
sqlalchemy_aio Documentation

Release 0.1

Frazer McLean

Feb 14, 2022

Contents

1	What is this?	1
2	Threading Model	3
3	Getting started	5
3.1	Asyncio	5
3.2	Trio	6
4	Contents	9
4.1	DDL	9
4.2	API Reference	10
4.3	Limitations	14
4.4	Contributing	14
5	Indices and tables	15
	Index	17

CHAPTER 1

What is this?

It's *not* an `asyncio` implementation of SQLAlchemy or the drivers it uses. `sqlalchemy_aio` lets you use SQLAlchemy by running operations in a separate thread.

If you're already using `run_in_executor()` to execute SQLAlchemy tasks, `sqlalchemy_aio` will work well with similar performance. If performance is critical, perhaps [asyncpg](#) can help.

CHAPTER 2

Threading Model

Explicit connections (*engine.connect()*) each run in their own thread. The engine uses a single worker thread, including implicit connections (e.g. *engine.execute()*).

3.1 Asyncio

```
import asyncio

from sqlalchemy_aio import ASYNCIO_STRATEGY

from sqlalchemy import (
    Column, Integer, MetaData, Table, Text, create_engine, select)
from sqlalchemy.schema import CreateTable, DropTable


async def main():
    engine = create_engine(
        # In-memory sqlite database cannot be accessed from different
        # threads, use file.
        'sqlite:///test.db', strategy=ASYNCIO_STRATEGY
    )

    metadata = MetaData()
    users = Table(
        'users', metadata,
        Column('id', Integer, primary_key=True),
        Column('name', Text),
    )

    # Create the table
    await engine.execute(CreateTable(users))

    conn = await engine.connect()

    # Insert some users
    await conn.execute(users.insert().values(name='Jeremy Goodwin'))
    await conn.execute(users.insert().values(name='Natalie Hurley'))
```

(continues on next page)

(continued from previous page)

```

await conn.execute(users.insert().values(name='Dan Rydell'))
await conn.execute(users.insert().values(name='Casey McCall'))
await conn.execute(users.insert().values(name='Dana Whitaker'))

result = await conn.execute(users.select(users.c.name.startswith('D')))
d_users = await result.fetchall()

await conn.close()

# Print out the users
for user in d_users:
    print('Username: %s' % user[users.c.name])

# Supports context async managers
async with engine.connect() as conn:
    async with conn.begin() as trans:
        assert await conn.scalar(select([1])) == 1

await engine.execute(DropTable(users))

if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())

```

3.2 Trio

```

import trio
from sqlalchemy_ao import TRIO_STRATEGY

from sqlalchemy import (
    Column, Integer, MetaData, Table, Text, create_engine, select)
from sqlalchemy.schema import CreateTable, DropTable

async def main():
    engine = create_engine(
        # In-memory sqlite database cannot be accessed from different
        # threads, use file.
        'sqlite:///test.db', strategy=TRIO_STRATEGY
    )

    metadata = MetaData()
    users = Table(
        'users', metadata,
        Column('id', Integer, primary_key=True),
        Column('name', Text),
    )

    # Create the table
    await engine.execute(CreateTable(users))

    conn = await engine.connect()

```

(continues on next page)

(continued from previous page)

```
# Insert some users
await conn.execute(users.insert().values(name='Jeremy Goodwin'))
await conn.execute(users.insert().values(name='Natalie Hurley'))
await conn.execute(users.insert().values(name='Dan Rydell'))
await conn.execute(users.insert().values(name='Casey McCall'))
await conn.execute(users.insert().values(name='Dana Whitaker'))

result = await conn.execute(users.select(users.c.name.startswith('D')))
d_users = await result.fetchall()

await conn.close()

# Print out the users
for user in d_users:
    print('Username: %s' % user[users.c.name])

# Supports context async managers
async with engine.connect() as conn:
    async with conn.begin() as trans:
        assert await conn.scalar(select([1])) == 1

await engine.execute(DropTable(users))

if __name__ == '__main__':
    trio.run(main)
```


4.1 DDL

Because of some of the limitations in the SQLAlchemy API, it's not possible to asynchronously create tables using `sqlalchemy.schema.Table.create()` or `sqlalchemy.schema.MetaData.create_all()`.

Instead of:

```
users = Table('users', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', String),
)

users.create(engine)
```

you can use `sqlalchemy.schema.CreateTable` or `AsyncEngine.run_in_thread()`:

```
await engine.execute(CreateTable(users))
```

```
await engine.run_in_thread(users.create, engine.sync_engine)
```

For `MetaData.create_all()`, instead of:

```
metadata.create_all(engine)
```

you have to do:

```
await engine.run_in_thread(metadata.create_all, engine.sync_engine)
```

4.2 API Reference

class sqlalchemy_ao.base.**AsyncEngine** (*pool, dialect, url, logging_name=None, echo=None, execution_options=None, **kwargs*)

begin (*close_with_result=False*)

Like `Engine.begin`, but returns an asynchronous context manager.

