asyncqlio Documentation

Release 0.1.1.dev128

Laura Dickinson

Feb 23, 2018

Contents

asyncqlio is a Python 3.5+ for SQL Relational Databases, that allows access from async code.

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)
- MySQL (aiomysql)
- SQLite3 (sqlite3)

Other databases may be supported in the future.

CHAPTER 1

Contents

1.1 Changelog

1.1.1 0.2.0

- Prevent `getattr` on relationships from causing massive recursion errors.
- Separate parts of *`Session`* out into a new *`SessionBase`*.
- Add new `DDLSession`, inherited from *`SessionBase`*.
- Fix `Sqlite3Transaction.execute()` when no params are passed.
- Add *`Session.truncate()`* to truncate tables. Falls back to DELETE FROM if it can't.
- Add `TableMeta.truncate()` to truncate tables, which calls *`Session.truncate()`*.
- Add *`BaseResultSet.flatten()`* to flatten a result set.
- Fix the `aiomysql` connector to use ANSI sql mode.
- Add new *`Index`* representing an index in a database. (PR #30)
- Add `TableMeta.create()` to create tables from *`Table`* classes. (PR #30)
- Add *`Table.generate_schema()`* to create a representative Python class from the table. (PR #30)
- Add `DDLSession.get_indexes()` to get *`Index`* objects from an existing database. (PR #30)
- Add `DDLSession.create_index()` to create an index on an existing database table. (PR #30)
- Support *`Index`* objects in `DDLSession.create_table()`.
- Actually generate foreign keys upon table creation.
- Add *`Serial`*, *`BigSerial`*, and *`SmallSerial`* types to support automatic incrementation. (issue #17, PR #34)
- Add *`UpsertQuery`*. (issue #32, PR #38)

- Change *DatabaseInterface.emit_param()* to globally keep track of the param counter, which simplifies a lot of operator code.

1.1.2 0.1.0 (released 2017-07-30)

- Initial release.

1.2 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")
```

1.2.1 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.

1.2.2 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.

