clickhouse-sqlalchemy Documentation

Release 0.2.4

clickhouse-sqlalchemy authors

Jun 30, 2023

Contents

1 User's Guide **3**
- 1.1 Installation . . . 3
- 1.2 Quickstart . . . 4
- 1.3 Connection configuration . . . 5
- 1.4 Features . . . 7
- 1.5 Types . . . 22
- 1.6 Migrations . . . 22

2 Additional Notes **25**
- 2.1 Development . . . 25
- 2.2 Changelog . . . 26
- 2.3 License . . . 26
- 2.4 How to Contribute . . . 26

Release 0.2.4.

Supported SQLAlchemy: 1.4.

Welcome to clickhouse-sqlalchemy's documentation. Get started with *Installation* and then get an overview with the *Quickstart* where common queries are described.

CHAPTER 1

User's Guide

This part of the documentation focuses on step-by-step instructions for development with clickhouse-sqlalchemy.

It assumes that you have experience with SQLAlchemy. Consider its docs at the first, if that is not so. Experience with ClickHouse is also highly recommended.

ClickHouse server provides a lot of interfaces. This dialect supports:

- HTTP interface (port 8123 by default);
- Native (TCP) interface (port 9000 by default).

Each interface has it's own support by corresponding "driver":

- **http** via `requests`
- **native** via `clickhouse-driver` or via `asynch` for async support

Native driver is recommended due to rich `clickhouse-driver` support. HTTP driver has poor development support compare to native driver. However default driver is still `http`.

1.1 Installation

1.1.1 Python Version

Clickhouse-sqlalchemy supports Python 2.7 and newer.

1.1.2 Dependencies

These distributions will be installed automatically when installing clickhouse-sqlalchemy:

- clickhouse-driver ClickHouse Python Driver with native (TCP) interface support.
- requests a simple and elegant HTTP library.
- ipaddress backport ipaddress module.

- asynch An asyncio ClickHouse Python Driver with native (TCP) interface support.

If you are planning to use `clickhouse-driver` with compression you should also install compression extras as well. See clickhouse-driver documentation.

1.1.3 Installation from PyPI

The package can be installed using `pip`:

```
pip install clickhouse-sqlalchemy
```

1.1.4 Installation from github

Development version can be installed directly from github:

```
pip install git+https://github.com/xzkostyan/clickhouse-sqlalchemy@master
↪#egg=clickhouse-sqlalchemy
```

1.2 Quickstart

This page gives a good introduction to clickhouse-sqlalchemy. It assumes you already have clickhouse-sqlalchemy installed. If you do not, head over to the *Installation* section.

It should be pointed that session must be created with `clickhouse_sqlalchemy.make_session`. Otherwise `session.query` and `session.execute` will not have ClickHouse SQL extensions. The same is applied to `Table` and `get_declarative_base`.

Let's define some table, insert data into it and query inserted data.

```
from sqlalchemy import create_engine, Column, MetaData

from clickhouse_sqlalchemy import (
    Table, make_session, get_declarative_base, types, engines
)

uri = 'clickhouse+native://localhost/default'

engine = create_engine(uri)
session = make_session(engine)
metadata = MetaData(bind=engine)

Base = get_declarative_base(metadata=metadata)

class Rate(Base):
    day = Column(types.Date, primary_key=True)
    value = Column(types.Int32)

    __table_args__ = (
        engines.Memory(),
    )

# Emits CREATE TABLE statement
Rate.__table__.create()
```

Now it's time to insert some data

```
from datetime import date, timedelta

from sqlalchemy import func

today = date.today()
rates = [
    {'day': today - timedelta(i), 'value': 200 - i}
    for i in range(100)
]
```

Let's query inserted data

```
session.execute(Rate.__table__.insert(), rates)

session.query(func.count(Rate.day)) \
    .filter(Rate.day > today - timedelta(20)) \
    .scalar()
```

Now you are ready to *configure your connection* and see more ClickHouse *features* support.

