AceQL HTTP 2.0 - Python Client SDK

Fundamentals

This document describes how to use the AceQL SDK / module and gives some details about how it operates with the AceQL Server side.

The AceQL SDK / module allows you to wrap the AceQL HTTP APIs to access remote SQL databases and/or SQL databases in the cloud by simply including standard Python SQL calls in your code, just like you would do for any local database. There is zero learning curve and usage is straightforward.

The AceQL Server operation is described in AceQL HTTP Server Installation and Configuration Guide, whose content is sometimes referred to in his User Guide.

On the remote side, like the AceQL Server access to the SQL database using Java JDBC, we will sometimes use the JDBC terminology (ResultSet, etc.) in this document. Nevertheless, knowledge of Java or JDBC is *not* a requirement.

Python Versions & DB-API 2.0

The module supports Python 2.6–2.7 & 3.4–3.7.

It provides a SQL interface compliant with the DB-API 2.0 specification described by **PEP 249**.

AceQL Server side compatiblity

This 2.0 SDK version is compatible with AceQL HTTP server side v2.0. It is not compatible with AceQL HTTP server side v1.0.

Installation

```
pip install aceql
```

Data transport

HTTP requests parameters are transported in UTF-8 format and JSON format is used for data and class transport

All requests are streamed:

- Output requests (from the client side) are streamed directly from the socket to the server to avoid buffering any content body
- Input responses (for the client side) are streamed directly from the socket to the server to efficiently read the response body

Large content (query results, Blobs/Clobs, etc.) is transferred using files. It is never loaded in memory. Streaming techniques are always used to read and write this content.

Best practices for fast response time

Every HTTP exchange between the client and server side is time-consuming, because the HTTP call is synchronous and waits for the server's response

Try to avoid coding SQL calls inside loops, as this can reduce execution speed. Each SQL call will send an http request and wait for the response from the server.

Note that AceQL is optimized as much as possible. A SELECT call returning a huge data volume will not consume memory on the server or client side: AceQL uses input stream and output stream I/O for data transfer.

Server JDBC ResultSet retrieval is as fast as possible :

- The ResultSet creation is done once on the server by the executeQuery.

- The rows are all dumped at once on the servlet output stream by the server
- The client side gets the ResultSet content as a file.

All data reading commands are executed locally on the client side with forward-only reading.

Datatypes

The main server side JDBC data types for columns are supported:

`Boolean`, `Blob/Clob`, `Integer`, `Short`, `Double`, `Float`, `BigDecimal`, `Long`, `String`, `Date`, `Time`, and `Timestamp`.

Note that the AceQL module does not allow you to specify data types to use; data types are implicitly chosen by the module.

Parameter values are automatically converted to their SQL equivalent. The following Python types can thus be sent to remote server without any problem:

Python type / class	**SQL type**
Tuple (None, SqlNullType.<SQL_TYPE>)	NULL
str	CHAR, VARCHAR
int	INTEGER, or BIGINT/NUMERIC, depending on size
bool	BIT, BOOL
float	REAL
date	DATE
datetime	TIMESTAMP
time	TIME
File Object	BLOB

NULL and BLOB types are explained in Advanced Usage.

This is how SQL types are converted to Python types by default:

SQL type	Python type / class
NULL	None
CHAR, VARCHAR	str
TINYINT, INTEGER	int
BIGINT, NUMERIC	Python 2: long Python 3: int
BOOL, BIT	bool
DATE	date
TIMESTAMP	datetime
TIME	time
BLOB	Response stream

State Management

AceQL supports two state management modes:

- The Stateful Mode
- The Stateless Mode

The Stateful Mode is the default when creating a session.

State Management is described in detail in:

AceQL HTTP Server Installation and Configuration Guide.

You can set the session State with the static method:

`Connection.set_stateless(bool value)`

Note that transactions and Connection modifiers calls are not allowed in Stateless mode and will raise an Error exception.

Usage

Quickstart

To use the module, just create a `Connection` object that represents the database:

```
import aceql

# URL of the AceQL server, Remote SQL database name
# & authentication info
host = "https://www.acme.com:9443/aceql"
database = "kawansoft_example"
username = "user1"
password = "password1"

connection = aceql.connect(host, database, username, password)
```

The schema of the database is here: kawansoft_example

Once you have a `Connection`, you can create a `Cursor` object and call its `execute()` method to perform SQL commands.

