Apache Ignite binary client Python API Documentation

Release 0.1.0

Apache Software Foundation (ASF)

Apr 14, 2021

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 2.7.0* (binary client protocol 1.2.0).

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:apache/ignite.git
$ 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, unless otherwise noted.

There is also a possibility to run examples alone with tests. For the explanation of testing, look up the *Testing* section.

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`.

Other *pytest* parameters:

`--timeout` timeout (in seconds) for each socket operation, including *connect*. Accepts integer or float value. Default is None (blocking mode),

`--username` and `--password` credentials to authenticate to Ignite cluster. Used in conjunction with *authenticationEnabled* property in cluster configuration.

`--examples` run the examples as one test. If you wish to run *only* the examples, supply also the name of the test function to *pytest* launcher:

```
$ pytest --examples ../tests/test_examples.py::test_examples
```

In this test assertion fails if any of the examples' processes ends with non-zero exit code.

Examples are not parameterized for the sake of simplicity. They always run with default parameters (host and port) regardless of any other *pytest* option.

Since failover, SSL and authentication examples are meant to be controlled by user or depend on special configuration of the Ignite cluster, they can not be automated.

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 -feM -o source/ ../ ../setup.py
$ 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

API Specification

The modules and subpackages listed here are the basis of a stable API of *pyignite*, intended for end users.

2.1 pyignite.client module

This module contains *Client* class, that lets you communicate with Apache Ignite cluster node by the means of Ignite binary client protocol.

To start the communication, you may connect to the node of their choice by instantiating the *Client* object and calling `connect()` method with proper parameters.

The whole storage room of Ignite cluster is split up into named structures, called caches. For accessing the particular cache in key-value style (a-la Redis or memcached) you should first create the *Cache* object by calling *create_cache()* or *get_or_create_cache()* methods, than call *Cache* methods. If you wish to create a cache object without communicating with server, there is also a *get_cache()* method that does just that.

For using Ignite SQL, call *sql()* method. It returns a generator with result rows.

register_binary_type() and *query_binary_type()* methods operates the local (class-wise) registry for Ignite Complex objects.

class pyignite.client.**Client**(*compact_footer: bool = None*, **args*, ***kwargs*)

Bases: pyignite.connection.Connection

This is a main *pyignite* class, that is build upon the `Connection`. In addition to the attributes, properties and methods of its parent class, *Client* implements the following features:

- cache factory. Cache objects are used for key-value operations,
- Ignite SQL endpoint,
- binary types registration endpoint.

__init__(*compact_footer: bool = None*, **args*, ***kwargs*)

Initialize client.

Parameters **compact_footer** – (optional) use compact (True, recommended) or full (False) schema approach when serializing Complex objects. Default is to use the same approach the server is using (None). Apache Ignite binary protocol documentation on this topic: https://apacheignite.readme.io/docs/binary-client-protocol-data-format#section-schema

compact_footer
This property remembers Complex object schema encoding approach when decoding any Complex object, to use the same approach on Complex object encoding.

Returns True if compact schema was used by server or no Complex object decoding has yet taken place, False if full schema was used.

create_cache(*settings: Union[str, dict]*) → pyignite.cache.Cache
Creates Ignite cache by name. Raises *CacheError* if such a cache is already exists.

Parameters **settings** – cache name or dict of cache properties' codes and values. All cache properties are documented here: *Cache Properties*. See also the *cache creation example*,

Returns `Cache` object.

get_binary_type(*binary_type: Union[str, int]*) → dict
Gets the binary type information from the Ignite server. This is quite a low-level implementation of Ignite thin client protocol's *OP_GET_BINARY_TYPE* operation. You would probably want to use `query_binary_type()` instead.

Parameters **binary_type** – binary type name or ID,

Returns

binary type description a dict with the following fields:

- *type_exists*: True if the type is registered, False otherwise. In the latter case all the following fields are omitted,
- *type_id*: Complex object type ID,
- *type_name*: Complex object type name,
- *affinity_key_field*: string value or None,
- *is_enum*: False in case of Complex object registration,
- *schemas*: a list, containing the Complex object schemas in format: OrderedDict[field name: field type hint]. A schema can be empty.

get_cache(*settings: Union[str, dict]*) → pyignite.cache.Cache
Creates Cache object with a given cache name without checking it up on server. If such a cache does not exist, some kind of exception (most probably *CacheError*) may be raised later.

Parameters **settings** – cache name or cache properties (but only *PROP_NAME* property is allowed),

Returns `Cache` object.

get_cache_names() → list
Gets existing cache names.

Returns list of cache names.

get_or_create_cache(*settings: Union[str, dict]*) → pyignite.cache.Cache
Creates Ignite cache, if not exist.

Parameters **settings** – cache name or dict of cache properties' codes and values. All cache properties are documented here: *Cache Properties*. See also the *cache creation example*,

Returns `Cache` object.

put_binary_type(*type_name: str*, *affinity_key_field: str = None*, *is_enum=False*, *schema: dict = None*)
Registers binary type information in cluster. Do not update binary registry. This is a literal implementation of Ignite thin client protocol's *OP_PUT_BINARY_TYPE* operation. You would probably want to use `register_binary_type()` instead.

