Apache Ignite binary client Python API Documentation

Release 0.1.0

Apache Software Foundation (ASF)

Aug 06, 2018

Contents:

CHAPTER 1

Basic Information

1.1 What it is

This is an Apache Ignite thin (binary protocol) client library, written in Python 3, abbreviated as *pyignite*.

Apache Ignite is a memory-centric distributed database, caching, and processing platform for transactional, analytical, and streaming workloads delivering in-memory speeds at petabyte scale.

Ignite binary client protocol provides user applications the ability to communicate with an existing Ignite cluster without starting a full-fledged Ignite node. An application can connect to the cluster through a raw TCP socket.

1.2 Prerequisites

- *Python 3.4* or above (3.6 is tested),
- Access to *Apache Ignite* node, local or remote. The current thin client version was tested on *Apache Ignite 2.5*.

1.3 Installation

1.3.1 for end user

If you want to use *pyignite* in your project, you may install it from PyPI:

```
$ pip install pyignite
```

1.3.2 for developer

If you want to run tests, examples or build documentation, clone the whole repository:

```
$ git clone git@github.com:nobitlost/ignite.git
$ git checkout ignite-7782
$ cd ignite/modules/platforms/python
$ pip install -e .
```

This will install the repository version of *pyignite* into your environment in so-called "develop" or "editable" mode. You may read more about editable installs in the *pip* manual.

Then run through the contents of *requirements* folder to install the the additional requirements into your working Python environment using

```
$ pip install -r requirements/<your task>.txt
```

You may also want to consult the setuptools manual about using *setup.py*.

1.4 Examples

Some examples of using pyignite are provided in *ignite/modules/platforms/python/examples* folder. They are extensively commented in the *Examples of usage* section of the documentation.

This code implies that it is run in the environment with *pyignite* package installed, and Apache Ignite node is running on localhost:10800.

1.5 Testing

Create and activate virtualenv environment. Run

```
$ cd ignite/modules/platforms/python
$ python ./setup.py pytest
```

This does not require *pytest* and other test dependencies to be installed in your environment.

Some or all tests require Apache Ignite node running on localhost:10800. To override the default parameters, use command line options `--ignite-host` and `--ignite-port`:

```
$ python ./setup.py pytest --addopts "--ignite-host=example.com --ignite-port=19840"
```

You can use each of these two options multiple times. All combinations of given host and port will be tested.

You can also test client against a server with SSL-encrypted connection. SSL-related *pytest* parameters are:

`--use-ssl` use SSL encryption,

`--ssl-certfile` a path to ssl certificate file to identify local party,

`--ssl-ca-certfile` a path to a trusted certificate or a certificate chain,

`--ssl-cert-reqs` determines how the remote side certificate is treated:

- `NONE` (ignore, default),
- `OPTIONAL` (validate, if provided),
- `REQUIRED` (valid remote certificate is required),

`--ssl-ciphers` ciphers to use,

`--ssl-version` SSL version:

- `TLSV1_1` (default),
- `TLSV1_2`.

1.6 Documentation

To recompile this documentation, do this from your virtualenv environment:

```
$ cd ignite/modules/platforms/python
$ pip install -r requirements/docs.txt
$ cd docs
$ make html
```

Then open ignite/modules/platforms/python/docs/generated/html/index.html in your browser.

If you feel that old version is stuck, do

```
$ cd ignite/modules/platforms/python/docs
$ make clean
$ sphinx-apidoc -M -o source/ ../pyignite
$ make html
```

And that should be it.

1.7 Licensing

This is a free software, brought to you on terms of the Apache License v2.

CHAPTER 2

Module Structure

The client library consists of several modules.

The most important for the end user are *connection* and *api*.

2.1 datatypes

Apache Ignite uses a sophisticated system of serializable data types to store and retrieve user data, as well as to manage the configuration of its caches through the Ignite binary protocol.

The complexity of data types varies from simple integer or character types to arrays, maps, collections and structures.

Each data type is defined by its code. *Type code* is byte-sized. Thus, every data object can be represented as a payload of fixed or variable size, logically divided into one or more fields, prepended by the *type_code* field.

Most of Ignite data types can be represented by some of the standard Python data type or class. Some of them, however, are conceptually alien, overly complex, or ambiguous to Python dynamic typing system.

The following table summarizes the notion of Apache Ignite data types, as well as their representation and handling in Python. For the nice description, as well as gory implementation details, you may follow the link to the parser/constructor class definition.

