



MariaDB[®]

**MariaDB ColumnStore Python API
Usage Documentation**

Release 1.1.6-bbacad5

MariaDB Corporation

Sep 10, 2018

CONTENTS

1	Licensing	1
1.1	Documentation Content	1
1.2	MariaDB ColumnStore Python API	1
2	Version History	2
3	Using pymcsapi	3
3.1	Usage Introduction	3
3.2	Windows 10 (x64) installation	3
3.3	Basic Bulk Insert	4
3.4	Advanced Bulk Insert	5
4	pymcsapi API Reference	7
4.1	ColumnStoreDriver Class	7
4.2	ColumnStoreBulkInsert Class	9
4.3	ColumnStoreSummary Class	16
4.4	ColumnStoreException Class	19
4.5	columnstore_data_convert_status constants	19
4.6	columnstore_data_type constants	20
4.7	ColumnStoreDateTime Class	21
4.8	ColumnStoreDecimal Class	22
4.9	ColumnStoreSystemCatalog Class	24
4.10	ColumnStoreSystemCatalogTable Class	24
4.11	ColumnStoreSystemCatalogColumn Class	25
	Index	28

LICENSING

1.1 Documentation Content



The pymcsapi documentation is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

1.2 MariaDB ColumnStore Python API

The MariaDB ColumnStore Python API (pymcsapi) is licensed under the [GNU Lesser General Public License, version 2.1](https://www.gnu.org/licenses/old-licenses/lgpl-2.1.html).

VERSION HISTORY

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

Version	Changes
1.1.6	<ul style="list-style-type: none"> • Python documentation added • Windows support added (Alpha)
1.1.5	<ul style="list-style-type: none"> • Changed the return type of <code>ColumnStoreBulkInsert.setColumn()</code> and <code>ColumnStoreBulkInsert.setNull()</code> to return a List of <code>ColumnStoreBulkInsert</code> object and return status. Before only the <code>ColumnStoreBulkInsert</code> object was returned.
1.1.4	<ul style="list-style-type: none"> • Make <code>ColumnStoreSystemCatalog.getTable()</code> and <code>ColumnStoreSystemCatalogTable.getColumn()</code> case insensitive • Add <code>ColumnStoreDriver.setDebug()</code> to enable debugging output to stderr
1.1.1	<ul style="list-style-type: none"> • Add <code>ColumnStoreBulkInsert.isActive()</code> • Make <code>ColumnStoreBulkInsert.rollback()</code> fail without error • Add <code>ColumnStoreBulkInsert.resetRow()</code> • <code>pymcsapi.ColumnStoreDateTime()</code> now uses <code>uint32_t</code> for every parameter • <code>ColumnStoreSystemCatalog</code> now uses <code>const</code> for the sub-class strings
1.1.0 β	<ul style="list-style-type: none"> • First beta release

USING PYMCSAPI

3.1 Usage Introduction

The Python bulk insert API (pymcsapi) is a wrapper around the C++ bulk insert API (mcsapi) generated by SWIG. As a result Python programmers can utilize the same functions to insert data into ColumnStore tables as C++ developers can do. pymcsapi is available for Python 2 and Python 3.

3.2 Windows 10 (x64) installation

After installing the MariaDB Bulk Write SDK in Windows 10 you have to manually add its libraries to your Python (x64) installation to use it.

Following variables will be used to represent the program installation paths needed:

Variable	Description	Example
%PythonInstallDir%	The base installation directory Python was installed to	C:\Python27
%Python3InstallDir%	The base installation directory Python 3 was installed to	C:\Python37
%mcsapiInstallDir%	The base installation directory of the Bulk Write SDK	C:\Program Files\MariaDB\ColumnStore Bulk Write SDK

3.2.1 Python 2.7

- copy libiconv.dll, libuv.dll, libxml2.dll and mcsapi.dll from %mcsapiInstallDir%\lib into %PythonInstallDir%\DLLs
- copy _pymcsapi.pyd from %mcsapiInstallDir%\lib\python into %PythonInstallDir%\DLLs
- copy columnStoreExporter.py, mcsapi_reserved_words.txt and pymcsapi.py from %mcsapiInstallDir%\lib\python into %PythonInstallDir%\Lib

3.2.2 Python 3.x

- copy libiconv.dll, libuv.dll, libxml2.dll and mcsapi.dll from %mcsapiInstallDir%\lib into %Python3InstallDir%\DLLs

- copy `_pymcsapi.pyd` from `%mcsapiInstallDir%\lib\python3` into `%Python3InstallDir%\DLLs`
- copy `columnStoreExporter.py`, `mcsapi_reserved_words.txt` and `pymcsapi.py` from `%mcsapiInstallDir%\lib\python3` into `%Python3InstallDir%\Lib`

Note: Unlike the Linux version of `pymcsapi3` the Windows version of `pymcsapi3` can only be used with the Python 3 release it was compiled with. Swig for Windows compiles against the `Python3x.lib` and not the main `Python3.lib`. Therefore, you might have to recompile `pymcsapi3` from scratch if you need to use a specific Python 3 release.

3.3 Basic Bulk Insert

In this example we will insert 1000 rows of two integer values into table `test.t1`. The full code for this can be found in the `example/basic_bulk_insert.py` file in the `mcsapi` codebase.

You will need the following table in the test database to execute this:

Listing 1: `example/basic_bulk_insert.sql`

```
1 CREATE TABLE `t1` (  
2   `a` int(11) DEFAULT NULL,  
3   `b` int(11) DEFAULT NULL  
4 ) ENGINE=Columnstore;
```

Listing 2: `example/basic_bulk_insert.py`

```
26 import pymcsapi
```

We need to import `pymcsapi` which is the main module to use `mcsapi`.

Listing 3: `example/basic_bulk_insert.py`

```
28 try:  
29     driver = pymcsapi.ColumnStoreDriver()
```

A new instance of `ColumnStoreDriver` is created which will attempt to find the `Columnstore.xml` configuration file by first searching for the environment variable `COLUMNSTORE_INSTALL_DIR` and then the default path of `/usr/local/mariadb/columnstore/etc/Columnstore.xml`. Alternatively we could provide a path as a parameter to `ColumnStoreDriver`.