Parameters

- **type_name** – name of the data type being registered,
- **affinity_key_field** – (optional) name of the affinity key field,
- **is_enum** – (optional) register enum if True, binary object otherwise. Defaults to False,
- **schema** – (optional) when register enum, pass a dict of enumerated parameter names as keys and an integers as values. When register binary type, pass a dict of field names: field types. Binary type with no fields is OK.

query_binary_type(*binary_type: Union[int, str], schema: Union[int, dict] = None, sync: bool = True*)
Queries the registry of Complex object classes.

Parameters

- **binary_type** – Complex object type name or ID,
- **schema** – (optional) Complex object schema or schema ID,
- **sync** – (optional) look up the Ignite server for registered Complex objects and create data classes for them if needed,

Returns found dataclass or None, if *schema* parameter is provided, a dict of {schema ID: dataclass} format otherwise.

register_binary_type(*data_class: Type[CT_co], affinity_key_field: str = None*)
Register the given class as a representation of a certain Complex object type. Discards autogenerated or previously registered class.

Parameters

- **data_class** – Complex object class,
- **affinity_key_field** – (optional) affinity parameter.

sql(*query_str: str*, *page_size: int = 1*, *query_args: Iterable[T_co] = None*, *schema: Union[int, str] = 'PUBLIC'*, *statement_type: int = 0*, *distributed_joins: bool = False*, *local: bool = False*, *replicated_only: bool = False*, *enforce_join_order: bool = False*, *collocated: bool = False*, *lazy: bool = False*, *include_field_names: bool = False*, *max_rows: int = -1*, *timeout: int = 0*)
Runs an SQL query and returns its result.

Parameters

- **query_str** – SQL query string,
- **page_size** – (optional) cursor page size. Default is 1, which means that client makes one server call per row,
- **query_args** – (optional) query arguments. List of values or (value, type hint) tuples,
- **schema** – (optional) schema for the query. Defaults to *PUBLIC*,
- **statement_type** – (optional) statement type. Can be:
  - StatementType.ALL any type (default),

  - StatementType.SELECT select,
  - StatementType.UPDATE update.
- **distributed_joins** – (optional) distributed joins. Defaults to False,
- **local** – (optional) pass True if this query should be executed on local node only. Defaults to False,
- **replicated_only** – (optional) whether query contains only replicated tables or not. Defaults to False,
- **enforce_join_order** – (optional) enforce join order. Defaults to False,
- **collocated** – (optional) whether your data is co-located or not. Defaults to False,
- **lazy** – (optional) lazy query execution. Defaults to False,
- **include_field_names** – (optional) include field names in result. Defaults to False,
- **max_rows** – (optional) query-wide maximum of rows. Defaults to -1 (all rows),
- **timeout** – (optional) non-negative timeout value in ms. Zero disables timeout (default),

Returns generator with result rows as a lists. If *include_field_names* was set, the first row will hold field names.

2.2 pyignite.cache module

class pyignite.cache.**Cache**(*client: Client*, *settings: Union[str, dict] = None*, *with_get: bool = False*, *get_only: bool = False*)

Bases: object

Ignite cache abstraction. Users should never use this class directly, but construct its instances with *create_cache()*, *get_or_create_cache()* or *get_cache()* methods instead. See *this example* on how to do it.

__init__(*client: Client*, *settings: Union[str, dict] = None*, *with_get: bool = False*, *get_only: bool = False*)

Initialize cache object.

Parameters

- **client** – Ignite client,
- **settings** – cache settings. Can be a string (cache name) or a dict of cache properties and their values. In this case PROP_NAME is mandatory,
- **with_get** – (optional) do not raise exception, if the cache is already exists. Defaults to False,
- **get_only** – (optional) do not communicate with Ignite server at all, only create Cache instance. Defaults to False.

cache_id

Cache ID.

Returns integer value of the cache ID.

clear(*keys: Union[list, NoneType] = None*)

Clears the cache without notifying listeners or cache writers.

Parameters **keys** – (optional) list of cache keys or (key, key type hint) tuples to clear (default: clear all).

clear_key(*key*, *key_hint: object = None*)
Clears the cache key without notifying listeners or cache writers.

Parameters

- **key** – key for the cache entry,
- **key_hint** – (optional) Ignite data type, for which the given key should be converted,

client
Ignite *Client* object.

Returns Client object, through which the cache is accessed.

contains_key(*key*, *key_hint=None*) → bool
Returns a value indicating whether given key is present in cache.

Parameters

- **key** – key for the cache entry. Can be of any supported type,
- **key_hint** – (optional) Ignite data type, for which the given key should be converted,

Returns boolean *True* when key is present, *False* otherwise.

contains_keys(*keys: Iterable[T_co]*) → bool
Returns a value indicating whether all given keys are present in cache.

Parameters **keys** – a list of keys or (key, type hint) tuples,

Returns boolean *True* when all keys are present, *False* otherwise.

destroy()
Destroys cache with a given name.

get(*key*, *key_hint: object = None*) → Any
Retrieves a value from cache by key.

Parameters

- **key** – key for the cache entry. Can be of any supported type,
- **key_hint** – (optional) Ignite data type, for which the given key should be converted,

Returns value retrieved.

get_all(*keys: list*) → list
Retrieves multiple key-value pairs from cache.