Example

```
async with engine.begin():
    await engine.execute(...)
```

connect ()

Like `Engine.connect`, but returns an awaitable that can also be used as an asynchronous context manager.

Examples

```
conn = await engine.connect()
await conn.execute(...)
await conn.close()
```

```
async with engine.connect() as conn:
    await conn.execute(...)
```

execute (**args, **kwargs*)

Like `Engine.execute`, but is a coroutine that returns an `AsyncioResultProxy`.

Example

```
result = await engine.execute(...)
data = await result.fetchall()
```

Warning: Make sure to explicitly call `AsyncioResultProxy.close()` if the `ResultProxy` has pending rows remaining otherwise it will be closed during garbage collection. With SQLite, this will raise an exception since the DBAPI connection was created in a different thread.

has_table (*table_name, schema=None*)

Like `Engine.has_table`, but is a coroutine.

run_callable (*callable_, *args, **kwargs*)

Like `Engine.run_callable`.

Warning: This method blocks. It exists so that we can warn the user if they try to use an async engine for table reflection:

```
Table(..., autoload_with=engine)
```

run_in_thread (*func*, **args*)

Run a synchronous function in the engine's worker thread.

Example

The following blocking function:

```
some_fn(engine.sync_engine)
```

can be called like this instead:

```
await engine.run_in_thread(some_fn, engine.sync_engine)
```

Parameters

- **func** – A synchronous function.
- **args** – Positional arguments to be passed to *func*. If you need to pass keyword arguments, then use `functools.partial()`.

scalar (**args*, ***kwargs*)

Like `Connection.scalar`, but is a coroutine.

sync_engine

Public property of the underlying SQLAlchemy engine.

table_names (*schema=None*, *connection: Optional[sqlalchemy_aio.base.AsyncConnection] = None*)

Like `Engine.table_names`, but is a coroutine.

class sqlalchemy_aio.base.**AsyncConnection** (*connection*, *worker*, *engine*)

Mostly like `sqlalchemy.engine.Connection` except some of the methods are coroutines.

begin ()

Like `Connection.begin`, but returns an awaitable that can also be used as an asynchronous context manager.

Examples

```
async with conn.begin() as trans:
    await conn.execute(...)
    await conn.execute(...)
```

```
trans = await conn.begin():
await conn.execute(...)
await conn.execute(...)
await trans.commit()
```

begin_nested ()

Like `Connection.begin_nested`, but returns an awaitable that can also be used as an asynchronous context manager.

See also:

`begin()` for examples.

close (**args*, ***kwargs*)

Like `Connection.close`, but is a coroutine.

closed

Like the `Connection.closed` attribute.

connect()

Like `Connection.connect`, but is a coroutine.

execute(*args, **kwargs)

Like `Connection.execute`, but is a coroutine that returns an `AsyncioResultProxy`.

Example

```
result = await conn.execute(...)
data = await result.fetchall()
```

Warning: Make sure to explicitly call `AsyncioResultProxy.close()` if the `ResultProxy` has pending rows remaining otherwise it will be closed during garbage collection. With SQLite, this will raise an exception since the DBAPI connection was created in a different thread.

in_transaction()

Like `Connection.in_transaction`.

run_callable(callable_, *args, **kwargs)

Like `Connection.run_callable`.

Warning: This method blocks. It exists so that we can warn the user if they try to use an async connection for table reflection:

```
Table(..., autoload_with=connection)
```

run_in_thread(func, *args)

Run a synchronous function in the connection's worker thread.

Example

The following blocking function:

```
some_fn(conn.sync_connection)
```

can be called like this instead:

```
await engine.run_in_thread(some_fn, conn.sync_connection)
```

Parameters

- **func** – A synchronous function.
- **args** – Positional arguments to be passed to *func*. If you need to pass keyword arguments, then use `functools.partial()`.

scalar(*args, **kwargs)

Like `Connection.scalar`, but is a coroutine.

sync_connection

Public property of the underlying SQLAlchemy connection.

class sqlalchemy_ao.base.**AsyncResultProxy** (*result_proxy, run_in_thread*)

Mostly like sqlalchemy.engine.ResultProxy except some of the methods are coroutines.

close()

Like ResultProxy.close, but is a coroutine.

fetchall()

Like ResultProxy.fetchall, but is a coroutine.

fetchmany (*size=None*)

Like ResultProxy.fetchmany, but is a coroutine.

fetchone()

Like ResultProxy.fetchone, but is a coroutine.

first()

Like ResultProxy.first, but is a coroutine.

inserted_primary_key

Like the ResultProxy.inserted_primary_key attribute.

keys()

Like ResultProxy.keys, but is a coroutine.

returns_rows

Like the ResultProxy.returns_rows attribute.

rowcount

Like the ResultProxy.rowcount attribute.

scalar()

Like ResultProxy.scalar, but is a coroutine.

class sqlalchemy_ao.base.**AsyncTransaction** (*transaction, run_in_thread*)

Mostly like sqlalchemy.engine.Transaction except some of the methods are coroutines.

close()

Like Transaction.close, but is a coroutine.

commit()

Like Transaction.commit, but is a coroutine.

rollback()

Like Transaction.rollback, but is a coroutine.

class sqlalchemy_ao.asyncio.**AsyncioEngine** (*pool, dialect, url, logging_name=None, echo=None, execution_options=None, **kwargs*)

Mostly like sqlalchemy.engine.Engine except some of the methods are coroutines.

class sqlalchemy_ao.exc.**AlreadyQuit**

Raised by ThreadWorker if an attempt is made to use it after its thread has quit.

class sqlalchemy_ao.exc.**BlockingWarning**

Emitted when an *AsyncEngine* or *AsyncConnection* is used in a blocking fashion accidentally.