1.3 Connection configuration

ClickHouse SQLAlchemy uses the following syntax for the connection string:

Where:

- **driver** is driver to use. Possible choices: `http`, `native`, `asynch`. `http` is default. When you omit driver http is used.
- **database** is database connect to. Default is `default`.
- **user** is database user. Defaults to `'default'`.
- **password** of the user. Defaults to `''` (no password).
- **port** can be customized if ClickHouse server is listening on non-standard port.

Additional parameters are passed to driver.

1.3.1 Common options

- **engine_reflection** controls table engine reflection during table reflection. Engine reflection can be very slow if you have thousand of tables. You can disable reflection by setting this parameter to `false`. Possible choices: `true`/`false`. Default is `true`.
- **server_version** can be used for eliminating initialization `select version()` query. Generally you shouldn't set this parameter and server version will be detected automatically.

1.3.2 Driver options

There are several options can be specified in query string.

HTTP

- **port** is port ClickHouse server is bound to. Default is `8123`.
- **timeout** in seconds. There is no timeout by default.
- **protocol** to use. Possible choices: `http`, `https`. `http` is default.
- **verify** controls certificate verification in `https` protocol. Possible choices: `true`/`false`. Default is `true`.

Simple DSN example:

```
clickhouse+http://host/db
```

DSN example for ClickHouse https port:

```
clickhouse+http://user:password@host:8443/db?protocol=https
```

When you are using *nginx* as proxy server for ClickHouse server connection string might look like:

```
clickhouse+http://user:password@host:8124/test?protocol=https
```

Where `8124` is proxy port.

If you need control over the underlying HTTP connection, pass a requests.Session instance to `create_engine()`, like so:

```
from sqlalchemy import create_engine
from requests import Session

engine = create_engine(
    'clickhouse+http://localhost/test',
    connect_args={'http_session': Session()}
)
```

Native

Please note that native connection **is not encrypted**. All data including user/password is transferred in plain text. You should use this connection over SSH or VPN (for example) while communicating over untrusted network.

Simple DSN example:

```
clickhouse+native://host/db
```

All connection string parameters are proxied to `clickhouse-driver`. See it's parameters.

Example DSN with LZ4 compression secured with Let's Encrypt certificate on server side:

```
import certify

dsn = (
    'clickhouse+native://user:pass@host/db?compression=lz4&'
    'secure=True&ca_certs={}'.format(certify.where())
)
```

Example with multiple hosts

```
clickhouse+native://wronghost/default?alt_hosts=localhost:9000
```

Asynch

Same as Native.

Simple DSN example:

```
clickhouse+asynch://host/db
```

All connection string parameters are proxied to `asynch`. See it's parameters.

1.4 Features

This section describes features that current dialect supports.

1.4.1 Tables and models definition

Both declarative and constructor-style tables supported:

```
from sqlalchemy import create_engine, Column, MetaData, literal

from clickhouse_sqlalchemy import (
    Table, make_session, get_declarative_base, types, engines
)

uri = 'clickhouse://default:@localhost/test'

engine = create_engine(uri)
session = make_session(engine)
metadata = MetaData(bind=engine)

Base = get_declarative_base(metadata=metadata)

class Rate(Base):
    day = Column(types.Date, primary_key=True)
    value = Column(types.Int32, comment='Rate value')
    other_value = Column(types.DateTime)

    __table_args__ = (
        engines.Memory(),
        {'comment': 'Store rates'}
    )

another_table = Table('another_rate', metadata,
    Column('day', types.Date, primary_key=True),
    Column('value', types.Int32, server_default=literal(1)),
    engines.Memory()
)
```

Tables created in declarative way have lowercase with words separated by underscores naming convention. But you can easy set you own via SQLAlchemy `__tablename__` attribute.

SQLAlchemy `func` proxy for real ClickHouse functions can be also used.