Parameters **keys** – list of keys or tuples of (key, key_hint),

Returns a dict of key-value pairs.

get_and_put(*key*, *value*, *key_hint=None*, *value_hint=None*) → Any
Puts a value with a given key to cache, and returns the previous value for that key, or null value if there was not such key.

Parameters

- **key** – key for the cache entry. Can be of any supported type,
- **value** – value for the key,
- **key_hint** – (optional) Ignite data type, for which the given key should be converted,
- **value_hint** – (optional) Ignite data type, for which the given value should be converted.

Returns old value or None.

get_and_put_if_absent(*key*, *value*, *key_hint=None*, *value_hint=None*)
Puts a value with a given key to cache only if the key does not already exist.

Parameters

- **key** – key for the cache entry. Can be of any supported type,
- **value** – value for the key,
- **key_hint** – (optional) Ignite data type, for which the given key should be converted,
- **value_hint** – (optional) Ignite data type, for which the given value should be converted,

Returns old value or None.

get_and_remove(*key*, *key_hint=None*) → Any
Removes the cache entry with specified key, returning the value.

Parameters

- **key** – key for the cache entry. Can be of any supported type,
- **key_hint** – (optional) Ignite data type, for which the given key should be converted,

Returns old value or None.

get_and_replace(*key*, *value*, *key_hint=None*, *value_hint=None*) → Any
Puts a value with a given key to cache, returning previous value for that key, if and only if there is a value currently mapped for that key.

Parameters

- **key** – key for the cache entry. Can be of any supported type,
- **value** – value for the key,
- **key_hint** – (optional) Ignite data type, for which the given key should be converted,
- **value_hint** – (optional) Ignite data type, for which the given value should be converted.

Returns old value or None.

get_size(*peek_modes=0*)
Gets the number of entries in cache.

Parameters **peek_modes** – (optional) limit count to near cache partition (PeekModes.NEAR), primary cache (PeekModes.PRIMARY), or backup cache (PeekModes.BACKUP). Defaults to all cache partitions (PeekModes.ALL),

Returns integer number of cache entries.

name
Lazy cache name.

Returns cache name string.

put(*key*, *value*, *key_hint: object = None*, *value_hint: object = None*)
Puts a value with a given key to cache (overwriting existing value if any).

Parameters

- **key** – key for the cache entry. Can be of any supported type,
- **value** – value for the key,
- **key_hint** – (optional) Ignite data type, for which the given key should be converted,
- **value_hint** – (optional) Ignite data type, for which the given value should be converted.

put_all(*pairs: dict*)
Puts multiple key-value pairs to cache (overwriting existing associations if any).

Parameters **pairs** – dictionary type parameters, contains key-value pairs to save. Each key or value can be an item of representable Python type or a tuple of (item, hint),

put_if_absent(*key*, *value*, *key_hint=None*, *value_hint=None*)
Puts a value with a given key to cache only if the key does not already exist.

Parameters

- **key** – key for the cache entry. Can be of any supported type,
- **value** – value for the key,
- **key_hint** – (optional) Ignite data type, for which the given key should be converted,
- **value_hint** – (optional) Ignite data type, for which the given value should be converted.

remove_all()
Removes all cache entries, notifying listeners and cache writers.

remove_if_equals(*key*, *sample*, *key_hint=None*, *sample_hint=None*)
Removes an entry with a given key if provided value is equal to actual value, notifying listeners and cache writers.

Parameters

- **key** – key for the cache entry,
- **sample** – a sample to compare the stored value with,
- **key_hint** – (optional) Ignite data type, for which the given key should be converted,
- **sample_hint** – (optional) Ignite data type, for whic the given sample should be converted.

remove_key(*key*, *key_hint=None*)
Clears the cache key without notifying listeners or cache writers.

Parameters

- **key** – key for the cache entry,
- **key_hint** – (optional) Ignite data type, for which the given key should be converted,

remove_keys(*keys: list*)
Removes cache entries by given list of keys, notifying listeners and cache writers.

Parameters **keys** – list of keys or tuples of (key, key_hint) to remove.

replace(*key*, *value*, *key_hint: object = None*, *value_hint: object = None*)
Puts a value with a given key to cache only if the key already exist.

Parameters

- **key** – key for the cache entry. Can be of any supported type,
- **value** – value for the key,
- **key_hint** – (optional) Ignite data type, for which the given key should be converted,
- **value_hint** – (optional) Ignite data type, for which the given value should be converted.

replace_if_equals(*key*, *sample*, *value*, *key_hint=None*, *sample_hint=None*, *value_hint=None*) → Any
Puts a value with a given key to cache only if the key already exists and value equals provided sample.

Parameters

- **key** – key for the cache entry,
- **sample** – a sample to compare the stored value with,
- **value** – new value for the given key,
- **key_hint** – (optional) Ignite data type, for which the given key should be converted,
- **sample_hint** – (optional) Ignite data type, for whic the given sample should be converted
- **value_hint** – (optional) Ignite data type, for which the given value should be converted,

Returns boolean *True* when key is present, *False* otherwise.

scan(*page_size: int = 1*, *partitions: int = -1*, *local: bool = False*)

Returns all key-value pairs from the cache, similar to *get_all*, but with internal pagination, which is slower, but safer.