Note: you are not obliged to actually use those parser/constructor classes. Pythonic types will suffice to interact with Apache Ignite binary API. However, in some rare cases of type ambiguity, as well as for the needs of interoperability, you may have to sneak one or the other class, along with your data, in to some API function as a *type conversion hint*.

type_code	Apache Ignite docs reference	Python type or class	Parser/constructor class
Primitive data types			
0x01	Byte	int	`ByteObject`
0x02	Short	int	`ShortObject`
0x03	Int	int	`IntObject`
0x04	Long	int	`LongObject`
0x05	Float	float	`FloatObject`

Continued on next page

Table 1 – continued from previous page

type_code	Apache Ignite docs reference	Python type or class	Parser/constructor class
0x06	Double	float	`DoubleObject`
0x07	Char	str	`CharObject`
0x08	Bool	bool	`BoolObject`
0x65	Null	NoneType	`Null`
Standard objects			
0x09	String	Str	`String`
0x0a	UUID	uuid.UUID	`UUIDObject`
0x21	Timestamp	tuple	`TimestampObject`
0x0b	Date	datetime.datetime	`DateObject`
0x24	Time	datetime.timedelta	`TimeObject`
0x1e	Decimal	decimal.Decimal	`DecimalObject`
0x1c	Enum	tuple	`EnumObject`
0x67	Binary enum	tuple	`BinaryEnumObject`
Arrays of primitives			
0x0c	Byte array	iterable/list	`ByteArrayObject`
0x0d	Short array	iterable/list	`ShortArrayObject` ‘
0x0e	Int array	iterable/list	`IntArrayObject`
0x0f	Long array	iterable/list	`LongArrayObject`
0x10	Float array	iterable/list	`FloatArrayObject` ‘
0x11	Double array	iterable/list	`DoubleArrayObject`
0x12	Char array	iterable/list	`CharArrayObject`
0x13	Bool array	iterable/list	`BoolArrayObject`
Arrays of standard objects			
0x14	String array	iterable/list	`StringArrayObject`
0x15	UUID array	iterable/list	`UUIDArrayObject`
0x22	Timestamp array	iterable/list	`TimestampArrayObject`
0x16	Date array	iterable/list	`DateArrayObject`
0x23	Time array	iterable/list	`TimeArrayObject`
0x1f	Decimal array	iterable/list	`DecimalArrayObject`
Object collections, special types, and complex object			
0x17	Object array	iterable/list	`ObjectArrayObject`
0x18	Collection	tuple	`CollectionObject`
0x19	Map	dict, collections.OrderedDict	`MapObject`
0x1d	Enum array	iterable/list	`EnumArrayObject`
0x67	Complex object	dict	`BinaryObject`
0x1b	Wrapped data	tuple	`WrappedDataObject`

All type codes are stored in module `pyignite.datatypes.type_codes`.

On top of all concrete parser/constructor classes, there are classes that do not have their corresponding Ignite binary types. These classes are used to simplify the task of encoding and/or decoding data.

2.1.1 `AnyDataObject`

It is an omnivorous data type that calls other classes' deserializers when decoding the byte stream. It also does some guesswork when serializing your Python data.

It is not overly smart or omnipotent though: it can not choose CharObject for you; it will use String. It will also use LongArrayObject for representing two-integer tuple, even if you mean Enum or Collection.

This is the summary of its type guessing:

Native data types	Ignite data object
None	`Null`
int	`LongObject`
float	`DoubleObject`
str, bytes	`String`
datetime.datetime	`DateObject`
datetime.timedelta	`TimeObject`
decimal.Decimal	`DecimalObject`
uuid.UUID	`UUIDObject`
iterable	`datatypes` will inspect its contents to find the right *ArrayObject class

Bottom line: use type hints when you need to pick up a certain data type for your data, not just store that data.

2.1.2 `Struct`

This class describes a sequence of binary fields with or without *type_id*. When *type_id* is expected, `AnyDataObject` can be used as a *Struct* member. Otherwise use payload classes like `Bool` instead of `BoolObject`.

Note that any standard object can accept `Null` in its position; you do not have to explicitly handle standard objects' nullability.

2.1.3 `StructArray`

An idiomatic construct of uniform *Struct* sequence, prepended by counter field. Counter is of type `Int` by default, but its type can be changed by parameterizing the *StructArray* object. Any integer data type is acceptable.

2.1.4 `AnyDataArray`

A sequence of `AnyDataObject` objects prepended by `Int` counter. Unlike `MapObject` it do not have common *type_id* or *type* fields.

2.2 `connection`

To connect to Ignite server socket, instantiate a `Connection` class with host name and port number. Connection will negotiate a handshake with the Ignite server and raise a `SocketError` in case of client/server API versions mismatch or data flow errors.

You can then pass a `Connection` instance to various API functions.

2.3 `api`

This is a collection of functions, split into four parts:

- `cache_config` allows you to manipulate caches;
- `key_value` brings a key-value-style data manipulation, similar to *memcached* or *Redis* APIs;
- `sql` gives you the ultimate power of SQL queries;
- `binary` allows you to query the Ignite registry of binary types or register your own binary type.

To construct client queries and process server responses, all API functions uses `Query` and `Response` base classes respectively under their hoods. These classes are a natural extension of the data type parsing/constructing module (`datatypes`) and uses all the power of the indigenous `AnyDataObject`.

Each function returns operation status and result data (or verbose error message) in `APIResult` object.

All data manipulations are handled with native Python data types, without the need for the end user to construct complex data objects or parse blobs.