Dialect-specific options

You can specify particular codec for column:

```
class Rate(Base):
    day = Column(types.Date, primary_key=True)
    value = Column(types.Int32)
    other_value = Column(
        types.DateTime,
        clickhouse_codec=('DoubleDelta', 'ZSTD')
    )

    __table_args__ = (
        engines.Memory(),
    )
```

```
CREATE TABLE rate (
    day Date,
    value Int32,
    other_value DateTime CODEC(DoubleDelta, ZSTD)
) ENGINE = Memory
```

`server_default` will render as `DEFAULT`

```
class Rate(Base):
    day = Column(types.Date, primary_key=True)
    value = Column(types.Int32)
    other_value = Column(
        types.DateTime, server_default=func.now()
    )

    __table_args__ = (
        engines.Memory(),
    )
```

```
CREATE TABLE rate (
    day Date,
    value Int32,
    other_value DateTime DEFAULT now()
) ENGINE = Memory
```

`MATERIALIZED` and `ALIAS` also supported

```
class Rate(Base):
    day = Column(types.Date, primary_key=True)
    value = Column(types.Int32)
    other_value = Column(
        types.DateTime, clickhouse_materialized=func.now()
    )

    __table_args__ = (
        engines.Memory(),
    )
```

```
CREATE TABLE rate (
    day Date,
    value Int32,
```

(continues on next page)

(continued from previous page)

```
    other_value DateTime MATERIALIZED now()
) ENGINE = Memory
```

```
class Rate(Base):
    day = Column(types.Date, primary_key=True)
    value = Column(types.Int32)
    other_value = Column(
        types.DateTime, clickhouse_alias=func.now()
    )

    __table_args__ = (
        engines.Memory(),
    )
```

```
CREATE TABLE rate (
    day Date,
    value Int32,
    other_value DateTime ALIAS now()
) ENGINE = Memory
```

You can also specify another column as default, materialized and alias

```
class Rate(Base):
    day = Column(types.Date, primary_key=True)
    value = Column(types.Int32)
    other_value = Column(types.Int32, server_default=value)

    __table_args__ = (
        engines.Memory(),
    )
```

```
CREATE TABLE rate (
    day Date,
    value Int32,
    other_value Int32 DEFAULT value
) ENGINE = Memory
```

Table Engines

Every table in ClickHouse requires engine. Engine can be specified in declarative `__table_args__`:

```
from sqlalchemy import create_engine, MetaData, Column
from clickhouse_sqlalchemy import (
    get_declarative_base, types, engines
)

engine = create_engine('clickhouse://localhost')
metadata = MetaData(bind=engine)
Base = get_declarative_base(metadata=metadata)

class Statistics(Base):
    date = Column(types.Date, primary_key=True)
    sign = Column(types.Int8)
    grouping = Column(types.Int32)
```

(continues on next page)

(continued from previous page)

```
    metric1 = Column(types.Int32)

    __table_args__ = (
        engines.CollapsingMergeTree(
            sign,
            partition_by=func.toYYYYMM(date),
            order_by=(date, grouping)
        ),
    )
```

Or in table:

```
from sqlalchemy import create_engine, MetaData, Column, text
from clickhouse_sqlalchemy import (
    get_declarative_base, types, engines
)

engine = create_engine('clickhouse+native://localhost/default')
metadata = MetaData(bind=engine)

statistics = Table(
    'statistics', metadata,
    Column('date', types.Date, primary_key=True),
    Column('sign', types.Int8),
    Column('grouping', types.Int32),
    Column('metric1', types.Int32),

    engines.CollapsingMergeTree(
        'sign',
        partition_by=text('toYYYYMM(date)'),
        order_by=('date', 'grouping')
    )
)
```

Engine parameters can be column variables or column names.

Note: SQLAlchemy functions can be applied to variables, but not to names.

This will work `partition_by=func.toYYYYMM(date)` and this will not: `partition_by=func.toYYYYMM('date')`. You should use `partition_by=text('toYYYYMM(date)')` in the second case.

Currently supported engines:

- *MergeTree
- Replicated*MergeTree
- Distributed
- Buffer
- View/MaterializedView
- Log/TinyLog
- Memory
- Null

- File

Each engine has it's own parameters. Please refer to ClickHouse documentation about engines.