Parameters

- **page_size** – (optional) page size. Default size is 1 (slowest and safest),
- **partitions** – (optional) number of partitions to query (negative to query entire cache),
- **local** – (optional) pass True if this query should be executed on local node only. Defaults to False,

Returns generator with key-value pairs.

select_row(*query_str: str*, *page_size: int = 1*, *query_args: Union[list, NoneType] = None*, *distributed_joins: bool = False*, *replicated_only: bool = False*, *local: bool = False*, *timeout: int = 0*)

Executes a simplified SQL SELECT query over data stored in the cache. The query returns the whole record (key and value).

Parameters

- **query_str** – SQL query string,
- **page_size** – (optional) cursor page size. Default is 1, which means that client makes one server call per row,
- **query_args** – (optional) query arguments,
- **distributed_joins** – (optional) distributed joins. Defaults to False,
- **replicated_only** – (optional) whether query contains only replicated tables or not. Defaults to False,
- **local** – (optional) pass True if this query should be executed on local node only. Defaults to False,
- **timeout** – (optional) non-negative timeout value in ms. Zero disables timeout (default),

Returns generator with key-value pairs.

settings

Lazy Cache settings. See the *example* of reading this property.

All cache properties are documented here: *Cache Properties*.

Returns dict of cache properties and their values.

2.3 Data Types

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 type 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 that parser/constructor classes are not instantiatable. The *class* here is used mostly as a sort of tupperware for organizing methods together.

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`
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`

Continued on next page

Table 1 – continued from previous page

type_code	Apache Ignite docs reference	Python type or class	Parser/constructor class
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	object	`BinaryObject`
0x1b	Wrapped data	tuple	`WrappedDataObject`

2.4 Cache Properties

The `prop_codes` module contains a list of ordinal values, that represent various cache settings.

Please refer to the Apache Ignite Data Grid documentation on cache synchronization, rebalance, affinity and other cache configuration-related matters.

Property name	Ordinal value	Property type	Description
Read/write cache properties, used to configure cache via *`create_cache()`* or *`get_or_create_`*			
PROP_NAME	0	str	Cache name.
PROP_CACHE_MODE	1	int	Cache mode:
PROP_CACHE_ATOMICITY_MODE	2	int	Cache atomici
PROP_BACKUPS_NUMBER	3	int	Number of ba
PROP_WRITE_SYNCHRONIZATION_MODE	4	int	Write synchro
PROP_COPY_ON_READ	5	bool	Copy-on-read
PROP_READ_FROM_BACKUP	6	bool	Read from bac
PROP_DATA_REGION_NAME	100	str	Data region na
PROP_IS_ONHEAP_CACHE_ENABLED	101	bool	Is OnHeap cac
PROP_QUERY_ENTITIES	200	list	A list of query
PROP_QUERY_PARALLELISM	201	int	Query parallel
PROP_QUERY_DETAIL_METRIC_SIZE	202	int	Query detail n
PROP_SQL_SCHEMA	203	str	SQL schema
PROP_SQL_INDEX_INLINE_MAX_SIZE	204	int	SQL index inl
PROP_SQL_ESCAPE_ALL	205	bool	Turns on SQL
PROP_MAX_QUERY_ITERATORS	206	int	Maximum nur
PROP_REBALANCE_MODE	300	int	Rebalance mo
PROP_REBALANCE_DELAY	301	int	Rebalance del
PROP_REBALANCE_TIMEOUT	302	int	Rebalance tim
PROP_REBALANCE_BATCH_SIZE	303	int	Rebalance bat
PROP_REBALANCE_BATCHES_PREFETCH_COUNT	304	int	Rebalance bat
PROP_REBALANCE_ORDER	305	int	Rebalance ord
PROP_REBALANCE_THROTTLE	306	int	Rebalance thr
PROP_GROUP_NAME	400	str	Group name
PROP_CACHE_KEY_CONFIGURATION	401	list	Cache key con
PROP_DEFAULT_LOCK_TIMEOUT	402	int	Default lock t

Table 2 – continued from

Property name	Ordinal value	Property type	Description
PROP_MAX_CONCURRENT_ASYNC_OPERATIONS	403	int	Maximum nu
PROP_PARTITION_LOSS_POLICY	404	int	Partition loss
PROP_EAGER_TTL	405	bool	Eager TTL
PROP_STATISTICS_ENABLED	406	bool	Statistics enab
Read-only cache properties. Can not be set, but only retrieved via `settings()`			
PROP_INVALIDATE	-1	bool	Invalidate

2.4.1 Query entity

A dict with all ot the following keys:

- *table_name*: SQL table name,
- *key_field_name*: name of the key field,
- *key_type_name*: name of the key type (Java type or complex object),
- *value_field_name*: name of the value field,
- *value_type_name*: name of the value type,
- *field_name_aliases*: a list of 0 or more dicts of aliases (see *Field name alias*),
- *query_fields*: a list of 0 or more query field names (see *Query field*),
- *query_indexes*: a list of 0 or more query indexes (see *Query index*).

Field name alias

- *field_name*: field name,
- *alias*: alias (str).

Query field

- *name*: field name,
- *type_name*: name of Java type or complex object,
- *is_key_field*: (optional) boolean value, *False* by default,
- *is_notnull_constraint_field*: boolean value,
- *default_value*: (optional) anything that can be converted to *type_name* type. *None* (`Null`) by default,
- *precision* (optional) decimal precision: total number of digits in decimal value. Defaults to -1 (use cluster default). Ignored for non-decimal SQL types (other than *java.math.BigDecimal*),
- *scale* (optional) decimal precision: number of digits after the decimal point. Defaults to -1 (use cluster default). Ignored for non-decimal SQL types.

