Cheat Sheet for UC Shared Cluster migration

Jakob Mund Nov 4, 2024

Table of Contents

API Surface

Internal DButils API: Command Context

Problem: Workloads which try to access the command access, e.g., to retrieve a job id, using

```python
dbutils.notebook.entry_point.getDbutils().notebook().getContext().toJson()
```

fail with a Py4j not whitelisted error on UC shared clusters.

Workaround:

- Instead of **.toJson()**, use **.safeToJson()**. This gives a subset of all command context information that can be securely shared on a shared cluster.
- Requires DBR 13.3 LTS+

spark.catalog.X (tableExists, listTables, setDefaultCatalog)

Problem: Commonly used functions in spark.catalog, such as tableExists, listTables, setDefault catalog are not allowed/whitelisted on shared clusters due to security reasons.

Solution:

- Shipped Available in DBR 14.2 or later.

Other workarounds:

- Instead of **tableExists**, use the following code:

```python
data# SQL workaround
def tableExistsSql(tablename):
  try:
    spark.sql(f"DESCRIBE TABLE {tablename};")
  except Exception as e:
    return False
  return True

tableExistsSql("jakob.jakob.my_table")
```

- Instead of **listTables**, use SHOW TABLES (allows to restrict database or pattern matching as well):

```python
spark.sql(“SHOW TABLES”)
```

- For **setDefaultCatalog**, use DBR 14.2 or later. On previous DBR, run

```
Python
spark.sql("USE CATALOG <catalog_name>")
```

Access Spark JVM from PySpark: sc._jvm, spark._jspark._jvm df._jdf, df.col._jcol, and friends.

Problem: With Spark Connect on Shared clusters it is no longer possible to directly access the host JVM from the Python process. This means it is no longer possible to interact with Java classes or instantiate arbitrary Java classes directly from Python similar to the code below.

```
Python
# Loading arbitrary Java
spark._jspark._jvm.com.my.custom.class.Name()
```

Alternative problem: Mixing pyspark and spark connect classes can cause a JVM_ATTRIBUTE_NOT_SUPPORTED.

```
Python
from pyspark.sql import DataFrame as SqlDataFrame
from pyspark.sql.connect.dataframe import DataFrame
from functools import reduce

df = spark.createDataFrame([('a', 1)], ['a', 'b'])
df2 = spark.createDataFrame([('a', 3)], ['a', 'b'])

print(type(df)) # is a pyspark.sql.connect.dataframe.DataFrame

x = reduce(DataFrame.unionByName, [df, df2])
y = reduce(SqlDataFrame.unionByName, [df, df2]) # fails with
JVM_ATTRIBUTE_NOT_SUPPORTED
display(x)
display(y)
```

Workarounds:

- Use Spark configurations / DataFrame API operations where possible
- If your intention is to use certain classes preloaded on the Spark driver JVM (most commonly JDBC Driver manager), they can attach the JDBC driver / other dependencies as a maven / jar library on the cluster, and use it from a Scala language notebook.

PySpark: spark.udf.registerJavaFunction

Problem: spark.udf.registerJavaFunction is not available. The function registers a Java UDF from the Python REPL.

Workarounds:

- For notebooks and jobs, use a %scala cell to register the Scala UDF using spark.udf.register. As Python and Scala share the execution context, the Scala UDF will be available from Python as well.
- If using IDEs (using Databricks Connect v2), the only option is to rewrite the UDF as a UC Python UDF.

RDDs: sc.parallelize & spark.read.json() to convert a JSON object into a DF

Problem: Our KB articles and the PySpark docs suggest that in order to convert JSON strings into a DF, users should call sc.parallelize to convert a JSON object to a RDD and then call spark.read.json to convert it to a DF. However, sc.parallelize() is not supported.