Engine settings can be passed as additional keyword arguments

```
engines.MergeTree(
    partition_by=date,
    key='value'
)
```

Will render to

```
MergeTree()
PARTITION BY date
SETTINGS key=value
```

More complex examples

```
engines.MergeTree(order_by=func.tuple_())

engines.MergeTree(
    primary_key=('device_id', 'timestamp'),
    order_by=('device_id', 'timestamp'),
    partition_by=func.toYYYYMM(timestamp)
)

engines.MergeTree(
    partition_by=text('toYYYYMM(date)'),
    order_by=('date', func.intHash32(x)),
    sample_by=func.intHash32(x)
)

engines.MergeTree(
    partition_by=date,
    order_by=(date, x),
    primary_key=(x, y),
    sample_by=func.random(),
    key='value'
)

engines.CollapsingMergeTree(
    sign,
    partition_by=date,
    order_by=(date, x)
)

engines.ReplicatedCollapsingMergeTree(
    '/table/path', 'name',
    sign,
    partition_by=date,
    order_by=(date, x)
)

engines.VersionedCollapsingMergeTree(
    sign, version,
    partition_by=date,
    order_by=(date, x),
)
```

(continues on next page)

(continued from previous page)

```
engines.SummingMergeTree(
    columns=(y, ),
    partition_by=date,
    order_by=(date, x)
)

engines.ReplacingMergeTree(
    version='version',
    partition_by='date',
    order_by=('date', 'x')
)
```

Tables can be reflected with engines

```
from sqlalchemy import create_engine, MetaData
from clickhouse_sqlalchemy import Table

engine = create_engine('clickhouse+native://localhost/default')
metadata = MetaData(bind=engine)

statistics = Table('statistics', metadata, autoload=True)
```

Note: Reflection is possible for tables created with modern syntax. Table with following engine can't be reflected.

Note: Engine reflection can take long time if your database have many tables. You can control engine reflection with **engine_reflection** connection parameter.

ON CLUSTER

`ON CLUSTER` clause will be automatically added to DDL queries (`CREATE TABLE`, `DROP TABLE`, etc.) if cluster is specified in `__table_args__`

```
class TestTable(...):
    ...

    __table_args__ = (
        engines.ReplicatedMergeTree(...),
        {'clickhouse_cluster': 'my_cluster'}
    )
```

TTL

`TTL` clause can be rendered during table creation

```
class TestTable(...):
    date = Column(types.Date, primary_key=True)
    x = Column(types.Int32)
```

(continues on next page)

(continued from previous page)

```
    __table_args__ = (
        engines.MergeTree(ttl=date + func.toIntervalDay(1)),
    )
```

```
CREATE TABLE test_table (date Date, x Int32)
ENGINE = MergeTree()
TTL date + toIntervalDay(1)
```

Deletion

```
from clickhouse_sqlalchemy.sql.ddl import ttl_delete

class TestTable(...):
    date = Column(types.Date, primary_key=True)
    x = Column(types.Int32)

    __table_args__ = (
        engines.MergeTree(
            ttl=ttl_delete(date + func.toIntervalDay(1))
        ),
    )
```

```
CREATE TABLE test_table (date Date, x Int32)
ENGINE = MergeTree()
TTL date + toIntervalDay(1) DELETE
```

Multiple clauses at once

```
from clickhouse_sqlalchemy.sql.ddl import (
    ttl_delete,
    ttl_to_disk,
    ttl_to_volume
)

ttl = [
    ttl_delete(date + func.toIntervalDay(1)),
    ttl_to_disk(date + func.toIntervalDay(1), 'hdd'),
    ttl_to_volume(date + func.toIntervalDay(1), 'slow'),
]

class TestTable(...):
    date = Column(types.Date, primary_key=True)
    x = Column(types.Int32)

    __table_args__ = (
        engines.MergeTree(ttl=ttl),
    )
```

```
CREATE TABLE test_table (date Date, x Int32)
ENGINE = MergeTree()
TTL date + toIntervalDay(1) DELETE,
    date + toIntervalDay(1) TO DISK 'hdd',
    date + toIntervalDay(1) TO VOLUME 'slow'
```