Query index

- *index_name*: index name,
- *index_type*: index type code as an integer value in unsigned byte range,

- *inline_size*: integer value,
- *fields*: a list of 0 or more indexed fields (see *Fields*).

Fields

- *name*: field name,
- *is_descending*: (optional) boolean value, *False* by default.

2.4.2 Cache key

A dict of the following format:

- *type_name*: name of the complex object,
- *affinity_key_field_name*: name of the affinity key field.

2.5 pygnite.exceptions module

exception pygnite.exceptions.**BinaryTypeError**
Bases: *pygnite.exceptions.CacheError*

A remote error in operation with Complex Object registry.

exception pygnite.exceptions.**CacheCreationError**
Bases: *pygnite.exceptions.CacheError*

This exception is raised, when any complex operation failed on cache creation phase.

exception pygnite.exceptions.**CacheError**
Bases: Exception

This exception is raised, whenever any remote Thin client operation returns an error.

exception pygnite.exceptions.**HandshakeError**
Bases: OSError

This exception is raised on Ignite binary protocol handshake failure, as defined in https://apacheignite.readme.io/docs/binary-client-protocol#section-handshake

exception pygnite.exceptions.**ParameterError**
Bases: Exception

This exception represents the parameter validation error in any *pyignite* method.

exception pygnite.exceptions.**ParseError**
Bases: Exception

This exception is raised, when *pyignite* is unable to build a query to, or parse a response from, Ignite node.

exception pygnite.exceptions.**ReconnectError**
Bases: Exception

This exception is raised by *Client.reconnect* method, when no more nodes are left to connect to. It is not meant to be an error, but rather a flow control tool, similar to *StopIteration*.

exception pyignite.exceptions.**SQLError**
Bases: *pyignite.exceptions.CacheError*

An error in SQL query.

CHAPTER 3

Examples of usage

File: get_and_put.py.

3.1 Key-value

3.1.1 Open connection

```
from pyignite import Client

client = Client()
client.connect('127.0.0.1', 10800)
```

3.1.2 Create cache

```
my_cache = client.create_cache('my cache')
```

3.1.3 Put value in cache

```
my_cache.put('my key', 42)
```

3.1.4 Get value from cache

```
result = my_cache.get('my key')
print(result)  # 42

result = my_cache.get('non-existent key')
print(result)  # None
```

3.1.5 Get multiple values from cache

```
result = my_cache.get_all([
    'my key',
    'non-existent key',
    'other-key',
])
print(result)  # {'my key': 42}
```

3.1.6 Type hints usage

File: type_hints.py

```
my_cache.put('my key', 42)
# value '42' takes 9 bytes of memory as a LongObject

my_cache.put('my key', 42, value_hint=ShortObject)
# value '42' takes only 3 bytes as a ShortObject

my_cache.put('a', 1)
# 'a' is a key of type String

my_cache.put('a', 2, key_hint=CharObject)
# another key 'a' of type CharObject was created

value = my_cache.get('a')
print(value)
# 1

value = my_cache.get('a', key_hint=CharObject)
print(value)
# 2

# now let us delete both keys at once
my_cache.remove_keys([
    'a',                # a default type key
    ('a', CharObject),  # a key of type CharObject
])
```

As a rule of thumb:

- when a *pyignite* method or function deals with a single value or key, it has an additional parameter, like *value_hint* or *key_hint*, which accepts a parser/constructor class,
- nearly any structure element (inside dict or list) can be replaced with a two-tuple of (said element, type hint).

Refer the *Data Types* section for the full list of parser/constructor classes you can use as type hints.

3.1.7 Scan

File: scans.py.

Cache's `scan()` method queries allows you to get the whole contents of the cache, element by element.

Let us put some data in cache.

```
my_cache.put_all({'key_{}'.format(v): v for v in range(20)})
# {
#     'key_0': 0,
#     'key_1': 1,
#     'key_2': 2,
#     ... 20 elements in total...
#     'key_18': 18,
#     'key_19': 19
# }

result = my_cache.scan()
```

scan() returns a generator, that yields two-tuples of key and value. You can iterate through the generated pairs in a safe manner:

```
for k, v in result:
    print(k, v)
# 'key_17' 17
# 'key_10' 10
# 'key_6' 6,
# ... 20 elements in total...
# 'key_16' 16
# 'key_12' 12
```

Or, alternatively, you can convert the generator to dictionary in one go:

```
print(dict(result))
# {
#     'key_17': 17,
#     'key_10': 10,
#     'key_6': 6,
#     ... 20 elements in total...
#     'key_16': 16,
#     'key_12': 12
# }
```

But be cautious: if the cache contains a large set of data, the dictionary may eat too much memory!

3.1.8 Do cleanup

Destroy created cache and close connection.

```
my_cache.destroy()
client.close()
```

3.2 SQL

File: sql.py.