Listing 4: `example/basic_bulk_insert.py`

```
30     bulk = driver.createBulkInsert("test", "t1", 0, 0)
```

Once we have the `ColumnStore` installation's configuration in the driver we use this to initiate a bulk insert using `ColumnStoreDriver.createBulkInsert()`. We are using the `test` database and the `t1` table. The remaining two parameters are unused for now and set to 0.

Listing 5: example/basic_bulk_insert.py

```

31     for i in range(0,1000):
32         bulk.setColumn(0, i)
33         bulk.setColumn(1, 1000-i)
34         bulk.writeRow()

```

A “for” loop is used to loop over 1000 arbitrary inserts in this example. We use `ColumnStoreBulkInsert.setColumn()` to specify that column 0 (column a) should be set to the integer from the “for” loop and column 1 (column b) is set to 1000 minus the integer from the “for” loop.

When we have added something to every column `ColumnStoreBulkInsert.writeRow()` is used to indicate we are finished with the row. The library won’t necessarily write the row at this stage, it buffers up to 100,000 rows by default.

Listing 6: example/basic_bulk_insert.py

```

35     bulk.commit()

```

At the end of the loop we execute `ColumnStoreBulkInsert.commit()` which will send any final rows and initiate the commit of the data. If we do not do this the transaction will be implicitly rolled back instead.

Listing 7: example/basic_bulk_insert.py

```

36     except RuntimeError as err:
37         print("Error caught: %s" % (err,))

```

If anything fails then we should catch `RuntimeError` to handle it.

3.4 Advanced Bulk Insert

In this example we will insert 2 rows in a more complex table. This will demonstrate using different kinds of data types, chained methods and getting the summary information at the end of a transaction.

You will need the following table in the test database to execute this:

Listing 8: example/advanced_bulk_insert.sql

```

1  CREATE TABLE `t2` (
2     `id` int(11) DEFAULT NULL,
3     `name` varchar(40) DEFAULT NULL,
4     `dob` date DEFAULT NULL,
5     `added` datetime DEFAULT NULL,
6     `salary` decimal(9,2) DEFAULT NULL
7 ) ENGINE=Columnstore;

```

Listing 9: example/advanced_bulk_insert.py

```

26 import pymcsapi
27
28 try:
29     driver = pymcsapi.ColumnStoreDriver()
30     bulk = driver.createBulkInsert("test", "t2", 0, 0)

```

As with the basic example we create an instance of the driver and use it to create a bulk insert instance.

Listing 10: example/advanced_bulk_insert.py

```

31 bulk.setColumn(0, 1)
32 bulk.setColumn(1, "Andrew")
33 bulk.setColumn(2, "1936-12-24")
34 bulk.setColumn(3, "2017-07-07 15:14:12")
35 bulk.setColumn(4, "15239.45");
36 bulk.writeRow()

```

This demonstrates setting several different data types using strings of data. The second column (column 1) is a VARCHAR(40) and is set to “Andrew”. The third column is a DATE column and the API will automatically convert this into a binary date format before transmitting it to ColumnStore. The fourth is a DATETIME and the fifth a DECIMAL column which again the API will convert from the strings into the binary format.

Listing 11: example/advanced_bulk_insert.py

```

37 bulk.setColumn(0, 2)[0].setColumn(1, "David")[0].setColumn(2, pymcsapi.
↳ ColumnStoreDateTime(1972, 5, 23))[0].setColumn(3, pymcsapi.ColumnStoreDateTime(2017,
↳ 7, 7, 15, 20, 18))[0].setColumn(4, pymcsapi.ColumnStoreDecimal(2347623, 2))[0].
↳ writeRow()

```

Many of the *ColumnStoreBulkInsert* methods return a pointer to the class and a return status which means multiple calls can be chained together in a similar way to ORM APIs. Here we use additional datatypes *ColumnStoreDateTime* and *ColumnStoreDecimal*.

ColumnStoreDateTime is used to create an entry for a DATE or DATETIME column. It can be used to define custom formats for dates and times using the *strptime* format.

A decimal is created using the *ColumnStoreDecimal* class. It can be set using a string, double or a pair of integers. The first integer is the precision and the second integer is the scale. So this number becomes 23476.23.

Listing 12: example/advanced_bulk_insert.py

```

38 bulk.commit()
39 summary = bulk.getSummary()
40 print("Execution time: %s" % (summary.getExecutionTime(),))
41 print("Rows inserted: %s" % (summary.getRowsInsertedCount(),))
42 print("Truncation count: %s" % (summary.getTruncationCount(),))
43 print("Saturated count: %s" % (summary.getSaturatedCount(),))
44 print("Invalid count: %s" % (summary.getInvalidCount(),))

```

After a commit or rollback we can obtain summary information from the bulk insert class. This is done using the *ColumnStoreBulkInsert.getSummary()* method which will return a reference *ColumnStoreSummary* class. In this example we get the number of rows inserted (or would be inserted if there was a rollback) and the execution time from the moment the bulk insert class is created until the commit or rollback is complete.

Listing 13: example/advanced_bulk_insert.py

```

45 except RuntimeError as err:
46     print("Error caught: %s" % (err,))

```

At the end we clean up in the same was as the basic bulk insert example.

PYMCSAPI API REFERENCE

4.1 ColumnStoreDriver Class

class ColumnStoreDriver

This is the parent class for pymcsapi. It uses the `Columnstore.xml` file to discover the layout of the ColumnStore cluster. It therefore needs to be able to discover the path to the ColumnStore installation.

4.1.1 ColumnStoreDriver()

`pymcsapi.ColumnStoreDriver()`

Creates an instance of the `ColumnStoreDriver`. This will search for the environment variable `COLUMNSTORE_INSTALL_DIR`, if this isn't found then the default path of `/usr/local/mariadb/columnstore/` is used.

Raises `RuntimeError` – When the `Columnstore.xml` file cannot be found or cannot be parsed

Example

```
1 import pymcsapi
2
3 try:
4     driver = pymcsapi.ColumnStoreDriver()
5 except RuntimeError as err:
6     print("Error caught: %s" % (err,))
```

`pymcsapi.ColumnStoreDriver(path)`

Creates an instance of `ColumnStoreDriver` using the specified path to the `Columnstore.xml` file (including filename).