```
# connects the database and initializes the backend
await db.connect()
```

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

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

1.3 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.

1.3.1 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*, *tblbases*, *class_body*, *register=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** (`bool`) – Should this table be registered in the TableMetadata?
- **table_name** – The name for this table.

metadata = None

The *TableMetadata* for this table.

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.

Return type *PrimaryKey*

mro() → list

return a type's method resolution order

1.3.2 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 `Integer`, and is set to be a primary key and unique. Of course, you can name it anything and add different type; all that matters is that the object is a *Column*.

Warning: Auto-incrementing columns is handled specially for databases that support the SERIAL type. To autoincrement a column on Sqlite3, use a type of *Integer* and `primary_key=True`. For other databases, use *Serial* (or Big/Small variants) of such.

class asyncqlio.orm.schema.column.**Column**(*type_*, ***, *primary_key=False*, *nullable=False*, *default=<object object>*, *index=True*, *unique=False*, *foreign_key=None*, *table=None*)

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** (`Union`[*ColumnType*, `Type`[*ColumnType*]]) – The *ColumnType* that represents the type of this column.
- **primary_key** (`bool`) – Is this column the table's Primary Key (the unique identifier that identifies each row)?
- **nullable** (`bool`) – Can this column be NULL?
- **default** (`Any`) – The client-side default for this column. If no value is provided when inserting, this value will automatically be added to the insert query.
- **index** (`bool`) – Should this column be indexed?
- **unique** (`bool`) – Is this column unique?
- **foreign_key** (`Optional`[*ForeignKey*]) – The *ForeignKey* associated with this column.

__init__(*type_*, ***, *primary_key=False*, *nullable=False*, *default=<object object>*, *index=True*, *unique=False*, *foreign_key=None*, *table=None*)

Parameters

- **type** (Union[*ColumnType*, Type[*ColumnType*]]) – The *ColumnType* that represents the type of this column.
- **primary_key** (bool) – Is this column the table's Primary Key (the unique identifier that identifies each row)?
- **nullable** (bool) – Can this column be NULL?
- **default** (Any) – The client-side default for this column. If no value is provided when inserting, this value will automatically be added to the insert query.
- **index** (bool) – Should this column be indexed?
- **unique** (bool) – Is this column unique?
- **foreign_key** (Optional[*ForeignKey*]) – 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.

indexed = None
If this Column is indexed.

unique = None
If this Column is unique.

foreign_key = None
The foreign key associated with this column.

__repr__()
Return repr(self).

__hash__()
Return hash(self).

__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.

table_name
The name of this column's table.

Return type str

autoincrement
Whether this column is set to autoincrement.

Return type bool

classmethod with_name(*name*, **args*, ***kwargs*)
Creates this column with a name already set.

Return type *Column*

get_ddl_sql()
Gets the DDL SQL for this column.

Return type str

generate_schema(*fp=None*)
Generates the library schema for this column.

Return type str

__eq__(*other*)
Return self==value.

Return type Union[*Eq*, bool]

__ne__(*other*)
Return self!=value.

Return type Union[*NEq*, bool]

__lt__(*other*)
Return self<value.

Return type *Lt*

__gt__(*other*)
Return self>value.

Return type *Gt*

__le__(*other*)
Return self<=value.

Return type *Lte*

__ge__(*other*)
Return self>=value.

Return type *Gte*

eq(*other*)
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.

Return type *Eq*

ne(*other*)
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.

Return type *NEq*

asc()
Returns the ascending sorter operator for this column.

Return type *AscSorter*

desc()
Returns the descending sorter operator for this column.

Return type *DescSorter*

set(*value*)
Sets this column in a bulk update.

Return type *ValueSetter*

incr(*value*)
Increments this column in a bulk update.

Return type *IncrementSetter*

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

decr(*value*)
Decrements this column in a bulk update.

Return type *DecrementSetter*

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

quoted_fullname_with_table(*table*)
Gets the quoted fullname with a table. This is used for columns with alias tables.

Parameters **table** (*TableMeta*) – The *Table* or *AliasedTable* to use.

Return type str

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

Return type *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).

__init_subclass__()
This method is called when a class is subclassed.

The default implementation does nothing. It may be overridden to extend subclasses.

__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=False*)
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** (bool) – Should the name be quoted?

Return type str

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*)
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.

index_name = None
The index name of this primary key, if any

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

Column 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()

Return type str

Returns The str SQL name of this type.

schema()

Return type str

Returns The library schema of this object.

If this is not defined, any arguments given to the type will not persist across schema generation.

validate_set(*row*, *value*)
Validates that the item being set is valid. This is called by the default `on_set`.

Parameters

- **row** (*Table*) – The row being set.

- **value** (Any) – The value to set.

Return type bool

Returns A bool indicating if this is valid or not.

store_value(*row*, *value*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** (*Table*) – The row to store in.
- **value** (Any) – The value to store in the row.

on_set(*row*, *value*)
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** (*Table*) – The row this value is being set on.
- **value** (Any) – The value being set.

Return type Any

on_get(*row*)
Called when a value is retrieved from this column.

Parameters **row** (*Table*) – The row that is being retrieved.

Return type Any

Returns The value of the row's internal storage.

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

Return type *ColumnType*

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

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

Return type *In*

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

Represents a VARCHAR() type.

size = None
The max size of this String.

sql()

Returns The str SQL name of this type.

schema()

Returns The library schema of this object.

If this is not defined, any arguments given to the type will not persist across schema generation.

validate_set(*row*, *value*)
Validates that the item being set is valid. This is called by the default `on_set`.

Parameters

