# MariaDB ColumnStore Spark API Usage Documentation

***Release 1.2.3-c2e35eb***

**MariaDB Corporation**

**Jan 28, 2019**

# CONTENTS

CHAPTER

ONE

# LICENSING

## 1.1 Documentation Content

The Spark mcsapi documentation is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

## 1.2 MariaDB ColumnStore Spark API

The MariaDB ColumnStore Spark API (spark_mcsapi) is licensed under the GNU Lesser General Public License, version 2.1.

CHAPTER

TWO

# VERSION HISTORY

This is a version history of Spark API interface changes. It does not include internal fixes and changes.

| Version | Changes |
|---|---|
| 1.2.3 | • Documentation added |

CHAPTER

THREE

# USING MCSAPI WITH SPARK

## 3.1 Usage Introduction

mcsapi for Spark is built upon the Java bulk insert API (javamcsapi) and provides export functions to ingest Spark DataFrames and RDDs into MariaDB ColumnStore.

## 3.2 Installation and Configuration

This document describes the installation and configuration of MariaDB ColumnStore 1.2, Apache Spark 2.4.0, and mcsapi for Spark in a dockerized lab environment. Production system installations need to follow the same steps. Installation and configuration commands and paths might change depending on your operating systems, software versions, and network setup.

### 3.2.1 Lab environment setup

The lab environment consists of:

- A multi node MariaDB ColumnStore 1.2 installation with 1 user module (UM) and 2 performance modules (PMs)
- A multi node Apache Spark 2.4 installation with 1 Spark driver and 2 Spark workers

It is defined through following docker-compose.yml configuration.

To start the lab environment download it and go to the folder containing the docker-compose.yml file. Then execute:

```
docker-compose up -d
```

This will spin up the environment with six containers.

### 3.2.2 Installation of mcsapi for Spark

To utilize mcsapi for Spark's functions you have to install it on every Spark worker node as well as the Spark driver. Therefore, you first have to set up the regarding software repository via:

```
docker exec -it {SPARK_MASTER | SPARK_WORKER_1 | SPARK_WORKER_2} bash #to get a shell␣
↪in the docker container instance
apt-get update
apt-get install -y apt-transport-https dirmngr wget
echo "deb https://downloads.mariadb.com/MariaDB/mariadb-columnstore-api/latest/repo/
↪debian9 stretch main" > /etc/apt/sources.list.d/mariadb-columnstore-api.list
```

(continues on next page)

(continued from previous page)

Then add the repository key and refresh the repositories via:

```
wget -qO - https://downloads.mariadb.com/MariaDB/mariadb-columnstore/MariaDB-
↪ColumnStore.gpg.key | apt-key add -
apt-get update
```

And finally install mcsapi for Spark and its dependencies:

```
apt-get install -y mariadb-columnstore-api-spark
```

It is further advised to install the MariaDB Java JDBC library on the Spark driver to be able to execute DDL.

```
cd ${SPARK_HOME}/jars
wget https://downloads.mariadb.com/Connectors/java/connector-java-2.3.0/mariadb-java-
↪client-2.3.0.jar
```

For other operating systems, please follow the dedicated installation document in our Knowledge Base.

### 3.2.3 Spark configuration

To configure Spark to use mcsapi for Spark two more actions need to be executed.

On the one hand, the Spark master's configuration needs to be adapted to utilize the newly introduced Java libraries for javamcsapi and mcsapi for Spark.

```
cd ${SPARK_HOME}/conf   # if ${SPARK_CONF_DIR} is set it needs to be used instead
echo "spark.driver.extraClassPath /usr/lib/javamcsapi.jar:/usr/lib/spark-scala-mcsapi-
↪connector.jar" >> spark-defaults.conf
echo "spark.executor.extraClassPath /usr/lib/javamcsapi.jar:/usr/lib/spark-scala-
↪mcsapi-connector.jar" >> spark-defaults.conf
```