Parameters `path` – The path to the `Columnstore.xml` (including filename)

Raises `RuntimeError` – When the `Columnstore.xml` file cannot be found or cannot be parsed

Example

```
1 import pymcsapi
2
3 try:
4     driver = pymcsapi.ColumnStoreDriver('/usr/local/mariadb/columnstore/etc/
↳ Columnstore.xml')
```

(continues on next page)

(continued from previous page)

```

5 except RuntimeError as err:
6     print("Error caught: %s" % (err,))

```

4.1.2 createBulkInsert()

ColumnStoreDriver.**createBulkInsert** (*db, table, mode, pm*)

Allocates and configures an instance of *ColumnStoreBulkInsert* to be used for bulk inserts with the ColumnStore installation reference by the driver.

Parameters

- **db** – The database name for the table to insert into
- **table** – The table name to insert into
- **mode** – Future use, must be set to 0
- **pm** – Future use, must be set to 0. For now batches of inserts use a round-robin between the PM servers.

Returns An instance of *ColumnStoreBulkInsert*

Raises **RuntimeError** – If a table lock cannot be acquired for the desired table

Example

```

1 import pymcsapi
2
3 try:
4     driver = pymcsapi.ColumnStoreDriver()
5     bulkInsert = driver.createBulkInsert("test", "t1", 0, 0);
6 except RuntimeError as err:
7     print("Error caught: %s" % (err,))

```

4.1.3 getVersion()

ColumnStoreDriver.**getVersion** ()

Returns the version of the mcsapi library in the format 1.0.0-0393456-dirty where 1.0.0 is the version number, 0393456 is the short git tag and dirty signifies there is uncommitted code making up this build.

Returns The mcsapi version string

Example

```

1 import pymcsapi
2
3 try:
4     driver = pymcsapi.ColumnStoreDriver()
5     print("mcsapi version: %s" % (driver.getVersion(),))
6 except RuntimeError as err:
7     print("Error caught: %s" % (err,))

```

4.1.4 setDebug()

`ColumnStoreDriver.setDebug(enabled)`

Enables/disables verbose debugging output which is sent to stderr upon execution.

Note: This is a global setting which will apply to all instances of all of the API's classes after it is set until it is turned off.

Parameters `enabled` – Set to `True` to enable and `False` to disable.

Example

```
1 import pymcsapi
2
3 try:
4     driver = pymcsapi.ColumnStoreDriver()
5     driver.setDebug(True)
6     # Debugging output is now enabled
7 except RuntimeError as err:
8     print("Error caught: %s" % (err,))
```

4.1.5 getSystemCatalog()

`ColumnStoreDriver.getSystemCatalog()`

Returns an instance of the ColumnStore system catalog which contains all of the ColumnStore table and column details

Returns The system catalog

Example

```
1 import pymcsapi
2
3 try:
4     driver = pymcsapi.ColumnStoreDriver()
5     sysCat = driver.getSystemCatalog()
6     table = sysCat.getTable("test", "t1")
7     print("t1 has %d columns" % (table.getColumnCount(),))
8 except RuntimeError as err:
9     print("Error caught: %s" % (err,))
```

4.2 ColumnStoreBulkInsert Class

class `ColumnStoreBulkInsert`

The bulk insert class is designed to rapidly insert data into a ColumnStore installation.

Note: An instance of this class should only be created from `ColumnStoreDriver`

Note: If an explicit commit is not given before the class is destroyed then an implicit rollback will be executed

Note: This class should be viewed as a single transaction. Once committed or rolled back the class cannot be used for any more operations beyond getting the summary. Further usage attempts will result in an exception being thrown.

4.2.1 getColumnCount()

`ColumnStoreBulkInsert.getColumnCount()`

Gets the number of columns in the table to be inserted into.

Returns A count of the number of columns

Example

This example can be used inside the try...except blocks in the *ColumnStoreDriver* examples.

```
1 ...
2 driver = pymcsapi.ColumnStoreDriver()
3 bulkInsert = driver.createBulkInsert(db, table, 0, 0)
4 # columnCount will now contain the number of columns in the table
5 columnCount = bulkInsert.getColumnCount()
6 ...
```

4.2.2 setColumn()

`ColumnStoreBulkInsert.setColumn(columnNumber, value)`

Sets a value for a given column.

Parameters

- **columnNumber** – The column number to set (starting from 0)
- **value** – The value to set this column

Returns A List of a pointer to the *ColumnStoreBulkInsert* class so that calls can be chained, and the return status.

Raises

- **RuntimeError** – If there is an error setting the column, such as truncation error when `ColumnStoreBulkInsert.setTruncateIsError()` is used or an invalid column number is supplied
- **RuntimeError** – If the transaction has already been closed

Example

This example can be used inside the try...except blocks in the *ColumnStoreDriver* examples.

```

1  ...
2  driver = pymcsapi.ColumnStoreDriver()
3  bulkInsert = driver.createBulkInsert(db, table, 0, 0)
4
5  # Create a decimal value
6  decimalVal = pymcsapi.ColumnStoreDecimal("3.14159")
7
8  # And an int value
9  intVal = 123456
10
11 # And a string value
12 strVal = "Hello World"
13
14 # Finally a date/time values
15 dateTime = pymcsapi.ColumnStoreDateTime("1999-01-01 23:23:23", "%Y-%m-%d %H:%M:%S")
16
17 nxt, status = bulkInsert.setColumn(0, intVal)
18 # Check conversion status
19 if status != pymcsapi.CONVERT_STATUS_NONE:
20     exit(1)
21 nxt, status = bulkInsert.setColumn(1, decimalVal)
22 # Check conversion status
23 if status != pymcsapi.CONVERT_STATUS_NONE:
24     exit(1)
25 nxt, status = bulkInsert.setColumn(2, strVal)
26 # Check conversion status
27 if status != pymcsapi.CONVERT_STATUS_NONE:
28     exit(1)
29 nxt, status = bulkInsert.setColumn(3, dateTime)
30 # Check conversion status
31 if status != pymcsapi.CONVERT_STATUS_NONE:
32     exit(1)
33
34 # Write this row ready to start another
35 bulkInsert.writeRow()
36
37 decimalVal.set("1.41421")
38 intVal = 654321
39 strVal = "dlroW olleH"
40 dateTime.set("2017-07-05 22:00:43", "%Y-%m-%d %H:%M:%S")
41
42 # A chained example
43 bulkInsert.setColumn(0, intVal)[0].setColumn(1, decimalVal)[0].setColumn(2,
↳strVal)[0].setColumn(3, dateTime)[0].writeRow()
44 ...