Custom engines

If some engine is not supported yet, you can add new one into your code in the following way:

```
from sqlalchemy import create_engine, MetaData, Column
from clickhouse_sqlalchemy import (
    Table, get_declarative_base, types
)
from clickhouse_sqlalchemy.engines.base import Engine

engine = create_engine('clickhouse://localhost/default')
metadata = MetaData(bind=engine)
Base = get_declarative_base(metadata=metadata)

class Kafka(Engine):
    def __init__(self, broker_list, topic_list):
        self.broker_list = broker_list
        self.topic_list = topic_list
        super(Kafka, self).__init__()

    @property
    def name(self):
        return (
            super(Kafka, self).name + '()' +
            '\nSETTINGS kafka_broker_list={},'
            '\nkafka_topic_list={}'.format(
                self.broker_list, self.topic_list
            )
        )

table = Table(
    'test', metadata,
    Column('x', types.Int32),
    Kafka(
        broker_list='host:port',
        topic_list = 'topic1,topic2,...'
    )
)
```

1.4.2 Materialized Views

Materialized Views can be defined in the same way as models. Definition consists from two steps:

- storage definition (table that will store data);
- `SELECT` query definition.

```
from clickhouse_sqlalchemy import MaterializedView, select

class Statistics(Base):
    date = Column(types.Date, primary_key=True)
    sign = Column(types.Int8, nullable=False)
    grouping = Column(types.Int32, nullable=False)
    metric1 = Column(types.Int32, nullable=False)

    __table_args__ = (
        engines.CollapsingMergeTree(
```

(continues on next page)

(continued from previous page)

```
            sign,
            partition_by=func.toYYYYMM(date),
            order_by=(date, grouping)
        ),
    )


# Define storage for Materialized View
class GroupedStatistics(Base):
    date = Column(types.Date, primary_key=True)
    metric1 = Column(types.Int32, nullable=False)

    __table_args__ = (
        engines.SummingMergeTree(
            partition_by=func.toYYYYMM(date),
            order_by=(date, )
        ),
    )


Stat = Statistics

# Define SELECT for Materialized View
MatView = MaterializedView(GroupedStatistics, select([
    Stat.date.label('date'),
    func.sum(Stat.metric1 * Stat.sign).label('metric1')
]).where(
    Stat.grouping > 42
).group_by(
    Stat.date
))

Stat.__table__.create()
MatView.create()
```

Defining materialized views in code is useful for further migrations. Autogeneration can reduce possible human errors in case of columns and materialized views.

Note: Currently it's not possible to detect **database** engine during startup. It's required to specify whether or not materialized view will use `TO [db.]name` syntax.

There are two database engines now: Ordinary and Atomic.

If your database has `Ordinary` engine inner table will be created automatically for materialized view. You can control name generation only by defining class for inner table with appropriate name. `class GroupedStatistics` in example above.

If your database has `Atomic` engine inner tables are not used for materialized view you must add `use_to` for materialized view object: `MaterializedView(..., use_to=True)`. You can optionally specify materialized view name with `name=...`. By default view name is table name with `mv_suffix='_mv'`.

Examples:

- `MaterializedView(TestTable, use_to=True)` is declaration of materialized view `test_table_mv`.
- `MaterializedView(TestTable, use_to=True, name='my_mv')` is declaration of materialized

view `my_mv`.

- `MaterializedView(TestTable, use_to=True, mv_suffix='_mat_view')` is declaration of materialized view `test_table_mat_view`.

