asyncqlio Documentation

Release 0.1.0.1

Laura Dickinson

Dec 14, 2017

High-level API

asyncqlio is a Python 3.5+ ORM for SQL Relational Databases, that uses `asyncio` for an async interface.

You can install the latest version of asyncqlio from PyPI with pip:

```
$ pip install asyncqlio
```

You can also install the development version of asyncqlio from Git:

```
$ pip install git+https://github.com/SunDwarf/asyncqlio.git
```

The development version is NOT guarenteed to be stable, or even working. Of course, asyncqlio by itself is useless without a driver to connect to a database. asyncqlio comes with modules that use other libraries as backend drivers to connect to these servers:

- PostgreSQL (asyncpg, aiopg): driver-psql
- MySQL (aiomysql): driver-mysql
- SQLite3 (sqlite3): driver-sqlite3

Other databases may be supported in the future.

CHAPTER 1

Contents

1.1 Connecting to your Database

When writing a application that uses a database, the first thing you need to do is to connect to the database. This is achieved through the usage of the *DatabaseInterface* provided by the library, using a **Data Source Name**.

```
db = DatabaseInterface("postgresql://myuser:mypassword@127.0.0.1:5432/db")
```

You can omit the DSN if you plan on providing it later.

1.1.1 The DSN

Each part of the DSN represents something:

- `postgresql` - The dialect this database is connecting to.
- `+asyncpg` (implicit, not shown) - The driver used to connect.
- `myuser` - The username to connect to the database through.
- `mypassword` - The password for the user. This can be omitted if the user does not have a password.
- `127.0.0.1` - The hostname or IP being connected to.
- `5432` - The port being connected to. If omitted, this will use the default port.
- `db` - The database name to load.

1.1.2 Opening the Connection

Creating a database object does not actually connect to the database; for that you must use *DatabaseInterface.connect()* to open a new connection or connection pool for usage in the database.

```
# if you specified the DSN earlier
await db.connect()
# otherwise
await db.connect(dsn)
```

Once connected, you can do a test query to verify everything works:

```
async with db.get_transaction() as t:
    print(await (await t.cursor("SELECT 1;")).fetch_row())
```

1.2 Tables and Columns

Each database in the real database is represented by a table class in the Python land. These table classes map virtual columns to real columns, allowing access of them in your code easily and intuitively.

Each table class subclasses an instance of the **table base** - a special object that stores some metadata about the current database the application is running. The function *table_base()* can be used to create a new table base object for subclassing.

```
from asyncqlio import table_base
Table = table_base()
```

Internally, this makes a clone of a *Table* object backed by the *TableMeta* which is then used to customize your tables.

1.2.1 Defining Tables

Tables are defined by making a new class inheriting from your table base object, corresponding to a table in the database.

```
class Server(Table, table_name="server"):
    ...
```

Note that `table_name` is passed explicitly here - this is optional. If no table name is passed a name will be automatically generated from the table name made all lowercase.

Table classes themselves are technically instances of *TableMeta*, and as such all the methods of a TableMeta object are available on a table object.

class asyncqlio.orm.schema.table.**TableMeta**(*tblname: str*, *tblbases: tuple*, *class_body: dict*, *register: bool = True*, **args*, ***kwargs*)

Bases: `type`

The metaclass for a table object. This represents the "type" of a table class.

Creates a new Table instance.

Parameters

- **register** – Should this table be registered in the TableMetadata?
- **table_name** – The name for this table.

columns

Returns A list of *Column* this Table has.

iter_relationships() → typing.Generator[[asyncqlio.orm.schema.relationship.Relationship, NoneType], NoneType]

Returns A generator that yields *Relationship* objects for this table.

iter_columns() → typing.Generator[[asyncqlio.orm.schema.column.Column, NoneType], NoneType]

Returns A generator that yields *Column* objects for this table.

get_column(*column_name: str*, *, *raise_si: bool = False*) → typing.Union[asyncqlio.orm.schema.column.Column, NoneType]

Gets a column by name.

Parameters **column_name** – The column name to lookup.

This can be one of the following:

- The column's `name`
- The column's `alias_name()` for this table

Returns The *Column* associated with that name, or None if no column was found.

get_relationship(*relationship_name*) → typing.Union[asyncqlio.orm.schema.relationship.Relationship, NoneType]

Gets a relationship by name.

Parameters **relationship_name** – The name of the relationship to get.

Returns The *Relationship* associated with that name, or None if it doesn't existr.

primary_key

Getter The *PrimaryKey* for this table.

Setter A new *PrimaryKey* for this table.

Note: A primary key will automatically be calculated from columns at define time, if any columns have `primary_key` set to True.

---

mro() → list

return a type's method resolution order

1.2.2 Adding Columns

Columns on tables are represented by *Column* objects - these objects are strictly only on the table classes and not the rows. They provide useful functions for query building and an easy way to map data from a request onto a table.

Columns are added to table objects with a simple attribute setting syntax. To add a column to a table, you only need to do this:

```
class Server(Table, table_name="server"):
    id = Column(Int(), primary_key=True, unique=True)
```

In this example, a column called `id` is added to the table with the type `Int`, and is set to be a primary key and unique. Of course, you can name it anything and add a different type; all that matters is that the object is a *Column*.

class asyncqlio.orm.schema.column.**Column**

Bases: `object`

Represents a column in a table in a database.

```
class MyTable(Table):
    id = Column(Integer, primary_key=True)
```

The `id` column will mirror the ID of records in the table when fetching, etc. and can be set on a record when storing in a table.

```
sess = db.get_session()
user = await sess.select(User).where(User.id == 2).first()

print(user.id)  # 2
```

Parameters

- **type** – The *ColumnType* that represents the type of this column.
- **primary_key** – Is this column the table's Primary Key (the unique identifier that identifies each row)?
- **nullable** – Can this column be NULL?
- **default** – The client-side default for this column. If no value is provided when inserting, this value will automatically be added to the insert query.
- **autoincrement** – Should this column auto-increment? This will create a serial sequence.
- **index** – Should this column be indexed?
- **unique** – Is this column unique?
- **foreign_key** – The *ForeignKey* associated with this column.

__init__()

Parameters

- **type** – The *ColumnType* that represents the type of this column.
- **primary_key** – Is this column the table's Primary Key (the unique identifier that identifies each row)?
- **nullable** – Can this column be NULL?
- **default** – The client-side default for this column. If no value is provided when inserting, this value will automatically be added to the insert query.
- **autoincrement** – Should this column auto-increment? This will create a serial sequence.
- **index** – Should this column be indexed?
- **unique** – Is this column unique?
- **foreign_key** – The *ForeignKey* associated with this column.

name = None

The name of the column. This can be manually set, or automatically set when set on a table.

table = None

The *Table* instance this Column is associated with.

type = None

The *ColumnType* that represents the type of this column.

default = None
The default for this column.

primary_key = None
If this Column is a primary key.

nullable = None
If this Column is nullable.

autoincrement = None
If this Column is to autoincrement.

indexed = None
If this Column is indexed.

unique = None
If this Column is unique.

foreign_key = None
The foreign key associated with this column.

__set_name__(*owner*, *name*)
Called to update the table and the name of this Column.

Parameters

- **owner** – The *Table* this Column is on.
- **name** – The str name of this table.

eq(*other*) → asyncqlio.orm.operators.Eq
Checks if this column is equal to something else.

Note: This is the easy way to check if a column equals another column in a WHERE clause, because the default __eq__ behaviour returns a bool rather than an operator.

---

ne(*other*) → asyncqlio.orm.operators.NEq
Checks if this column is not equal to something else.

Note: This is the easy way to check if a column doesn't equal another column in a WHERE clause, because the default __ne__ behaviour returns a bool rather than an operator.

---

asc() → asyncqlio.orm.operators.AscSorter
Returns the ascending sorter operator for this column.

desc() → asyncqlio.orm.operators.DescSorter
Returns the descending sorter operator for this column.

set(*value: typing.Any*) → asyncqlio.orm.operators.ValueSetter
Sets this column in a bulk update.

incr(*value: typing.Any*) → asyncqlio.orm.operators.IncrementSetter
Increments this column in a bulk update.

__add__(*other*)
Magic method for incr()

decr(*value: typing.Any*) → asyncqlio.orm.operators.DecrementSetter
Decrements this column in a bulk update.

__sub__(*other*)
Magic method for decr()

quoted_fullname_with_table(*table: asyncqlio.orm.schema.table.TableMeta*) → str
Gets the quoted fullname with a table. This is used for columns with alias tables.

Parameters **table** – The *Table* or *AliasedTable* to use.

Returns

quoted_name
Gets the quoted name for this column.

This returns the column name in "column" format.

quoted_fullname
Gets the full quoted name for this column.

This returns the column name in "table"."column" format.

foreign_column

Returns The foreign *Column* this is associated with, or None otherwise.

__delattr__
Implement delattr(self, name).

__dir__() → list
default dir() implementation

__format__()
default object formatter

__getattribute__
Return getattr(self, name).

__new__()
Create and return a new object. See help(type) for accurate signature.

__reduce__()
helper for pickle

__reduce_ex__()
helper for pickle

__setattr__
Implement setattr(self, name, value).

__sizeof__() → int
size of object in memory, in bytes

__str__
Return str(self).

__subclasshook__()
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__weakref__
list of weak references to the object (if defined)

alias_name(*table=None*, *quoted: bool = False*) → str
Gets the alias name for a column, given the table.

This is in the format of *t_<table name>_<column_name>*.

Parameters

- **table** – The *Table* to use to generate the alias name. This is useful for aliased tables.
- **quoted** – Should the name be quoted?

Returns A str representing the alias name.

Primary Keys

Tables can have primary keys, which uniquely identify rows in a table, and are made up of from 1 to N columns in the table. Typically keys with multiple columns are known as **compound primary keys**. For convenience, an object provides primary keys on table classes.

class asyncqlio.orm.schema.table.**PrimaryKey**(**cols: asyncqlio.orm.schema.column.Column*)

Bases: object

Represents the primary key of a table.

A primary key can be on any 1 to N columns in a table.

```
class Something(Table):
    first_id = Column(Integer)
    second_id = Column(Integer)

pkey = PrimaryKey(Something.first_id, Something.second_id)
Something.primary_key = pkey
```

Alternatively, the primary key can be automatically calculated by passing `primary_key=True` to columns in their constructor:

```
class Something(Table):
    id = Column(Integer, primary_key=True)

print(Something.primary_key)
```

Primary keys will be automatically generated on a table when multiple columns are marked as

columns = None
A list of *Column* that this primary key encompasses.

table = None
The table this primary key is bound to.

`primary_key` in the constructor, but a *PrimaryKey* object can be constructed manually and set on `Table.primary_key`.

Column Types

All columns in both SQL and Python have a type - the column type. This defines what data they store, what operators they can use, and so on. In asyncqlio, the first parameter passed to a column is its type; this gives it extra functionality and defines how it stores data passed to it both from the user and the database.

For implementing your own types, see creating-col-types.

exception asyncqlio.orm.schema.types.**ColumnValidationError**
Bases: *asyncqlio.exc.DatabaseException*

Raised when a column fails validation.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class asyncqlio.orm.schema.types.**ColumnType**
Bases: abc.ABC

Implements some underlying mechanisms for a *Column*.

The only method that is required to be implemented on children is *ColumnType.sql()* - which is used in CREATE TABLE declarations, etc. *ColumnType.on_set()*, *ColumnType.on_get()* and so on are not required to be implemented - the defaults will work fine.

The ColumnType is responsible for actually loading the data from the row's internal storage and to the user code.

```
# we hate fun
def on_get(self, row, column):
    return "lol"

...

# row is a random row object
# load the `fun` column which has this weird type
value = row.fun
print(value)  # "lol", regardless of what was stored in the database.
```

Accordingly, it is also responsible for storing the data into the row's internal storage.

```
def on_set(*args, **kwargs):
    return None

row.not_fun = 1
print(row.not_fun)  # None - no value was stored in the row
```

To actually insert a value into the row's storage table, use *ColumnType.store_value()*. Correspondingly, loading a value from the row's storage table can be achieved with ColumnType.load_value(). These functions should be used, as they are guarenteed to work across all versions.

Columns will proxy bad attribute accesses from the Column object to this type object - meaning types can implement custom operators, if applicable.

```
class User(Table):
    id = Column(MyWeirdType())

...

# MyWeirdType implements `.contains`
# the contains call is proxied to (MyWeirdType instance).contains("heck")
q = await sess.select(User).where(User.id.contains("heck")).first()
```

column
The column this type object is associated with.

sql() → str

Returns The str SQL name of this type.

validate_set(*row: asyncqlio.orm.schema.table.Table*, *value: typing.Any*) → bool
Validates that the item being set is valid. This is called by the default `on_set`.

Parameters

- **row** – The row being set.
- **value** – The value to set.