- **row** – The row being set.
- **value** (`Any`) – The value to set.

Returns A bool indicating if this is valid or not.

like(*other*)
Returns a LIKE operator, checking if this column is LIKE another string.

Parameters **other** (`str`) – The other string to check.

Return type *Like*

ilike(*other*)
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** (`str`) – The other string to check.

Return type `Union`[*ILike*, *HackyILike*]

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

Return type *ColumnType*

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

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

Return type *In*

on_get(*row*)
Called when a value is retrieved from this column.

Parameters **row** (*Table*) – The row that is being retrieved.

Return type `Any`

Returns The value of the row's internal storage.

on_set(*row*, *value*)
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** (*Table*) – The row this value is being set on.
- **value** (`Any`) – The value being set.

Return type `Any`

store_value(*row*, *value*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** (*Table*) – The row to store in.
- **value** (Any) – 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.

sql()

Returns The str SQL name of this type.

classmethod create_default()

Creates the default object for this table in the event that a type is passed to a column, instead of an instance.

Return type *ColumnType*

ilike(*other*)

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** (str) – The other string to check.

Return type Union[*ILike*, *HackyILike*]

in_(**args*)

Returns an IN operator, checking if a value in this column is in a tuple of items.

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

Return type *In*

like(*other*)

Returns a LIKE operator, checking if this column is LIKE another string.

Parameters **other** (str) – The other string to check.

Return type *Like*

on_get(*row*)

Called when a value is retrieved from this column.

Parameters **row** (*Table*) – The row that is being retrieved.

Return type Any

Returns The value of the row's internal storage.

on_set(*row*, *value*)
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** (*Table*) – The row this value is being set on.
- **value** (Any) – The value being set.

Return type Any

schema()

Returns The library schema of this object.

If this is not defined, any arguments given to the type will not persist across schema generation.

store_value(*row*, *value*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** (*Table*) – The row to store in.
- **value** (Any) – The value to store in the row.

validate_set(*row*, *value*)
Validates that the item being set is valid. This is called by the default `on_set`.

Parameters

- **row** – The row being set.
- **value** (Any) – The value to set.

Returns A bool indicating if this is valid or not.

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

Represents a BOOL type.

sql()

Returns The str SQL name of this type.

validate_set(*row*, *value*)
Validates that the item being set is valid. This is called by the default `on_set`.

Parameters

- **row** (*Table*) – The row being set.
- **value** (Any) – The value to set.

Returns A bool indicating if this is valid or not.

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

Return type *ColumnType*

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

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

Return type *In*

on_get(*row*)
Called when a value is retrieved from this column.

Parameters **row** (*Table*) – The row that is being retrieved.

Return type Any

Returns The value of the row's internal storage.

on_set(*row*, *value*)
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** (*Table*) – The row this value is being set on.
- **value** (Any) – The value being set.

Return type Any

schema()

Return type str

Returns The library schema of this object.

If this is not defined, any arguments given to the type will not persist across schema generation.

store_value(*row*, *value*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** (*Table*) – The row to store in.
- **value** (Any) – 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)

sql()

Returns The str SQL name of this type.

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

on_set(*row*, *value*)
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** (Any) – The value being set.

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

Return type *ColumnType*

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

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

Return type *In*

on_get(*row*)
Called when a value is retrieved from this column.

Parameters **row** (*Table*) – The row that is being retrieved.

Return type Any

Returns The value of the row's internal storage.

schema()

Return type str

Returns The library schema of this object.

If this is not defined, any arguments given to the type will not persist across schema generation.

store_value(*row*, *value*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** (*Table*) – The row to store in.
- **value** (Any) – The value to store in the row.

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

Represents a SMALLINT type.

sql()

Returns The str SQL name of this type.

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

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

Return type *ColumnType*

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

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

Return type *In*

on_get(*row*)
Called when a value is retrieved from this column.

Parameters **row** (*Table*) – The row that is being retrieved.

Return type Any

Returns The value of the row's internal storage.

on_set(*row*, *value*)
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** (Any) – The value being set.

schema()

Return type str

Returns The library schema of this object.

If this is not defined, any arguments given to the type will not persist across schema generation.

store_value(*row*, *value*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** (*Table*) – The row to store in.
- **value** (Any) – The value to store in the row.

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

Represents a BIGINT type.

sql()

Returns The str SQL name of this type.

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

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

Return type *ColumnType*

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

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

Return type *In*

on_get(*row*)
Called when a value is retrieved from this column.

Parameters **row** (*Table*) – The row that is being retrieved.

Return type Any

Returns The value of the row's internal storage.

on_set(*row*, *value*)
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** (Any) – The value being set.

schema()

Return type str

Returns The library schema of this object.

If this is not defined, any arguments given to the type will not persist across schema generation.

store_value(*row*, *value*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** (*Table*) – The row to store in.
- **value** (Any) – The value to store in the row.

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