These 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.

```
client = Client()
client.connect('127.0.0.1', 10800)
```

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 BOOLEAN,
    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,
]:
    client.sql(query)
```

Create indexes.

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

LANGUAGE_CREATE_INDEX = '''
CREATE INDEX idx_lang_country_code ON CountryLanguage (CountryCode)'''
```

(continues on next page)

(continued from previous page)

```
for query in [CITY_CREATE_INDEX, LANGUAGE_CREATE_INDEX]:
    client.sql(query)
```

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:
    client.sql(COUNTRY_INSERT_QUERY, query_args=row)

for row in CITY_DATA:
    client.sql(CITY_INSERT_QUERY, query_args=row)

for row in LANGUAGE_DATA:
    client.sql(LANGUAGE_INSERT_QUERY, query_args=row)
```

Data samples are 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)?

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

result = client.sql(MOST_POPULATED_QUERY)
print('Most 10 populated cities:')
for row in result:
    print(row)
# Most 10 populated cities:
# ['Mumbai (Bombay)', 10500000]
# ['Shanghai', 9696300]
# ['New York', 8008278]
# ['Peking', 7472000]
# ['Delhi', 7206704]
# ['Chongqing', 6351600]
# ['Tianjin', 5286800]
# ['Calcutta [Kolkata]', 4399819]
# ['Wuhan', 4344600]
# ['Harbin', 4289800]
```

The `sql()` method returns a generator, that yields the resulting rows.

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

If you set the *include_field_names* argument to *True*, the `sql()` method will generate a list of column names as a first yield. You can access field names with Python built-in *next* function.

```
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 = client.sql(
    MOST_POPULATED_IN_3_COUNTRIES_QUERY,
    include_field_names=True,
)
print('Most 10 populated cities in USA, India and China:')
print(next(result))
print('----------------------------------------')
for row in result:
    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]
# ['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 = client.sql(
    CITY_INFO_QUERY,
    query_args=[3802],
    include_field_names=True,
)
field_names = next(result)
field_data = list(*result)

print('City info:')
for field_name, field_value in zip(field_names*len(field_data), field_data):
    print('{}: {}'.format(field_name, field_value))
# City info:
# ID: 3802
```

(continues on next page)

(continued from previous page)

```
# 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 {} IF EXISTS'''

for table_name in [
    CITY_TABLE_NAME,
    LANGUAGE_TABLE_NAME,
    COUNTRY_TABLE_NAME,
]:
    result = client.sql(DROP_TABLE_QUERY.format(table_name))
```

3.3 Complex objects

File: binary_basics.py.

Complex object (that is often called 'Binary object') is an Ignite data type, that is designed to represent a Java class. It have the following features:

- have a unique ID (type id), which is derives from a class name (type name),
- have one or more associated schemas, that describes its inner structure (the order, names and types of its fields). Each schema have its own ID,
- have an optional version number, that is aimed towards the end users to help them distinguish between objects of the same type, serialized with different schemas.

Unfortunately, these distinctive features of the Complex object have few to no meaning outside of Java language. Python class can not be defined by its name (it is not unique), ID (object ID in Python is volatile; in CPython it is just a pointer in the interpreter's memory heap), or complex of its fields (they do not have an associated data types, moreover, they can be added or deleted in run-time). For the *pyignite* user it means that for all purposes of storing native Python data it is better to use Ignite `CollectionObject` or `MapObject` data types.

However, for interoperability purposes, *pyignite* has a mechanism of creating special Python classes to read or write Complex objects. These classes have an interface, that simulates all the features of the Complex object: type name, type ID, schema, schema ID, and version number.

Assuming that one concrete class for representing one Complex object can severely limit the user's data manipulation capabilities, all the functionality said above is implemented through the metaclass: `GenericObjectMeta`. This metaclass is used automatically when reading Complex objects.

```
from pyignite import Client, GenericObjectMeta
from pyignite.datatypes import *

client = Client()
client.connect('localhost', 10800)

person_cache = client.get_or_create_cache('person')

person = person_cache.get(1)
print(person.__class__.__name__)
```

(continues on next page)

(continued from previous page)

```
# Person

print(person)
# Person(first_name='Ivan', last_name='Ivanov', age=33, version=1)
```

Here you can see how `GenericObjectMeta` uses attrs package internally for creating nice *__init__()* and *__repr__()* methods.

You can reuse the autogenerated class for subsequent writes:

```
Person = person.__class__

person_cache.put(
    1, Person(first_name='Ivan', last_name='Ivanov', age=33)
)
```

`GenericObjectMeta` can also be used directly for creating custom classes:

```
class Person(metaclass=GenericObjectMeta, schema=OrderedDict([
    ('first_name', String),
    ('last_name', String),
    ('age', IntObject),
])):
    pass
```

Note how the *Person* class is defined. *schema* is a `GenericObjectMeta` metaclass parameter. Another important `GenericObjectMeta` parameter is a *type_name*, but it is optional and defaults to the class name ('Person' in our example).

Note also, that *Person* do not have to define its own attributes, methods and properties (*pass*), although it is completely possible.

Now, when your custom *Person* class is created, you are ready to send data to Ignite server using its objects. The client will implicitly register your class as soon as the first Complex object is sent. If you intend to use your custom class for reading existing Complex objects' values before all, you must register said class explicitly with your client:

```
client.register_binary_type(Person)
```

Now, when we dealt with the basics of *pyignite* implementation of Complex Objects, let us move on to more elaborate examples.

3.3.1 Read

File: read_binary.py.