Returns A bool indicating if this is valid or not.

store_value(*row: asyncqlio.orm.schema.table.Table*, *value: typing.Any*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** – The row to store in.
- **value** – The value to store in the row.

on_set(*row: asyncqlio.orm.schema.table.Table*, *value: typing.Any*) → typing.Any
Called when a value is a set on this column.

This is the default method - it will call *ColumnType.validate_set()* to validate the type before storing it. This is useful for simple column types.

Parameters

- **row** – The row this value is being set on.
- **value** – The value being set.

on_get(*row: asyncqlio.orm.schema.table.Table*) → typing.Any
Called when a value is retrieved from this column.

Parameters **row** – The row that is being retrieved.

Returns The value of the row's internal storage.

classmethod create_default() → asyncqlio.orm.schema.types.ColumnType
Creates the default object for this table in the event that a type is passed to a column, instead of an instance.

in_(**args*) → asyncqlio.orm.operators.In
Returns an IN operator, checking if a value in this column is in a tuple of items.

Parameters **args** – The items to check.

class asyncqlio.orm.schema.types.**String**(*size: int = -1*)
Bases: *asyncqlio.orm.schema.types.ColumnType*

Represents a VARCHAR() type.

size = None
The max size of this String.

like(*other: str*) → asyncqlio.orm.operators.Like
Returns a LIKE operator, checking if this column is LIKE another string.

Parameters **other** – The other string to check.

ilike(*other: str*) → typing.Union[asyncqlio.orm.operators.ILike, asyncqlio.orm.operators.HackyILike]
Returns an ILIKE operator, checking if this column is case-insensitive LIKE another string.

> **Warning:** This is not supported in all DB backends.

Parameters **other** – The other string to check.

create_default() → asyncqlio.orm.schema.types.ColumnType
Creates the default object for this table in the event that a type is passed to a column, instead of an instance.

in_(**args*) → asyncqlio.orm.operators.In
Returns an IN operator, checking if a value in this column is in a tuple of items.

Parameters **args** – The items to check.

on_get(*row: asyncqlio.orm.schema.table.Table*) → typing.Any
Called when a value is retrieved from this column.

Parameters **row** – The row that is being retrieved.

Returns The value of the row's internal storage.

on_set(*row: asyncqlio.orm.schema.table.Table*, *value: typing.Any*) → typing.Any
Called when a value is a set on this column.

This is the default method - it will call *ColumnType.validate_set()* to validate the type before storing it. This is useful for simple column types.

Parameters

- **row** – The row this value is being set on.
- **value** – The value being set.

store_value(*row: asyncqlio.orm.schema.table.Table*, *value: typing.Any*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** – The row to store in.
- **value** – The value to store in the row.

class asyncqlio.orm.schema.types.**Text**
Bases: *asyncqlio.orm.schema.types.String*

Represents a TEXT type. TEXT type columns are very similar to String type objects, except that they have no size limit.

Note: This is preferable to the String type in some databases.

> **Warning:** This is deprecated in MSSQL.

create_default() → asyncqlio.orm.schema.types.ColumnType
Creates the default object for this table in the event that a type is passed to a column, instead of an instance.

ilike(*other: str*) → typing.Union[asyncqlio.orm.operators.ILike, asyncqlio.orm.operators.HackyILike]
Returns an ILIKE operator, checking if this column is case-insensitive LIKE another string.

> **Warning:** This is not supported in all DB backends.

Parameters **other** – The other string to check.

in_(**args*) → asyncqlio.orm.operators.In
Returns an IN operator, checking if a value in this column is in a tuple of items.

Parameters **args** – The items to check.

like(*other: str*) → asyncqlio.orm.operators.Like
Returns a LIKE operator, checking if this column is LIKE another string.

Parameters **other** – The other string to check.

on_get(*row: asyncqlio.orm.schema.table.Table*) → typing.Any
Called when a value is retrieved from this column.

Parameters **row** – The row that is being retrieved.

Returns The value of the row's internal storage.

on_set(*row: asyncqlio.orm.schema.table.Table*, *value: typing.Any*) → typing.Any
Called when a value is a set on this column.

This is the default method - it will call *ColumnType.validate_set()* to validate the type before storing it. This is useful for simple column types.

Parameters

- **row** – The row this value is being set on.
- **value** – The value being set.

store_value(*row: asyncqlio.orm.schema.table.Table*, *value: typing.Any*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** – The row to store in.
- **value** – The value to store in the row.

class asyncqlio.orm.schema.types.**Boolean**
Bases: *asyncqlio.orm.schema.types.ColumnType*

Represents a BOOL type.

create_default() → asyncqlio.orm.schema.types.ColumnType
Creates the default object for this table in the event that a type is passed to a column, instead of an instance.

in_(**args*) → asyncqlio.orm.operators.In
Returns an IN operator, checking if a value in this column is in a tuple of items.

Parameters **args** – The items to check.

on_get(*row: asyncqlio.orm.schema.table.Table*) → typing.Any
Called when a value is retrieved from this column.

Parameters **row** – The row that is being retrieved.

Returns The value of the row's internal storage.

on_set(*row: asyncqlio.orm.schema.table.Table*, *value: typing.Any*) → typing.Any
Called when a value is a set on this column.

This is the default method - it will call *ColumnType.validate_set()* to validate the type before storing it. This is useful for simple column types.

Parameters

- **row** – The row this value is being set on.
- **value** – The value being set.

store_value(*row: asyncqlio.orm.schema.table.Table*, *value: typing.Any*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** – The row to store in.
- **value** – The value to store in the row.

class asyncqlio.orm.schema.types.**Integer**
Bases: *asyncqlio.orm.schema.types.ColumnType*

Represents an INTEGER type.

Warning: This represents a 32-bit integer (2**31-1 to -2**32)

validate_set(*row*, *value: typing.Any*)
Checks if this int is in range for the type.

create_default() → asyncqlio.orm.schema.types.ColumnType
Creates the default object for this table in the event that a type is passed to a column, instead of an instance.

in_(**args*) → asyncqlio.orm.operators.In
Returns an IN operator, checking if a value in this column is in a tuple of items.

Parameters **args** – The items to check.

on_get(*row: asyncqlio.orm.schema.table.Table*) → typing.Any
Called when a value is retrieved from this column.

Parameters **row** – The row that is being retrieved.

Returns The value of the row's internal storage.

store_value(*row: asyncqlio.orm.schema.table.Table*, *value: typing.Any*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** – The row to store in.
- **value** – The value to store in the row.

class asyncqlio.orm.schema.types.**SmallInt**
Bases: *asyncqlio.orm.schema.types.Integer*

Represents a SMALLINT type.

create_default() → asyncqlio.orm.schema.types.ColumnType
Creates the default object for this table in the event that a type is passed to a column, instead of an instance.

in_(**args*) → asyncqlio.orm.operators.In
Returns an IN operator, checking if a value in this column is in a tuple of items.

Parameters **args** – The items to check.

on_get(*row: asyncqlio.orm.schema.table.Table*) → typing.Any
Called when a value is retrieved from this column.

Parameters **row** – The row that is being retrieved.

Returns The value of the row's internal storage.

store_value(*row: asyncqlio.orm.schema.table.Table*, *value: typing.Any*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** – The row to store in.
- **value** – The value to store in the row.

class asyncqlio.orm.schema.types.**BigInt**
Bases: *asyncqlio.orm.schema.types.Integer*

Represents a BIGINT type.

create_default() → asyncqlio.orm.schema.types.ColumnType
Creates the default object for this table in the event that a type is passed to a column, instead of an instance.

in_(**args*) → asyncqlio.orm.operators.In
Returns an IN operator, checking if a value in this column is in a tuple of items.

Parameters **args** – The items to check.

on_get(*row: asyncqlio.orm.schema.table.Table*) → typing.Any
Called when a value is retrieved from this column.

Parameters **row** – The row that is being retrieved.

Returns The value of the row's internal storage.

store_value(*row: asyncqlio.orm.schema.table.Table*, *value: typing.Any*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** – The row to store in.
- **value** – The value to store in the row.

class asyncqlio.orm.schema.types.**Timestamp**
Bases: *asyncqlio.orm.schema.types.ColumnType*

Represents a TIMESTAMP type.

create_default() → asyncqlio.orm.schema.types.ColumnType
Creates the default object for this table in the event that a type is passed to a column, instead of an instance.

in_(**args*) → asyncqlio.orm.operators.In
Returns an IN operator, checking if a value in this column is in a tuple of items.

Parameters **args** – The items to check.

on_get(*row: asyncqlio.orm.schema.table.Table*) → typing.Any
Called when a value is retrieved from this column.

Parameters **row** – The row that is being retrieved.

Returns The value of the row's internal storage.

on_set(*row: asyncqlio.orm.schema.table.Table*, *value: typing.Any*) → typing.Any
Called when a value is a set on this column.

This is the default method - it will call *ColumnType.validate_set()* to validate the type before storing it. This is useful for simple column types.

Parameters

- **row** – The row this value is being set on.
- **value** – The value being set.

store_value(*row: asyncqlio.orm.schema.table.Table*, *value: typing.Any*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** – The row to store in.
- **value** – The value to store in the row.

1.2.3 Row Objects

In asyncqlio, a row object is simply an instance of a *Table*. To create one, you can call the table object (much like creating a normal instance of a class):

```
row = User()
```

To provide values for the columns, you can pass keyword arguments to the constructor corresponding with the names of the columns, like so:

```
row = User(id=1, name="heck")
```

1.3 Sessions

Sessions are one of the key parts of interacting with the database. They provide a wrapper around a transaction object, providing an API which uses table row instances and query objects to interacting with the database connected to your application.

1.3.1 Creating a session

Creating a new *Session* that is bound to the current database interface is simple via the usage of *DatabaseInterface.get_session()*.

```
# create a session bound to our current database
# this will automatically provide the ability to get transactions from the db
sess = db.get_session()

# alternatively, you can create your own sessions bound to the database interface
# providng a custom subclass or so on
session = Session(bind=db)
```

Using the session requires beginning it; behind the scenes this will acquire a new transaction object from the current connector, and emit a BEGIN statement to start the transaction.

```
# begin and connect the session
await session.begin()
```

The session object also supports the `async with` protocol, meaning you can have it automatically open and close without calling the begin/close methods.

```
async with session:
    ...

# or alternatively
async with db.get_session() as sess:
    ...
```

1.3.2 Running SQL

The most basic thing you can do with a session is to run some SQL code, using either *Session.execute()* or *Session.cursor()*. The former is used for queries without a result, the latter is used to execute and return a result.

For example, to fetch the result of the sum `1 + 1`, you would use:

```
cursor = await session.cursor("SELECT 1+1;")
```

This returns an instance of the the low-level object *BaseResultSet*. To fetch the result, you can use *BaseResultSet.fetch_row()*:

```
result = await cursor.fetch_row()
answer = result["?column?"]  # postgres example
answer = list(result.values())[0]  # or the list form for cross-db compatability
```

1.3.3 Inserting Rows

The session is the one-stop gateway to inserting, updating, or even deleting Row Objects . There are several methods used: *Session.add()*, *Session.merge()*, and *Session.remove()* are the high level methods.

- *Session.add()* is used for new rows, or rows that have been retrieved from a query
- *Session.merge()* is used for rows that already exist in the database
- *Session.remove()* is used to delete rows that exist in the database.

For example, to add a user to the DB:

```
u = User(id=1, name="heck")
await session.add(u)
```

You can also update a user in the database as long as the row you're providing has a primary key, and you use the `merge` method:

```
u = User(id=1)
u.name = "not heck"
await session.merge(u)
```

1.3.4 Querying with the Session

See querying for an explanation of how to query using the session object.

class asyncqlio.orm.session.**Session**(*bind: asyncqlio.db.DatabaseInterface*)

Bases: object

Sessions act as a temporary window into the database. They are responsible for creating queries, inserting and updating rows, etc.

Sessions are bound to a *DatabaseInterface* instance which they use to get a transaction and execute queries in.

```
# get a session from our db interface
sess = db.get_session()
```

Parameters **bind** – The *DatabaseInterface* instance we are bound to.

transaction = None

The current *BaseTransaction* this Session is associated with. The transaction is used for making queries and inserts, etc.

select

Creates a new SELECT query that can be built upon.

Returns A new *SelectQuery*.

insert

Creates a new INSERT INTO query that can be built upon.

Returns A new *InsertQuery*.

update

Creates a new bulk UPDATE query that can be built upon.

Returns A new *BulkUpdateQuery*.

delete

Creates a new bulk DELETE query that can be built upon.

Returns A new *BulkDeleteQuery*.

coroutine start() → asyncqlio.orm.session.Session

Starts the session, acquiring a transaction connection which will be used to modify the DB.

This **must** be called before using the session.

```
sess = db.get_session()
await sess.start()
```

Note: When using `async with`, this is automatically called.

checkpoint(*checkpoint_name: str*)
Sets a new checkpoint.

Parameters **checkpoint_name** – The name of the checkpoint to use.

uncheckpoint(*checkpoint_name: str*)
Releases a checkpoint.