```

4.2.3 setNull()

`ColumnStoreBulkInsert.setNull(columnNumber)`

Sets a NULL for a given column.

Parameters `columnNumber` – The column number to set (starting from 0)

Returns A List of a pointer to the `ColumnStoreBulkInsert` class so that calls can be chained, and the return status

Raises

- **RuntimeError** – If there is an error setting the column, such as an invalid column number is supplied
- **RuntimeError** – If the transaction has already been closed

Example

This example can be used inside the try...except blocks in the *ColumnStoreDriver* examples.

```
1 ...
2 driver = pymcsapi.ColumnStoreDriver()
3 bulkInsert = driver.createBulkInsert(db, table, 0, 0)
4
5 # Set an whole row of NULLs
6 bulkInsert.setNull(0)[0].setNull(1)[0].setNull(2)[0].setNull(3)[0].writeRow()
7 ...
```

4.2.4 resetRow()

`ColumnStoreBulkInsert.resetRow()`

Resets everything that has been set for the current row. This method should be used to clear the row memory without using `ColumnStoreBulkInsert.writeRow()`.

Returns A pointer to the *ColumnStoreBulkInsert* class so that calls can be chained

Raises **RuntimeError** – If the transaction has already been closed

4.2.5 writeRow()

`ColumnStoreBulkInsert.writeRow()`

States that a row is ready to be written.

Note: The row may not be written at this stage. The library will batch an amount of rows together before sending them, by default data is only sent to the server every 100,000 rows or `ColumnStoreBulkInsert.commit()` is called. Data is not committed with `writeRow()`, it has to be explicitly committed at the end of the transaction.

Returns A pointer to the *ColumnStoreBulkInsert* class so that calls can be chained

Raises

- **RuntimeError** – If there has been an error during the write at the network level
- **RuntimeError** – If there has been an error during the write at the remote server level
- **RuntimeError** – If the transaction has already been closed

Example

This example can be used inside the try...except blocks in the *ColumnStoreDriver* examples.

```

1  ...
2  driver = pymcsapi.ColumnStoreDriver()
3  bulkInsert = driver.createBulkInsert(db, table, 0, 0)
4
5  # Set values for a 2 int column table
6  bulkInsert.setColumn(0, 123456)
7  bulkInsert.setColumn(1, 654321)
8
9  # Write the row
10 bulkInsert.writeRow()
11 ...

```

4.2.6 commit()

`ColumnStoreBulkInsert.commit()`

Commits the data to the table.

Note: After making this call the transaction is completed and the class should not be used for anything but `ColumnStoreBulkInsert.getSummary()` or `ColumnStoreBulkInsert.isActive()`. Attempts to use it again will trigger an exception.

Note: If the commit fails a rollback will be executed automatically upon deletion of the `ColumnStoreBulkInsert` object.

Raises

- **RuntimeError** – If there has been an error during the write at the network level
- **RuntimeError** – If there has been an error during the write at the remote server level
- **RuntimeError** – If the transaction has already been closed

Example

This example can be used inside the try...except blocks in the `ColumnStoreDriver` examples.

```

1  ...
2  driver = pymcsapi.ColumnStoreDriver()
3  bulkInsert = driver.createBulkInsert(db, table, 0, 0)
4
5  # Set values for a 2 int column table
6  bulkInsert.setColumn(0, 123456)
7  bulkInsert.setColumn(1, 654321)
8
9  # Write the row
10 bulkInsert.writeRow()
11
12 # Commit the transaction
13 bulkInsert.commit()
14
15 # This WILL throw an exception if uncommented

```

(continues on next page)

(continued from previous page)

```

16 # bulkInsert.setColumn(0, 99999)
17 ...

```

4.2.7 rollback()

`ColumnStoreBulkInsert.rollback()`

Rolls back the data written to the table. If the transaction has already been committed or rolled back this will just return without error.

Note: After making this call the transaction is completed and the class should not be used for anything but `ColumnStoreBulkInsert.getSummary()` or `ColumnStoreBulkInsert.isActive()`. Attempts to use it again will trigger an exception.

Raises

- **RuntimeError** – If there has been an error during the write at the network level
- **RuntimeError** – If there has been an error during the write at the remote server level

Example

This example can be used inside the try...except blocks in the `ColumnStoreDriver` examples.

```

1 ...
2 driver = pymcsapi.ColumnStoreDriver()
3 bulkInsert = driver.createBulkInsert(db, table, 0, 0)
4
5 # Set values for a 2 int column table
6 bulkInsert.setColumn(0, 123456)
7 bulkInsert.setColumn(1, 654321)
8
9 # Write the row
10 bulkInsert.writeRow()
11
12 # Rollback the transaction
13 bulkInsert.rollback()
14
15 # This WILL throw an exception if uncommented
16 # bulkInsert.setColumn(0, 99999)
17 ...

```

4.2.8 isActive()

`ColumnStoreBulkInsert.isActive()`

Returns whether or not the bulk insert transaction is still active.

Returns True if the transaction is still active, False if it has been committed or rolled back

4.2.9 getSummary()

`ColumnStoreBulkInsert.getSummary()`

Gets the summary information for this bulk write transaction.

Returns The summary object

Example

This example can be used inside the `try...except` blocks in the `ColumnStoreDriver` examples.

```

1  ...
2  driver = pymcsapi.ColumnStoreDriver()
3  bulkInsert = driver.createBulkInsert(db, table, 0, 0)
4
5  # Set values for a 2 int column table
6  bulkInsert.setColumn(0, 123456)
7  bulkInsert.setColumn(1, 654321)
8
9  # Write the row
10 bulkInsert.writeRow()
11
12 # Rollback the transaction
13 bulkInsert.rollback()
14
15 # Get the summary
16 summary = bulkInsert.getSummary()
17
18 # Get the number of inserted rows before they were rolled back
19 rows = summary.getRowsInsertedCount()
20 ...