Represents a SERIAL type.

This type does not exist in SQLite; integer primary keys autoincrement already.

sql()

Returns The str SQL name of this type.

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

Return type *ColumnType*

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

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

Return type *In*

on_get(*row*)
Called when a value is retrieved from this column.

Parameters **row** (*Table*) – The row that is being retrieved.

Return type Any

Returns The value of the row's internal storage.

on_set(*row*, *value*)
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** (Any) – The value being set.

schema()

Return type str

Returns The library schema of this object.

If this is not defined, any arguments given to the type will not persist across schema generation.

store_value(*row*, *value*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** (*Table*) – The row to store in.
- **value** (Any) – The value to store in the row.

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

class asyncqlio.orm.schema.types.**BigSerial**
Bases: *asyncqlio.orm.schema.types.Serial*, *asyncqlio.orm.schema.types.BigInt*

Represents a BIGSERIAL type.

sql()

Returns The str SQL name of this type.

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

Return type *ColumnType*

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

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

Return type *In*

on_get(*row*)
Called when a value is retrieved from this column.

Parameters **row** (*Table*) – The row that is being retrieved.

Return type Any

Returns The value of the row's internal storage.

on_set(*row*, *value*)
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** (Any) – The value being set.

schema()

Return type str

Returns The library schema of this object.

If this is not defined, any arguments given to the type will not persist across schema generation.

store_value(*row*, *value*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** (*Table*) – The row to store in.
- **value** (Any) – The value to store in the row.

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

class asyncqlio.orm.schema.types.**SmallSerial**
Bases: *asyncqlio.orm.schema.types.Serial*, *asyncqlio.orm.schema.types.SmallInt*

Represents a SMALLSERIAL type.

sql()

Returns The str SQL name of this type.

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

Return type *ColumnType*

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

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

Return type *In*

on_get(*row*)
Called when a value is retrieved from this column.

Parameters **row** (*Table*) – The row that is being retrieved.

Return type Any

Returns The value of the row's internal storage.

on_set(*row*, *value*)
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** (Any) – The value being set.

schema()

Return type str

Returns The library schema of this object.

If this is not defined, any arguments given to the type will not persist across schema generation.

store_value(*row*, *value*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** (*Table*) – The row to store in.
- **value** (Any) – The value to store in the row.

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

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

Represents a REAL type.

sql()

Returns The str SQL name of this type.

validate_set(*row*, *value*)
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.

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

Return type *ColumnType*

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

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

Return type *In*

on_get(*row*)
Called when a value is retrieved from this column.

Parameters **row** (*Table*) – The row that is being retrieved.

Return type Any

Returns The value of the row's internal storage.

on_set(*row*, *value*)
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** (*Table*) – The row this value is being set on.
- **value** (Any) – The value being set.

Return type Any

schema()

Return type str

Returns The library schema of this object.

If this is not defined, any arguments given to the type will not persist across schema generation.

store_value(*row*, *value*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** (*Table*) – The row to store in.
- **value** (Any) – The value to store in the row.

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

Represents a TIMESTAMP type.

sql()

Returns The str SQL name of this type.

validate_set(*row*, *value*)
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.

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

Return type *ColumnType*

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

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

Return type *In*

on_get(*row*)
Called when a value is retrieved from this column.

Parameters **row** (*Table*) – The row that is being retrieved.

Return type Any

Returns The value of the row's internal storage.

on_set(*row*, *value*)
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** (*Table*) – The row this value is being set on.
- **value** (Any) – The value being set.

Return type Any

schema()

Return type str

Returns The library schema of this object.

If this is not defined, any arguments given to the type will not persist across schema generation.

store_value(*row*, *value*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** (*Table*) – The row to store in.
- **value** (Any) – The value to store in the row.

class asyncqlio.orm.schema.types.**Numeric**(*precision=10*, *scale=0*, *asdecimal=True*)
Bases: *asyncqlio.orm.schema.types.ColumnType*

Represents a NUMERIC type.

Note: *Numeric* type columns are similar to *Real* objects, except Numeric represents an explicit known value (DECIMAL, NUMERIC, DOUBLE, etc) and not a floating point type like REAL, or FLOAT. Use *Real* for floating points.

Parameters

- **precision** (int) – Total number of digits stored, excluding the the decimal.
- **scale** (int) – Number of digits to be stored after the decimal point.
- **asdecimal** (bool) – Set value as Python decimal.Decimal, floats are used when false.

sql()

Returns The str SQL name of this type.

schema()

Returns The library schema of this object.

If this is not defined, any arguments given to the type will not persist across schema generation.

on_get(*row*)
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*, *value*)
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** (Any) – The value being set.

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

Return type *ColumnType*

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

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

Return type *In*

store_value(*row*, *value*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** (*Table*) – The row to store in.
- **value** (Any) – The value to store in the row.

validate_set(*row*, *value*)
Validates that the item being set is valid. This is called by the default `on_set`.

Parameters

- **row** (*Table*) – The row being set.
- **value** (Any) – The value to set.

Return type bool

Returns A bool indicating if this is valid or not.

1.3.3 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")
```