Following sample shows how to insert 3 new customers using prepared statements:

```
sql = "insert into customer values (?, ?, ?, ?, ?, ?, ?, ?)"
params = (1, 'Sir', 'John', 'Smith I', '1 Madison Ave', 'New York',
          'NY 10010', '+1 212-586-7001')
cursor.execute(sql, params)
rows_inserted = cursor.rowcount

sql = "insert into customer values (?, ?, ?, ?, ?, ?, ?, ?)"
params = (2, 'Sir', 'William', 'Smith II', '1 Madison Ave', 'New York',
          'NY 10010', '+1 212-586-7002')
cursor.execute(sql, params)
rows_inserted += cursor.rowcount

sql = "insert into customer values (?, ?, ?, ?, ?, ?, ?, ?)"
params = (3, 'Sir', 'William', 'Smith III', '1 Madison Ave', 'New York',
          'NY 10010', '+1 212-586-7003')
cursor.execute(sql, params)
rows_inserted += cursor.rowcount

print("rows inserted: " + str(rows_inserted))
```

which returns:

```
rows inserted: 3
```

The `cursor.execute()` sends the SQL order and the parameters to the server who executes it on.

We view the first inserted customer:

```
sql = "select * from customer where customer_id = ?"
params = (1,)
cursor.execute(sql, params)
row = cursor.fetchone()
print (row)
```

which returns:

```
(1, 'Sir ', 'John', 'Smith', '1 Madison Ave', 'New York', 'NY 10010  ', '+1 212-586-7000')
```

The remote result set is downloaded into a file that is read line per line at each `Cursor.fetchone()` call.

We have access to the name and type of each column:

```
for desc in cursor.description:
    print(desc[0] + ", " + desc[1])
```

Which returns:

```
customer_id, INTEGER
customer_title, CHAR
fname, VARCHAR
lname, VARCHAR
addressline, VARCHAR
town, VARCHAR
zipcode, CHAR
phone, VARCHAR
```

It's recommended to close the `Cursor` at end of SELECT usage in order to release the underlying file stream and delete the associated temp file:

```
cursor.close()
```

We view now all the customers and apply good practice to force the closing of `Cursor`:

```
with closing(connection.cursor()) as cursor:
    sql = "select * from customer where customer_id >= ? order by customer_id"
    params = (1,)
    cursor.execute(sql, params)
    print("rows: " + str(cursor.rowcount))

    rows = cursor.fetchall()

    for row in rows:
        print(row)
```

Which returns:

```
(1, 'Sir ', 'John', 'Smith', '1 Madison Ave', 'New York', 'NY 10010  ', '+1 212-586-7001')
(2, 'Sir ', 'William', 'Smith II', '1 Madison Ave', 'New York', 'NY 10010  ', '+1 212-586-7002')
(3, 'Sir ', 'William', 'Smith III', '1 Madison Ave', 'New York', 'NY 10010  ', '+1 212-586-
7003')
rows: 3
```

At end of our session, it is highly recommended to close the `Connection`:

```
# Make sure connection is always closed in order to close and release
# server connection into the pool:
connection.close()
```

Handling Exceptions

Except for `TypeError`, Exceptions thrown are always an instance of `Error`

The `Error` contains 5 pieces of information:

Info	Description
Reason	The error message. Retrieved with `reason` property.
Error Type	See below for description. Retrieved with `error_type` property.
Exception	The original `Exception` that is the cause, if any. Retrieved with `cause` property.
Http Status Code	See below for description. Retrieved with `http_status_code` property.
Server Exception	The Java Exception Stack Trace thrown on server side, if any. Retrieved with `remote_stack_trace` property.

The error type

The error type allows you to get the type of error, and where the error occurred. It is retrieved with the `Error.error_type` property.

Error Type Value	Description
0	The error occurred locally on the client side. See `http_status_code` property for more info. Typical cases: no Internet connection, proxy authentication required.
1	The error is due to a JDBC Exception. It was raised by the remote JDBC Driver and is rerouted by AceQL as is. The JDBC error message is accessible via `reason` property. Typical case: an error in the SQL statement. Examples: wrong table or column name.
2	The error was raised by the AceQL Server. This means that the AceQL Server expected a value or parameter that was not sent by the client side. Typical cases: misspelling in URL parameter, missing required request parameters, JDBC Connection expiration, etc. The detailed error message is accessible via `reason` property. See below for most common AceQL Server error messages.
3	The AceQL Server forbade the execution of the SQL statement for a security reason. For security reasons, `reason` property gives access to voluntarily vague details.
4	The AceQL Server is on failure and raised an unexpected Java Exception. The stack track is included and accessible via `remote_stack_trace` property.