You can specify cluster for materialized view in inner table definition.

```
class GroupedStatistics(...):
    ...

    __table_args__ = (
        engines.ReplicatedSummingMergeTree(...),
        {'clickhouse_cluster': 'my_cluster'}
    )
```

1.4.3 Basic DDL support

You can emit simple DDL. Example `CREATE` / `DROP` table:

```
table = Rate.__table__
table.create()
another_table.create()

another_table.drop()
table.drop()
```

1.4.4 Query method chaining

Common `order_by`, `filter`, `limit`, `offset`, etc. are supported alongside with ClickHouse specific `final` and others.

```
session.query(func.count(Rate.day)) \
    .filter(Rate.day > today - timedelta(20)) \
    .scalar()

session.query(Rate.value) \
    .order_by(Rate.day.desc()) \
    .first()

session.query(Rate.value) \
    .order_by(Rate.day) \
    .limit(10) \
    .all()

session.query(func.sum(Rate.value)) \
    .scalar()
```

1.4.5 INSERT

Simple batch INSERT:

```
from datetime import date, timedelta
from sqlalchemy import func
```

(continues on next page)

(continued from previous page)

```
today = date.today()
rates = [
    {'day': today - timedelta(i), 'value': 200 - i}
    for i in range(100)
]

# Emits single INSERT statement.
session.execute(table.insert(), rates)
```

INSERT FROM SELECT statement:

```
from sqlalchemy import cast

# Labels must be present.
select_query = session.query(
    Rate.day.label('day'),
    cast(Rate.value * 1.5, types.Int32).label('value')
).subquery()

# Emits single INSERT FROM SELECT statement
session.execute(
    another_table.insert()
    .from_select(['day', 'value'], select_query)
)
```

1.4.6 UPDATE and DELETE

SQLAlchemy's update statement are mapped into ClickHouse's `ALTER UPDATE`

```
tbl = Table(...)
session.execute(t1.update().where(t1.c.x == 25).values(x=5))
```

or

```
tbl = Table(...)
session.execute(update(t1).where(t1.c.x == 25).values(x=5))
```

becomes

```
ALTER TABLE ... UPDATE x=5 WHERE x = 25
```

Delete statement is also supported and mapped into `ALTER DELETE`

```
tbl = Table(...)
session.execute(t1.delete().where(t1.c.x == 25))
```

or

```
tbl = Table(...)
session.execute(delete(t1).where(t1.c.x == 25))
```

becomes

```
ALTER TABLE ... DELETE WHERE x = 25
```

Many other SQLAlchemy features are supported out of the box. UNION ALL example:

```
from sqlalchemy import union_all

select_rate = session.query(
    Rate.day.label('date'),
    Rate.value.label('x')
)
select_another_rate = session.query(
    another_table.c.day.label('date'),
    another_table.c.value.label('x')
)

union_all(select_rate, select_another_rate) \
    .execute() \
    .fetchone()
```

1.4.7 SELECT extensions

Dialect supports some ClickHouse extensions for `SELECT` query.

SAMPLE

```
session.query(table.c.x).sample(0.1)
```

or

```
select([table.c.x]).sample(0.1)
```

becomes

```
SELECT ... FROM ... SAMPLE 0.1
```

LIMIT BY

```
session.query(table.c.x).order_by(table.c.x) \
    .limit_by([table.c.x], offset=1, limit=2)
```

or

```
select([table.c.x]).order_by(table.c.x) \
    .limit_by([table.c.x], offset=1, limit=2)
```

becomes

```
SELECT ... FROM ... ORDER BY ... LIMIT 1, 2 BY ...
```

Lambda

```
from clickhouse_sqlalchemy.ext.clauses import Lambda

session.query(
    func.arrayFilter(
        Lambda(lambda x: x.like('%World%')),
        literal(
            ['Hello', 'abc World'],
            types.Array(types.String)
        )
    ).label('test')
)
```

becomes

```
SELECT arrayFilter(
    x -> x LIKE '%%World%%',
    ['Hello', 'abc World']
) AS test
```

JOIN

ClickHouse's join is bit more powerful than usual SQL join. In this dialect join is parametrized with following arguments:

- type: `INNER|LEFT|RIGHT|FULL|CROSS`
- strictness: `OUTER|SEMI|ANTI|ANY|ASOF`
- distribution: `GLOBAL`