CHAPTER 3

Examples of usage

3.1 Key-value

3.1.1 Open connection

```
from pyignite.api import (
    cache_create, cache_destroy, cache_get, cache_put, cache_get_names
)
from pyignite.connection import Connection

conn = Connection()
conn.connect('127.0.0.1', 10800)
```

3.1.2 Create cache

```
cache_create(conn, 'my cache')
```

3.1.3 Put value in cache

```
result = cache_put(conn, 'my cache', 'my key', 42)
print(result.message)  # "Success"
```

3.1.4 Get value from cache

```
result = cache_get(conn, 'my cache', 'my key')
print(result.value)  # "42"
```

(continues on next page)

(continued from previous page)

```
result = cache_get(conn, 'my cache', 'non-existent key')
print(result.value)  # None
```

3.1.5 List keys in cache

```
result = cache_get_names(conn, 'my cache')
print(result.value)  # ['my key']
```

3.1.6 Type hints usage

```
cache_put(conn, 'my cache', 'my key', 42)
# value '42' takes 9 bytes of memory as a LongObject

cache_put(conn, 'my cache', 'my key', 42, value_hint=ShortObject)
# value '42' takes only 3 bytes as a ShortObject

cache_put(conn, 'my cache', 'a', 1)
# 'a' is a key of type String

cache_put(conn, 'my cache', 'a', 2, key_hint=CharObject)
# another key 'a' of type CharObject was created

# now let us delete both keys at once
cache_remove_keys(conn, 'my cache', [
    'a',                # a default type key
    ('a', CharObject),  # a key of type CharObject
])

cache_destroy(conn, 'my cache')
conn.close()
```

3.1.7 Scan queries

Scan queries allows you to browse cache contents with pagination.

```
cache_create(conn, 'my cache')

page_size = 10

cache_put_all(conn, 'my cache', {
        'key_{}'.format(v): v for v in range(page_size * 2)
    })

# {
#     'key_0': 0,
#     'key_1': 1,
#     'key_2': 2,
#     ... 20 elements in total...
#     'key_18': 18,
#     'key_19': 19
# }
```

(continues on next page)

(continued from previous page)

```
result = scan(conn, 'my cache', page_size)
print(dict(result.value))
# {
#     'cursor': 1,
#     'data': {
#         'key_4': 4,
#         'key_2': 2,
#         'key_8': 8,
#         ... 10 elements on page...
#         'key_0': 0,
#         'key_7': 7
#     },
#     'more': True
# }
```

Subsequent scans could be made using cursor ID.

```
cursor = result.value['cursor']
result = scan_cursor_get_page(conn, cursor)
print(result.value)
# {
#     'data': {
#         'key_15': 15,
#         'key_17': 17,
#         'key_11': 11,
#         ... another 10 elements...
#         'key_19': 19,
#         'key_16': 16
#     },
#     'more': False
# }
```

When cursor have no more data, it gets automatically destroyed.

```
result = scan_cursor_get_page(conn, cursor)
print(result.message)
# Failed to find resource with id: 1
```

If your cursor still holds some data, but you have no use of it anymore, you may destroy it manually.

```
resource_close(conn, cursor)
```

3.1.8 Do cleanup

Destroy created cache and close connection.

```
cache_destroy(conn, 'my cache')
conn.close()
```

3.2 SQL

This examples are similar to the ones given in the Apache Ignite SQL Documentation: Getting Started.

3.2.1 Setup

First let us establish a connection and create a database schema.

```
SCHEMA_NAME = 'PUBLIC'

# establish connection
conn = Connection()
conn.connect('127.0.0.1', 10800)

# create schema
cache_get_or_create(conn, SCHEMA_NAME)
```

Then create tables. Begin with *Country* table, than proceed with related tables *City* and *CountryLanguage*.

```
COUNTRY_CREATE_TABLE_QUERY = '''CREATE TABLE Country (
    Code CHAR(3) PRIMARY KEY,
    Name CHAR(52),
    Continent CHAR(50),
    Region CHAR(26),
    SurfaceArea DECIMAL(10,2),
    IndepYear SMALLINT(6),
    Population INT(11),
    LifeExpectancy DECIMAL(3,1),
    GNP DECIMAL(10,2),
    GNPOld DECIMAL(10,2),
    LocalName CHAR(45),
    GovernmentForm CHAR(45),
    HeadOfState CHAR(60),
    Capital INT(11),
    Code2 CHAR(2)
)'''
CITY_CREATE_TABLE_QUERY = '''CREATE TABLE City (
    ID INT(11),
    Name CHAR(35),
    CountryCode CHAR(3),
    District CHAR(20),
    Population INT(11),
    PRIMARY KEY (ID, CountryCode)
) WITH "affinityKey=CountryCode"'''
LANGUAGE_CREATE_TABLE_QUERY = '''CREATE TABLE CountryLanguage (
    CountryCode CHAR(3),
    Language CHAR(30),
    IsOfficial CHAR(2),
    Percentage DECIMAL(4,1),
    PRIMARY KEY (CountryCode, Language)
) WITH "affinityKey=CountryCode"'''
for query in [
    COUNTRY_CREATE_TABLE_QUERY,
    CITY_CREATE_TABLE_QUERY,
    LANGUAGE_CREATE_TABLE_QUERY,
]:
    sql_fields(conn, SCHEMA_NAME, query, PAGE_SIZE)
```