```

4.2.10 setTruncateIsError()

`ColumnStoreBulkInsert::setTruncateIsError(set)`

Sets whether or not a truncation of CHAR/VARCHAR data is an error. It is disabled by default.

Parameters `set` – True to enable, False to disable

Example

This example can be used inside the `try...except` blocks in the `ColumnStoreDriver` examples.

```

1  ...
2  driver = pymcsapi.ColumnStoreDriver()
3  bulkInsert = driver.createBulkInsert(db, table, 0, 0)
4
5  bulkInsert.setTruncateIsError(True)
6
7  # A short string that will insert fine
8  bulkInsert.setColumn(0, "Short string")
9
10 # This long string will truncate on my VARCHAR(20) and throw an exception

```

(continues on next page)

(continued from previous page)

```
11 bulkInsert.setColumn(1, "This is a long string test to demonstrate_
    ↳setTruncateIsError() ")
12 ...
```

4.2.11 setBatchSize()

`ColumnStoreBulkInsert.setBatchSize(batchSize)`

Future use, this has not been implemented yet

4.3 ColumnStoreSummary Class

class ColumnStoreSummary

A class containing the summary information for a transaction. An instance of this should be obtained from `ColumnStoreBulkInsert.getSummary()`.

4.3.1 getExecutionTime()

`ColumnStoreSummary.getExecutionTime()`

Returns the total time for the transaction in seconds, from creation of the `ColumnStoreBulkInsert` class until commit or rollback.

Returns The total execution time in seconds

Example

This example can be used inside the `try...except` blocks in the `ColumnStoreDriver` examples.

```
1 ...
2 driver = pymcsapi.ColumnStoreDriver()
3 bulkInsert = driver.createBulkInsert(db, table, 0, 0)
4
5 # Set values for a 2 int column table
6 bulkInsert.setColumn(0, 123456)
7 bulkInsert.setColumn(1, 654321)
8
9 # Write the row
10 bulkInsert.writeRow()
11
12 # Rollback the transaction
13 bulkInsert.rollback()
14
15 # Get the summary
16 summary = bulkInsert.getSummary()
17
18 # Get the execution time for the transaction
19 execTime = summary.getExecutionTime()
20 ...
```

4.3.2 getRowsInsertedCount()

ColumnStoreSummary.**getRowsInsertedCount** ()

Returns the number of rows inserted during the transaction or failed to insert for a rollback.

Returns The total number of rows

Example

This example can be used inside the try...except blocks in the *ColumnStoreDriver* examples.

```

1  ...
2  driver = pymcsapi.ColumnStoreDriver()
3  bulkInsert = driver.createBulkInsert(db, table, 0, 0)
4
5  # Set values for a 2 int column table
6  bulkInsert.setColumn(0, 123456)
7  bulkInsert.setColumn(1, 654321)
8
9  # Write the row
10 bulkInsert.writeRow()
11
12 # Rollback the transaction
13 bulkInsert.rollback()
14
15 # Get the summary
16 summary = bulkInsert.getSummary()
17
18 # Get the number of inserted rows before they were rolled back
19 rows = summary.getRowsInsertedCount()
20 ...

```

4.3.3 getTruncationCount()

ColumnStoreSummary.**getTruncationCount** ()

Returns the number of truncated CHAR/VARCHAR values during the transaction.

Returns The total number of truncated values

Example

This example can be used inside the try...except blocks in the *ColumnStoreDriver* examples.

```

1  ...
2  driver = pymcsapi.ColumnStoreDriver()
3  bulkInsert = driver.createBulkInsert(db, table, 0, 0)
4
5  # Set values for a 2 int column table
6  bulkInsert.setColumn(0, 123456)
7  bulkInsert.setColumn(1, 654321)
8
9  # Write the row
10 bulkInsert.writeRow()
11

```

(continues on next page)

(continued from previous page)

```

12 # Rollback the transaction
13 bulkInsert.rollback()
14
15 # Get the summary
16 summary = bulkInsert.getSummary()
17
18 # Get the number of truncated values before they were rolled back
19 truncateCount = summary.getTruncationCount()
20 ...

```

4.3.4 getSaturatedCount()

ColumnStoreSummary.**getSaturatedCount** ()

Returns the number of saturated values during the transaction.

Returns The total number of saturated values

Example

This example can be used inside the try...except blocks in the *ColumnStoreDriver* examples.

```

1 ...
2 driver = pymcsapi.ColumnStoreDriver()
3 bulkInsert = driver.createBulkInsert(db, table, 0, 0)
4
5 # Set values for a 2 int column table
6 bulkInsert.setColumn(0, 123456)
7 bulkInsert.setColumn(1, 2147483650)
8
9 # Write the row
10 bulkInsert.writeRow()
11
12 # Rollback the transaction
13 bulkInsert.rollback()
14
15 # Get the summary
16 summary = bulkInsert.getSummary()
17
18 # Get the number of saturated values before they were rolled back
19 saturatedCount = summary.getSaturatedCount()
20 ...

```

4.3.5 getInvalidCount()

ColumnStoreSummary.**getInvalidCount** ()

Returns the number of invalid values during the transaction.

Note: An invalid value is one where a data conversion during `ColumnStoreBulkInsert.setValue()` was not possible. When this happens a 0 or empty string is used instead and the status value set accordingly.

Returns The total number of invalid values

Example

This example can be used inside the try...except blocks in the *ColumnStoreDriver* examples.

```

1  ...
2  driver = pymcsapi.ColumnStoreDriver()
3  bulkInsert = driver.createBulkInsert(db, table, 0, 0)
4
5  # Set values for a 2 int column table
6  bulkInsert.setColumn(0, 123456);
7  # This is a DATE column, which is invalid to set as a date.
8  # The result will be the date set to '0000-00-00'
9  # and a invalid counter increment
10 bulkInsert.setColumn(1, 123456)
11
12 # Write the row
13 bulkInsert.writeRow()
14
15 # Rollback the transaction
16 bulkInsert.rollback()
17
18 # Get the summary
19 summary = bulkInsert.getSummary()
20
21 # Get the number of invalid values before they were rolled back
22 invalidCount = summary.getInvalidCount()
23 ...