Parameters **checkpoint_name** – The name of the checkpoint to release.

commit()
Commits the current session, running inserts/updates/deletes.

This will **not** close the session; it can be re-used after a commit.

rollback(*checkpoint: str = None*)
Rolls the current session back. This is useful if an error occurs inside your code.

Parameters **checkpoint** – The checkpoint to roll back to, if applicable.

close()
Closes the current session.

> **Warning:** This will **NOT COMMIT ANY DATA**. Old data will die.

fetch(*sql: str*, *params=None*)
Fetches a single row.

execute(*sql: str, params: typing.Union[typing.Mapping[str, typing.Any], typing.Iterable[typing.Any]] = None*)
Executes SQL inside the current session.

This is part of the **low-level API.**

Parameters

- **sql** – The SQL to execute.
- **params** – The parameters to use inside the query.

cursor(*sql: str, params: typing.Union[typing.Mapping[str, typing.Any], typing.Iterable[typing.Any]] = None*)
Executes SQL inside the current session, and returns a new `BaseResultSet`.

Parameters

- **sql** – The SQL to execute.
- **params** – The parameters to use inside the query.

insert_now(*row: asyncqlio.orm.schema.table.Table*) → typing.Any
Inserts a row NOW.

> **Warning:** This will only generate the INSERT statement for the row now. Only *Session.commit()* will actually commit the row to storage.
>
> Also, tables with auto-incrementing fields will only have their first field filled in outside of Postgres databases.

Parameters **row** – The *Table* instance to insert.

Returns The row, with primary key included.

update_now(*row: asyncqlio.orm.schema.table.Table*) → asyncqlio.orm.schema.table.Table
Updates a row NOW.

> **Warning:** This will only generate the UPDATE statement for the row now. Only *Session.commit()* will actually commit the row to storage.

Parameters **row** – The *Table* instance to update.

Returns The *Table* instance that was updated.

delete_now(*row: asyncqlio.orm.schema.table.Table*) → asyncqlio.orm.schema.table.Table
Deletes a row NOW.

coroutine run_select_query(*query: asyncqlio.orm.query.SelectQuery*)
Executes a select query.

> **Warning:** Unlike the other *run_*_query* methods, this method should not be used without a good reason; it creates a special class that is used for the query.
>
> Use `SelectQuery.first` or `SelectQuery.all`.

Parameters **query** – The *SelectQuery* to use.

Returns A `_ResultGenerator` for this query.

coroutine run_insert_query(*query: asyncqlio.orm.query.InsertQuery*)
Executes an insert query.

Parameters **query** – The *InsertQuery* to use.

Returns The list of rows that were inserted.

coroutine run_update_query(*query: asyncqlio.orm.query.BaseQuery*)
Executes an update query.

Parameters **query** – The *RowUpdateQuery* or *BulkUpdateQuery* to execute.

coroutine run_delete_query(*query: asyncqlio.orm.query.RowDeleteQuery*)
Executes a delete query.

Parameters **query** – The *RowDeleteQuery* or *BulkDeleteQuery* to execute.

coroutine add(*row: asyncqlio.orm.schema.table.Table*) → asyncqlio.orm.schema.table.Table
Adds a row to the current transaction. This will emit SQL that will generate an INSERT or UPDATE statement, and then update the primary key of this row.

> **Warning:** This will only generate the INSERT statement for the row now. Only *Session.commit()* will actually commit the row to storage.

Parameters **row** – The *Table* instance object to add to the transaction.

Returns The *Table* instance with primary key filled in, if applicable.

coroutine merge(*row: asyncqlio.orm.schema.table.Table*) → asyncqlio.orm.schema.table.Table
Merges a row with a row that already exists in the database.

This should be used for rows that have a primary key, but were not returned from `Session.select()`.

Parameters **row** – The *Table* instance to merge.

Returns The *Table* instance once updated.

coroutine remove(*row: asyncqlio.orm.schema.table.Table*) → asyncqlio.orm.schema.table.Table
Removes a row from the database.

Parameters **row** – The *Table* instance to remove.

1.4 Low Level Basics

asyncqlio's low-level API is a database-agnostic SQL API that provides developers the ability to execute SQL code without worrying about the underlying driver.

```
from asyncqlio.db import DatabaseInterface

# create the database object to connect to the server.
db = DatabaseInterface("postgresql+asyncpg://joku@127.0.0.1/joku")

async def main():
    # connect to the database with db.connect
    await db.connect()
    # create a transaction to execute sql inside of
    async with db.get_transaction() as trans:
        # run a query
        results: BaseResultSet = await trans.cursor("SELECT 1;")
        row = await results.fetch_row()  # row with 1
```

1.5 Transactions

Transactions are the way of executing queries without affecting the rest of the database. **All** agnostic connections require the usage of a transaction to execute SQL (it is possible to execute SQL purely on a connection using the driver-specific API, but this is not supported).

The *BaseTransaction* object is used to abstract away Database API transaction objects into a common format that can be used in every dialect. To get a new transaction that is bound to the current connection, use *DatabaseInterface.get_transaction()*:

```
# tr is a new transaction object
tr: BaseTransaction = db.get_transaction()
# this is connected to the current database's connections
# and will execute on said connection
```

Transactions MUST be started before execution can happen; this can be achieved with *BaseTransaction.begin()*.

```
# start the transaction
# this will usually emit a BEGIN or START TRANSACTION command underneath
await tr.begin()
```

SQL can be emitted in the transaction with the usage of *BaseTransaction.execute()* and *BaseTransaction.cursor()*.

```
# update some data
await tr.execute('UPDATE "user" SET level = 3 WHERE "user".xp < 1000')
# select some rows
rows = await tr.cursor('SELECT * FROM "user" WHERE level > 5')
```

BaseTransaction.cursor() returns rows from a select query in the form of a `BaseResultSet()`. ResultSets can be iterated over asynchronously with `async for` to select each dict-like row:

```
async for row in rows:
    print(row.keys(), row.values())
```

Once done with the transaction, you can commit it to flush the changes, or you can rollback to revert any changes.

```
if all_went_good:
    # it all went good, save changes
    await tr.commit()
else:
    # not all went good, rollback changes
    await tr.rollback()
```

Transactions support the `async for` protocol, which will automatically begin and commit/rollback as appropriate.

class asyncqlio.backends.base.**BaseTransaction**(*connector: asyncqlio.backends.base.BaseConnector*)

Bases: `asyncqlio.meta.AsyncABC`

The base class for a transaction. This represents a database transaction (i.e SQL statements guarded with a BEGIN and a COMMIT/ROLLBACK).

Children classes must implement:

- *BaseTransaction.begin()*
- *BaseTransaction.rollback()*
- *BaseTransaction.commit()*
- *BaseTransaction.execute()*
- *BaseTransaction.cursor()*
- *BaseTransaction.close()*

Additionally, some extra methods can be implemented:

- *BaseTransaction.create_savepoint()*
- *BaseTransaction.release_savepoint()*

These methods are not required to be implemented, but will raise `NotImplementedError` if they are not.

This class takes one parameter in the constructor: the `BaseConnector` used to connect to the DB server.

coroutine begin()
Begins the transaction, emitting a BEGIN instruction.

coroutine rollback(*checkpoint: str = None*)
Rolls back the transaction.

Parameters **checkpoint** – If provided, the checkpoint to rollback to. Otherwise, the entire transaction will be rolled back.

coroutine commit()
Commits the current transaction, emitting a COMMIT instruction.

coroutine execute(*sql: str*, *params: typing.Iterable = None*)
Executes SQL in the current transaction.

Parameters

- **sql** – The SQL statement to execute.
- **params** – Any parameters to pass to the query.

coroutine close()
Called at the end of a transaction to cleanup.

coroutine cursor(*sql: str*, *params: typing.Iterable = None*) → asyncqlio.backends.base.BaseResultSet
Executes SQL and returns a database cursor for the rows.

Parameters

- **sql** – The SQL statement to execute.
- **params** – Any parameters to pass to the query.

Returns The *BaseResultSet* returned from the query, if applicable.

create_savepoint(*name: str*)
Creates a savepoint in the current transaction.

> **Warning:** This is not supported in all DB engines. If so, this will raise NotImplementedError.

Parameters **name** – The name of the savepoint to create.

release_savepoint(*name: str*)
Releases a savepoint in the current transaction.

Parameters **name** – The name of the savepoint to release.

class asyncqlio.backends.base.**BaseResultSet**
Bases: collections.abc.AsyncIterator, asyncqlio.meta.AsyncABC

The base class for a result set. This represents the results from a database query, as an async iterable.

Children classes must implement:

- *BaseResultSet.keys*
- BaseResultSet.fetch_row
- BaseResultSet.fetch_many

keys

Returns An iterable of keys that this query contained.

coroutine fetch_row() → typing.Mapping[str, typing.Any]
Fetches the **next row** in this query.

This should return None if the row could not be fetched.

coroutine fetch_many(*n: int*) → typing.List[typing.Mapping[str, typing.Any]]
Fetches the **next N rows** in this query.

Parameters **n** – The number of rows to fetch.

coroutine close()
Closes this result set.

CHAPTER 2

Autogenerated Documentation

These docs are automatically generated from the source code using the autosummary module.

2.1 asyncqlio

This is **automatically generated** API documentation for the *asyncqlio* module. Main package for asyncqlio - a Python 3.5+ async ORM built on top of asyncio.

db	The main Database object.
orm	The core code for the ORM.
backends	SQL driver backends for asyncqlio.
exc	Exceptions for asyncqlio.
meta	Useful metamagic classes, such as async ABCs.

2.1.1 asyncqlio.db

The main Database object. This is the "database interface" to the actual DB server.

Classes

DatabaseInterface(dsn: str = None, *, ...)	The "database interface" to your database.

class asyncqlio.db.**DatabaseInterface**(*dsn: str = None*, *, *loop: asyncio.events.AbstractEventLoop = None*)

Bases: object

The "database interface" to your database. This provides the actual connection to the DB server, including things such as querying, inserting, updating, et cetera.

Creating a new database object is simple:

```
# pass the DSN in the constructor
dsn = "postgresql://postgres:B07_L1v3s_M4tt3r_T00@127.0.0.1/mydb"
my_database = DatabaseInterface(dsn)
# or provide it in the `.connect()` call
await my_database.connect(dsn)
```

Parameters **dsn** – The *Data Source Name <http://whatis.techtarget.com/definition/data-source-name-DSN>_* to connect to the database on.

connector = None
The current connector instance.

dialect = None
The current Dialect instance.

connected
Checks if this DB is connected.

bind_tables(*md: asyncqlio.orm.schema.table.TableMetadata*)
Binds tables to this DB instance.

coroutine connect(*dsn: str = None*, ***kwargs*) → asyncqlio.backends.base.BaseConnector
Connects the interface to the database server.

Note: For SQLite3 connections, this will just open the database for reading.

Parameters **dsn** – The Data Source Name to connect to, if it was not specified in the constructor.

Returns The `BaseConnector` established.

emit_param(*name: str*) → str
Emits a param in the format that the DB driver specifies.

Parameters **name** – The name of the parameter to emit.

Returns A str representing the emitted param.

get_transaction(***kwargs*) → asyncqlio.backends.base.BaseTransaction
Gets a low-level *BaseTransaction*.

```
async with db.get_transaction() as transaction:
    results = await transaction.cursor("SELECT 1;")
```

get_session(***kwargs*) → asyncqlio.orm.session.Session
Gets a new *Session* bound to this instance.

coroutine close()
Closes the current database interface.

coroutine get_db_server_info()
Gets DB server info.

Warning: This is **not** supported on SQLite3 connections.

2.1.2 asyncqlio.orm

The core code for the ORM.

schema	Code for ORM schema objects.
ddl	
query	Classes for query objects.
session	
inspection	Inspection module - contains utilities for inspecting Table objects and Row objects.
operators	Classes for operators returned from queries.

asyncqlio.orm.schema

Code for ORM schema objects.

table	
column	
relationship	Relationship helpers.
types	
decorators	Decorator helpers for tables.

asyncqlio.orm.schema.table

Functions

table_base(name: str = , ...)	Gets a new base object to use for OO-style tables.

Classes

AliasedTable(alias_name: str, ...)	Represents an "aliased table".
PrimaryKey(...)	Represents the primary key of a table.
Table(**kwargs)	The "base" class for all tables.
TableMeta(tblname: str, tblbases: tuple, ...)	The metaclass for a table object.
TableMetadata()	The root class for table metadata.

class asyncqlio.orm.schema.table.**TableMetadata**

Bases: object

The root class for table metadata. This stores a registry of tables, and is responsible for calculating relationships etc.

```
meta = TableMetadata()
Table = table_base(metadata=meta)
```

tables = None

A registry of table name -> table object for this metadata.

bind = None

The DB object bound to this metadata.

register_table(*tbl: asyncqlio.orm.schema.table.TableMeta*, *, *autosetup_tables: bool = False*) → asyncqlio.orm.schema.table.TableMeta
Registers a new table object.

