⇒ SQL UDF :-

Cmd 5

Markdown

User-Defined Functions

User Defined Functions (UDFs) in Spark SQL allow you to register custom SQL logic as functions in a database, making these methods reusable anywhere SQL can be run on Databricks. These functions are registered natively in SQL and maintain all of the optimizations of Spark when applying custom logic to large datasets.

At minimum, creating a SQL UDF requires a function name, optional parameters, the type to be returned, and some custom logic.

Below, a simple function named `sale_announcement` takes an `item_name` and `item_price` as parameters. It returns a string that announces a sale for an item at 80% of its original price.

Cmd 6

SQL

```sql
CREATE OR REPLACE FUNCTION sale_announcement(item_name STRING, item_price INT)
RETURNS STRING
RETURN concat("The ", item_name, " is on sale for $", round(item_price * 0.8, 0));

SELECT *, sale_announcement(name, price) AS message FROM item_lookup
```

⇒ To see details of a UDF :-

DESCRIBE FUNCTION EXTENDED <udf_name>

⇒ UDF persists b/w notebooks, jobs (environments).

⇒ UDFs exists as objects in the metastore, and is governed by same table, views ACLs.

Simple Control Flow Functions

Combining SQL UDFs with control flow in the form of `CASE` / `WHEN` clauses provides optimized execution for control flows within SQL workloads. The standard SQL syntactic construct `CASE` / `WHEN` allows the evaluation of multiple conditional statements with alternative outcomes based on table contents.

Here, we demonstrate wrapping this control flow logic in a function that will be reusable anywhere we can execute SQL.

Cmd 12

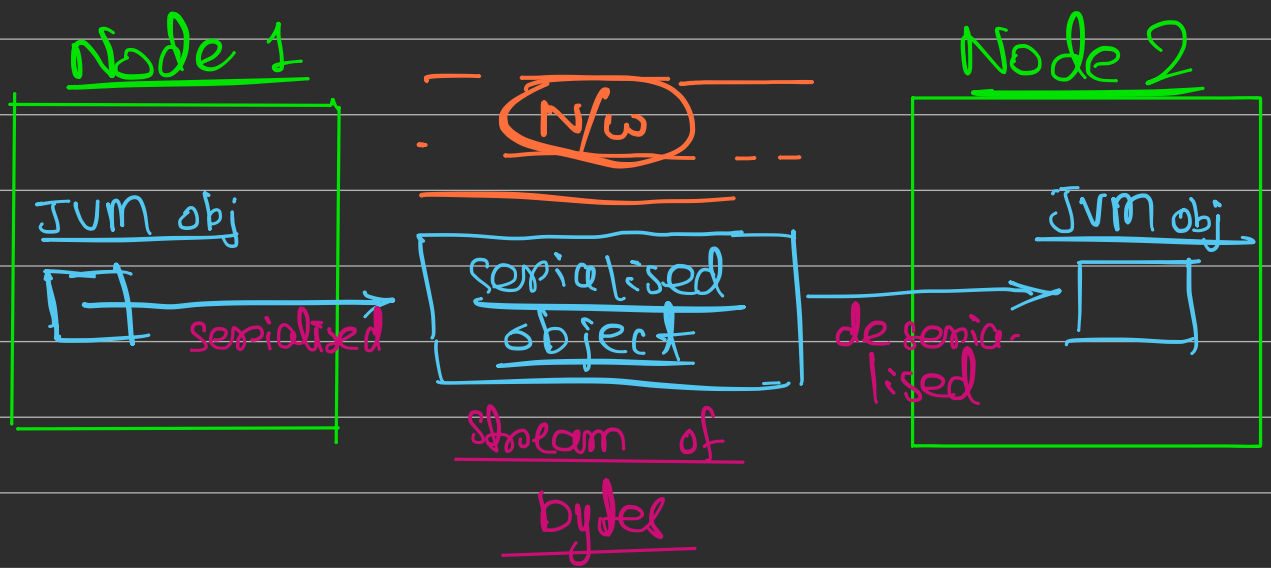
```sql
CREATE OR REPLACE FUNCTION item_preference(name STRING, price INT)
RETURNS STRING
RETURN CASE
  WHEN name = "Standard Queen Mattress" THEN "This is my default mattress"
  WHEN name = "Premium Queen Mattress" THEN "This is my favorite mattress"
  WHEN price > 100 THEN concat("I'd wait until the ", name, " is on sale for $", round(price * 0.8, 0))
  ELSE concat("I don't need a ", name)
END;

SELECT *, item_preference(name, price) FROM item_lookup
```

Serialisation :-

- RDDs, Dataframes, Datasets, are all called JVM objects. These JVM objects exists as partitions in the nodes.

- In a Cluster, there are multiple nodes, and each node has their own partitions. When Wide Transformations are applied, there is shuffling of data, i.e, the data in the form of JVM objects are transferred from one node to other, through network.

Now, JVM objects cannot be transferred through n/w as it is.
To transfer JVM objects b/w nodes, they need to be Serialized first.



Serialization :-

Process of conversion of a JVM object into a Stream of Bytes.

Deserialization :-

Construction of a JVM object from the received Stream of Bytes.

⇒ Python UDF :-

⇒ We created a normal function :-

```
def first_letter(var):
    return var[0]
```

⇒ This function can be registered as a PySpark UDF using udf() function as :-

```
first_letter_udf = udf(first_letter)
```

registered UDF — python function name

This serialises the function and sends it to executors to be able to transform DF records.

⇒ Applying UDF on columns :-

```
from pyspark.sql.functions import col
display(<DF>.select(first_letter_udf(col("<col_name>"))
```

first_letter_udf → UDF name
col("<col_name>") → column

⇒ Registering UDF to use in SQL :-

⇒ spark.udf.register() can be used to register a python function, to both as an SQL udf and a python UDF simultaneously.

Syntax :-

```
python_udf = spark.udf.register("sql_udf", first_letter)
```

python_udf → python udf registered name
"sql_udf" → name by which udf is registered in sql
first_letter → python function name

```
%python
<DF>.select(python_udf(col("<col_name>")))
```

```
%sql
SELECT sql_udf(username) FROM users;
```

↓

column on which it is to be applied.

Markdown

Use Decorator Syntax (Python Only)

Alternatively, you can define and register a UDF using Python decorator syntax. The `@udf` decorator parameter is the Column datatype the function returns.

You will no longer be able to call the local Python function (i.e., `first_letter_udf("annagray@kaufman.com")` will not work).

This example also uses Python type hints, which were introduced in Python 3.5. Type hints are not required for this example, but instead serve as "documentation" to help developers use the function correctly. They are used in this example to emphasize that the UDF processes one record at a time, taking a single `str` argument and returning a `str` value.

Cmd 18

```
# Our input/output is a string
@udf("string")
def first_letter_udf(email: str) -> str:
    return email[0]
```

→ Python type hints

Cmd 19

And let's use our decorator UDF here.

Cmd 20

```
from pyspark.sql.functions import col

sales_df = spark.table("sales")
display(sales_df.select(first_letter_udf(col("email"))))
```

Important Points :-

- A UDF
  - ⇒ Cannot be optimised by Catalyst optimiser.
  - ⇒ Is serialised and sent to executors.
  - ⇒ Row data is first deserialised and then UDF process deserialised data (objects). Then again, the result is serialized to Spark's native format

⇒ For Python UDFs, additional interprocess communication overhead occurs b/w Python Interpreter & Executor