```

4.4 ColumnStoreException Class

Currently pymcsapi hasn't that sophisticated exception handling than mcsapi has. All mcsapi ColumnStoreErrors are parsed by SWIG to RuntimeError.

4.5 columnstore_data_convert_status constants

These are the constants for the status values used in *ColumnStoreBulkInsert.setColumn()* and *ColumnStoreBulkInsert.setNull()* to signify the status of any data conversion that occurred during setting.

`pymcsapi.CONVERT_STATUS_NONE`

There was no problems during the conversion or no conversion.

`pymcsapi.CONVERT_STATUS_SATURATED`

The value was saturated during the conversion, the maximum/minimum was used instead.

`pymcsapi.CONVERT_STATUS_INVALID`

The value was invalid during the conversion, 0 or empty string was used instead.

`pymcsapi.CONVERT_STATUS_TRUNCATED`

The value was truncated.

4.6 columnstore_data_type constants

These are the constants for the data types as returned by *ColumnStoreSystemCatalogColumn.getType()*.

`pymcsapi.DATA_TYPE_BIT`

BIT data type

`pymcsapi.DATA_TYPE_TINYINT`

TINYINT data type

`pymcsapi.DATA_TYPE_CHAR`

CHAR data type

`pymcsapi.DATA_TYPE_SMALLINT`

SMALLINT data type

`pymcsapi.DATA_TYPE_DECIMAL`

DECIMAL data type

`pymcsapi.DATA_TYPE_MEDINT`

MEDIUMINT data type

`pymcsapi.DATA_TYPE_INT`

INT data type

`pymcsapi.DATA_TYPE_FLOAT`

FLOAT data type

`pymcsapi.DATA_TYPE_DATE`

DATE data type

`pymcsapi.DATA_TYPE_BIGINT`

BIGINT data type

`pymcsapi.DATA_TYPE_DOUBLE`

DOUBLE data type

`pymcsapi.DATA_TYPE_DATETIME`

DATETIME data type

`pymcsapi.DATA_TYPE_VARCHAR`

VARCHAR data type

`pymcsapi.DATA_TYPE_VARBINARY`

VARBINARY data type

`pymcsapi.DATA_TYPE_CLOB`

Unused

`pymcsapi.DATA_TYPE_BLOB`

BLOB data type

`pymcsapi.DATA_TYPE_UTINYINT`

UNSIGNED TINYINT data type

`pymcsapi.DATA_TYPE_USMALLINT`

UNSIGNED SMALLINT data type

`pymcsapi.DATA_TYPE_UDECIMAL`

UNSIGNED DECIMAL data type

`pymcsapi.DATA_TYPE_UMEDINT`
UNSIGNED MEDIUMINT data type

`pymcsapi.DATA_TYPE_UINT`
UNSIGNED INT data type

`pymcsapi.DATA_TYPE_UFLOAT`
UNSIGNED FLOAT data type

`pymcsapi.DATA_TYPE_UBIGINT`
UNSIGNED BIGINT data type

`pymcsapi.DATA_TYPE_UDOUBLE`
UNSIGNED DOUBLE data type

`pymcsapi.DATA_TYPE_TEXT`
TEXT data type

4.7 ColumnStoreDateTime Class

class ColumnStoreDateTime

A class which is used to contain a date/time used to set DATE or DATETIME columns using `ColumnStoreBulkInsert.setColumn()`

4.7.1 ColumnStoreDateTime()

`pymcsapi.ColumnStoreDateTime()`
Sets the date/time to 0000-00-00 00:00:00.

`pymcsapi.ColumnStoreDateTime(time)`
Sets the date/time the value of the input string.

Parameters `time` – The date/time to set

Raises `RuntimeError` – When an invalid date or time is supplied

`pymcsapi.ColumnStoreDateTime(dateTime, format)`
Sets the date/time based on a given string and format.

Parameters

- **dateTime** – A string containing the date/time to set
- **format** – The format specifier for the date/time string. This uses the `strptime` format.

Raises `RuntimeError` – When an invalid date or time is supplied

`pymcsapi.ColumnStoreDateTime(year, month, day, hour, minute, second, microsecond)`
Sets the date/time based on a given set of intergers

Note: Microseconds are for future usage, they are not currently supported in ColumnStore.

Parameters

- **year** – The year
- **month** – The month of year

- **day** – The day of month
- **hour** – The hour
- **minute** – The minute
- **second** – The second
- **microsecond** – The microseconds

Raises **RuntimeError** – When an invalid date or time is supplied

4.7.2 set()

`ColumnStoreDateTime.set (time)`

Sets the date/time using the value of the input string.

Parameters **time** – The date/time to set

Returns True if the date/time is valid, False if it is not

`ColumnStoreDateTime.set (dateTime, format)`

Sets the date/time based on a given string and format.

Parameters

- **dateTime** – A string containing the date/time to set
- **format** – The format specifier for the date/time string. This uses the `strptime` format.

Returns True if the date/time is valid, False if it is not

4.8 ColumnStoreDecimal Class

class `ColumnStoreDecimal`

A class which is used to contain a non-lossy decimal format used to set `DECIMAL` columns using `ColumnStoreBulkInsert.setColumn()`.

4.8.1 ColumnStoreDecimal()

`pymcsapi.ColumnStoreDecimal ()`

Sets the decimal to 0.

`pymcsapi.ColumnStoreDecimal (value)`

Sets the decimal to an supplied integer value.

Parameters **value** – The value to set

Raises **RuntimeError** – When an invalid value is supplied

`pymcsapi.ColumnStoreDecimal (value)`

Sets the decimal to the contents of a supplied string value (such as "3.14159").

Parameters **value** – The value to set

Raises **RuntimeError** – When an invalid value is supplied

`pymcsapi.ColumnStoreDecimal` (*value*)
Sets the decimal to the contents of a supplied double value.

Note: In the internally this uses the string method so the performance may be lower than expected.