These row objects have their column values accessible via attribute access:

```
print(row.id)  # prints 1
```

Row objects also produced from queries, and act exactly the same.

1.4 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.

1.4.1 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:
    ...
```

1.4.2 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
```

1.4.3 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)
```

1.4.4 Querying with the Session

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

class asyncqlio.orm.session.**Session**(*bind*, ***kwargs*)

Bases: *asyncqlio.orm.session.SessionBase*

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** (*DatabaseInterface*) – The *DatabaseInterface* instance we are bound to.

select

Creates a new SELECT query that can be built upon.

Return type *SelectQuery*

Returns A new *SelectQuery*.

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

Return type *InsertQuery*

Returns A new *InsertQuery*.

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

Return type *BulkUpdateQuery*

Returns A new *BulkUpdateQuery*.

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

Return type *BulkDeleteQuery*

Returns A new *BulkDeleteQuery*.

coroutine add(*self*, *row*)
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** (*Table*) – The *Table* instance object to add to the transaction.

Return type *Table*

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

close(*, *has_error=False*)
Closes the current session.

Parameters **has_error** (bool) – If this session had an error. Internal usage only.

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

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

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

Return type *SessionBase*

cursor(*sql*, *params=None*)
Executes SQL inside the current session, and returns a new BaseResultSet.

Parameters

- **sql** (str) – The SQL to execute.
- **params** (Union[Mapping[str, Any], Iterable[Any], None]) – The parameters to use inside the query.

Return type *BaseResultSet*

delete_now(*row*)
Deletes a row NOW.

Return type *Table*

execute(*sql*, *params=None*)
Executes SQL inside the current session.

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

Parameters

- **sql** (str) – The SQL to execute.
- **params** (Union[Mapping[str, Any], Iterable[Any], None]) – The parameters to use inside the query.

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

Return type Mapping[~KT, +VT_co]

insert_now(*row*)
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** (*Table*) – The *Table* instance to insert.

Return type Any

Returns The row, with primary key included.

coroutine merge(*self*, *row*)
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** (*Table*) – The *Table* instance to merge.

Return type *Table*

Returns The *Table* instance once updated.

coroutine remove(*self*, *row*)
Removes a row from the database.

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

Return type *Table*

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

Parameters **checkpoint** (Optional[str]) – The checkpoint to roll back to, if applicable.

Return type *SessionBase*

coroutine run_delete_query(*self*, *query*)
Executes a delete query.

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

coroutine run_insert_query(*self*, *query*)
Executes an insert query.

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

Returns The list of rows that were inserted.

coroutine run_select_query(*self*, *query*)
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** (*SelectQuery*) – The *SelectQuery* to use.

Returns A `_ResultGenerator` for this query.

coroutine run_update_query(*self*, *query*)
Executes an update query.

Parameters **query** (*BaseQuery*) – The *RowUpdateQuery* or *BulkUpdateQuery* to execute.

coroutine start(*self*)
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()  # or get_ddl_session etc
await sess.start()
```