Create indexes.

```
CITY_CREATE_INDEX = '''
CREATE INDEX idx_country_code ON city (CountryCode)'''
```

(continues on next page)

(continued from previous page)

```
LANGUAGE_CREATE_INDEX = '''
CREATE INDEX idx_lang_country_code ON CountryLanguage (CountryCode)'''
for query in [CITY_CREATE_INDEX, LANGUAGE_CREATE_INDEX]:
    sql_fields(conn, SCHEMA_NAME, query, PAGE_SIZE)
```

Fill tables with data.

```
COUNTRY_INSERT_QUERY = '''INSERT INTO Country(
    Code, Name, Continent, Region,
    SurfaceArea, IndepYear, Population,
    LifeExpectancy, GNP, GNPOld,
    LocalName, GovernmentForm, HeadOfState,
    Capital, Code2
) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)'''
CITY_INSERT_QUERY = '''INSERT INTO City(
    ID, Name, CountryCode, District, Population
) VALUES (?, ?, ?, ?, ?)'''
LANGUAGE_INSERT_QUERY = '''INSERT INTO CountryLanguage(
    CountryCode, Language, IsOfficial, Percentage
) VALUES (?, ?, ?, ?)'''
for row in COUNTRY_DATA:
    sql_fields(
        conn,
        SCHEMA_NAME,
        COUNTRY_INSERT_QUERY,
        PAGE_SIZE,
        query_args=row,
    )

for row in CITY_DATA:
    sql_fields(
        conn,
        SCHEMA_NAME,
        CITY_INSERT_QUERY,
        PAGE_SIZE,
        query_args=row,
    )

for row in LANGUAGE_DATA:
    sql_fields(
        conn,
        SCHEMA_NAME,
        LANGUAGE_INSERT_QUERY,
        PAGE_SIZE,
        query_args=row,
    )
```

Data samples is taken from Ignite GitHub repository.

That concludes the preparation of data. Now let us answer some questions.

3.2.2 What are the 10 largest cities in our data sample (population-wise)?

```
PAGE_SIZE = 5
MOST_POPULATED_QUERY = '''
```

(continues on next page)

(continued from previous page)

```
SELECT name, population FROM City ORDER BY population DESC LIMIT 10'''

result = sql_fields(
    conn,
    SCHEMA_NAME,
    MOST_POPULATED_QUERY,
    PAGE_SIZE,
)
print('Most 10 populated cities:')
for row in result.value['data']:
    print(row)

# Most 10 populated cities:
# ['Mumbai (Bombay)', 10500000]
# ['Shanghai', 9696300]
# ['New York', 8008278]
# ['Peking', 7472000]
# ['Delhi', 7206704]
```

We were happy with `sql_fields()` so far. But this time we configured *PAGE_SIZE* to be 5, but requested 10 rows in the query. To get the rest of the rows we should use `sql_fields_cursor_get_page()` repeatedly.

```
cursor = result.value['cursor']
field_count = result.value['field_count']
while result.value['more']:
    print('... continue on next page...')
    result = sql_fields_cursor_get_page(conn, cursor, field_count)
    for row in result.value['data']:
        print(row)

# ... continue on next page...
# ['Chongqing', 6351600]
# ['Tianjin', 5286800]
# ['Calcutta [Kolkata]', 4399819]
# ['Wuhan', 4344600]
# ['Harbin', 4289800]
```

3.2.3 What are the 10 most populated cities throughout the 3 chosen countries?

```
MOST_POPULATED_IN_3_COUNTRIES_QUERY = '''
SELECT country.name as country_name, city.name as city_name, MAX(city.population) AS␣
↪max_pop FROM country
    JOIN city ON city.countrycode = country.code
    WHERE country.code IN ('USA','IND','CHN')
    GROUP BY country.name, city.name ORDER BY max_pop DESC LIMIT 10
'''

result = sql_fields(
    conn,
    SCHEMA_NAME,
    MOST_POPULATED_IN_3_COUNTRIES_QUERY,
    PAGE_SIZE,
    include_field_names=True,
)
```

(continues on next page)

(continued from previous page)

```
print('Most 10 populated cities in USA, India and China:')
print(result.value['fields'])
print('-----------------------------------------')
for row in result.value['data']:
    print(row)

cursor = result.value['cursor']
field_count = len(result.value['fields'])
while result.value['more']:
    print('... continue on next page...')
    result = sql_fields_cursor_get_page(conn, cursor, field_count)
    for row in result.value['data']:
        print(row)

# Most 10 populated cities in USA, India and China:
# ['COUNTRY_NAME', 'CITY_NAME', 'MAX_POP']
# -----------------------------------------
# ['India', 'Mumbai (Bombay)', 10500000]
# ['China', 'Shanghai', 9696300]
# ['United States', 'New York', 8008278]
# ['China', 'Peking', 7472000]
# ['India', 'Delhi', 7206704]
# ... continue on next page...
# ['China', 'Chongqing', 6351600]
# ['China', 'Tianjin', 5286800]
# ['India', 'Calcutta [Kolkata]', 4399819]
# ['China', 'Wuhan', 4344600]
# ['China', 'Harbin', 4289800]
```

