SPIP: Python Stored Procedures

Authors: Allison Wang (allison.wang@databricks.com) Ruifeng Zheng (ruifengz@apache.org)
SPIP Shepard - Hyukjin Kwon (gurwls223@apache.org)

Q1. What are you trying to do? Articulate your objectives using absolutely no jargon.

Stored procedures are an extension of the ANSI SQL standard. They play a crucial role in improving the capabilities of SQL by encapsulating complex logic into reusable routines. Stored procedures are widely supported in many systems, such as PostgreSQL, MySQL, and Presto. We aim to extend Spark SQL by introducing support for stored procedures, starting with Python as the procedural language. This addition will allow users to execute procedural programs, leveraging programming constructs of Python to perform tasks with complex logic. Additionally, users will be able to persist these procedural routines in catalogs such as HMS for future reuse. By providing this functionality, we intend to empower Spark users to seamlessly integrate with Python routines within their SQL workflows.

Q2. What problem is this proposal NOT designed to solve?

This proposal is not designed to support stored procedures in languages other than Python. It also does not intend to provide Python APIs for stored procedures. These functionalities can be added in the future.

Q3. How is it done today, and what are the limits of current practice?

Currently, Spark does not support stored procedures. It only supports session-level user-defined functions in Scala, Java, Python, and R. For more complex tasks, developers often resort to multiple SQL commands or external tools. By introducing stored procedures, we can significantly increase the functionality of Spark SQL to handle procedural logic and make it more powerful to use. Also, this will offer a means for users to retain procedural logic in the catalog for future applications.

Q4. What is new in your approach and why do you think it will be successful?

We are going to introduce Spark SQL APIs for users to create and use stored procedures with Python as the procedural language. This will be successful because it merges Python's popularity and flexibility with Spark SQL, making it more powerful and versatile to use.

Q5. Who cares? If you are successful, what difference will it make?

Anyone who uses Spark will benefit from this feature. Compared with spark-submit, stored procedures offer a more user-friendly experience by enabling users to create and execute custom Python logic directly within SQL. This allows for a seamless integration into their existing workflows. Additionally, this feature will not only augment the power of Spark SQL but also pave the way for the potential support of more procedural languages in the future.

Q6. What are the risks?

There are a few risks. One is to ensure optimal performance when executing Python code within Spark SQL, maintaining SQL's robustness and stability, and avoiding unnecessary complexities that might burden the end-users. Also, we need to ensure it remains user-friendly and straightforward to debug.

Q7. How long will it take?

We anticipate a development window of approximately four to six months, factoring in testing, feedback, and improvements. The introduction of this feature is planned for the next major Spark release.

Q8. What are the mid-term and final “exams” to check for success?

Mid-term: Demonstrating basic stored procedures in Spark SQL using Python for procedural logic.
Final exam: A comprehensive and seamlessly integrated stored procedure capability in Spark SQL

Appendix A. Proposed API Changes. Optional section defining APIs changes, if any. Backward and forward compatibility must be taken into account.

Appendix B. Optional Design Sketch: How are the goals going to be accomplished? Give sufficient technical detail to allow a contributor to judge whether it's likely to be feasible. Note that this is not a full design document.

CREATE PROCEDURE

```
CREATE [OR REPLACE] PROCEDURE [IF NOT EXISTS]
   procedure_name ( [ parameter [, ...] ] )
   [ characteristic [...] ]
    AS procedure_body

parameter
    [parameter_mode] parameter_name data_type [DEFAULT
default_expression] [COMMENT parameter_comment]

parameter_mode
   { IN | INOUT | OUT }

characteristic
  { LANGUAGE PYTHON |
    NOT DETERMINISTIC |
    COMMENT procedure_comment |
    MODIFIES SQL DATA }
```

CALL

```
CALL procedure_name( [ argument [, …] ] )
```

DESCRIBE PROCEDURE

```
{ DESC | DESCRIBE } PROCEDURE [ EXTENDED ] procedure_name
```

DROP PROCEDURE

```
DROP PROCEDURE [IF EXISTS] procedure_name
```

SHOW PROCEDURES

```
SHOW PROCEDURES [ { FROM | IN } schema_name ]
                [ [ LIKE ] { procedure_name | regex_pattern } ]
```

Example

A simple example of Python stored procedure:

```
> CREATE OR REPLACE PROCEDURE area_of_rectangle
     (IN x INT, IN y INT, OUT area INT, INOUT acc INT)
    LANGUAGE PYTHON
    AS $$
      area = x * y
      acc = acc + area
    $$;

> CALL area_of_rectangle(5, 10, None, 10);
  area  acc
  ----  ---
    50   60
```

An example using spark session inside the Python stored procedure:

Python

```
// start of SQL commands, using SQL
CREATE OR REPLACE PROCEDURE my_etl_proc
   (IN table_name STRING, OUT status STRING)
LANGUAGE PYTHON
AS

$$  // <- start of procedure body, using Python

import pandas as pd
```

```
# Establish a remote Spark session
from pyspark.sql.session import SparkSession
spark = SparkSession.builder.getOrCreate()

# Load data
red_wine = spark.read.load("....csv", format="csv", sep=";",
inferSchema="true", header="true",).toPandas()

white_wine = spark.read.load("....csv",format="csv", sep=";",
inferSchema="true", header="true",).toPandas()

red_wine['is_red'] = 1
white_wine['is_red'] = 0

df = pd.concat([red_wine, white_wine], axis=0)

# Remove spaces from column names
df.rename(columns=lambda x: x.replace(' ', '_'), inplace=True)

spark.createDataFrame(df).write.mode("overwrite").format("delta").saveAs
Table(f"{table_name}")

num_rows = int(df.shape[0])
status = "success"

$$;  // <- end of procedure body

CALL my_etl_proc("wine_quality", None);
```

Another ML example:

Python

```
// start of SQL commands, using SQL
CREATE PROCEDURE compute_feature_importances(
      INOUT table_name STRING,
      OUT train_score DOUBLE,
```

```
      OUT test_score DOUBLE,
      OUT auc_score DOUBLE,
      OUT feature_importances MAP<STRING, FLOAT>)
LANGUAGE PYTHON
AS

$$  // <- start of procedure body, using Python

import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import roc_auc_score
from sklearn.model_selection import train_test_split

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

data = spark.sql(f"select * from {table_name}").toPandas()

X = data.drop(["quality"], axis=1)
y = data.quality

# Split out the training data
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.8,
random_state=123)

n_estimators = 10
model = RandomForestClassifier(n_estimators=n_estimators,
random_state=np.random.RandomState(123))
model.fit(X_train, y_train)

predictions_test = model.predict_proba(X_test)
train_score = float(model.score(X_train, y_train))
test_score = float(model.score(X_test, y_test))

feature_importances = pd.DataFrame(model.feature_importances_,
index=X_train.columns.tolist(), columns=['importance'])
feature_importances = feature_importances.sort_values('importance',
ascending=False)['importance'].to_dict()

$$; // <- end of procedure body

CALL compute_feature_importances('wine_quality', null, null, null, null);
```