Here are some examples

```
session.query(t1.c.x, t2.c.x).join(
    t2,
    t1.c.x == t2.c.y,
    type='inner',
    strictness='all',
    distribution='global'
)
```

or

```
select([t1.c.x, t2.c.x]).join(
    t2,
    t1.c.x == t2.c.y,
    type='inner',
    strictness='all',
    distribution='global'
)
```

becomes

```
SELECT ... FROM ... GLOBAL ALL INNER JOIN ... ON ...
```

You can also control join parameters with native SQLAlchemy options as well: `isouter` and `full`.

```
session.query(t1.c.x, t2.c.x).join(
    t2,
    t1.c.x == t2.c.y,
    isouter=True,
    full=True
)
```

becomes

```
SELECT ... FROM ... FULL OUTER JOIN ... ON ...
```

ARRAY JOIN

```
session.query(...).array_join(...)
```

or

```
select([...]).array_join(...)
```

becomes

```
SELECT ... FROM ... ARRAY JOIN ...
```

WITH CUBE/ROLLUP/TOTALS

```
session.query(table.c.x).group_by(table.c.x).with_cube()
session.query(table.c.x).group_by(table.c.x).with_rollup()
session.query(table.c.x).group_by(table.c.x).with_totals()
```

or

```
select([table.c.x]).group_by(table.c.x).with_cube()
select([table.c.x]).group_by(table.c.x).with_rollup()
select([table.c.x]).group_by(table.c.x).with_totals()
```

becomes (respectively)

```
SELECT ... FROM ... GROUP BY ... WITH CUBE
SELECT ... FROM ... GROUP BY ... WITH ROLLUP
SELECT ... FROM ... GROUP BY ... WITH TOTALS
```

FINAL

Note: Currently `FINAL` clause is supported only for table specified in `FROM` clause.

```
session.query(table.c.x).final().group_by(table.c.x)
```

or

```
select([table.c.x]).final().group_by(table.c.x)
```

becomes

```
SELECT ... FROM ... FINAL GROUP BY ...
```

1.4.8 Miscellaneous

Batching

You may want to fetch very large result sets in chunks.

```
session.query(...).yield_per(N)
```

Attention: This supported only in native driver.

In this case clickhouse-driver's `execute_iter` is used and setting `max_block_size` is set into `N`.

There is side effect. If next query will be emitted before end of iteration over query with yield there will be an error. Example

```
def gen(session):
    yield from session.query(...).yield_per(N)

rv = gen(session)

# There will be an error
session.query(...).all()
```

To avoid this side effect you should create another session

```
class another_session():
    def __init__(self, engine):
        self.engine = engine
        self.session = None

    def __enter__(self):
        self.session = make_session(self.engine)
        return self.session

    def __exit__(self, *exc_info):
        self.session.close()

def gen(session):
    with another_session(session.bind) as new_session:
        yield from new_session.query(...).yield_per(N)

rv = gen(session)

# There will be no error
session.query(...).all()
```

Execution options

Attention: This supported only in native and asynch drivers.

You can override default ClickHouse server settings and pass desired settings with `execution_options`. Set lower priority to query and limit max number threads to execute the request

```
settings = {'max_threads': 2, 'priority': 10}

session.query(...).execution_options(settings=settings)
```

You can pass external tables to ClickHouse server with `execution_options`

```
table = Table(
    'ext_table1', metadata,
    Column('id', types.UInt64),
    Column('name', types.String),
    clickhouse_data=[(x, 'name' + str(x)) for x in range(10)],
    extend_existing=True
)

session.query(func.sum(table.c.id)) \
    .execution_options(external_tables=[table])
    .scalar()
```

1.5 Types

The following ClickHouse types are supported by clickhouse-sqlalchemy.

TODO.

1.6 Migrations

Since version 0.1.10 clickhouse-sqlalchemy has alembic support. This support allows autogenerate migrations from source code with some limitations. This is the main advantage comparing to other migration tools when you need to write plain migrations by yourself.