Ignite SQL uses Complex objects internally to represent keys and rows in SQL tables. Normally SQL data is accessed via queries (see *SQL*), so we will consider the following example solely for the demonstration of how Binary objects (not Ignite SQL) work.

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

```
result = client.get_cache_names()
print(result)
# [
#     'SQL_PUBLIC_CITY',
```

(continues on next page)

(continued from previous page)

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

We can see that Ignite created a cache for each of our tables. 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 `settings` property.

```
city_cache = client.get_or_create_cache('SQL_PUBLIC_CITY')
print(city_cache.settings[PROP_NAME])
# 'SQL_PUBLIC_CITY'

print(city_cache.settings[PROP_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. The *City* table's key fields are stored using *key_type_name* type, and the other fields *value_type_name* type.

Now when we have the cache, in which the SQL data resides, and the names of the key and value data types, we can read the data without using SQL functions and verify the correctness of the result.

```
result = city_cache.scan()
print(next(result))
# (
#     SQL_PUBLIC_CITY_6fe650e1_700f_4e74_867d_58f52f433c43_KEY(
#         ID=1890,
#         COUNTRYCODE='CHN',
#         version=1
#     ),
#     SQL_PUBLIC_CITY_6fe650e1_700f_4e74_867d_58f52f433c43(
#         NAME='Shanghai',
#         DISTRICT='Shanghai',
#         POPULATION=9696300,
#         version=1
#     )
# )
```

What we see is a tuple of key and value, extracted from the cache. Both key and value are represent Complex objects. The dataclass names are the same as the *value_type_name* and *key_type_name* cache settings. The objects' fields correspond to the SQL query.

3.3.2 Create

File: create_binary.py.

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 table cache.

```
client = Client()
client.connect('127.0.0.1', 10800)

student_cache = client.create_cache({
        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,
                    },
                    {
                        'name': 'NAME',
                        'type_name': 'java.lang.String',
                    },
                    {
                        'name': 'LOGIN',
                        'type_name': 'java.lang.String',
                    },
                    {
                        'name': 'AGE',
                        'type_name': 'java.lang.Integer',
                    },
                    {
                        'name': 'GPA',
                        'type_name': 'java.math.Double',
                    },
                ],
                'query_indexes': [],
```

(continues on next page)

(continued from previous page)

```
                'value_type_name': 'SQL_PUBLIC_STUDENT_TYPE',
                'value_field_name': None,
            },
        ],
    })
```

2. Define Complex object data class.

```
class Student(
    metaclass=GenericObjectMeta,
    type_name='SQL_PUBLIC_STUDENT_TYPE',
    schema=OrderedDict([
        ('NAME', String),
        ('LOGIN', String),
        ('AGE', IntObject),
        ('GPA', DoubleObject),
    ])
):
    pass
```

3. Insert row.

```
student_cache.put(
    1,
    Student(LOGIN='jdoe', NAME='John Doe', AGE=17, GPA=4.25),
    key_hint=IntObject
)
```

Now let us make sure that our cache really can be used with SQL functions.

```
result = client.sql(
    r'SELECT * FROM Student',
    include_field_names=True
)
print(next(result))
# ['SID', 'NAME', 'LOGIN', 'AGE', 'GPA']

print(*result)
# [1, 'John Doe', 'jdoe', 17, 4.25]
```

Note, however, that the cache we create can not be dropped with DDL command.

```
# DROP_QUERY = 'DROP TABLE Student'
# client.sql(DROP_QUERY)
#
# pyignite.exceptions.SQLError: class org.apache.ignite.IgniteCheckedException:
# Only cache created with CREATE TABLE may be removed with DROP TABLE
# [cacheName=SQL_PUBLIC_STUDENT]
```

It should be deleted as any other key-value cache.

```
student_cache.destroy()
```

3.3.3 Migrate

File: migrate_binary.py.

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 get the vouchers' cache.

```
client = Client()
client.connect('127.0.0.1', 10800)

accounting = client.get_or_create_cache('accounting')
```

If you do not store the schema of the Complex object in code, you can obtain it as a dataclass property with *query_binary_type()* method.

```
data_classes = client.query_binary_type('ExpenseVoucher')
print(data_classes)
# {
#     -231598180: <class '__main__.ExpenseVoucher'>
# }

s_id, data_class = data_classes.popitem()
schema = data_class.schema
```

Let us modify the schema and create a new Complex object class with an updated schema.

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


# define new data class
class ExpenseVoucherV2(
    metaclass=GenericObjectMeta,
    type_name='ExpenseVoucher',
    schema=schema,
):
    pass
```

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

```
def migrate(cache, data, new_class):
    """ Migrate given data pages. """
    for key, old_value in data:
        # read data
        print(old_value)
        # ExpenseVoucher(
        #     date=datetime(2017, 9, 21, 0, 0),
        #     reported=True,
        #     purpose='Praesent eget fermentum massa',
        #     sum=Decimal('666.67'),
```

(continues on next page)