Parameters

- **tbl** – The table to register.
- **autosetup_tables** – Should tables be setup again?

get_table(*table_name: str*) → typing.Type[asyncqlio.orm.schema.table.Table]
Gets a table from the current metadata.

Parameters **table_name** – The name of the table to get.

Returns A *Table* object.

setup_tables()
Sets up the tables for usage in the ORM.

resolve_aliases()
Resolves all alias tables on relationship objects.

resolve_backrefs()
Resolves back-references.

resolve_floating_relationships()
Resolves any "floating" relationships - i.e any relationship/foreign keys that don't directly reference a column object.

class asyncqlio.orm.schema.table.**TableMeta**(*tblname: str*, *tblbases: tuple*, *class_body: dict*, *register: bool = True*, **args*, ***kwargs*)
Bases: type

The metaclass for a table object. This represents the "type" of a table class.

Creates a new Table instance.

Parameters

- **register** – Should this table be registered in the TableMetadata?
- **table_name** – The name for this table.

columns

Returns A list of *Column* this Table has.

iter_relationships() → typing.Generator[[asyncqlio.orm.schema.relationship.Relationship, NoneType], NoneType]

Returns A generator that yields *Relationship* objects for this table.

iter_columns() → typing.Generator[[asyncqlio.orm.schema.column.Column, NoneType], NoneType]

Returns A generator that yields *Column* objects for this table.

get_column(*column_name: str*, *, *raise_si: bool = False*) → typing.Union[asyncqlio.orm.schema.column.Column, NoneType]
Gets a column by name.

Parameters **column_name** – The column name to lookup.

This can be one of the following:

- The column's `name`
- The column's `alias_name()` for this table

Returns The *Column* associated with that name, or None if no column was found.

get_relationship(*relationship_name*) → typing.Union[asyncqlio.orm.schema.relationship.Relationship, NoneType]
Gets a relationship by name.

Parameters **relationship_name** – The name of the relationship to get.

Returns The *Relationship* associated with that name, or None if it doesn't existr.

primary_key

Getter The *PrimaryKey* for this table.

Setter A new *PrimaryKey* for this table.

Note: A primary key will automatically be calculated from columns at define time, if any columns have `primary_key` set to True.

---

mro() → list
return a type's method resolution order

class asyncqlio.orm.schema.table.**Table**(***kwargs*)
Bases: object

The "base" class for all tables. This class is not actually directly used; instead `table_base()` should be called to get a fresh clone.

table = None
The actual table that this object is an instance of.

get_old_value(*column: asyncqlio.orm.schema.column.Column*)
Gets the old value from the specified column in this row.

get_column_value(*column: asyncqlio.orm.schema.column.Column*, *return_default: bool = True*)
Gets the value from the specified column in this row.

Parameters

- **column** – The column.
- **return_default** – If this should return the column default, or NO_VALUE.

store_column_value(*column: asyncqlio.orm.schema.column.Column*, *value: typing.Any*, *track_history: bool = True*)
Updates the value of a column in this row.

This will also update the history of the value, if applicable.

get_relationship_instance(*relation_name: str*)
Gets a 'relationship instance'.

Parameters **relation_name** – The name of the relationship to load.

to_dict(*, *include_attrs: bool = False*) → dict
Converts this row to a dict, indexed by Column.

Parameters **include_attrs** – Should this include row_attrs?

asyncqlio.orm.schema.table.**table_base**(*name: str = 'Table'*, *meta: typing.Union[asyncqlio.orm.schema.table.TableMetadata, NoneType] = None*)

Gets a new base object to use for OO-style tables. This object is the parent of all tables created in the object-oriented style; it provides some key configuration to the relationship calculator and the DB object itself.

To use this object, you call this function to create the new object, and subclass it in your table classes:

```
Table = table_base()

class User(Table):
    ...
```

Binding the base object to the database object is essential for querying:

```
# ensure the table is bound to that database
db.bind_tables(Table)

# now we can do queries
sess = db.get_session()
user = await sess.select(User).where(User.id == 2).first()
```

Each Table object is associated with a database interface, which it uses for special querying inside the object, such as `Table.get()`.

```
class User(Table):
    id = Column(Integer, primary_key=True)
    ...

db.bind_tables(Table)
# later on, in some worker code
user = await User.get(1)
```

Parameters

- **name** – The name of the new class to produce. By default, it is `Table`.
- **meta** – The *TableMetadata* to use as metadata.

Returns A new Table class that can be used for OO tables.

class asyncqlio.orm.schema.table.**AliasedTable**(*alias_name: str*, *table: typing.Type[asyncqlio.orm.schema.table.Table]*)

Bases: object

Represents an "aliased table". This is a transparent proxy to a *TableMeta* table, and will create the right Table objects when called.

```
class User(Table):
    id = Column(Integer, primary_key=True, autoincrement=True)
    username = Column(String, nullable=False, unique=True)
    password = Column(String, nullable=False)

NotUser = AliasedTable("not_user", User)
```

Parameters

- **alias_name** – The name of the alias for this table.
- **table** – The *TableMeta* used to alias this table.

get_column(*column_name: str*) → asyncqlio.orm.schema.column.Column
Gets a column by name from the specified table.

This will use the base *TableMeta.get_column()*, and then search for columns via their alias name using this table.

class asyncqlio.orm.schema.table.**PrimaryKey**(**cols: asyncqlio.orm.schema.column.Column*)

Bases: object

Represents the primary key of a table.

A primary key can be on any 1 to N columns in a table.

```
class Something(Table):
    first_id = Column(Integer)
    second_id = Column(Integer)

pkey = PrimaryKey(Something.first_id, Something.second_id)
Something.primary_key = pkey
```

Alternatively, the primary key can be automatically calculated by passing `primary_key=True` to columns in their constructor:

```
class Something(Table):
    id = Column(Integer, primary_key=True)

print(Something.primary_key)
```

columns = None
A list of *Column* that this primary key encompasses.

table = None
The table this primary key is bound to.

asyncqlio.orm.schema.column

Classes

AliasedColumn(...)	Represents a column on an aliased table.
Column	Represents a column in a table in a database.

class asyncqlio.orm.schema.column.**AliasedColumn**(*alias_table: asyncqlio.orm.schema.table.AliasedTable*, *column: asyncqlio.orm.schema.column.Column*)

Bases: object

Represents a column on an aliased table.

Parameters

- **alias_table** – The alias table this column is a member of.
- **column** – The Column object this aliased column proxies.

class asyncqlio.orm.schema.column.**Column**
Bases: object

Represents a column in a table in a database.

```
class MyTable(Table):
    id = Column(Integer, primary_key=True)
```

The `id` column will mirror the ID of records in the table when fetching, etc. and can be set on a record when storing in a table.

```
sess = db.get_session()
user = await sess.select(User).where(User.id == 2).first()

print(user.id)  # 2
```

Parameters

- **type** – The *ColumnType* that represents the type of this column.
- **primary_key** – Is this column the table's Primary Key (the unique identifier that identifies each row)?
- **nullable** – Can this column be NULL?
- **default** – The client-side default for this column. If no value is provided when inserting, this value will automatically be added to the insert query.
- **autoincrement** – Should this column auto-increment? This will create a serial sequence.
- **index** – Should this column be indexed?
- **unique** – Is this column unique?
- **foreign_key** – The *ForeignKey* associated with this column.

name = None
The name of the column. This can be manually set, or automatically set when set on a table.

table = None
The *Table* instance this Column is associated with.

type = None
The *ColumnType* that represents the type of this column.

default = None
The default for this column.

primary_key = None
If this Column is a primary key.

nullable = None
If this Column is nullable.

autoincrement = None
If this Column is to autoincrement.

indexed = None
If this Column is indexed.

unique = None
If this Column is unique.

foreign_key = None
The foreign key associated with this column.

eq(*other*) → asyncqlio.orm.operators.Eq
Checks if this column is equal to something else.

Note: This is the easy way to check if a column equals another column in a WHERE clause, because the default __eq__ behaviour returns a bool rather than an operator.

---

ne(*other*) → asyncqlio.orm.operators.NEq
Checks if this column is not equal to something else.

Note: This is the easy way to check if a column doesn't equal another column in a WHERE clause, because the default __ne__ behaviour returns a bool rather than an operator.

---

asc() → asyncqlio.orm.operators.AscSorter
Returns the ascending sorter operator for this column.

desc() → asyncqlio.orm.operators.DescSorter
Returns the descending sorter operator for this column.

set(*value: typing.Any*) → asyncqlio.orm.operators.ValueSetter
Sets this column in a bulk update.

incr(*value: typing.Any*) → asyncqlio.orm.operators.IncrementSetter
Increments this column in a bulk update.

decr(*value: typing.Any*) → asyncqlio.orm.operators.DecrementSetter
Decrements this column in a bulk update.

quoted_fullname_with_table(*table: asyncqlio.orm.schema.table.TableMeta*) → str
Gets the quoted fullname with a table. This is used for columns with alias tables.

Parameters **table** – The *Table* or *AliasedTable* to use.

Returns

quoted_name
Gets the quoted name for this column.

This returns the column name in "column" format.

quoted_fullname
Gets the full quoted name for this column.

This returns the column name in "table"."column" format.

foreign_column

Returns The foreign *Column* this is associated with, or None otherwise.

alias_name(*table=None*, *quoted: bool = False*) → str
Gets the alias name for a column, given the table.

This is in the format of *t_<table name>_<column_name>*.

Parameters

- **table** – The *Table* to use to generate the alias name. This is useful for aliased tables.
- **quoted** – Should the name be quoted?

Returns A str representing the alias name.

asyncqlio.orm.schema.relationship

Relationship helpers.

Classes

BaseLoadedRelationship(...)	Provides some common methods for specific relationship type subclasses.
ForeignKey(...)	Represents a foreign key object in a column.
JoinLoadedOTMRelationship(...)	Represents a join-loaded one to many relationship.
JoinLoadedOTORelationship(...)	Represents a joined one<-to->one relationship.
Relationship(...)	Represents a relationship to another table object.
SelectLoadedRelationship(...)	A relationship object that uses a separate SELECT statement to load follow-on tables.

class asyncqlio.orm.schema.relationship.**ForeignKey**(*foreign_column: typing.Union[asyncqlio.orm.schema.column.Column, str]*)

Bases: object

Represents a foreign key object in a column. This allows linking multiple tables together relationally.

```
class Server(Table):
    id = Column(Integer, primary_key=True, autoincrement=True)
    name = Column(String)

    owner_id = Column(Integer, foreign_key=ForeignKey("User.id")
```

Parameters **foreign_column** – Either a *Column* representing the foreign column, or a str in the format `<table object name>.<column name>`.

foreign_column = None
The *Column* object this FK references.

column = None
The *Column* object this FK is associated with.

class asyncqlio.orm.schema.relationship.**Relationship**(*left: typing.Union[asyncqlio.orm.schema.column.Column, str], right: typing.Union[asyncqlio.orm.schema.column.Column, str], *, load: str = 'select', use_iter: bool = True, back_ref: str = None, table_alias: str = None*)

Bases: object

Represents a relationship to another table object.

This object provides an easy, object-oriented interface to a foreign key relationship between two tables, the left table and the right table. The left table is the "parent" table, and the right table is the "child" table; effectively creating a one to many/many to one relationship between the two tables.

To create a relationship, there must be a column in the child table that represents the primary key of a parent table; this is the foreign key column, and will be used to load the other table.

```
class User(Table):
    # id is the primary key of the parent table
    id = Column(Integer, auto_increment=True)
    name = Column(String)

    # this is the relationship joiner; it uses id as the left key, and user_id as␣
↪the right
    # this will create a join between the two tables
    inventory = Relationship(left=id, right="InventoryItem.user_id")

class InventoryItem(Table):
    id = Column(BigInteger, auto_increment=True)

    # user_id is the "foreign key" - it references the column User.id
    user_id = Column(Integer, foreign_key=ForeignKey(User.id)
```

Once created, the new relationship object can be used to iterate over the child objects, using `async for`:

```
user = await sess.select.from_(User).where(User.id == 1).first()
async for item in user.inventory:
    ...
```

By default, the relationship will use a SELECT query to load the items; this can be changed to a joined query when loading any table rows, by changing the `load` param. The possible values of this param are:

- `select` - Emits a SELECT query to load child items.
- `joined` - Emits a join query to load child items.

For all possible options, see Relationship Loading.

Parameters

- **left** – The left-hand column (the Column on this table) in this relationship.
- **right** – The right-hand column (the Column on the foreign table) in this relationship.
- **load** – The way to load this relationship.

  The default is "select" - this means that a separate select statement will be issued to iterate over the rows of the relationship.

  For all possible options, see Relationship Loading.
- **use_iter** – Should this relationship use the iterable format?

  This controls if this relationship is created as one to many, or as a many to one/one to one relationship.
- **back_ref** – The "back reference" to add to the right table.

  This will automatically add a relationship to the right table with the specified name, and automatically fill it when querying over said relationship.
- **table_alias** – The table alias to use when joining.