Python
```python
json_content1 = "{'json_col1': 'hello', 'json_col2': 32}"
json_content2 = "{'json_col1': 'hello', 'json_col2': 'world'}"



json_list = []
json_list.append(json_content1)
json_list.append(json_content2)

df = spark.read.json(sc.parallelize(json_list))
display(df)
```

Solution: Use json.loads instead

Python
```python
from pyspark.sql import Row
import json
# Sample JSON data as a list of dictionaries (similar to JSON objects)
json_data_str = response.text
json_data = [json.loads(json_data_str)]

# Convert dictionaries to Row objects
rows = [Row(**json_dict) for json_dict in json_data]
# Create DataFrame from list of Row objects
df = spark.createDataFrame(rows)
# Show the DataFrame
df.display()
```

RDDs: Empty Dataframe via sc.emptyRDD()

Problem: RDD API is not supported. However, the community commonly suggests to use the following code to create an empty dataframe (where *schema* is a schema definition)

Java
```java
val schema = StructType( StructField("k", StringType, true) ::
StructField("v", IntegerType, false) :: Nil)

spark.createDataFrame(sc.emptyRDD[Row], schema)
```

Solution:

Java
```java
import org.apache.spark.sql.types.{StructType, StructField, StringType,
IntegerType}

val schema = StructType( StructField("k", StringType, true) ::
StructField("v", IntegerType, false) :: Nil)
spark.createDataFrame(new java.util.ArrayList[Row](), schema)
```

Python
```python
from pyspark.sql.types import StructType, StructField, StringType

schema = StructType([StructField("k", StringType(), True)])
spark.createDataFrame([], schema)
```

RDDs: mapPartitions (expensive initialization logic + cheaper operations per row)

Problem: UC shared clusters using Spark Connect for the communication between the Python / Scala programs and the Spark Server, which makes RDDs no longer accessible.

A typical use case for RDDs is to execute expensive initialization logic only once and then perform cheaper operations per row. Such a use case can be calling an external service or initializing encryption logic.

Python
```python
# Replacing RDD calls
def expensive_map_partitions(rows):
  # Expensive operations
  client = initialize()
  for row in rows:
     yield [row[0], client.call(row[1])]

df.rdd.mapPartitions(expensive_map_partitions)
```

Solution: Rewrite RDD operations using Dataframe API. For instance, the above example can be rewritten using PySpark native Arrow UDFs.

Python
```python
from uuid import uuid4
from typing import Iterator, Tuple
import pandas as pd
from pyspark.sql.functions import pandas_udf, lit, col

# This is a vectorized pandas UDF that is used to repartition and efficiently
execute
# data on a per batch basis.
# Declare the function and create the UDF
@pandas_udf("long")  # type: ignore[call-overload]
def heavy_processing(iterator: Iterator[Tuple[pd.Series, pd.Series]]) ->
Iterator[pd.Series]:
    # First val is symbolic for any kind of heavy processing of the data. The
whole
    # series is converted to this batch only.
    first_val = None

    # Iterate over the input series
    for a, b in iterator:
        pdf = pd.DataFrame(zip(a,b))
        yield pdf.apply(lambda row: row[0] * row[1], axis=1)


spark.range(10).withColumn("id2", lit(5)).select(heavy_processing(col("id"),
col("id2"))).show()
```

RDDs: map()

Problem: UC shared clusters using Spark Connect for the communication between the Python / Scala programs and the Spark Server, which makes RDDs no longer accessible.

A typical use case for RDDs is to transform columns using arbitrary lambda function:

Python
```python
df = spark.createDataFrame([("Alice", 100), ("Bob", 200)], ["name",
"age"])
df.rdd.map(lambda row: f"{row.name=} {row.age=}").collect()
```

Solution: Rewrite RDD operations using Dataframe API. For instance, the above example can be rewritten using PySpark native Arrow UDFs.

Python
```python
## Setup convenience function for DataFrame.map equivalent to rdd.map
from pyspark.sql.connect.dataframe import DataFrame
from pyspark.sql.functions import udf, struct
from pyspark.sql.types import StringType, StructType
```

```
class RddLike:
  def __init__(self, generator):
      self.gen = generator
  def collect(self):
      return [val[0] for val in self.gen.collect()]

def map(self, func, returnType=StringType()):
   return RddLike(self.select(udf(func, returnType,
useArrow=True)(struct("*"))))

DataFrame.map = map

##
df = spark.createDataFrame([("Alice", 100), ("Bob", 200)], ["name",
"age"])
df.map(lambda row: f"{row.name=} {row.age=}").collect()
```

SparkContext (sc) & sqlContext

Problem: Spark Context (sc) & sqlContext are not available by design due to UC shared cluster architecture and SparkConnect.