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

Return type *SessionBase*

coroutine truncate(*self*, *table*, *, *cascade=False*)
Truncates a table.

Parameters

- **table** (`Type`[*Table*]) – The table to truncate.
- **cascade** (`bool`) – If this truncate should cascade to other tables.

update_now(*row*)
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** (*Table*) – The *Table* instance to update.

Return type *Table*

Returns The *Table* instance that was updated.

1.5 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
```

1.6 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*)
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.

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

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

Parameters **name** (`str`) – The name of the savepoint to create.

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

Parameters **name** (`str`) – The name of the savepoint to release.

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

coroutine close(*self*, *, *has_error=False*)
Called at the end of a transaction to cleanup. The connection will be released if there's no error; otherwise it will be closed.

Parameters **has_error** (`bool`) – If the transaction has an error.

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

coroutine cursor(*self*, *sql*, *params=None*)
Executes SQL and returns a database cursor for the rows.

Parameters

- **sql** (str) – The SQL statement to execute.
- **params** (Optional[Iterable[+T_co]]) – Any parameters to pass to the query.

Return type *BaseResultSet*

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

coroutine execute(*self*, *sql*, *params=None*)
Executes SQL in the current transaction.

Parameters

- **sql** (str) – The SQL statement to execute.
- **params** (Optional[Iterable[+T_co]]) – Any parameters to pass to the query.

coroutine rollback(*self*, *checkpoint=None*)
Rolls back the transaction.

Parameters **checkpoint** (Optional[str]) – If provided, the checkpoint to rollback to. Otherwise, the entire transaction will be rolled back.

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

Return type Iterable[str]

Returns An iterable of keys that this query contained.

coroutine close()
Closes this result set.

coroutine fetch_many(*self*, *n*)
Fetches the **next N rows** in this query.

Parameters **n** (int) – The number of rows to fetch.

Return type DictRow

coroutine fetch_row(*self*)
Fetches the **next row** in this query.

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

Return type DictRow

coroutine flatten(*self*)
Flattens this ResultSet.

Return type List[DictRow]

Returns A list of DictRow objects.

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[, connector])	The "database interface" to your database.

class asyncqlio.db.**DatabaseInterface**(*dsn*, *connector=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)
# then connect
await my_database.connect()
```

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

dialect = None
The current Dialect instance.

connector = None
The current connector instance.

connected
Checks if this DB is connected.

bind_tables(*md*)
Binds tables to this DB instance.

emit_param(*name=None*)
Emits a param in the format that the DB driver specifies.

Parameters **name** (Optional[str]) – The name to use. If this is None, a name will automatically be used, and no name param will be returned.

Return type Union[Tuple[str, str], str]

Returns The emitted param, and the name of the param emitted.

get_transaction(***kwargs*)
Gets a low-level *BaseTransaction*.

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

Return type *BaseTransaction*

get_session(***kwargs*)
Gets a new *Session* bound to this instance.

Return type *Session*

get_ddl_session(***kwargs*)
Gets a new DDLSession bound to this instance.