  This will rename the joined table to allow selecting specific rows in tables with multiple relationships to the same table.

left_column = None
The left column for this relationship.

right_column = None
The right column for this relationship.

use_iter = None
If this relationship uses the iterable format.

owner_table = None
The owner table for this relationship.

back_reference = None
The back-reference for this relationship.

load_type = None
The load type for this relationship.

our_column
Gets the local column this relationship refers to.

foreign_column
Gets the foreign column this relationship refers to.

join_columns
Gets the "join" columns of this relationship, i.e the columns that link the two columns.

get_instance(*row: asyncqlio.orm.schema.table.Table*, *session*)
Gets a new "relationship" instance.

class asyncqlio.orm.schema.relationship.**BaseLoadedRelationship**(*rel: asyncqlio.orm.schema.relationship.Relationshi row: asyncqlio.orm.schema.table.Table*, *session*)

Bases: object

Provides some common methods for specific relationship type subclasses.

Parameters

- **rel** – The *Relationship* that lies underneath this object.
- **row** – The TableRow this is being loaded from.
- **session** – The *Session* this object is attached to.

coroutine add(*row: asyncqlio.orm.schema.table.Table*)
Adds a row to this relationship.

> **Warning:** This will run an immediate insert/update of this row; if the parent row for this relationship is not inserted it will run an immediate insert on the parent.

Parameters **row** – The TableRow object to add to this relationship.

coroutine remove(*row: asyncqlio.orm.schema.table.Table*)
Removes a row from this query.

> **Warning:** This will run an immediate UPDATE of this row to remove the foreign key.

Parameters **row** – The TableRow object to remove from this relationship.

set_rows(*rows: typing.List[asyncqlio.orm.schema.table.Table]*)
Sets the rows for this relationship. This is an internal method, and not to be used in user code.

class asyncqlio.orm.schema.relationship.**SelectLoadedRelationship**(*rel: asyncqlio.orm.schema.relationship.Relation*, *row: asyncqlio.orm.schema.table.Table*, *session*)

Bases: *asyncqlio.orm.schema.relationship.BaseLoadedRelationship*

A relationship object that uses a separate SELECT statement to load follow-on tables.

Parameters

- **rel** – The *Relationship* that lies underneath this object.
- **row** – The TableRow this is being loaded from.
- **session** – The *Session* this object is attached to.

query

Gets the query for this relationship, allowing further customization. For example, to change the order of the rows returned:

```
async for child in parent.children.query.order_by(Child.age):
    ...
```

coroutine add(*row: asyncqlio.orm.schema.table.Table*)

Adds a row to this relationship.

Warning: This will run an immediate insert/update of this row; if the parent row for this relationship is not inserted it will run an immediate insert on the parent.

Parameters **row** – The TableRow object to add to this relationship.

coroutine remove(*row: asyncqlio.orm.schema.table.Table*)

Removes a row from this query.

Warning: This will run an immediate UPDATE of this row to remove the foreign key.

Parameters **row** – The TableRow object to remove from this relationship.

set_rows(*rows: typing.List[asyncqlio.orm.schema.table.Table]*)

Sets the rows for this relationship. This is an internal method, and not to be used in user code.

class asyncqlio.orm.schema.relationship.**JoinLoadedOTMRelationship**(*rel: asyncqlio.orm.schema.relationship.Relatio*, *row: asyncqlio.orm.schema.table.Table*, *session*)

Bases: *asyncqlio.orm.schema.relationship.BaseLoadedRelationship*

Represents a join-loaded one to many relationship.

Parameters

- **rel** – The *Relationship* that lies underneath this object.
- **row** – The TableRow this is being loaded from.
- **session** – The *Session* this object is attached to.

coroutine add(*row: asyncqlio.orm.schema.table.Table*)
Adds a row to this relationship.

> **Warning:** This will run an immediate insert/update of this row; if the parent row for this relationship is not inserted it will run an immediate insert on the parent.

Parameters **row** – The `TableRow` object to add to this relationship.

coroutine remove(*row: asyncqlio.orm.schema.table.Table*)
Removes a row from this query.

> **Warning:** This will run an immediate UPDATE of this row to remove the foreign key.

Parameters **row** – The `TableRow` object to remove from this relationship.

class asyncqlio.orm.schema.relationship.**JoinLoadedOTORelationship**(*rel: asyncqlio.orm.schema.relationship.Relatio row: asyncqlio.orm.schema.table.Table, session*)
Bases: *asyncqlio.orm.schema.relationship.BaseLoadedRelationship*

Represents a joined one<-to->one relationship.

Parameters

- **rel** – The *Relationship* that lies underneath this object.
- **row** – The `TableRow` this is being loaded from.
- **session** – The *Session* this object is attached to.

coroutine set(*row: asyncqlio.orm.schema.table.Table*)
Sets the row for this one-to-one relationship.

> **Warning:** This will run an immediate insert/update of this row; if the parent row for this relationship is not inserted it will run an immediate insert on the parent.

Parameters **row** – The `TableRow` to set.

asyncqlio.orm.schema.types

Classes

BigInt()	Represents a BIGINT type.
Boolean()	Represents a BOOL type.
ColumnType()	Implements some underlying mechanisms for a *Column*.
Integer()	Represents an INTEGER type.
SmallInt()	Represents a SMALLINT type.
String(size: int = -1)	Represents a VARCHAR() type.

Continued on next page

Table 2.9 – continued from previous page

Text()	Represents a TEXT type.
Timestamp()	Represents a TIMESTAMP type.

Exceptions

ColumnValidationError	Raised when a column fails validation.

exception asyncqlio.orm.schema.types.**ColumnValidationError**

Bases: *asyncqlio.exc.DatabaseException*

Raised when a column fails validation.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class asyncqlio.orm.schema.types.**ColumnType**

Bases: abc.ABC

Implements some underlying mechanisms for a *Column*.

The only method that is required to be implemented on children is *ColumnType.sql()* - which is used in CREATE TABLE declarations, etc. *ColumnType.on_set()*, *ColumnType.on_get()* and so on are not required to be implemented - the defaults will work fine.

The ColumnType is responsible for actually loading the data from the row's internal storage and to the user code.

```
# we hate fun
def on_get(self, row, column):
    return "lol"

...

# row is a random row object
# load the `fun` column which has this weird type
value = row.fun
print(value)  # "lol", regardless of what was stored in the database.
```

Accordingly, it is also responsible for storing the data into the row's internal storage.

```
def on_set(*args, **kwargs):
    return None

row.not_fun = 1
print(row.not_fun)  # None - no value was stored in the row
```

To actually insert a value into the row's storage table, use *ColumnType.store_value()*. Correspondingly, loading a value from the row's storage table can be achieved with ColumnType.load_value(). These functions should be used, as they are guarenteed to work across all versions.

Columns will proxy bad attribute accesses from the Column object to this type object - meaning types can implement custom operators, if applicable.

```
class User(Table):
    id = Column(MyWeirdType())

...
```

```
# MyWeirdType implements `.contains`
# the contains call is proxied to (MyWeirdType instance).contains("heck")
q = await sess.select(User).where(User.id.contains("heck")).first()
```

column
The column this type object is associated with.

sql() → str

Returns The str SQL name of this type.

validate_set(*row: asyncqlio.orm.schema.table.Table*, *value: typing.Any*) → bool
Validates that the item being set is valid. This is called by the default `on_set`.

Parameters

- **row** – The row being set.
- **value** – The value to set.

Returns A bool indicating if this is valid or not.

store_value(*row: asyncqlio.orm.schema.table.Table*, *value: typing.Any*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** – The row to store in.
- **value** – The value to store in the row.

on_set(*row: asyncqlio.orm.schema.table.Table*, *value: typing.Any*) → typing.Any
Called when a value is a set on this column.

This is the default method - it will call *ColumnType.validate_set()* to validate the type before storing it. This is useful for simple column types.

Parameters

- **row** – The row this value is being set on.
- **value** – The value being set.

on_get(*row: asyncqlio.orm.schema.table.Table*) → typing.Any
Called when a value is retrieved from this column.

Parameters **row** – The row that is being retrieved.

Returns The value of the row's internal storage.

classmethod create_default() → asyncqlio.orm.schema.types.ColumnType
Creates the default object for this table in the event that a type is passed to a column, instead of an instance.

in_(**args*) → asyncqlio.orm.operators.In
Returns an IN operator, checking if a value in this column is in a tuple of items.

Parameters **args** – The items to check.

class asyncqlio.orm.schema.types.**String**(*size: int = -1*)
Bases: *asyncqlio.orm.schema.types.ColumnType*

Represents a VARCHAR() type.

size = None
The max size of this String.

like(*other: str*) → asyncqlio.orm.operators.Like
Returns a LIKE operator, checking if this column is LIKE another string.

Parameters **other** – The other string to check.

ilike(*other: str*) → typing.Union[asyncqlio.orm.operators.ILike, asyncqlio.orm.operators.HackyILike]
Returns an ILIKE operator, checking if this column is case-insensitive LIKE another string.

> **Warning:** This is not supported in all DB backends.

Parameters **other** – The other string to check.

create_default() → asyncqlio.orm.schema.types.ColumnType
Creates the default object for this table in the event that a type is passed to a column, instead of an instance.

in_(**args*) → asyncqlio.orm.operators.In
Returns an IN operator, checking if a value in this column is in a tuple of items.

Parameters **args** – The items to check.

on_get(*row: asyncqlio.orm.schema.table.Table*) → typing.Any
Called when a value is retrieved from this column.

Parameters **row** – The row that is being retrieved.

Returns The value of the row's internal storage.

on_set(*row: asyncqlio.orm.schema.table.Table*, *value: typing.Any*) → typing.Any
Called when a value is a set on this column.

This is the default method - it will call *ColumnType.validate_set()* to validate the type before storing it. This is useful for simple column types.

Parameters

- **row** – The row this value is being set on.
- **value** – The value being set.

store_value(*row: asyncqlio.orm.schema.table.Table*, *value: typing.Any*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** – The row to store in.
- **value** – The value to store in the row.

class asyncqlio.orm.schema.types.**Text**
Bases: *asyncqlio.orm.schema.types.String*

Represents a TEXT type. TEXT type columns are very similar to String type objects, except that they have no size limit.

Note: This is preferable to the String type in some databases.

> **Warning:** This is deprecated in MSSQL.

create_default() → asyncqlio.orm.schema.types.ColumnType
Creates the default object for this table in the event that a type is passed to a column, instead of an instance.

ilike(*other: str*) → typing.Union[asyncqlio.orm.operators.ILike, asyncqlio.orm.operators.HackyILike]
Returns an ILIKE operator, checking if this column is case-insensitive LIKE another string.

> **Warning:** This is not supported in all DB backends.

Parameters **other** – The other string to check.

in_(**args*) → asyncqlio.orm.operators.In
Returns an IN operator, checking if a value in this column is in a tuple of items.

Parameters **args** – The items to check.

like(*other: str*) → asyncqlio.orm.operators.Like
Returns a LIKE operator, checking if this column is LIKE another string.

Parameters **other** – The other string to check.

on_get(*row: asyncqlio.orm.schema.table.Table*) → typing.Any
Called when a value is retrieved from this column.

Parameters **row** – The row that is being retrieved.

Returns The value of the row's internal storage.

on_set(*row: asyncqlio.orm.schema.table.Table*, *value: typing.Any*) → typing.Any
Called when a value is a set on this column.

This is the default method - it will call *ColumnType.validate_set()* to validate the type before storing it. This is useful for simple column types.

Parameters

- **row** – The row this value is being set on.
- **value** – The value being set.

store_value(*row: asyncqlio.orm.schema.table.Table*, *value: typing.Any*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** – The row to store in.
- **value** – The value to store in the row.

class asyncqlio.orm.schema.types.**Boolean**
Bases: *asyncqlio.orm.schema.types.ColumnType*

Represents a BOOL type.

create_default() → asyncqlio.orm.schema.types.ColumnType
Creates the default object for this table in the event that a type is passed to a column, instead of an instance.

in_(**args*) → asyncqlio.orm.operators.In
Returns an IN operator, checking if a value in this column is in a tuple of items.

Parameters **args** – The items to check.

on_get(*row: asyncqlio.orm.schema.table.Table*) → typing.Any
Called when a value is retrieved from this column.

Parameters **row** – The row that is being retrieved.

Returns The value of the row's internal storage.

on_set(*row: asyncqlio.orm.schema.table.Table*, *value: typing.Any*) → typing.Any
Called when a value is a set on this column.

This is the default method - it will call *ColumnType.validate_set()* to validate the type before storing it. This is useful for simple column types.

Parameters

- **row** – The row this value is being set on.
- **value** – The value being set.

store_value(*row: asyncqlio.orm.schema.table.Table*, *value: typing.Any*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** – The row to store in.
- **value** – The value to store in the row.

class asyncqlio.orm.schema.types.**Integer**
Bases: *asyncqlio.orm.schema.types.ColumnType*

Represents an INTEGER type.