3.2.4 Display all the information about a given city

```
CITY_INFO_QUERY = '''SELECT * FROM City WHERE id = ?'''

result = sql_fields(
    conn,
    SCHEMA_NAME,
    CITY_INFO_QUERY,
    PAGE_SIZE,
    query_args=[3802],
    include_field_names=True,
)
print('City info:')
for field_name, field_value in zip(
    result.value['fields'],
    result.value['data'][0],
):
    print('{}: {}'.format(field_name, field_value))

# City info:
# ID: 3802
# NAME: Detroit
# COUNTRYCODE: USA
# DISTRICT: Michigan
# POPULATION: 951270
```

Finally, delete the tables used in this example with the following queries:

```
DROP_TABLE_QUERY = '''DROP TABLE {}'''
for table_name in [
    CITY_TABLE_NAME,
    LANGUAGE_TABLE_NAME,
    COUNTRY_TABLE_NAME,
]:
    result = sql_fields(
        conn,
        SCHEMA_NAME,
        DROP_TABLE_QUERY.format(table_name),
        PAGE_SIZE,
    )
    print('Deleting `{}`: {}'.format(table_name, result.message))

# Deleting `City`: Success
# Deleting `CountryLanguage`: Success
# Deleting `Country`: Success
```

3.3 Complex objects

3.3.1 Read

Complex object (that is often called 'Binary object') is used to represent user-defined complex data types. It have the following features:

- have a unique ID,
- have an associated schema, that describes its inner structure (the order and types of its fields).

The schemas are stored in Ignite metadata storage. That is why Complex object must be registered with the Ignite cluster before use.

The most obvious example of Complex object usage is SQL tables.

In the *previous examples* we have created some SQL tables. Let us do it again and examine the Ignite storage afterwards.

```
result = cache_get_names(conn)
print(result.value)

# [
#     'SQL_PUBLIC_CITY',
#     'SQL_PUBLIC_COUNTRY',
#     'PUBLIC',
#     'SQL_PUBLIC_COUNTRYLANGUAGE'
# ]
```

We can see that Ignite created a cache for each of our table. The caches are conveniently named using '*SQL_<schema name>_<table name>*' pattern.

Now let us examine a configuration of a cache that contains SQL data using a `cache_get_configuration()` function.

```
result = cache_get_configuration(conn, 'SQL_PUBLIC_CITY')
print(dict(result.value))

# {
#     'name': 'SQL_PUBLIC_CITY',
#     'sql_schema': 'PUBLIC',
#     'cache_key_configuration': [
#         {
#             'type_name': 'SQL_PUBLIC_CITY_9ac8e17a_2f99_45b7_958e_06da32882e9d_KEY',
#             'affinity_key_field_name': 'COUNTRYCODE'
#         }
#     ],
#     'query_entities': [
#         {
#             'key_type_name': 'SQL_PUBLIC_CITY_9ac8e17a_2f99_45b7_958e_06da32882e9d_
↪KEY',
#             'value_type_name': 'SQL_PUBLIC_CITY_9ac8e17a_2f99_45b7_958e_06da32882e9d
↪',
#             'table_name': 'CITY',
#             'query_fields': [
#                 ...
#             ],
#             'field_name_aliases': [
#                 ...
#             ],
#             'query_indexes': []
#         }
#     ]
# }
```

The values of *value_type_name* and *key_type_name* are names of the binary types, in which the *Cities* table rows' values and keys are stored. Let us check the types' registration and properties.

```
key_binary_type_name = result.value['query_entities'][0]['key_type_name']
key_binary_type_id = entity_id(key_binary_type_name)

value_binary_type_name = result.value['query_entities'][0]['value_type_name']
value_binary_type_id = entity_id(value_binary_type_name)

print(key_binary_type_id, value_binary_type_id)

# -996482981 -1295865797

result = get_binary_type(conn, key_binary_type_id)
print(result.value['type_exists'])

# True

result = get_binary_type(conn, value_binary_type_id)
print(result.value['type_exists'])

# True
```

Let us take a closer look to the value type.

```
print(result.value)
```

(continues on next page)

(continued from previous page)

```
# {
#     'type_exists': True,
#     'type_id': -1295865797,
#     'type_name': 'SQL_PUBLIC_CITY_9ac8e17a_2f99_45b7_958e_06da32882e9d',
#     'affinity_key_field': None,
#     'binary_fields': [
#         {'field_name': 'NAME', 'type_id': 9, 'field_id': 3373707},
#         {'field_name': 'DISTRICT', 'type_id': 9, 'field_id': 288961422},
#         {'field_name': 'POPULATION', 'type_id': 3, 'field_id': -2023558323}
#     ],
#     'is_enum': False,
#     'schema': [
#         {
#             'schema_id': 275495165,
#             'schema_fields': [
#                 {'schema_field_id': 3373707},
#                 {'schema_field_id': 288961422},
#                 {'schema_field_id': -2023558323}
#             ]
#         }
#     ]
# }
```

We have 3 fields in the row value: *Name*, *District*, and *Population*. The complex primary key field, *ID* + *CountryCode*, is in the row key.