On the other hand, mcsapi for Spark needs information about the ColumnStore cluster to write data to. This information needs to be provided in form of a Columnstore.xml configuration file. This needs to be copied from ColumnStore's um1 node to the Spark master and each Spark worker node.

```
docker cp COLUMNSTORE_UM_1:/usr/local/mariadb/columnstore/etc/Columnstore.xml .
docker exec -it {SPARK_MASTER | SPARK_WORKER_1 | SPARK_WORKER_2} mkdir -p /usr/local/
↪mariadb/columnstore/etc
docker cp Columnstore.xml {SPARK_MASTER | SPARK_WORKER_1 | SPARK_WORKER_2}:/usr/local/
↪mariadb/columnstore/etc
```

More information about creating appropriate Columnstore.xml configuration files and Spark configuration changes can be found in our Knowledge Base.

### 3.2.4 Firewall setup

In production environments with installed firewalls you have to ensure that the Spark master and worker nodes can reach TCP port 3306 on the ColumnStore user modules, and TCP ports 8616, 8630, and 8800 on the ColumnStore performance modules. The lab environment is already fully configured, therefore there is nothing to do in this case.

### 3.2.5 Finishing note

Note that the configured Spark container aren't persistent. Once the container are stopped you have to install and configure mcsapi for Spark again. You could use `docker commit` to save your changes. Feel free to check out our *Interactive test environments* if you want to tinker around further with mcsapi for Spark.

## 3.3 A simple DataFrame export application

In this example we will export a synthetic DataFrame out of Spark into a non existing table in MariaDB ColumnStore. The full code for this can be found in the ExportDataFrame.scala file in the mcsapi codebase.

First, all needed libraries are imported and the SparkContext is created.

Listing 1: ExportDataFrame.scala

```
17 import org.apache.spark.sql.{SparkSession,DataFrame}
18 import java.util.Properties
19 import java.sql.{DriverManager,Connection,PreparedStatement,SQLException}
20 import com.mariadb.columnstore.api.connector.ColumnStoreExporter
21
22 object DataFrameExportExample {
23         def main(args: Array[String]) {
24                 // get a Spark session
25                 val spark = SparkSession.builder.appName("DataFrame export into␣
↪MariaDB ColumnStore").getOrCreate()
```

Second, an example DataFrame that is going to be exported into ColumnStore is created. The DataFrame consists of two columns. One containing numbers from 0-127 and the other containing its ASCII character representation.

Listing 2: ExportDataFrame.scala

```
27                 // generate a sample DataFrame to be exported
28                 import spark.implicits._
29                 val df = spark.sparkContext.makeRDD(0 to 127).map(i => (i, i.toChar.
↪toString)).toDF("number", "ASCII_representation")
```

Third, the JDBC connection necessary to execute DML statements to create the target table is set up.

Listing 3: ExportDataFrame.scala

```
31                 // set the variables for a JDBC connection
32                 var host = "um1"
33                 var user = "root"
34                 var password = ""
35
36                 val url = "jdbc:mysql://"+host+":3306/"
37                 val connectionProperties = new Properties()
38                 connectionProperties.put("user", user)
39                 connectionProperties.put("password", password)
40                 connectionProperties.put("driver", "org.mariadb.jdbc.Driver")
```

Fourth, the target table "spark_export" is created in the database "test". Note that `ColumnStoreExporter.generateTableStatement` infers a suitable DML statement based on the DataFrame's structure.

Listing 4: ExportDataFrame.scala

```
42                  // generate the target table
43                  val createTableStatement = ColumnStoreExporter.
↪generateTableStatement(df, "test", "spark_export")
44
45                  var connection: Connection = null
46                  try {
47                          connection = DriverManager.getConnection(url,␣
↪connectionProperties)
48                          val statement = connection.createStatement
49                          statement.executeQuery("CREATE DATABASE IF NOT EXISTS test")
50                          statement.executeQuery(createTableStatement)
51                  }catch {
52                          case e: Exception => System.err.println("error during create␣
↪table statement execution: " + e)
53                  } finally {
54                          connection.close()
55                  }
```

Finally, the DataFrame gets imported into MariaDB ColumnStore by bypassing the SQL layer and injecting the data directly through MariaDB's Bulk Write SDK.