Most common AceQL server messages

AceQL Sever Error Messages (Error.error_type = 2)
AceQL main servlet not found in path
An error occurred during Blob download
An error occurred during Blob upload
Blob directory defined in DatabaseConfigurator.getBlobDirectory() does not exist
Connection is invalidated (probably expired)
Database does not exist
Invalid blob_id. Cannot be used to create a file
Invalid blob_id. No Blob corresponding to blob_id
Invalid session_id
Invalid username or password
No action found in request
Operation not allowed in stateless mode
Unable to get a Connection
Unknown SQL action or not supported by software

HTTP Status Codes

The HTTP Status Code is accessible with the `Error.http_status_code` property. The HTTP Status Code is 200 (OK) on successful completion calls.

When an error occurs:

- If error type is 0, the HTTP Status Code is returned by the client side and may take all possible values in a malformed HTTP call.
- If error type is > 0, the HTTP Status Code can take one the following values returned by the server side:

HTTP Status Code	Description
400 (BAD REQUEST)	Missing element in URL path Missing request parameters All JDBC errors raised by the remote JDBC Driver
401 (UNAUTHORIZED)	Invalid username or password in connect Invalid session_id The AceQL Server forbade the execution of the SQL statement for security reasons
404 (NOT_FOUND)	BLOB directory does not exist on server BLOB file not found on server
500 (INTERNAL_SERVER_ERROR)	The AceQL Server is on failure and raised an unexpected Java Exception

Advanced Usage

Managing NULL values

Setting NULL values

`NULL` values are handled in a specific way, because the remote server must know the type of the `NULL` value.

To create a `NULL` value parameter, create a tuple of 2 elements:

- First value is `None`.
- Second value is a one of the `SqlNullType` constants that defines the type of the parameter.

This 2 elements tuple is then inserted in the tuple of the prepared statement parameters:

```
sql = "insert into customer values (?, ?, ?, ?, ?, ?, ?, ?)"
params = (4, 'Sir', 'William', 'Smith IV', '1 Madison Ave',
          'New York', 'NY 10010', (None, SqlNullType.VARCHAR))
cursor.execute(sql, params)
```

Reading NULL values

A `NULL` column value is returned as `None`:

```
sql = "select * from customer_3 where customer_id = ? order by customer_id"
params = (4,)
cursor.execute(sql, params)
row = cursor.fetchone()
print (row)
```

Execution will return:

```
(4, 'Sir ', 'William', 'Smith IV', '1 Madison Ave', 'New York', 'NY 10010  ', None)
```

In this AceQL module version: there is no difference for string columns between a real NULL in the database and the "NULL" string.

Transactions

Transactions are supported by the module. Because the remote server executes JDBC code, client code must follow the JDBC requirement to set the auto commit mode to false prior executing a transaction.

This is done with `Cursor.set_auto_commit(False)`. It is good practice to always reset auto commit mode to true at end of your transactions. Not that it auto commit mode state is undefined when a `Connection` is created with `aceql.connect()` call.

Transaction example:

```
# To do prior transaction
self.connection.set_auto_commit(False)

cursor = self.connection.cursor()

try:
    # Create a Customer
    sql = "insert into customer values (?, ?, ?, ?, ?, ?, ?, ?)"
    params = (customer_id, 'Sir', 'John', 'Smith', '1 Madison Ave',
              'New York', 'NY 10010', '+1 212-586-7000')
    cursor.execute(sql, params)

    # Create an Order for this Customer
    sql = "insert into orderlog values ( ?, ?, ?, ?, ?, ?, ?, ?, ? )"

    the_datetime = datetime.now()
    the_date = the_datetime.date()

    # (None, SqlNullType.BLOB) means to set the jpeg_image BLOB
    # column to NULL on server:
    params = (customer_id, item_id, "Item Description", 9999,
              the_date, the_datetime, (None, SqlNullType.BLOB), 1, 2)
    cursor.execute(sql, params)

    self.connection.commit()
except Error as e:
    print(e)
    self.connection.rollback()
    raise e
finally:
    self.connection.set_auto_commit(True) # Good practice
    cursor.close()
```