To support this theory let us try to read the data without using SQL functions.

```
result = scan(conn, 'SQL_PUBLIC_CITY', 1)
print(result.value['data'])

# {
#     (b'... Some binary data...', 0): (b'... Some more binary data...', 0)
# }
```

Not exactly what we expected. That's because the Binary objects are always come wrapped in a content-agnostic `WrappedDataObject`. We need to take an additional step to explicitly decode it.

```
wrapped_value = list(result.value['data'].values())[0]
binary_obj = unwrap_binary(conn, wrapped_value)
print(binary_obj)

# {
#     'version': 1,
#     'type_id': -1295865797,
#     'hash_code': 819840247,
#     'schema_id': 275495165,
#     'fields': {
#         'NAME': 'Shanghai',
#         'DISTRICT': 'Shanghai',
#         'POPULATION': 9696300
#     }
# }
```

3.3.2 Create

Now, that we aware of the internal structure of the Ignite SQL storage, we can create a table and put data in it using only key-value functions.

For example, let us create a table to register High School students: a rough equivalent of the following SQL DDL statement:

```
CREATE TABLE Student (
    sid CHAR(9),
    name VARCHAR(20),
    login CHAR(8),
    age INTEGER(11),
    gpa REAL
)
```

These are the necessary steps to perform the task.

1. Create scheme cache.

```
cache_get_or_create(conn, 'PUBLIC')
```

2. Create table cache.

```
cache_create_with_config(conn, {
        PROP_NAME: 'SQL_PUBLIC_STUDENT',
        PROP_SQL_SCHEMA: 'PUBLIC',
        PROP_QUERY_ENTITIES: [
            {
                'table_name': 'Student'.upper(),
                'key_field_name': 'SID',
                'key_type_name': 'java.lang.Integer',
                'field_name_aliases': [],
                'query_fields': [
                    {
                        'name': 'SID',
                        'type_name': 'java.lang.Integer',
                        'is_key_field': True,
                        'is_notnull_constraint_field': True,
                        'default_value': None,
                    },
                    {
                        'name': 'NAME',
                        'type_name': 'java.lang.String',
                        'is_key_field': False,
                        'is_notnull_constraint_field': False,
                        'default_value': None,
                    },
                    {
                        'name': 'LOGIN',
                        'type_name': 'java.lang.String',
                        'is_key_field': False,
                        'is_notnull_constraint_field': False,
                        'default_value': None,
                    },
                    {
                        'name': 'AGE',
                        'type_name': 'java.lang.Integer',
```

(continues on next page)

(continued from previous page)

```
                        'is_key_field': False,
                        'is_notnull_constraint_field': False,
                        'default_value': None,
                    },
                    {
                        'name': 'GPA',
                        'type_name': 'java.math.Double',
                        'is_key_field': False,
                        'is_notnull_constraint_field': False,
                        'default_value': None,
                    },
                ],
                'query_indexes': [],
                'value_type_name': 'SQL_PUBLIC_STUDENT_TYPE',
                'value_field_name': None,
            },
        ],
    })
```

3. Register binary type.

```
result = put_binary_type(
    conn,
    'SQL_PUBLIC_STUDENT_TYPE',
    schema={
        'NAME': String,
        'LOGIN': String,
        'AGE': IntObject,
        'GPA': DoubleObject,
    }
)

type_id = result.value['type_id']
schema_id = result.value['schema_id']
```

4. Insert row.

```
cache_put(
    conn,
    'SQL_PUBLIC_STUDENT',
    key=1,
    key_hint=IntObject,
    value={
        'version': 1,
        'type_id': type_id,
        'schema_id': schema_id,
        'fields': {
            'LOGIN': 'jdoe',
            'NAME': 'John Doe',
            'AGE': (17, IntObject),
            'GPA': 4.25,
        },
    },
    value_hint=BinaryObject,
)
```

Now read the row using an SQL function.

```
result = sql_fields(
    conn,
    'PUBLIC',
    'SELECT * FROM Student',
    1,
    include_field_names=True,
)
print(result.value)

# {
#     'more': False,
#     'data': [
#         [1, 'John Doe', 'jdoe', 17, 4.25]
#     ],
#     'fields': ['SID', 'NAME', 'LOGIN', 'AGE', 'GPA'],
#     'cursor': 1
# }
```

3.3.3 Migrate

Suppose we have an accounting app that stores its data in key-value format. Our task would be to introduce the following changes to the original expense voucher's format and data:

- rename *date* to *expense_date*,
- add *report_date*,
- set *report_date* to the current date if *reported* is True, None if False,
- delete *reported*.