Listing 5: ExportDataFrame.scala

```
57                  // export the DataFrame
58                  ColumnStoreExporter.export("test", "spark_export", df)
59                  spark.stop()
60          }
61  }
```

## 3.4 Application compilation and execution

To submit last section's sample application to your Spark setup you first have to compile it into a Java archive. We'll use sbt for this purpose, but other build tools like Maven would work as well.

Install sbt on the system which you want to use for compiling. Then create a sbt build project with following file structure:

```
./build.sbt
./lib/spark-scala-mcsapi-connector.jar
./src/main/scala/ColumnStoreExporter.scala
```

The build file build.sbt contains the build information, the *lib* directory contains the *mcsapi for Scala* library as unmanaged dependency, and the *src/main/scala/* directory contains the actual code to be executed.

Listing 6: build.sbt

```
name := "ColumnStoreExporter Example"

version := "1.0"

scalaVersion := "2.11.12"

libraryDependencies += "org.apache.spark" %% "spark-sql" % "2.4.0"
```

To compile the project execute:

```
sbt package
```

This will build the application `target/scala/columnstoreexporter-example_2.11-1.0.jar` that can be copied to the Spark master to be executed.

```
docker cp target/scala/columnstoreexporter-example_2.11-1.0.jar SPARK_MASTER:/root
docker exec -it SPARK_MASTER spark-submit --master spark://master:7077 /root/
↪columnstoreexporter-example_2.11-1.0.jar
```

## 3.5 Interactive test environments

Feel free to check out our interactive test environments for pymcsapi, javamcsapi, mcsapi for Spark, and mcsapi for PySpark.

- Zeppelin environment: A multi node ColumnStore environment featuring Apache Spark and Zeppelin UI
- Jupyter environment: A multi node ColumnStore environment featuring Apache Spark and Jupyter UI

CHAPTER

FOUR

# API REFERENCE

## 4.1 ColumnStoreExporter Object

### 4.1.1 Methods

#### generateTableStatement

public String **`generateTableStatement`** (DataFrame *dataFrame*)

Returns a DML CREATE TABLE statement without database prefix based on the schema of the submitted DataFrame. The table name is set to "spark_export".

**Parameters**

- **`dataFrame`** – The DataFrame from whom the structure for the generated table statement will be inferred.

public String **`generateTableStatement`** (DataFrame *dataFrame*, String *database*)

Returns a DML CREATE TABLE statement with database prefix based on the schema of the submitted DataFrame. The table name is set to "spark_export".

**Parameters**

- **`dataFrame`** – The DataFrame from whom the structure for the generated table statement will be inferred.
- **`database`** – The database name used in the generated table statement.

---

**Note:** The submitted database name will automatically be parsed into the ColumnStore naming convention, if not already compatible.

---

public String **`generateTableStatement`** (DataFrame *dataFrame*, String *database*, String *table*)

Returns a DML CREATE TABLE statement for database.table based on the schema of the submitted DataFrame.

**Parameters**

- **`dataFrame`** – The DataFrame from whom the structure for the generated table statement will be inferred.
- **`database`** – The database name used in the generated table statement.
- **`table`** – The table name used in the generated table statement.

**Note:** The submitted database and table names will automatically be parsed into the ColumnStore naming convention, if not already compatible.

public String **generateTableStatement** (DataFrame *dataFrame*, String *database*, String *table*, bool *determineTypeLength*)

Returns a DML CREATE TABLE statement for database.table based on the schema (and content) of the submitted DataFrame.

**Parameters**

- **dataFrame** – The DataFrame from whom the structure for the generated table statement will be inferred.
- **database** – The database name used in the generated table statement.
- **table** – The table name used in the generated table statement.
- **determineTypeLength** – If set to true the content of the DataFrame will be analysed to determine the best SQL datatype for each column. Otherwise reasonable default types will be used.

