to break complex filters into pieces.
  - **df.filter(col(“column_name”) != “apple”).filter(col(“column_name”) !=”pear”)**
    - Only return rows where the value of “column_name” column is neither “apple” nor ”pear”

Rows

- A row in a dataframe
- The fields can be accessed in two ways
    - **row.key_name**
    - **row[“key_name”]**

```
>>> Person = Row("name", "age")
>>> Person
<Row('name', 'age')>
>>> 'name' in Person
True
>>> 'wrong_key' in Person
False
>>> Person("Alice", 11)
Row(name='Alice', age=11)
```


Antra

collect()

- df.collect() will return an array (Python list when using Pyspark) of Rows in the dataframe
- We can then loop through this list and access the content of each row.
  - Recall if we loop through a DF directly, we get the individual columns, not rows

```
rows = df.collect()

for row in rows:
  val1 = row["col_name_1"]
  val2 = row["col_name_2"]
  #some additional logic....
```

take(n)

- **df.take(n)** is the same as **df.limit(n).collect()**
- Returns the first n rows of the df as a list of rows.

Datetime Manipulation

- **unix_timestamp(**“col_name”,”pattern”**)** will convert the the column to Unix timestamp (in seconds) according to the pattern and return as a long
- **.withColumn(**“col_name”**,col.cast(**“new_type”**))** can be used to cast column to different type.
- Combining these, we have…

```
df.withColumn("col_name",unix_timestamp(col("col_name"),"datetime_pattern").cast("timestamp"))
```

- Available datetime_pattern(**SimpleDateFormat**) can be found at:
  - https://docs.oracle.com/javase/tutorial/i18n/format/simpleDateFormat.html

Datetime Manipulation

- After we have a timestamp column we can apply functions like year(), month()... to extract additional information from this column.
- Note year() and month() are both from pyspark.sql.functions and they both take a column as input.
-

```
(pageviewsDF
  .select( month(col("capturedAt")).alias("month"), year(col("capturedAt")).alias("year"))
  .distinct()
  .show()
)
```

▸ (2) Spark Jobs

```
+-----+----+
|month|year|
+-----+----+
|    3|2015|
|    4|2015|
+-----+----+
```

Aggregate Functions

- One way to do aggregation is by using .groupBy() first, then chain it with the aggregation we want
  - Returns **GroupedData**
  - GroupedData support a variety of aggregation methods.

```
print(
  pageviewsDF
    .groupBy( col("site") )
)
```

```
<pyspark.sql.group.GroupedData object at 0x7f701fc78f10>
```

```
display(
  pageviewsDF
    .groupBy( col("site") )
    .sum("requests")
)
```

Aggregate Functions

- Another way is to use SQL-like verbs
- We can specify the aggregation and columns we want in a select()
- There is no option to do a groupBy here since .select() works on a DF and not on GroupedData

```
(df
  .select( sum( col("1")), count(col("2")), avg(col("3")), min(col("4")), max(col("5")) )
  .show()
)
```