> **Warning:** This represents a 32-bit integer (2**31-1 to -2**32)

validate_set(*row*, *value: typing.Any*)
Checks if this int is in range for the type.

create_default() → asyncqlio.orm.schema.types.ColumnType
Creates the default object for this table in the event that a type is passed to a column, instead of an instance.

in_(**args*) → asyncqlio.orm.operators.In
Returns an IN operator, checking if a value in this column is in a tuple of items.

Parameters **args** – The items to check.

on_get(*row: asyncqlio.orm.schema.table.Table*) → typing.Any
Called when a value is retrieved from this column.

Parameters **row** – The row that is being retrieved.

Returns The value of the row's internal storage.

store_value(*row: asyncqlio.orm.schema.table.Table*, *value: typing.Any*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** – The row to store in.
- **value** – The value to store in the row.

class asyncqlio.orm.schema.types.**SmallInt**
Bases: *asyncqlio.orm.schema.types.Integer*

Represents a SMALLINT type.

create_default() → asyncqlio.orm.schema.types.ColumnType
Creates the default object for this table in the event that a type is passed to a column, instead of an instance.

in_(**args*) → asyncqlio.orm.operators.In
Returns an IN operator, checking if a value in this column is in a tuple of items.

Parameters **args** – The items to check.

on_get(*row: asyncqlio.orm.schema.table.Table*) → typing.Any
Called when a value is retrieved from this column.

Parameters **row** – The row that is being retrieved.

Returns The value of the row's internal storage.

store_value(*row: asyncqlio.orm.schema.table.Table*, *value: typing.Any*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** – The row to store in.
- **value** – The value to store in the row.

class asyncqlio.orm.schema.types.**BigInt**
Bases: *asyncqlio.orm.schema.types.Integer*

Represents a BIGINT type.

create_default() → asyncqlio.orm.schema.types.ColumnType
Creates the default object for this table in the event that a type is passed to a column, instead of an instance.

in_(**args*) → asyncqlio.orm.operators.In
Returns an IN operator, checking if a value in this column is in a tuple of items.

Parameters **args** – The items to check.

on_get(*row: asyncqlio.orm.schema.table.Table*) → typing.Any
Called when a value is retrieved from this column.

Parameters **row** – The row that is being retrieved.

Returns The value of the row's internal storage.

store_value(*row: asyncqlio.orm.schema.table.Table*, *value: typing.Any*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** – The row to store in.
- **value** – The value to store in the row.

class asyncqlio.orm.schema.types.**Timestamp**
Bases: *asyncqlio.orm.schema.types.ColumnType*

Represents a TIMESTAMP type.

create_default() → asyncqlio.orm.schema.types.ColumnType
Creates the default object for this table in the event that a type is passed to a column, instead of an instance.

in_(**args*) → asyncqlio.orm.operators.In
Returns an IN operator, checking if a value in this column is in a tuple of items.

Parameters **args** – The items to check.

on_get(*row: asyncqlio.orm.schema.table.Table*) → typing.Any
Called when a value is retrieved from this column.

Parameters **row** – The row that is being retrieved.

Returns The value of the row's internal storage.

on_set(*row: asyncqlio.orm.schema.table.Table*, *value: typing.Any*) → typing.Any
Called when a value is a set on this column.

This is the default method - it will call *ColumnType.validate_set()* to validate the type before storing it. This is useful for simple column types.

Parameters

- **row** – The row this value is being set on.
- **value** – The value being set.

store_value(*row: asyncqlio.orm.schema.table.Table*, *value: typing.Any*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** – The row to store in.
- **value** – The value to store in the row.

asyncqlio.orm.schema.decorators

Decorator helpers for tables.

Functions

hidden(func)	Marks a function as "hidden" - i.e the function can only be resolved on the *Table* it exists on, and **NOT** the TableRow associated with said table.
row_attr(func)	Marks a function as a "row attribute" - something that resolves on a TableRow as well as the *Table* it is a member of.

asyncqlio.orm.schema.decorators.**row_attr**(*func*)
Marks a function as a "row attribute" - something that resolves on a TableRow as well as the *Table* it is a member of.

```
class User(Table):
    ...

    @row_attr
    def full_name(self: TableRow):
        # this will resolve on TableRow.full_name
        return self.first_name + self.last_name
```

This allows writing attributes that resolve from table rows. Otherwise, they are treated as regular functions that need to be called on a TableRow object.

Parameters **func** – The function to decorate.

Returns A wrapper function. The original function can be found via `.__wrapped__`.

asyncqlio.orm.schema.decorators.**hidden**(*func*)

Marks a function as "hidden" - i.e the function can only be resolved on the *Table* it exists on, and **NOT** the TableRow associated with said table.

asyncqlio.orm.ddl

asyncqlio.orm.query

Classes for query objects.

Classes

BaseQuery(sess: asyncqlio.orm.session.Session)	A base query object.
BulkDeleteQuery(...)	Represents a **bulk delete query**.
BulkQuery(sess: asyncqlio.orm.session.Session)	Represents a **bulk query**.
BulkUpdateQuery(...)	Represents a **bulk update query**.
InsertQuery(sess: asyncqlio.orm.session.Session)	Represents an INSERT query.
ResultGenerator(...)	A helper class that will generate new results from a query when iterated over.
RowDeleteQuery(...)	Represents a row deletion query.
RowUpdateQuery(...)	Represents a **row update query**.
SelectQuery(...)	Represents a SELECT query, which fetches data from the database.

class asyncqlio.orm.query.**BaseQuery**(*sess: asyncqlio.orm.session.Session*)

Bases: asyncqlio.meta.AsyncABC

A base query object.

Parameters **sess** – The *Session* associated with this query.

generate_sql() → typing.Tuple[str, typing.Mapping[str, typing.Any]]

Generates the SQL for this query. :return: A two item tuple, the SQL to use and a mapping of params to pass.

coroutine run()

Runs this query.

class asyncqlio.orm.query.**ResultGenerator**(*q: asyncqlio.orm.query.SelectQuery*)

Bases: collections.abc.AsyncIterator

A helper class that will generate new results from a query when iterated over.

Parameters **q** – The *SelectQuery* to use.

coroutine flatten() → typing.List[asyncqlio.orm.schema.table.Table]
Flattens this query into a single list.

class asyncqlio.orm.query.**SelectQuery**(*session: asyncqlio.orm.session.Session*)
Bases: *asyncqlio.orm.query.BaseQuery*

Represents a SELECT query, which fetches data from the database.

This is not normally created by user code directly, but rather as a result of a `Session.select()` call.

```
sess = db.get_session()
async with sess:
    query = sess.select.from_(User)  # query is instance of SelectQuery
    # alternatively, but not recommended
    query = sess.select(User)
```

However, it is possible to create this class manually:

```
query = SelectQuery(db.get_session()
query.set_table(User)
query.add_condition(User.id == 2)
user = await query.first()
```

table = None
The table being queried.

conditions = None
A list of conditions to fulfil.

row_limit = None
The limit on the number of rows returned from this query.

row_offset = None
The offset to start fetching rows from.

orderer = None
The column to order by.

get_required_join_paths()
Gets the required join paths for this query.

generate_sql() → typing.Tuple[str, dict]
Generates the SQL for this query.

coroutine first() → asyncqlio.orm.schema.table.Table
Gets the first result that matches from this query.

Returns A *Table* instance representing the first item, or None if no item matched.

coroutine all() → asyncqlio.orm.query.ResultGenerator
Gets all results that match from this query.

Returns A *ResultGenerator* that can be iterated over.

map_columns(*results: typing.Mapping[str, typing.Any]*) → asyncqlio.orm.schema.table.Table
Maps columns in a result row to a *Table* instance object.

Parameters **results** – A single row of results from the query cursor.

Returns A new *Table* instance that represents the row returned.

map_many(**rows: typing.Mapping[str, typing.Any]*)
Maps many records to one row.

This will group the records by the primary key of the main query table, then add additional columns as appropriate.

from_(*tbl*) → asyncqlio.orm.query.SelectQuery
Sets the table this query is selecting from.

Parameters **tbl** – The *Table* object to select.

Returns This query.

where(**conditions: asyncqlio.orm.operators.BaseOperator*) → asyncqlio.orm.query.SelectQuery
Adds a WHERE clause to the query. This is a shortcut for *SelectQuery.add_condition()*.

```
sess.select.from_(User).where(User.id == 1)
```

Parameters **conditions** – The conditions to use for this WHERE clause.

Returns This query.

limit(*row_limit: int*) → asyncqlio.orm.query.SelectQuery
Sets a limit of the number of rows that can be returned from this query.

Parameters **row_limit** – The maximum number of rows to return.

Returns This query.

offset(*offset: int*) → asyncqlio.orm.query.SelectQuery
Sets the offset of rows to start returning results from/

Parameters **offset** – The row offset.

Returns This query.

order_by(**col: typing.Union[asyncqlio.orm.schema.column.Column, asyncqlio.orm.operators.Sorter], sort_order: str = 'asc'*)
Sets the order by clause for this query.

The argument provided can either be a *Column*, or a *Sorter* which is provided by *Column.asc()* / *Column.desc()*. By default, `asc` is used when passing a column.

set_table(*tbl*) → asyncqlio.orm.query.SelectQuery
Sets the table to query on.

Parameters **tbl** – The *Table* object to set.

Returns This query.

add_condition(*condition: asyncqlio.orm.operators.BaseOperator*) → asyncqlio.orm.query.SelectQuery
Adds a condition to the query/

Parameters **condition** – The *BaseOperator* to add.

Returns This query.

class asyncqlio.orm.query.**InsertQuery**(*sess: asyncqlio.orm.session.Session*)
Bases: *asyncqlio.orm.query.BaseQuery*

Represents an INSERT query.

rows_to_insert = None
A list of rows to generate the insert statements for.

coroutine run() → typing.List[asyncqlio.orm.schema.table.Table]
Runs this query.

Returns A list of inserted `md_table.Table`.

rows(**rows: asyncqlio.orm.schema.table.Table*) → asyncqlio.orm.query.InsertQuery
Adds a set of rows to the query.

Parameters **rows** – The rows to insert.

Returns This query.

add_row(*row: asyncqlio.orm.schema.table.Table*) → asyncqlio.orm.query.InsertQuery
Adds a row to this query, allowing it to be executed later.

Parameters **row** – The *Table* instance to use for this query.

Returns This query.

generate_sql() → typing.List[typing.Tuple[str, tuple]]
Generates the SQL statements for this insert query.

This will return a list of two-item tuples to execute:

- The SQL query+params to emit to actually insert the row

class asyncqlio.orm.query.**BulkQuery**(*sess: asyncqlio.orm.session.Session*)
Bases: *asyncqlio.orm.query.BaseQuery*

Represents a **bulk query**.

This allows adding conditionals to the query.

conditions = None
The list of conditions to query by.

table(*table: typing.Type[asyncqlio.orm.schema.table.Table]*)
Sets the table for this query.

where(**conditions: asyncqlio.orm.operators.ComparisonOp*)
Sets the conditions for this query.

set_table(*table: typing.Type[asyncqlio.orm.schema.table.Table]*)
Sets a table on this query.

add_condition(*condition: asyncqlio.orm.operators.BaseOperator*)
Adds a condition to this query.

generate_sql() → typing.Tuple[str, typing.Mapping[str, typing.Any]]
Generates the SQL for this query. :return: A two item tuple, the SQL to use and a mapping of params to pass.

coroutine run()
Runs this query.

class asyncqlio.orm.query.**BulkUpdateQuery**(*sess: asyncqlio.orm.session.Session*)
Bases: *asyncqlio.orm.query.BulkQuery*

Represents a **bulk update query**. This updates many rows based on certain criteria.

```
query = BulkUpdateQuery(session)

# style 1: manual
query.set_table(User)
query.add_condition(User.xp < 300)
```

```
# add on a value
query.set_update(User.xp + 100)
# or set a value
query.set_update(User.xp.set(300))
await query.run()

# style 2: builder
await query.table(User).where(User.xp < 300).set(User.xp + 100).run()
await query.table(User).where(User.xp < 300).set(User.xp, 300).run()
```

setting = None
The thing to set on the updated rows.

set(*setter*, *value: typing.Any = None*)
Sets a column in this query.

set_update(*update*)
Sets the update for this query.

generate_sql()
Generates the SQL for this query.

add_condition(*condition: asyncqlio.orm.operators.BaseOperator*)
Adds a condition to this query.

set_table(*table: typing.Type[asyncqlio.orm.schema.table.Table]*)
Sets a table on this query.

table(*table: typing.Type[asyncqlio.orm.schema.table.Table]*)
Sets the table for this query.

where(**conditions: asyncqlio.orm.operators.ComparisonOp*)
Sets the conditions for this query.

class asyncqlio.orm.query.**BulkDeleteQuery**(*sess: asyncqlio.orm.session.Session*)
Bases: *asyncqlio.orm.query.BulkQuery*

Represents a **bulk delete query**. This deletes many rows based on criteria.

```
query = BulkDeleteQuery(session)

# style 1: manual
query.set_table(User)
query.add_condition(User.xp < 300)
await query.run()

# style 2: builder
await query.table(User).where(User.xp < 300).run()
await query.table(User).where(User.xp < 300).run()
```

add_condition(*condition: asyncqlio.orm.operators.BaseOperator*)
Adds a condition to this query.

set_table(*table: typing.Type[asyncqlio.orm.schema.table.Table]*)
Sets a table on this query.

table(*table: typing.Type[asyncqlio.orm.schema.table.Table]*)
Sets the table for this query.

where(**conditions: asyncqlio.orm.operators.ComparisonOp*)
Sets the conditions for this query.

class asyncqlio.orm.query.**RowUpdateQuery**(*sess: asyncqlio.orm.session.Session*)
Bases: *asyncqlio.orm.query.BaseQuery*

Represents a **row update query**. This is **NOT** a bulk update query - it is used for updating specific rows.

rows_to_update = None
The list of rows to update.

coroutine run()
Executes this query.

rows(**rows: asyncqlio.orm.schema.table.Table*) → asyncqlio.orm.query.RowUpdateQuery
Adds a set of rows to the query.