(continued from previous page)

```
        #     recipient='John Doe',
        #     cashier_id=8,
        #     version=1
        # )

        # create new binary object
        new_value = new_class()

        # process data
        new_value.sum = old_value.sum
        new_value.purpose = old_value.purpose
        new_value.recipient = old_value.recipient
        new_value.cashier_id = old_value.cashier_id
        new_value.expense_date = old_value.date
        new_value.report_date = date.today() if old_value.reported else None

        # replace data
        cache.put(key, new_value)

        # verify data
        verify = cache.get(key)
        print(verify)
        # ExpenseVoucherV2(
        #     purpose='Praesent eget fermentum massa',
        #     sum=Decimal('666.67'),
        #     recipient='John Doe',
        #     cashier_id=8,
        #     expense_date=datetime(2017, 9, 21, 0, 0),
        #     report_date=datetime(2018, 8, 29, 0, 0),
        #     version=1,
        # )


# migrate data
result = accounting.scan()
migrate(accounting, result, ExpenseVoucherV2)

# cleanup
accounting.destroy()
client.close()
```

At this moment all the fields, defined in both of our schemas, can be available in the resulting binary object, depending on which schema was used when writing it using `put()` or similar methods. Ignite Binary API do not have the method to delete Complex object schema; all the schemas ever defined will stay in cluster until its shutdown.

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, you can rename the field or create a new Complex object.

3.4 Failover

File: failover.py.

When connection to the server is broken or timed out, *Client* object propagates an original exception (*OSError* or *SocketError*), but keeps its constructor's parameters intact and tries to reconnect transparently.

When there's no way for *Client* to reconnect, it raises a special *ReconnectError* exception.

The following example features a simple node list traversal failover mechanism. Gather 3 Ignite nodes on *localhost* into one cluster and run:

```
from pyignite import Client
from pyignite.datatypes.cache_config import CacheMode
from pyignite.datatypes.prop_codes import *
from pyignite.exceptions import SocketError


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

client = Client(timeout=4.0)
client.connect(nodes)
print('Connected to {}'.format(client))

my_cache = client.get_or_create_cache({
    PROP_NAME: 'my_cache',
    PROP_CACHE_MODE: CacheMode.REPLICATED,
})
my_cache.put('test_key', 0)

# abstract main loop
while True:
    try:
        # do the work
        test_value = my_cache.get('test_key')
        my_cache.put('test_key', test_value + 1)
    except (OSError, SocketError) as e:
        # recover from error (repeat last command, check data
        # consistency or just continue  depends on the task)
        print('Error: {}'.format(e))
        print('Last value: {}'.format(my_cache.get('test_key')))
        print('Reconnected to {}'.format(client))
```

Then try shutting down and restarting nodes, and see what happens.

```
# Connected to 127.0.0.1:10800
# Error: [Errno 104] Connection reset by peer
# Last value: 6999
# Reconnected to 127.0.0.1:10801
# Error: Socket connection broken.
# Last value: 12302
# Reconnected to 127.0.0.1:10802
# Error: [Errno 111] Client refused
# Traceback (most recent call last):
#     ...
# pyignite.exceptions.ReconnectError: Can not reconnect: out of nodes
```

Client reconnection do not require an explicit user action, like calling a special method or resetting a parameter. Note, however, that reconnection is lazy: it happens only if (and when) it is needed. In this example, the automatic

reconnection happens, when the script checks upon the last saved value:

```
        print('Last value: {}'.format(my_cache.get('test_key')))
```

It means that instead of checking the connection status it is better for *pyignite* user to just try the supposed data operations and catch the resulting exception.

`connect()` method accepts any iterable, not just list. It means that you can implement any reconnection policy (round-robin, nodes prioritization, pause on reconnect or graceful backoff) with a generator.

pyignite comes with a sample `RoundRobin` generator. In the above example try to replace

```
client.connect(nodes)
```

with

```
client.connect(RoundRobin(nodes, max_reconnects=20))
```

The client will try to reconnect to node 1 after node 3 is crashed, then to node 2, et c. At least one node should be active for the `RoundRobin` to work properly.

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: *`Client`*.

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

```
from pyignite import Client

client = Client(use_ssl=True)
client.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 import Client

client = Client(
    use_ssl=True,
    ssl_keyfile='etc/.ssl/keyfile.key',
    ssl_certfile='etc/.ssl/certfile.crt',
)
client.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 import Client

client = Client(
    use_ssl=True,
    ssl_ca_certfile='etc/.ssl/ca_certs',
    ssl_cert_reqs=ssl.CERT_REQUIRED,
)
client.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.

3.6 Password authentication

To authenticate you must set *authenticationEnabled* property to *true* and enable persistance in Ignite XML configuration file, as described in Authentication section of Ignite documentation.

Be advised that sending credentials over the open channel is greatly discouraged, since they can be easily intercepted. Supplying credentials automatically turns SSL on from the client side. It is highly recommended to secure the connection to the Ignite server, as described in *SSL/TLS* example, in order to use password authentication.

Then just supply *username* and *password* parameters to `Client` constructor.

```
from pyignite import Client

client = Client(username='ignite', password='ignite')
client.connect('ignite-example.com', 10800)
```

If you still do not wish to secure the connection is spite of the warning, then disable SSL explicitly on creating the client object:

```
client = Client(username='ignite', password='ignite', use_ssl=False)
```

Note, that it is not possible for Ignite thin client to obtain the cluster's authentication settings through the binary protocol. Unexpected credentials are simply ignored by the server. In the opposite case, the user is greeted with the following message:

```
# pyignite.exceptions.HandshakeError: Handshake error: Unauthenticated sessions are␣
↪prohibited.
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index