First obtain the binary type ID. It can be calculated as a hashcode of the binary type name in lower case.

```
result = get_binary_type(conn, 'ExpenseVoucher')
```

Then obtain the initial schema.

```
# {
#     'type_id': -1171639466,
#     'type_name': 'ExpenseVoucher',
#     'is_enum': False,
#     'affinity_key_field': None,
#     'binary_fields': [
#         {'type_id': 11, 'field_id': 3076014, 'field_name': 'date'},
#         {'type_id': 8, 'field_id': -427039533, 'field_name': 'reported'},
#         {'type_id': 9, 'field_id': -220463842, 'field_name': 'purpose'},
#         {'type_id': 30, 'field_id': 114251, 'field_name': 'sum'},
#         {'type_id': 9, 'field_id': 820081177, 'field_name': 'recipient'},
#         {'type_id': 4, 'field_id': -2030736361, 'field_name': 'cashier_id'},
#     ],
#     'schema': {
#         -231598180: [
#             3076014,
#             -427039533,
#             -220463842,
#             114251,
```

(continues on next page)

(continued from previous page)

```
#             820081177,
#             -2030736361,
#         ],
#     },
#     'type_exists': True,
# }

schema = OrderedDict([
```

The binary type *ExpenseVoucher* has 6 fields and one schema. All the fields are present in that one schema. Note also, that each field has an ID (which is also calculated as a hascode of its name in lower case) and a type ID. Field type ID can be either ordinal value of one of the `type_codes` or an ID of the registered binary type.

Let us modify the schema dictionary and update the type.

```
    for field in result.value['binary_fields']
])

schema['expense_date'] = schema['date']
del schema['date']
schema['report_date'] = DateObject
del schema['reported']
schema['sum'] = DecimalObject

result = put_binary_type(
    conn,
    'ExpenseVoucher',
    schema=schema,
)
new_schema_id = result.value['schema_id']

result = get_binary_type(conn, type_id)
print(result.value)

# {
#     'type_id': -1171639466,
#     'type_name': 'ExpenseVoucher',
#     'is_enum': False,
#     'affinity_key_field': None,
#     'binary_fields': [
#         {'type_id': 11, 'field_id': 3076014, 'field_name': 'date'},
#         {'type_id': 8, 'field_id': -427039533, 'field_name': 'reported'},
#         {'type_id': 9, 'field_id': -220463842, 'field_name': 'purpose'},
#         {'type_id': 30, 'field_id': 114251, 'field_name': 'sum'},
#         {'type_id': 9, 'field_id': 820081177, 'field_name': 'recipient'},
#         {'type_id': 4, 'field_id': -2030736361, 'field_name': 'cashier_id'},
#         {'type_id': 11, 'field_id': 1264342837, 'field_name': 'expense_date'},
#         {'type_id': 11, 'field_id': -247041063, 'field_name': 'report_date'},
#     ], 'schema': {
#         -231598180: [
#             3076014,
#             -427039533,
#             -220463842,
#             114251,
#             820081177,
#             -2030736361,
#         ],
```

(continues on next page)

(continued from previous page)

```
#         547629991: [
#             -220463842,
#             114251,
#             820081177,
#             -2030736361,
#             1264342837,
#             -247041063,
#         ]
#     },
#     'type_exists': True,
# }
```

Now our binary type have two schemes. The old scheme (ID=-231598180) remained unchanged, while the new scheme (ID=547629991) has only those fields specified in the most recent `put_binary_type()` call. None of the binary fields were actually removed, but two newly described fields, *expense_date* and *report_date*, were added.

Now migrate the data from the old schema to the new one.

```
def migrate(data):
    """ Migrate given data pages. """
    for key, value in data.items():
        # read data
        fields = unwrap_binary(conn, value)['fields']
        print(dict(fields))

        # {
        #     'cashier_id': 8,
        #     'date': datetime.datetime(2017, 9, 21, 0, 0),
        #     'sum': Decimal('666.67'),
        #     'reported': True,
        #     'purpose': 'Praesent eget fermentum massa',
        #     'recipient': 'John Doe',
        # }

        # process data
        fields['expense_date'] = fields['date']
        del fields['date']
        fields['report_date'] = date.today() if fields['reported'] else None
        del fields['reported']

        # replace data
        cache_put(
            conn,
            'accounting',
            key,
            {
                'version': 1,
                'type_id': type_id,
                'schema_id': new_schema_id,
                'fields': fields,
            },
            value_hint=BinaryObject,
        )

        # verify data
```

(continues on next page)

(continued from previous page)

```
        verify = cache_get(conn, 'accounting', key)
        print(dict(unwrap_binary(conn, verify.value)['fields']))

        # {
        #     'cashier_id': 8,
        #     'sum': Decimal('666.67'),
        #     'report_date': datetime.datetime(2018, 7, 24, 0, 0),
        #     'expense_date': datetime.datetime(2017, 9, 21, 0, 0),
        #     'recipient': 'John Doe',
        #     'purpose': 'Praesent eget fermentum massa',
        # }


# migrate data
result = scan(conn, 'accounting', 2)
migrate(result.value['data'])

cursor = result.value['cursor']
while result.value['more']:
    result = scan_cursor_get_page(conn, cursor)
    migrate(result.value['data'])
```

As you can see, old or new fields are available in the resulting binary object, depending on which schema was used when writing them using `cache_put()`.