Parameters **rows** – The rows to insert.

Returns This query.

add_row(*row: asyncqlio.orm.schema.table.Table*) → asyncqlio.orm.query.RowUpdateQuery
Adds a row to this query, allowing it to be executed later.

Parameters **row** – The *Table* instance to use for this query.

Returns This query.

generate_sql() → typing.List[typing.Tuple[str, tuple]]
Generates the SQL statements for this row update query.

This will return a list of two-item tuples to execute:

- The SQL query+params to emit to actually insert the row

class asyncqlio.orm.query.**RowDeleteQuery**(*sess: asyncqlio.orm.session.Session*)
Bases: *asyncqlio.orm.query.BaseQuery*

Represents a row deletion query. This is **NOT** a bulk delete query - it is used for deleting specific rows.

rows_to_delete = None
The list of rows to delete.

rows(**rows: asyncqlio.orm.schema.table.Table*) → asyncqlio.orm.query.RowDeleteQuery
Adds a set of rows to the query.

Parameters **rows** – The rows to insert.

Returns This query.

add_row(*row: asyncqlio.orm.schema.table.Table*)
Adds a row to this query.

Parameters **row** – The *Table* instance

Returns

generate_sql() → typing.List[typing.Tuple[str, tuple]]
Generates the SQL statements for this row delete query.

This will return a list of two-item tuples to execute:

- The SQL query+params to emit to actually insert the row

asyncqlio.orm.session

Functions

enforce_open(func)	

Classes

Session(bind: asyncqlio.db.DatabaseInterface)	Sessions act as a temporary window into the database.
SessionState	An enumeration.

class asyncqlio.orm.session.**SessionState**
Bases: enum.Enum

An enumeration.

class asyncqlio.orm.session.**Session**(*bind: asyncqlio.db.DatabaseInterface*)
Bases: object

Sessions act as a temporary window into the database. They are responsible for creating queries, inserting and updating rows, etc.

Sessions are bound to a *DatabaseInterface* instance which they use to get a transaction and execute queries in.

```
# get a session from our db interface
sess = db.get_session()
```

Parameters **bind** – The *DatabaseInterface* instance we are bound to.

transaction = None
The current *BaseTransaction* this Session is associated with. The transaction is used for making queries and inserts, etc.

select
Creates a new SELECT query that can be built upon.

Returns A new *SelectQuery*.

insert
Creates a new INSERT INTO query that can be built upon.

Returns A new *InsertQuery*.

update
Creates a new bulk UPDATE query that can be built upon.

Returns A new *BulkUpdateQuery*.

delete
Creates a new bulk DELETE query that can be built upon.

Returns A new *BulkDeleteQuery*.

coroutine start() → asyncqlio.orm.session.Session
Starts the session, acquiring a transaction connection which will be used to modify the DB.

This **must** be called before using the session.

```
sess = db.get_session()
await sess.start()
```

Note: When using `async with`, this is automatically called.

checkpoint(*checkpoint_name: str*)
Sets a new checkpoint.

Parameters **checkpoint_name** – The name of the checkpoint to use.

uncheckpoint(*checkpoint_name: str*)
Releases a checkpoint.

Parameters **checkpoint_name** – The name of the checkpoint to release.

commit()
Commits the current session, running inserts/updates/deletes.

This will **not** close the session; it can be re-used after a commit.

rollback(*checkpoint: str = None*)
Rolls the current session back. This is useful if an error occurs inside your code.

Parameters **checkpoint** – The checkpoint to roll back to, if applicable.

close()
Closes the current session.

Warning: This will **NOT COMMIT ANY DATA**. Old data will die.

fetch(*sql: str*, *params=None*)
Fetches a single row.

execute(*sql: str, params: typing.Union[typing.Mapping[str, typing.Any], typing.Iterable[typing.Any]] = None*)
Executes SQL inside the current session.

This is part of the **low-level API.**

Parameters

- **sql** – The SQL to execute.
- **params** – The parameters to use inside the query.

cursor(*sql: str, params: typing.Union[typing.Mapping[str, typing.Any], typing.Iterable[typing.Any]] = None*)
Executes SQL inside the current session, and returns a new `BaseResultSet`.

Parameters

- **sql** – The SQL to execute.
- **params** – The parameters to use inside the query.

insert_now(*row: asyncqlio.orm.schema.table.Table*) → typing.Any
Inserts a row NOW.

> **Warning:** This will only generate the INSERT statement for the row now. Only *Session.commit()* will actually commit the row to storage.
>
> Also, tables with auto-incrementing fields will only have their first field filled in outside of Postgres databases.

Parameters **row** – The *Table* instance to insert.

Returns The row, with primary key included.

update_now(*row: asyncqlio.orm.schema.table.Table*) → asyncqlio.orm.schema.table.Table
Updates a row NOW.

> **Warning:** This will only generate the UPDATE statement for the row now. Only *Session.commit()* will actually commit the row to storage.

Parameters **row** – The *Table* instance to update.

Returns The *Table* instance that was updated.

delete_now(*row: asyncqlio.orm.schema.table.Table*) → asyncqlio.orm.schema.table.Table
Deletes a row NOW.

coroutine run_select_query(*query: asyncqlio.orm.query.SelectQuery*)
Executes a select query.

> **Warning:** Unlike the other *run_*_query* methods, this method should not be used without a good reason; it creates a special class that is used for the query.
>
> Use `SelectQuery.first` or `SelectQuery.all`.

Parameters **query** – The *SelectQuery* to use.

Returns A `_ResultGenerator` for this query.

coroutine run_insert_query(*query: asyncqlio.orm.query.InsertQuery*)
Executes an insert query.

Parameters **query** – The *InsertQuery* to use.

Returns The list of rows that were inserted.

coroutine run_update_query(*query: asyncqlio.orm.query.BaseQuery*)
Executes an update query.

Parameters **query** – The *RowUpdateQuery* or *BulkUpdateQuery* to execute.

coroutine run_delete_query(*query: asyncqlio.orm.query.RowDeleteQuery*)
Executes a delete query.

Parameters **query** – The *RowDeleteQuery* or *BulkDeleteQuery* to execute.

coroutine add(*row: asyncqlio.orm.schema.table.Table*) → asyncqlio.orm.schema.table.Table
Adds a row to the current transaction. This will emit SQL that will generate an INSERT or UPDATE statement, and then update the primary key of this row.

> **Warning:** This will only generate the INSERT statement for the row now. Only *Session.commit()* will actually commit the row to storage.

Parameters **row** – The *Table* instance object to add to the transaction.

Returns The *Table* instance with primary key filled in, if applicable.

coroutine merge(*row: asyncqlio.orm.schema.table.Table*) → asyncqlio.orm.schema.table.Table
Merges a row with a row that already exists in the database.

This should be used for rows that have a primary key, but were not returned from `Session.select()`.

Parameters **row** – The *Table* instance to merge.

Returns The *Table* instance once updated.

coroutine remove(*row: asyncqlio.orm.schema.table.Table*) → asyncqlio.orm.schema.table.Table
Removes a row from the database.

Parameters **row** – The *Table* instance to remove.

asyncqlio.orm.inspection

Inspection module - contains utilities for inspecting Table objects and Row objects.

Functions

get_pk(...)	Gets the primary key for a Table row.
get_row_history(...)	Gets the history for the specified row.
get_row_session(...)	Gets the *Session* associated with a `TableRow`.

asyncqlio.orm.inspection.**get_row_session**(*row: asyncqlio.orm.schema.table.Table*) → asyncqlio.orm.session.Session
Gets the *Session* associated with a `TableRow`.

Parameters **row** – The *Table* instance to inspect.

asyncqlio.orm.inspection.**get_row_history**(*row: asyncqlio.orm.schema.table.Table*) → typing.Dict[asyncqlio.orm.schema.column.Column, typing.Dict[str, typing.Any]]
Gets the history for the specified row.

This returns a dict, indexed by Column, with values being another dict with *old* and *new* keys that represent the old and new values of the item.

asyncqlio.orm.inspection.**get_pk**(*row: asyncqlio.orm.schema.table.Table*, *as_tuple: bool = True*)
Gets the primary key for a Table row.

Parameters

- **row** – The *Table* instance to extract the PK from.
- **as_tuple** – Should this PK always be returned as a tuple?

asyncqlio.orm.operators

Classes for operators returned from queries.

Functions

requires_bop(...)	A decorator that marks a magic method as requiring another BaseOperator.

Classes

And(*ops: asyncqlio.orm.operators.BaseOperator)	Represents an AND operator in a query.
AscSorter(...)	
BaseOperator	The base operator class.
BasicSetter(...)	Represents a basic setting operation.
ColumnValueMixin(...)	A mixin that specifies that an operator takes both a Column and a Value as arguments.
ComparisonOp(...)	A helper class that implements easy generation of comparison-based operators.
DecrementSetter(...)	Represents a decrement setter.
DescSorter(...)	
Eq(...)	Represents an equality operator.
Gt(...)	Represents a more than operator.
Gte(...)	Represents a more than or equals to operator.
HackyILike(...)	A "hacky" ILIKE operator for databases that do not support it.
ILike(...)	Represents an ILIKE operator.
In(...)	
IncrementSetter(...)	Represents an increment setter.
Like(...)	Represents a LIKE operator.
Lt(...)	Represents a less than operator.
Lte(...)	Represents a less than or equals to operator.
NEq(...)	Represents a non-equality operator.
OperatorResponse(sql: str, parameters: dict)	A storage class for the generated SQL from an operator.
Or(*ops: asyncqlio.orm.operators.BaseOperator)	Represents an OR operator in a query.
Sorter(...)	A generic sorter operator, for use in ORDER BY.
ValueSetter(...)	Represents a value setter (`col = 1`).

class asyncqlio.orm.operators.**OperatorResponse**(*sql: str*, *parameters: dict*)

Bases: object

A storage class for the generated SQL from an operator.

Parameters

- **sql** – The generated SQL for this operator.
- **parameters** – A dict of parameters to use for this response.

asyncqlio.orm.operators.**requires_bop**(*func*) → typing.Callable[[asyncqlio.orm.operators.BaseOperator, asyncqlio.orm.operators.BaseOperator], typing.Any]

A decorator that marks a magic method as requiring another BaseOperator.

Parameters **func** – The function to decorate.

Returns A function that returns NotImplemented when the class required isn't specified.

class asyncqlio.orm.operators.**BaseOperator**
Bases: abc.ABC

The base operator class.

get_param(*emitter: typing.Callable[[str], str], counter: itertools.count*) → typing.Tuple[str, str]
Gets the next parameter.

Parameters

- **emitter** – A function that emits a parameter name that can be formatted in a SQL query.
- **counter** – The counter for parameters.

generate_sql(*emitter: typing.Callable[[str], str], counter: itertools.count*) → asyncqlio.orm.operators.OperatorResponse
Generates the SQL for an operator.

Parameters must be generated using the emitter callable.

Parameters

- **emitter** – A callable that can be used to generate param placeholders in a query.
- **counter** – The current "parameter number".

Returns A *OperatorResponse* representing the result.

Warning: The param name and the param can be empty if none is to be returned.

class asyncqlio.orm.operators.**And**(**ops: asyncqlio.orm.operators.BaseOperator*)
Bases: *asyncqlio.orm.operators.BaseOperator*

Represents an AND operator in a query.

This will join multiple other *BaseOperator* objects together.

get_param(*emitter: typing.Callable[[str], str], counter: itertools.count*) → typing.Tuple[str, str]
Gets the next parameter.

Parameters

- **emitter** – A function that emits a parameter name that can be formatted in a SQL query.
- **counter** – The counter for parameters.

class asyncqlio.orm.operators.**Or**(**ops: asyncqlio.orm.operators.BaseOperator*)
Bases: *asyncqlio.orm.operators.BaseOperator*

Represents an OR operator in a query.

