Learning Spark

UDFs and Cache

💻 Master's degree, Universitat de Lleida

/bluetab

an IBM Company

User Defined Functions

What?

- While Apache Spark has a large amount of built-in functions, the flexibility of Spark allows for data engineers (us) anad ataga scientists to define our own functions too, using User Defined Functions.
- Probably you knew them, because they are also compatible with some RDBMS.
- To use them, in Pyspark, we create a function in Python syntax, and wrap it with Pyspark method udf() or register it as udf and use it on DataFrame and SQL respectively.

User Defined Functions

Why?

- UDFs are used to extend the functions of the framework and reuse these functions on multiple DataFrame.
- Example of use:
  - Let's imagine we want to convert every first letter of a word in a name string (inside a column) to a capital case; PySpark build-in features don't have this function hence you can create it a UDF and reuse this as needed on many Data Frames.
- PySpark SQL provides several predefined common functions and many more new functions are added with every release. hence, It is best to check before you reinventing the wheel.
- When we are creating UDFs, we need to design them very carefully. Otherwise we will come across optimization and performance issues.
- LET'S CODE

User Defined Functions

Special Handling

- PySpark/Spark does not guarantee the order of evaluation of subexpressions meaning expressions are not guarantee to evaluated left-to-right or in any other fixed order.
- PySpark reorders the execution for query optimization and planning hence, AND, OR, WHERE and HAVING expression will have side effects.
- BUT, sometimes, in justified occasions, we will need to make an execution in a specified order.
- For example, in the next example, if there's some value in "name" column which is null, the execution can fail:

```
spark.sql("select convertUDF(name) as Name from NAMES " + \
          "where name is not null and convertUDF(name) like '%John%'") \
     .show(truncate=False)
```

- It's always best practice to check for null inside a UDF function rather than checking for null outside.
- In any case, if we can't do a null check in UDF at lease use IF or CASE WHEN to check for null and call UDF conditionally.

Caching and Persistence of Data

- What is the difference? In Spark, none.
- persist() call provides more control over how and where our data is stored (memory or disk, serialized or unserialised…) than cache() call.
- Both contribute to better performances for frequently accessed DataFrames or tables.

Caching and Persistence of Data

DataFrame.cache()

- It will store as many of the partitions are in memory across Spark executors as memory allows.
- While a DF may be fractionally cached, partitions cannot be fractionally cached.
- However, if not all our partitions are cached, when we want to access the data again, the partitions that are not cached,. will have to be recomputed, slowing down our Spark job.
- When we cach o persist a DF, it’s is not cached until we invoke an action.

Caching and Persistence of Data

DataFrame.persist(StorageLevel.LEVEL)

- As here we can define the storage level, it provides more control over how our data is caches.

StorageLevel	Description
MEMORY_ONLY	Data is stored directly as objects and stored only in memory.
MEMORY_ONLY_SER	Data is serialized as compact byte array representation and stored only in memory. To use it, it has to be deserialized at a cost.
MEMORY_AND_DISK	Data is stored directly as objects in memory, but if there's insufficient memory the rest is serialized and stored on disk.
DISK_ONLY	Data is serialized and stored on disk.
OFF_HEAP	Data is stored off-heap. Off-heap memory is used in Spark for storage and query execution; see "Configuring Spark executors' memory and the shuffle service" on page 178.
MEMORY_AND_DISK_SER	Like MEMORY_AND_DISK, but data is serialized when stored in memory. (Data is always serialized when stored on disk.)

Caching and Persistence of Data

When?

- Common use cases for caching are scenarios where we will want to access a large data set repeatedly for queries or transformations. Some examples include:
  - DataFrames commonly used during iterative ML training
  - DF accessed commonly for doing frequent transformations during ETL or building data pipelines.

Caching and Persistence of Data

When not?

- Not all use cases distaste the need to cache. Some scenarios that may not warrant caching our DF include:
  - DF that are too big to fit in memory
  - An inexpensive transformation on a DF not requiring frequent use, regardless of size.
- As a general rule, we should use memory caching judiciously, as it can incur resource costs in serializing and deserializing, depending on the StorageLevel used.

/bluetab
an IBM Company

¡Gracias!

alba.lamas@bluetab.net

¡Síguenos!

https://bluetab.net/

https://www.linkedin.com/company/bluetab/