Note: It is necessary to notice that ClickHouse doesn't have transactions. Therefore migration will not rolled back after some command with an error. And schema will remain in partially migrated state.

Autogenerate **will detect**:

- Table and materialized view additions, removals.
- Column additions, removals.
- Column and table comment additions, removals.

Example project with migrations https://github.com/xzkostyan/clickhouse-sqlalchemy-alembic-example.

1.6.1 Requirements

Minimal versions:

- ClickHouse server 21.11.11.1
- clickhouse-sqlalchemy 0.1.10
- alembic 1.5.x

You can always write you migrations with pure alembic's `op.execute` if autogenerate is not possible for your schema objects or your are using `clickhouse-sqlalchemy<0.1.10`.

1.6.2 Limitations

Common limitations:

- Engines are not added into `op.create_table`.
- `Nullable(T)` columns generation via `Column(..., nullable=True)` is not supported.

Currently `ATTACH MATERIALIZED VIEW` with modified `SELECT` statement doesn't work for `Atomic` engine.

1.6.3 Migration adjusting

You can and should adjust migrations after autogeneration.

Following parameters can be specified for:

- `op.detach_mat_view`: `if_exists`, `on_cluster`, `permanently`.
- `op.attach_mat_view`: `if_not_exists`, `on_cluster`.
- `op.create_mat_view`: `if_not_exists`, `on_cluster`, `populate`.

See ClickHouse's Materialized View documentation.

For `op.add_column` you can add:

- `AFTER name_after`: `op.add_column(..., sa.Column(..., clickhouse_after=sa.text('my_column')))`.

CHAPTER 2

Additional Notes

Legal information, changelog and contributing are here for the interested.

2.1 Development

2.1.1 Test configuration

In `setup.cfg` you can find ClickHouse server ports, credentials, logging level and another options than can be tuned during local testing.

2.1.2 Running tests locally

Install desired Python version with system package manager/pyenv/another manager.

Install test requirements and build package:

```
python testsrequire.py && python setup.py develop
```

ClickHouse on host machine

Install desired versions of `clickhouse-server` and `clickhouse-client` on your machine.

Run tests:

```
pytest -v
```

ClickHouse in docker

Create container desired version of `clickhouse-server`:

```
docker run --rm -p 127.0.0.1:9000:9000 -p 127.0.0.1:8123:8123 --name test-
↪clickhouse-server clickhouse/clickhouse-server:$VERSION
```

Create container with the same version of `clickhouse-client`:

```
docker run --rm --entrypoint "/bin/sh" --name test-clickhouse-client --link␣
↪test-clickhouse-server:clickhouse-server clickhouse/clickhouse-client:
↪$VERSION -c 'while :; do sleep 1; done'
```

Create `clickhouse-client` script on your host machine:

```
echo -e '#!/bin/bash\n\ndocker exec test-clickhouse-client clickhouse-client
↪"$@"' | sudo tee /usr/local/bin/clickhouse-client > /dev/null
sudo chmod +x /usr/local/bin/clickhouse-client
```

After it container `test-clickhouse-client` will communicate with `test-clickhouse-server` transparently from host machine.

Set `host=clickhouse-server` in `setup.cfg`.

Add entry in hosts file:

```
echo '127.0.0.1 clickhouse-server' | sudo tee -a /etc/hosts > /dev/null
```

And run tests:

```
pytest -v
```

`pip` will automatically install all required modules for testing.

2.1.3 GitHub Actions in forked repository

Workflows in forked repositories can be used for running tests.

Workflows don't run in forked repositories by default. You must enable GitHub Actions in the **Actions** tab of the forked repository.

2.2 Changelog

Changelog is available in github repo.

2.3 License

clickhouse-sqlalchemy is distributed under the MIT license.

2.4 How to Contribute

1. Check for open issues or open a fresh issue to start a discussion around a feature idea or a bug.
2. Fork the repository on GitHub to start making your changes to the **master** branch (or branch off of it).
3. Write a test which shows that the bug was fixed or that the feature works as expected.

4. Send a pull request and bug the maintainer until it gets merged and published.