Workaround:

- Use *spark* variable to interact with the *SparkSession* instance

Limitations:

- The Spark JVM cannot be accessed directly from the Python / Scala REPL, only via Spark commands. This means that sc._jvm commands will fail by design.
- The following sc commands are not supported: *emptyRDD*, *range*, *init_batched_serializer*, *parallelize*, *pickleFile*, *textFile*, *wholeTextFiles*, *binaryFiles*, *binaryRecords*, *sequenceFile*, *newAPIHadoopFile*, *newAPIHadoopRDD*, *hadoopFile*, *hadoopRDD*, *union*, *runJob*, *setSystemProperty*, *uiWebUrl*, *stop*, *setJobGroup*, *setLocalProperty*, *getConf*

Spark Conf - sparkContext.getConf

Problem:

- sparkContext, df.sparkContext, sc.sparkContext and similar APIs are not available by design. Examples using that are literally the most popular suggestions on StackOverflow, e.g. this one

Workaround:

- Use spark.conf instead

SparkContext - SetJobDescription()

Problem: `sc.setJobDescription("String")` are not available by design due to UC shared cluster architecture and SparkConnect.

Workaround:

- Use tags instead if possible [PySpark docs]
- `spark.addTag()` can attach a tag, and `getTags()` and `interruptTag(tag)` can be used to act upon the presence/absence of a tag. These APIs only work with Spark Connect and will not work in "Assigned" access mode.
- Requires DBR 14.1+

Spark Log Levels

Problem: In Single-User and no isolation clusters it is possible to access the Spark Context to dynamically set the log level across drivers and executors directly. On shared clusters, this method was not accessible from the Spark Context, and in DBR 14+ the Spark Context is no longer available.

```python
sc.setLogLevel("INFO")
```

Solution: To control the log level without providing a log4j.conf, it is now possible to use a Spark configuration value in the cluster settings. Use Spark Log Levels by setting to `spark.log.level` to `DEBUG, WARN, INFO, ERROR` as a Spark configuration value in the cluster settings.

RecursionError / Protobuf maximum nesting level exceeded (for deeply nested expressions / queries)

Problem:

When recursively creating deeply nested DataFrames and expressions using the PySpark DataFrame API, it is possible that in certain cases either one of the following might occur:

1. Python exception: RecursionError: maximum recursion depth exceeded
2. SparkConnectGprcException: Protobuf maximum nesting level exceeded

Solution:
To circumvent the problem identify deeply nested code-paths and rewrite them using linear expressions / subqueries or temporary views.

For example: Instead of recursively calling `df.withColumn,` call `df.withColumns(dict)` instead.

Running work in DBFS using Shared cluster

Problem:
When using DBFS with shared cluster using FUSE service, it cannot reach the filesystem and generates file not found error

Example:
Here are some examples when using shared cluster access to DBFS fails

- with open('/dbfs/test/sample_file.csv', 'r') as file:
- ls -ltr /dbfs/test
- cat /dbfs/test/sample_file.csv

Solution:
Either Use -

- Databricks UC Volume instead of using DBFS **(Recommended)**
- Update the code to use dbutils or spark which directly accesses storage access path that are granted access to DBFS from shared cluster

Known Bugs

1. `display(df)` with images is currently not working. Running display on a DataFrame with image metadata yields a `[UC_COMMAND_NOT_SUPPORTED.WITHOUT_RECOMMENDATION]` error message.

2. Pandas UDF do not support row-wise `yield` in generator function for vectorized UDF. Execution fails with `Output rows does not match input rows` exception.
3. MLFlow on Databricks has integrations with APIs that are not allowlisted.
    a. https://databricks.atlassian.net/browse/ES-967258
    b. https://databricks.slack.com/archives/C05ME686MAB/p1705085555528029
4. input_file_name() is not supported in Unity Catalog

    ```
    withColumn("RECORD_FILE_NAME", col("_metadata.file_name"))
    ```
    Will work for spark.read to get the file name, or:
    ```
    .withColumn("RECORD_FILE_NAME", col("_metadata.file_path"))
    ```
    To get the whole file path

Feature Limitations

Instance Profiles

Problem: Instance profiles (AWS) and service principals/managed identity (Azure) are not supported from the Python/Scala REPL or UDFs, e.g. using boto3, Instance profiles are only from init scripts and (internally) from Spark.