Parameters *value* – The value to set

Raises `RuntimeError` – When an invalid value is supplied

`pymcsapi.ColumnStoreDecimal` (*number, scale*)
Sets the decimal to a given number and scale. For example for the value 3.14159 you would set the number to 314159 and the scale to 5.

Parameters

- **number** – The number to set
- **scale** – The scale for the number

Raises `RuntimeError` – When an invalid number/scale is supplied

4.8.2 set()

`ColumnStoreDecimal.set` (*value*)
Sets the decimal to an supplied integer value.

Parameters *value* – The value to set

Returns Always returns `True`

`ColumnStoreDecimal.set` (*value*)
Sets the decimal to the contents of a supplied string value (such as "3.14159").

Parameters *value* – The value to set

Returns `True` if the conversion was successful or `False` if it failed

`ColumnStoreDecimal.set` (*value*)
Sets the decimal to the contents of a supplied double value.

Note: In the internally this uses the string method so the performance may be lower than expected.

Parameters *value* – The value to set

Returns `True` if the conversion was successful or `False` if it failed

`ColumnStoreDecimal.set` (*number, scale*)
Sets the decimal to a given number and scale. For example for the value 3.14159 you would set the number to 314159 and the scale to 5.

Parameters

- **number** – The number to set
- **scale** – The scale for the number

Returns `True` if the conversion was successful or `False` if it failed

4.9 ColumnStoreSystemCatalog Class

class ColumnStoreSystemCatalog

A class which contains the ColumnStore system catalog of tables and columns. It should be instantiated using `ColumnStoreDriver.getSystemCatalog()`.

Note: The system catalog stores schema, table and column names as lower case and therefore the functions only return lower case names. Since version 1.1.4 we make case insensitive matches.

4.9.1 getTable()

`ColumnStoreSystemCatalog.getTable(schemaName, tableName)`

Gets the table information for a specific table.

Parameters

- **schemaName** – The schema the table is in
- **tableName** – The name of the table

Returns The table information

Raises `RuntimeError` – If the table is not found in the system catalog

4.10 ColumnStoreSystemCatalogTable Class

class ColumnStoreSystemCatalogTable

A class which contains the system catalog information for a specific table. It should be instantiated using `ColumnStoreSystemCatalog.getTable()`.

Note: The system catalog stores schema, table and column names as lower case and therefore the functions only return lower case names. Since version 1.1.4 we make case insensitive matches.

4.10.1 getSchemaName()

`ColumnStoreSystemCatalogTable.getSchemaName()`

Retrieves the database schema name for the table

Returns The schema name

4.10.2 getTableName()

`ColumnStoreSystemCatalogTable.getTableName()`

Retrieves the table name for the table

Returns The table name

4.10.3 getOID()

`ColumnStoreSystemCatalogTable.getOID()`

Retrieves the ColumnStore object ID for the table.

Returns The object ID for the table

4.10.4 getColumnCount()

`ColumnStoreSystemCatalogTable.getColumnCount()`

Retrieves the number of columns in the table

Returns The number of columns in the table

4.10.5 getColumn()

`ColumnStoreSystemCatalogTable.getColumn(columnName)`

Retrieves the column information for a specified column by name

Parameters `columnName` – The name of the column to retrieve

Returns The column information

Raises `RuntimeError` – If the column is not found

`ColumnStoreSystemCatalogTable.getColumn(columnNumber)`

Retrieves the column information for a specified column by number starting at zero

Parameters `columnNumber` – The number of the column to retrieve starting at 0

Returns The column information

Raises `RuntimeError` – If the column is not found

4.11 ColumnStoreSystemCatalogColumn Class

class `ColumnStoreSystemCatalogColumn`

A class containing information about a specific column in the system catalog. Should be instantiated using `ColumnStoreSystemCatalogTable.getColumn()`.

Note: The system catalog stores schema, table and column names as lower case and therefore the functions only return lower case names. Since version 1.1.4 we make case insensitive matches.

4.11.1 getOID()

`ColumnStoreSystemCatalogColumn.getOID()`

Retrieves the ColumnStore object ID for the column

Returns The column object ID

4.11.2 getColumnName()

ColumnStoreSystemCatalogColumn.**getColumnName**()

Retrieves the name of the column

Returns The column name

4.11.3 getDictionaryOID()

ColumnStoreSystemCatalogColumn.**getDictionaryOID**()

Retrieves the dictionary object ID for the column (or 0 if there is no dictionary)

Returns The dictionary object ID or 0 for no dictionary

4.11.4 getType()

ColumnStoreSystemCatalogColumn.**getType**()

Retrieves the data type for the column

Returns The data type for the column

4.11.5 getWidth()

ColumnStoreSystemCatalogColumn.**getWidth**()

Retrieves the width in bytes for the column

Returns The width in bytes

4.11.6 getPosition()

ColumnStoreSystemCatalogColumn.**getPosition**()

Retrieves the column's position in the table. The sequence of columns in the table is sorted on object ID, columns may be out-of-order if an ALTER TABLE has inserted one in the middle of the table.

Returns The column's position in the table

4.11.7 getDefaultValue()

ColumnStoreSystemCatalogColumn.**getDefaultValue**()

Retrieves the default value for the column in text. The value is empty for no default.

Returns The column's default value

4.11.8 isAutoincrement()

ColumnStoreSystemCatalogColumn.**isAutoincrement**()

Retrieves whether or not this column is an autoincrement column.

Returns `true` if this column is autoincrement, `false` if it isn't

4.11.9 getPrecision()

`ColumnStoreSystemCatalogColumn.getPrecision()`

Retrieves the decimal precision for the column.

Returns The decimal precision

4.11.10 getScale()

`ColumnStoreSystemCatalogColumn.getScale()`

Retrieves the decimal scale for the column.