For example, it is emitted in this case:

```
engine = create_engine(..., strategy=TRIO_STRATEGY)
Table(..., autoload_with=engine)
```

class sqlalchemy_ao.exc.SQLAlchemyAioDeprecationWarning
Emitted for deprecated functionality.

4.3 Limitations

There are two reasons stuff isn't implemented in sqlalchemy_ao.

First, because we haven't gotten there yet. For these items you should *file bugs or send pull requests*.

Second, some items can't be implemented because of limitations in SQLAlchemy, there's almost always a workaround though.

- *Table creation*

4.4 Contributing

As an open source project, sqlalchemy_ao welcomes contributions of many forms.

Examples of contributions include:

- Code patches
- Documentation improvements
- Bug reports and patch reviews

We welcome pull requests and tickets on [github](#)!

CHAPTER 5

Indices and tables

- `genindex`
- `modindex`
- `search`

A

AlreadyQuit (*class in sqlalchemy_ao.exc*), 13
 AsyncConnection (*class in sqlalchemy_ao.base*), 11
 AsyncEngine (*class in sqlalchemy_ao.base*), 10
 AsyncioEngine (*class in sqlalchemy_ao.asyncio*), 13
 AsyncResultProxy (*class in sqlalchemy_ao.base*), 13
 AsyncTransaction (*class in sqlalchemy_ao.base*), 13

B

begin() (*sqlalchemy_ao.base.AsyncConnection method*), 11
 begin() (*sqlalchemy_ao.base.AsyncEngine method*), 10
 begin_nested() (*sqlalchemy_ao.base.AsyncConnection method*), 11
 BlockingWarning (*class in sqlalchemy_ao.exc*), 13

C

close() (*sqlalchemy_ao.base.AsyncConnection method*), 11
 close() (*sqlalchemy_ao.base.AsyncResultProxy method*), 13
 close() (*sqlalchemy_ao.base.AsyncTransaction method*), 13
 closed (*sqlalchemy_ao.base.AsyncConnection attribute*), 11
 commit() (*sqlalchemy_ao.base.AsyncTransaction method*), 13
 connect() (*sqlalchemy_ao.base.AsyncConnection method*), 12
 connect() (*sqlalchemy_ao.base.AsyncEngine method*), 10

E

execute() (*sqlalchemy_ao.base.AsyncConnection method*), 12

execute() (*sqlalchemy_ao.base.AsyncEngine method*), 10

F

fetchall() (*sqlalchemy_ao.base.AsyncResultProxy method*), 13
 fetchmany() (*sqlalchemy_ao.base.AsyncResultProxy method*), 13
 fetchone() (*sqlalchemy_ao.base.AsyncResultProxy method*), 13
 first() (*sqlalchemy_ao.base.AsyncResultProxy method*), 13

H

has_table() (*sqlalchemy_ao.base.AsyncEngine method*), 10

I

in_transaction() (*sqlalchemy_ao.base.AsyncConnection method*), 12
 inserted_primary_key (*sqlalchemy_ao.base.AsyncResultProxy attribute*), 13

K

keys() (*sqlalchemy_ao.base.AsyncResultProxy method*), 13

R

returns_rows (*sqlalchemy_ao.base.AsyncResultProxy attribute*), 13
 rollback() (*sqlalchemy_ao.base.AsyncTransaction method*), 13
 rowcount (*sqlalchemy_ao.base.AsyncResultProxy attribute*), 13
 run_callable() (*sqlalchemy_ao.base.AsyncConnection method*), 12
 run_callable() (*sqlalchemy_ao.base.AsyncEngine method*), 10

`run_in_thread()` (*sqlalchemy_ao.base.AsyncConnection*
 method), 12
`run_in_thread()` (*sqlalchemy_ao.base.AsyncEngine*
 method), 10

S

`scalar()` (*sqlalchemy_ao.base.AsyncConnection*
 method), 12
`scalar()` (*sqlalchemy_ao.base.AsyncEngine* *method*),
 11
`scalar()` (*sqlalchemy_ao.base.AsyncResultProxy*
 method), 13
`SQLAlchemyAioDeprecationWarning` (*class in*
 sqlalchemy_ao.exc), 13
`sync_connection` (*sqlalchemy_ao.base.AsyncConnection*
 attribute), 12
`sync_engine` (*sqlalchemy_ao.base.AsyncEngine* *at-*
 tribute), 11

T

`table_names()` (*sqlalchemy_ao.base.AsyncEngine*
 method), 11