Solution:
- Shipped DBR 15.4 LTS: use UC service credentials instead

Other Workarounds
- For accessing cloud storage (S3, ADLS, GCS), use storage credentials and external locations.
- For accessing non-storage cloud services (e.g., AWS secrets manager, etc), use
    - (AWS) Consider other ways to authenticate with boto3, e.g., by passing credentials from Databricks secrets directly to boto3 as a parameter, or loading them as environment variables. This page contains more information. Please note that unlike instance profiles, those methods do not provide short-lived credentials out of the box, and customers are responsible for rotating secrets according to their security needs.

    - (Azure) Consider creating a Databricks secrets scope backed by Azure vault, and authenticate to external systems using secrets from that secrets scope

    - (GCP) Some customers have used an external secrets store (like Hashicorp vault) and loaded credentials upon cluster startup into the cluster's environment variables. Please note that those are usually not short-lived credentials, and customers are responsible for securing and rotation secrets according to their security needs.

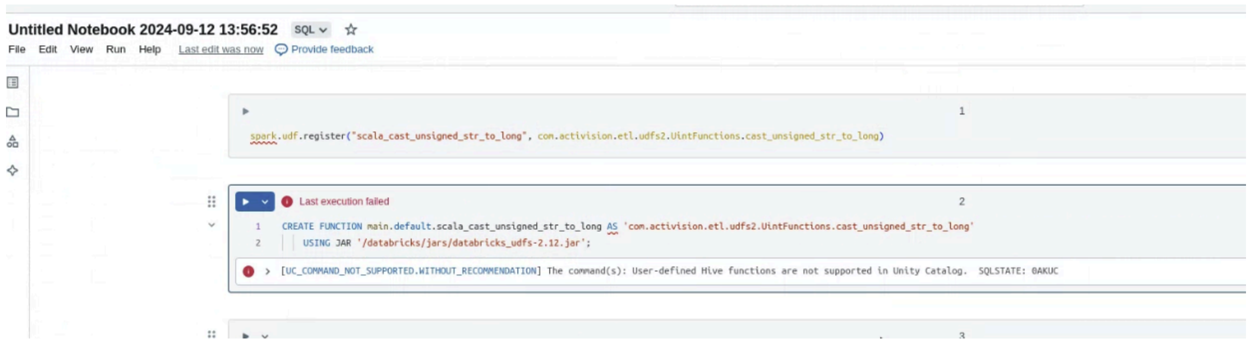
Accessing (untrusted) data sources, e.g., Oracle (via JDBC drivers)

Problem:
Accessing third-party databases—other than MySQL, PostgreSQL, Amazon Redshift, Snowflake, Microsoft SQL Server, Azure Synapse (SQL Data Warehouse) and Google BigQuery—will require additional permissions on a shared cluster if the user is not a workspace admin. This is due to the drivers not guaranteeing user isolation, e.g., as the driver writes data from multiple users to a widely accessible temp directory.

Workaround:
- Granting ANY FILE permissions will allow users to access untrusted databases. Note that ANY FILE will still enforce ACLs on any tables or external (storage) locations governed by Unity Catalog.
- This requires DBR 12.2 or later (DBR 12.1 or before is blocked on the network layer)

Hive UDFs in Scala

Problem: Hive UDFs, i.e., create a UDF from a jar overriding the Hive UDF interface, cannot be used.


Untitled Notebook 2024-09-12 13:56:52
SQL
File Edit View Run Help Last edit was now Provide feedback
Last execution failed

```
spark.udf.register("scala_cast_unsigned_str_to_long", com.activision.etl.udfs2.UintFunctions.cast_unsigned_str_to_long)
```

```
CREATE FUNCTION main.default.scala_cast_unsigned_str_to_long AS 'com.activision.etl.udfs2.UintFunctions.cast_unsigned_str_to_long'
  USING JAR '/databricks/jars/databricks_udfs-2.12.jar';
```

```
[UC_COMMAND_NOT_SUPPORTED.WITHOUT_RECOMMENDATION] The command(s): User-defined Hive functions are not supported in Unity Catalog.  SQLSTATE: 0AKUC
```

Workarounds:
- [Use session-based Scala UDFs](). If those UDFs should be shared among notebooks, extract them into a separate file and include them as the first step in jobs / notebooks. Requires DBR 14.2 or later.
- Use Python UDFs in UC. If the code can be (easily) rewritten in Python, consider creating a persistent [Python UDF in Unity Catalog](). The UC Python UDF is persistent until explicitly dropped and can be created and called from all UC-enabled compute (interactive clusters,

jobs clusters, DLT, SQL warehouses (w/ Pro/Serverless SKU))