Returns The decimal scale

4.11.11 isNullable()

`ColumnStoreSystemCatalogColumn.isNullable()`

Retrieves whether or not the column can be set to NULL

Returns True if the column can be NULL or False if it can not

4.11.12 compressionType()

`ColumnStoreSystemCatalogColumn.compressionType()`

Retrieves the compression type for the column. 0 means no compression and 2 means Snappy compression

Returns The compression type for the column

C

COLUMNSTORE_INSTALL_DIR, 4, 7
 ColumnStoreBulkInsert (built-in class), 9
 ColumnStoreDateTime (built-in class), 21
 ColumnStoreDateTime() (pymcsapi method), 21
 ColumnStoreDecimal (built-in class), 22
 ColumnStoreDecimal() (pymcsapi method), 22, 23
 ColumnStoreDriver (built-in class), 7
 ColumnStoreDriver() (pymcsapi method), 7
 ColumnStoreSummary (built-in class), 16
 ColumnStoreSystemCatalog (built-in class), 24
 ColumnStoreSystemCatalogColumn (built-in class), 25
 ColumnStoreSystemCatalogTable (built-in class), 24
 commit() (ColumnStoreBulkInsert method), 13
 compressionType() (ColumnStoreSystemCatalogColumn method), 27
 createBulkInsert() (ColumnStoreDriver method), 8

E

environment variable
 COLUMNSTORE_INSTALL_DIR, 4, 7

G

getColumn() (ColumnStoreSystemCatalogTable method), 25
 getColumnCount() (ColumnStoreBulkInsert method), 10
 getColumnCount() (ColumnStoreSystemCatalogTable method), 25
 getColumnName() (ColumnStoreSystemCatalogColumn method), 26
 getDefaultValue() (ColumnStoreSystemCatalogColumn method), 26
 getDictionaryOID() (ColumnStoreSystemCatalogColumn method), 26
 getExecutionTime() (ColumnStoreSummary method), 16
 getInvalidCount() (ColumnStoreSummary method), 18
 getOID() (ColumnStoreSystemCatalogColumn method), 25
 getPosition() (ColumnStoreSystemCatalogColumn method), 26
 getPrecision() (ColumnStoreSystemCatalogColumn method), 27

getRowsInsertedCount() (ColumnStoreSummary method), 17
 getSaturatedCount() (ColumnStoreSummary method), 18
 getScale() (ColumnStoreSystemCatalogColumn method), 27
 getSchemaName() (ColumnStoreSystemCatalogTable method), 24
 getSummary() (ColumnStoreBulkInsert method), 15
 getSystemCatalog() (ColumnStoreDriver method), 9
 getTable() (ColumnStoreSystemCatalog method), 24
 getTableName() (ColumnStoreSystemCatalogTable method), 24
 getTruncationCount() (ColumnStoreSummary method), 17
 getType() (ColumnStoreSystemCatalogColumn method), 26
 getVersion() (ColumnStoreDriver method), 8
 getWidth() (ColumnStoreSystemCatalogColumn method), 26

I

isActive() (ColumnStoreBulkInsert method), 14
 isAutoincrement() (ColumnStoreSystemCatalogColumn method), 26
 isNullable() (ColumnStoreSystemCatalogColumn method), 27

P

pymcsapi.CONVERT_STATUS_TRUNCATED (built-in variable), 19
 pymcsapi.CONVERT_STATUS_INVALID (built-in variable), 19
 pymcsapi.CONVERT_STATUS_NONE (built-in variable), 19
 pymcsapi.CONVERT_STATUS SATURATED (built-in variable), 19
 pymcsapi.DATA_TYPE_BIGINT (built-in variable), 20
 pymcsapi.DATA_TYPE_BIT (built-in variable), 20
 pymcsapi.DATA_TYPE_BLOB (built-in variable), 20
 pymcsapi.DATA_TYPE_CHAR (built-in variable), 20
 pymcsapi.DATA_TYPE_CLOB (built-in variable), 20
 pymcsapi.DATA_TYPE_DATE (built-in variable), 20

[pymcsapi.DATA_TYPE_DATETIME](#) (built-in variable),
[20](#)
[pymcsapi.DATA_TYPE_DECIMAL](#) (built-in variable),
[20](#)
[pymcsapi.DATA_TYPE_DOUBLE](#) (built-in variable), [20](#)
[pymcsapi.DATA_TYPE_FLOAT](#) (built-in variable), [20](#)
[pymcsapi.DATA_TYPE_INT](#) (built-in variable), [20](#)
[pymcsapi.DATA_TYPE_MEDINT](#) (built-in variable), [20](#)
[pymcsapi.DATA_TYPE_SMALLINT](#) (built-in variable),
[20](#)
[pymcsapi.DATA_TYPE_TEXT](#) (built-in variable), [21](#)
[pymcsapi.DATA_TYPE_TINYINT](#) (built-in variable), [20](#)
[pymcsapi.DATA_TYPE_UBIGINT](#) (built-in variable), [21](#)
[pymcsapi.DATA_TYPE_UDECIMAL](#) (built-in variable),
[20](#)
[pymcsapi.DATA_TYPE_UDOUBLE](#) (built-in variable),
[21](#)
[pymcsapi.DATA_TYPE_UFLOAT](#) (built-in variable), [21](#)
[pymcsapi.DATA_TYPE_UINT](#) (built-in variable), [21](#)
[pymcsapi.DATA_TYPE_UMEDINT](#) (built-in variable),
[20](#)
[pymcsapi.DATA_TYPE_USMALLINT](#) (built-in variable),
[20](#)
[pymcsapi.DATA_TYPE_UTINYINT](#) (built-in variable),
[20](#)
[pymcsapi.DATA_TYPE_VARBINARY](#) (built-in variable),
[20](#)
[pymcsapi.DATA_TYPE_VARCHAR](#) (built-in variable),
[20](#)

R

[resetRow\(\)](#) (ColumnStoreBulkInsert method), [12](#)
[rollback\(\)](#) (ColumnStoreBulkInsert method), [14](#)

S

[set\(\)](#) (ColumnStoreDateTime method), [22](#)
[set\(\)](#) (ColumnStoreDecimal method), [23](#)
[setBatchSize\(\)](#) (ColumnStoreBulkInsert method), [16](#)
[setColumn\(\)](#) (ColumnStoreBulkInsert method), [10](#)
[setDebug\(\)](#) (ColumnStoreDriver method), [9](#)
[setNull\(\)](#) (ColumnStoreBulkInsert method), [11](#)

W

[writeRow\(\)](#) (ColumnStoreBulkInsert method), [12](#)