This will join multiple other *BaseOperator* objects together.

get_param(*emitter: typing.Callable[[str], str], counter: itertools.count*) → typing.Tuple[str, str]
Gets the next parameter.

Parameters

- **emitter** – A function that emits a parameter name that can be formatted in a SQL query.
- **counter** – The counter for parameters.

class asyncqlio.orm.operators.**Sorter**(**columns: asyncqlio.orm.schema.column.Column*)
Bases: *asyncqlio.orm.operators.BaseOperator*

A generic sorter operator, for use in ORDER BY.

sort_order
The sort order for this row; ASC or DESC.

get_param(*emitter: typing.Callable[[str], str], counter: itertools.count*) → typing.Tuple[str, str]
Gets the next parameter.

Parameters

- **emitter** – A function that emits a parameter name that can be formatted in a SQL query.
- **counter** – The counter for parameters.

class asyncqlio.orm.operators.**ColumnValueMixin**(*column: asyncqlio.orm.schema.column.Column, value: typing.Any*)
Bases: object

A mixin that specifies that an operator takes both a Column and a Value as arguments.

```
class MyOp(BaseOperator, ColumnValueMixin):
    ...

# myop is constructed MyOp(col, value)
```

class asyncqlio.orm.operators.**BasicSetter**(*column: asyncqlio.orm.schema.column.Column, value: typing.Any*)
Bases: *asyncqlio.orm.operators.BaseOperator*, *asyncqlio.orm.operators.ColumnValueMixin*

Represents a basic setting operation. Used for bulk queries.

set_operator

Returns The "setting" operator to use when generating the SQL.

get_param(*emitter: typing.Callable[[str], str], counter: itertools.count*) → typing.Tuple[str, str]
Gets the next parameter.

Parameters

- **emitter** – A function that emits a parameter name that can be formatted in a SQL query.
- **counter** – The counter for parameters.

class asyncqlio.orm.operators.**ValueSetter**(*column: asyncqlio.orm.schema.column.Column, value: typing.Any*)
Bases: *asyncqlio.orm.operators.BasicSetter*

Represents a value setter (`col = 1`).

get_param(*emitter: typing.Callable[[str], str], counter: itertools.count*) → typing.Tuple[str, str]
Gets the next parameter.

Parameters

- **emitter** – A function that emits a parameter name that can be formatted in a SQL query.
- **counter** – The counter for parameters.

class asyncqlio.orm.operators.**IncrementSetter**(*column: asyncqlio.orm.schema.column.Column*, *value: typing.Any*)

Bases: *asyncqlio.orm.operators.BasicSetter*

Represents an increment setter. (`col = col + 1`)

get_param(*emitter: typing.Callable[[str], str]*, *counter: itertools.count*) → typing.Tuple[str, str]

Gets the next parameter.

Parameters

- **emitter** – A function that emits a parameter name that can be formatted in a SQL query.
- **counter** – The counter for parameters.

class asyncqlio.orm.operators.**DecrementSetter**(*column: asyncqlio.orm.schema.column.Column*, *value: typing.Any*)

Bases: *asyncqlio.orm.operators.BasicSetter*

Represents a decrement setter.

get_param(*emitter: typing.Callable[[str], str]*, *counter: itertools.count*) → typing.Tuple[str, str]

Gets the next parameter.

Parameters

- **emitter** – A function that emits a parameter name that can be formatted in a SQL query.
- **counter** – The counter for parameters.

class asyncqlio.orm.operators.**ComparisonOp**(*column: asyncqlio.orm.schema.column.Column*, *value: typing.Any*)

Bases: *asyncqlio.orm.operators.ColumnValueMixin*, *asyncqlio.orm.operators.BaseOperator*

A helper class that implements easy generation of comparison-based operators.

To customize the operator provided, set the value of `operator` in the class body.

get_param(*emitter: typing.Callable[[str], str]*, *counter: itertools.count*) → typing.Tuple[str, str]

Gets the next parameter.

Parameters

- **emitter** – A function that emits a parameter name that can be formatted in a SQL query.
- **counter** – The counter for parameters.

class asyncqlio.orm.operators.**Eq**(*column: asyncqlio.orm.schema.column.Column*, *value: typing.Any*)

Bases: *asyncqlio.orm.operators.ComparisonOp*

Represents an equality operator.

get_param(*emitter: typing.Callable[[str], str]*, *counter: itertools.count*) → typing.Tuple[str, str]

Gets the next parameter.

Parameters

- **emitter** – A function that emits a parameter name that can be formatted in a SQL query.
- **counter** – The counter for parameters.

class asyncqlio.orm.operators.**NEq**(*column: asyncqlio.orm.schema.column.Column*, *value: typing.Any*)

Bases: *asyncqlio.orm.operators.ComparisonOp*

Represents a non-equality operator.

get_param(*emitter: typing.Callable[[str], str]*, *counter: itertools.count*) → typing.Tuple[str, str]

Gets the next parameter.

Parameters

- **emitter** – A function that emits a parameter name that can be formatted in a SQL query.
- **counter** – The counter for parameters.

class asyncqlio.orm.operators.**Lt**(*column: asyncqlio.orm.schema.column.Column*, *value: typing.Any*)

Bases: *asyncqlio.orm.operators.ComparisonOp*

Represents a less than operator.

get_param(*emitter: typing.Callable[[str], str]*, *counter: itertools.count*) → typing.Tuple[str, str]

Gets the next parameter.

Parameters

- **emitter** – A function that emits a parameter name that can be formatted in a SQL query.
- **counter** – The counter for parameters.

class asyncqlio.orm.operators.**Gt**(*column: asyncqlio.orm.schema.column.Column*, *value: typing.Any*)

Bases: *asyncqlio.orm.operators.ComparisonOp*

Represents a more than operator.

get_param(*emitter: typing.Callable[[str], str]*, *counter: itertools.count*) → typing.Tuple[str, str]

Gets the next parameter.

Parameters

- **emitter** – A function that emits a parameter name that can be formatted in a SQL query.
- **counter** – The counter for parameters.

class asyncqlio.orm.operators.**Lte**(*column: asyncqlio.orm.schema.column.Column*, *value: typing.Any*)

Bases: *asyncqlio.orm.operators.ComparisonOp*

Represents a less than or equals to operator.

get_param(*emitter: typing.Callable[[str], str]*, *counter: itertools.count*) → typing.Tuple[str, str]

Gets the next parameter.

Parameters

- **emitter** – A function that emits a parameter name that can be formatted in a SQL query.
- **counter** – The counter for parameters.

class asyncqlio.orm.operators.**Gte**(*column: asyncqlio.orm.schema.column.Column*, *value: typing.Any*)

Bases: *asyncqlio.orm.operators.ComparisonOp*

Represents a more than or equals to operator.

get_param(*emitter: typing.Callable[[str], str]*, *counter: itertools.count*) → typing.Tuple[str, str]

Gets the next parameter.

Parameters

- **emitter** – A function that emits a parameter name that can be formatted in a SQL query.
- **counter** – The counter for parameters.

class asyncqlio.orm.operators.**Like**(*column: asyncqlio.orm.schema.column.Column*, *value: typing.Any*)

Bases: *asyncqlio.orm.operators.ComparisonOp*

Represents a LIKE operator.

get_param(*emitter: typing.Callable[[str], str], counter: itertools.count*) → typing.Tuple[str, str]

Gets the next parameter.

Parameters

- **emitter** – A function that emits a parameter name that can be formatted in a SQL query.
- **counter** – The counter for parameters.

class asyncqlio.orm.operators.**ILike**(*column: asyncqlio.orm.schema.column.Column*, *value: typing.Any*)

Bases: *asyncqlio.orm.operators.ComparisonOp*

Represents an ILIKE operator.

> **Warning:** This operator is not natively supported on all dialects. If used on a dialect that doesn't support it, it will fallback to a lowercase LIKE.

get_param(*emitter: typing.Callable[[str], str], counter: itertools.count*) → typing.Tuple[str, str]

Gets the next parameter.

Parameters

- **emitter** – A function that emits a parameter name that can be formatted in a SQL query.
- **counter** – The counter for parameters.

class asyncqlio.orm.operators.**HackyILike**(*column: asyncqlio.orm.schema.column.Column*, *value: typing.Any*)

Bases: *asyncqlio.orm.operators.BaseOperator*, *asyncqlio.orm.operators.ColumnValueMixin*

A "hacky" ILIKE operator for databases that do not support it.

get_param(*emitter: typing.Callable[[str], str], counter: itertools.count*) → typing.Tuple[str, str]

Gets the next parameter.

Parameters

- **emitter** – A function that emits a parameter name that can be formatted in a SQL query.
- **counter** – The counter for parameters.

2.1.3 asyncqlio.backends

SQL driver backends for asyncqlio.

postgresql	PostgreSQL backends.

Continued on next page

Table 2.18 – continued from previous page

sqlite3	
mysql	

asyncqlio.backends.postgresql

PostgreSQL backends.

asyncpg

Classes

PostgresqlDialect	The dialect for Postgres.

class asyncqlio.backends.postgresql.**PostgresqlDialect**
Bases: asyncqlio.backends.base.BaseDialect

The dialect for Postgres.

asyncqlio.backends.sqlite3

Classes

Sqlite3Dialect	The dialect for sqlite3.

class asyncqlio.backends.sqlite3.**Sqlite3Dialect**
Bases: asyncqlio.backends.base.BaseDialect

The dialect for sqlite3.

asyncqlio.backends.mysql

Classes

MysqlDialect	The dialect for MySQL.

class asyncqlio.backends.mysql.**MysqlDialect**
Bases: asyncqlio.backends.base.BaseDialect

The dialect for MySQL.

2.1.4 asyncqlio.exc

Exceptions for asyncqlio.

Exceptions

DatabaseException	The base class for ALL exceptions.
IntegrityError	Raised when a column's integrity is not preserved (e.g.
NoSuchColumnError	Raised when a non-existing column is requested.
OperationalError	Raised when an operational error has occurred.
SchemaError	Raised when there is an error in the database schema.

exception asyncqlio.exc.**DatabaseException**
Bases: Exception

The base class for ALL exceptions.

Catch this if you wish to catch any custom exception raised inside the lib.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception asyncqlio.exc.**SchemaError**
Bases: *asyncqlio.exc.DatabaseException*

Raised when there is an error in the database schema.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception asyncqlio.exc.**IntegrityError**
Bases: *asyncqlio.exc.DatabaseException*

Raised when a column's integrity is not preserved (e.g. null or unique violations).

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception asyncqlio.exc.**OperationalError**
Bases: *asyncqlio.exc.DatabaseException*

Raised when an operational error has occurred.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception asyncqlio.exc.**NoSuchColumnError**
Bases: *asyncqlio.exc.DatabaseException*

Raised when a non-existing column is requested.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

2.1.5 asyncqlio.meta

Useful metamagic classes, such as async ABCs.

Functions

make_proxy(name: str)	Makes a proxy object for magic methods.
proxy_to_getattr(*magic_methods: str)	Proxies a method to to __getattr__ when it would not be normally proxied.

Classes

AsyncABC	
AsyncABCMeta(name, bases, methods)	Metaclass that gives all of the features of an abstract base class, but additionally enforces coroutine correctness on subclasses.
AsyncInstanceType(name, bases, methods)	Metaclass that allows for asynchronous instance initialization and the __init__() method to be defined as a coroutine.
AsyncObject	

asyncqlio.meta.**make_proxy**(*name: str*)
Makes a proxy object for magic methods.

asyncqlio.meta.**proxy_to_getattr**(**magic_methods: str*)
Proxies a method to to __getattr__ when it would not be normally proxied.

This is used for magic methods that are slot loaded (__setattr__ etc.)

Parameters **magic_methods** – The magic methods to proxy to getattr.

class asyncqlio.meta.**AsyncABCMeta**(*name*, *bases*, *methods*)
Bases: abc.ABCMeta

Metaclass that gives all of the features of an abstract base class, but additionally enforces coroutine correctness on subclasses. If any method is defined as a coroutine in a parent, it must also be defined as a coroutine in any child.

mro() → list
return a type's method resolution order

register(*subclass*)
Register a virtual subclass of an ABC.

Returns the subclass, to allow usage as a class decorator.

class asyncqlio.meta.**AsyncInstanceType**(*name*, *bases*, *methods*)
Bases: *asyncqlio.meta.AsyncABCMeta*

Metaclass that allows for asynchronous instance initialization and the __init__() method to be defined as a coroutine. Usage: class Spam(metaclass=AsyncInstanceType):

async def __init__(self, x, y): self.x = x self.y = y

async def main(): s = await Spam(2, 3) . . .

mro() → list
return a type's method resolution order

register(*subclass*)
Register a virtual subclass of an ABC.

Returns the subclass, to allow usage as a class decorator.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

C

D

E

F

G

H

I

J

S

T

U

V

W