**Note:** The submitted database and table names will automatically be parsed into the ColumnStore naming convention, if not already compatible.

### export

public void **export** (String *database*, String *table*, DataFrame *df*)

Exports the given DataFrame into an existing ColumnStore database.table using the default Columnstore.xml configuration.

**Parameters**

- **database** – The target database the DataFrame is exported into.
- **table** – The target table the DataFrame is exported into.
- **df** – The DataFrame to export.

**Note:** To guarantee that the DataFrame import into ColumnStore is a single transaction, that is rollbacked in case of error, the DataFrame is first collected at the Spark master and from there written to the ColumnStore system. Therefore, it needs to fit into the memory of the Spark master.

**Note:** The schema of the DataFrame to export and the ColumnStore table to import have to match. Otherwise, the import will fail.

public void **export** (String *database*, String *table*, DataFrame *df*, String *configuration*)

Exports the given DataFrame into an existing ColumnStore database.table using a specific Columnstore.xml configuration.

**Parameters**

- **database** – The target database the DataFrame is exported into.
- **table** – The target table the DataFrame is exported into.

- **df** – The DataFrame to export.
- **configuration** – Path to the Columnstore.xml configuration to use for the export.

**Note:** To guarantee that the DataFrame import into ColumnStore is a single transaction, that is rollbacked in case of error, the DataFrame is first collected at the Spark master and from there written to the ColumnStore system. Therefore, it needs to fit into the memory of the Spark master.

**Note:** The schema of the DataFrame to export and the ColumnStore table to import have to match. Otherwise, the import will fail.

### exportFromWorkers

public void **exportFromWorkers** (String *database*, String *table*, RDD *rdd*)

Exports the given RDD into an existing ColumnStore database.table from the worker nodes using the default Columnstore.xml configuration.

**Parameters**

- **database** – The target database the RDD is exported into.
- **table** – The target table the RDD is exported into.
- **rdd** – The RDD to export.

**Note:** Each partition of the RDD is imported as single transaction into ColumnStore. In case of an error only partitions in which the error occurred are rolled back. Already committed partitions will remain in the database.

**Note:** The schema of the RDD to export and the ColumnStore table to import have to match. Otherwise, the import will fail.

public void **exportFromWorkers** (String *database*, String *table*, RDD *rdd*, List<Int> *partitions*)

Exports the given partitions of the RDD into an existing ColumnStore database.table from the worker nodes using the default Columnstore.xml configuration.

**Parameters**

- **database** – The target database the RDD is exported into.
- **table** – The target table the RDD is exported into.
- **rdd** – The RDD to export.
- **partitions** – List of partitions identified by their integer to be exported. If an empty List is submitted all partitions are exported.

**Note:** Each partition of the RDD is imported as single transaction into ColumnStore. In case of an error only partitions in which the error occurred are rolled back. Already committed partitions will remain in the database.

**Note:** The schema of the RDD to export and the ColumnStore table to import have to match. Otherwise, the import will fail.

public void **exportFromWorkers**(String *database*, String *table*, RDD *rdd*, List<Int> *partitions*, String *configuration*)

Exports the given partitions of the RDD into an existing ColumnStore database.table from the worker nodes using a specific Columnstore.xml configuration.

**Parameters**

- **database** – The target database the RDD is exported into.
- **table** – The target table the RDD is exported into.
- **rdd** – The RDD to export.
- **partitions** – List of partitions identified by their integer to be exported. If an empty List is submitted all partitions are exported.
- **configuration** – Path to the Columnstore.xml configuration to use for the export.

**Note:** Each partition of the RDD is imported as single transaction into ColumnStore. In case of an error only partitions in which the error occurred are rolled back. Already committed partitions will remain in the database.

**Note:** The schema of the RDD to export and the ColumnStore table to import have to match. Otherwise, the import will fail.

### parseTableColumnNameToCSConvention

public String **parseTableColumnNameToCSConvention**(String *input*)

Parses the input String according to the ColumnStore naming convention and returns it.

**Parameters**

- **input** – The String that is going to be parsed.

# INDEX