Proxies

The AceQL module support proxies, using the proxy syntax of Requests. The aceql module uses Requests for HTTP communications with the remote server:

```
import aceql
from aceql import *

proxies = {
  'http': 'http://10.10.1.10:3128',
  'https': 'http://10.10.1.10:1080',
}

# Create a Connection using a proxy:
connection = aceql.connect(host, database,
                           username, password, proxies=proxies)
```

Authenticated proxies are supported. Just create an `aceql.ProxyAuth` instance and pass it to `aceql.connect()`:

```
import aceql
from aceql import *

proxies = {
  'http': 'http://10.10.1.10:3128',
  'https': 'http://10.10.1.10:1080',
}

# The proxy authentication info:
auth = ProxyAuth("proxyUsername", "proxyPassword")

# Create a Connection using an authenticated proxy:
connection = aceql.connect(host, database,
                           username, password,
                           proxies=proxies, auth=auth)
```

The AceQL module uses requests-toolbelt for authenticated proxy management.

Timeouts

Use static method `Connection.set_timeout(timeout)` to define a timeout in seconds

If no timeout is specified explicitly, requests do not time out. (For more info: timeouts are implemented with Requests Timeouts.)

BLOB management

The AceQL module supports BLOB creation and reading. Methods are implemented using streaming techniques to keep low memory consumption. CLOBs are not supported in this version.

BLOB creation

BLOB creation is supported by passing a tuple with a File Object as parameter of a prepared statement:

```
sql = "insert into orderlog values ( ?, ?, ?, ?, ?, ?, ?, ?, ? )"

filename = os.getcwd() + sep + "item_1_image.png"
fd = open(filename, "rb") # File will be closed by AceQL
blob_tuple = (fd, )

params = (1, 1, "Item 1 Description", 9999,
          datetime.now() , datetime.now().date(), blob_tuple, 1, 2)
cursor.execute(sql, params)
```

BLOB reading

BLOB reading is supported through `Cursor.get_blob_stream(column_index)`. The stream can then be read with a `for` loop that iterates on the `respsonse`, using syntax provided by Requests:

```
sql = "select customer_id, item_id, jpeg_image from orderlog " \
      "where customer_id = ? and item_id = ?"
params = (1, 1)
cursor.execute(sql, params)
row = cursor.fetchone()

# You can get BLOB length if you want to use a progress indicator
blob_length = cursor.get_blob_length(2)
print("blob length: " + str(blob_length))

# Get the stream to the remote BLOB
response = cursor.get_blob_stream(2)

# Download is streamed and writen into filename
filename = os.path.expanduser("~") + sep + "jpeg_image.jpg"
with open(filename, 'wb') as fd:
    for chunk in response.iter_content(chunk_size=2048):
        fd.write(chunk)

stat_info = os.stat(filename)
print("file length: " + str(stat_info.st_size))
```

Managing BLOB upload progress

You may want to give your users a progress bar when uploading BLOBs.

The `ProgressIndicator.Percent` property allows you to get the current percent of upload. Value will be incremented automatically during upload.

To activate the update mechanism:

1/ Set the long BLOB length along the File Object in the tuple of the BLOB prepared statement parameter:

```
file_length = os.stat(filename).st_size

fd = open(filename, "rb")
blob_tuple = (fd, file_length)
```

2/ Create your `ProgressIndicator` instance and enter it to the `Connection` instance before the `Cursor.execute(sql, params)` call :

```
progress_indicator = ProgressIndicator()
connection.set_progress_indicator(progress_indicator)
```

You then can read `ProgressIndicator.percent` property in your watching thread.

Code sample:

```
with closing(connection.cursor()) as cursor:
    filename = os.getcwd() + sep + "item_1_image.jpg"
    file_length = os.stat(filename).st_size

    fd = open(filename, "rb")
    blob_tuple = (fd, file_length)

    progress_indicator = ProgressIndicator()
    connection.set_progress_indicator(progress_indicator)

    sql = "insert into orderlog values ( ?, ?, ?, ?, ?, ?, ?, ?, ? )"

    params = (1, 1, "Item 1 Description", 9999,
              datetime.now() , datetime.now().date(),
              blob_tuple, 1, 2)

    # cursor.execute() uploads BLOB by chunks and increments
    # ProgressIndicator.percent property
    cursor.execute(sql, params)
```