This versioning mechanism is quite simple and robust, but it have its limitations. The main thing is: you can not change the type of the existing field. If you try, you will be greeted with the following message:

```
`org.apache.ignite.binary.BinaryObjectException: Wrong value has been set
[typeName=SomeType, fieldName=f1, fieldType=String, assignedValueType=int]`
```

As an alternative (which feels more like a workaround) you can rename the field or create a new schema.

3.4 Failover

When connection to the server is broken or timed out, `Connection` object raises an appropriate exception, but keeps its constructor's parameters intact, so user can reconnect, and the connection object remains valid.

The following example features a simple round-robin failover mechanism. Launch 3 Ignite nodes on *localhost* and run:

```
from pyignite.api import (
    cache_get, cache_put, cache_get_or_create_with_config,
)
from pyignite.connection import Connection
from pyignite.datatypes.cache_config import CacheMode
from pyignite.datatypes.prop_codes import *


MAX_ERRORS = 20
nodes = [
    ('127.0.0.1', 10800),
    ('127.0.0.1', 10801),
    ('127.0.0.1', 10802),
]
```

(continues on next page)

(continued from previous page)

```
node_idx = err_count = 0
conn = Connection(timeout=3.0)
conn.connect(*nodes[node_idx])

while True:
    try:
        # reconnect
        conn.connect(*nodes[node_idx])
        print('Connected to node {}'.format(node_idx))
        while True:
            # proceed with initializing or modifying data
            cache_get_or_create_with_config(conn, {
                PROP_NAME: 'failover_test',
                PROP_CACHE_MODE: CacheMode.REPLICATED,
            })
            result = cache_get(conn, 'failover_test', 'test_value')
            cache_put(
                conn,
                'failover_test',
                'test_value',
                result.value + 1 if result.value else 1
            )
    except Exception as e:
        # count errors
        err_count += 1
        if err_count > MAX_ERRORS:
            print('Too many disconnects! Exiting.')
            break
        # switch to another node
        node_idx = node_idx + 1
        if node_idx >= len(nodes):
            node_idx = 0
        print(
            '"{}" just happened; switching to node {}.'.format(e, node_idx)
        )
```

Then try shutting down and restarting nodes, and see what happens. At least one node should remain active.

```
# Connected to node 0
# "Socket connection broken." just happened; switching to node 1.
# Connected to node 1
# "Socket connection broken." just happened; switching to node 2.
# "[Errno 111] Connection refused" just happened; switching to node 0.
# Connected to node 0
```

3.5 SSL/TLS

There are some special requirements for testing SSL connectivity.

The Ignite server must be configured for securing the binary protocol port. The server configuration process can be split up into these basic steps:

1. Create a key store and a trust store using Java keytool. When creating the trust store, you will probably need a client X.509 certificate. You will also need to export the server X.509 certificate to include in the client chain of trust.

2. Turn on the *SslContextFactory* for your Ignite cluster according to this document: Securing Connection Between Nodes.
3. Tell Ignite to encrypt data on its thin client port, using the settings for ClientConnectorConfiguration. If you only want to encrypt connection, not to validate client's certificate, set *sslClientAuth* property to *false*. You'll still have to set up the trust store on step 1 though.

Client SSL settings is summarized here: `Connection`.

To use the SSL encryption without certificate validation just *use_ssl*.

```
from pyignite.connection import Connection

conn = Connection(use_ssl=True)
conn.connect('127.0.0.1', 10800)
```

To identify the client, create an SSL keypair and a certificate with openssl command and use them in this manner:

```
from pyignite.connection import Connection

conn = Connection(
    use_ssl=True,
    ssl_keyfile='etc/.ssl/keyfile.key',
    ssl_certfile='etc/.ssl/certfile.crt',
)
conn.connect('ignite-example.com', 10800)
```

To check the authenticity of the server, get the server certificate or certificate chain and provide its path in the *ssl_ca_certfile* parameter.

```
import ssl

from pyignite.connection import Connection

conn = Connection(
    use_ssl=True,
    ssl_ca_certfile='etc/.ssl/ca_certs',
    ssl_cert_reqs=ssl.CERT_REQUIRED,
)
conn.connect('ignite-example.com', 10800)
```

You can also provide such parameters as the set of ciphers (*ssl_ciphers*) and the SSL version (*ssl_version*), if the defaults (`ssl._DEFAULT_CIPHERS` and TLS 1.1) do not suit you.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search