Return type DDLSession

coroutine close()
Closes the current database interface.

coroutine connect(*self*, ***kwargs*)
Connects the interface to the database server.

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

Return type BaseConnector

Returns The BaseConnector established.

coroutine get_db_server_version(*self*)
Gets the version of the DB server.

Return type str

2.1.2 asyncqlio.orm

The core code for the ORM.

schema	Code for ORM schema objects.
ddl	Data Domain Language helpers.
query	Classes for query objects.
session	Classes for session objects.
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	Table objects.
column	Classes for column objects.
index	Represents
relationship	Relationship objects.
types	Column types.
decorators	Decorator helpers for tables.

asyncqlio.orm.schema.table

Table objects.

Functions

table_base([name, meta])	Gets a new base object to use for OO-style tables.

Classes

AliasedTable(alias_name, table)	Represents an “aliased table”.
PrimaryKey(*cols)	Represents the primary key of a table.
Table(**kwargs)	The “base” class for all tables.
TableMeta(tblname, tblbases, class_body[, . . .])	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

Return type *DatabaseInterface*

Returns The *DatabaseInterface* bound to this metadata.

register_table(*tbl*, *, *autosetup_tables=False*)
Registers a new table object.

Parameters

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

Return type *TableMeta*

get_table(*table_name*)
Gets a table from the current metadata.

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

Return type Type[*Table*]

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.

generate_primary_key_indexes()
Generates an index for the primary key of each table, if the dialect creates one.

New in version 0.2.0.

generate_unique_column_indexes()
Generates an index for columns marked as unique in each table, if the dialect creates them.

New in version 0.2.0.

class asyncqlio.orm.schema.table.**TableMeta**(*tblname*, *tblbases*, *class_body*, *register=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** (bool) – Should this table be registered in the TableMetadata?
- **table_name** – The name for this table.

metadata = None
The *TableMetadata* for this table.

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.

Return type *PrimaryKey*

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.

classmethod iter_relationships()

Return type Generator[*Relationship*, None, None]

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

Changed in version 0.2.0: Moved from *TableMeta* to *Table* as a classmethod.

classmethod iter_columns()

Return type Generator[*Column*, None, None]

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

Changed in version 0.2.0: Moved from *TableMeta* to *Table* as a classmethod.

classmethod iter_indexes()

Return type Generator[*Index*, None, None]

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

Changed in version 0.2.0: Moved from *TableMeta* to *Table* as a classmethod.

classmethod explicit_indexes(*cls*)

Return type Generator[*Index*, None, None]

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

Only manually added indexes are yielded from this generator; that is, it ignores primary key indexes, unique column indexes, relationship indexes, etc

Changed in version 0.2.0: Moved from *TableMeta* to *Table* as a classmethod.

classmethod get_column(*column_name*)
Gets a column by name.

Parameters **column_name** (str) – The column name to lookup.

This can be one of the following:

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

Return type Optional[*Column*]

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

Changed in version 0.2.0: Moved from *TableMeta* to *Table* as a classmethod.

classmethod get_relationship(*relationship_name*)
Gets a relationship by name.

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

Return type Optional[*Relationship*]

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

Changed in version 0.2.0: Moved from *TableMeta* to *Table* as a classmethod.

classmethod get_index(*index_name*)
Gets an index by name.

Parameters **index_name** – The name of the index to get.

Return type Optional[*Index*]

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

Changed in version 0.2.0: Moved from *TableMeta* to *Table* as a classmethod.

primary_key
Gets the primary key for this row.

If this table only has one primary key column, this property will be a single value. If this table has multiple columns in a primary key, this property will be a tuple.

Return type Union[Any, Iterable[Any]]

get_column_value(*column*, *return_default=True*)
Gets the value from the specified column in this row.

> **Warning:** This method should not be used by user code; it is for types to interface with only.

Parameters

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

store_column_value(*column*, *value*, ***, *track_history=True*)
Updates the value of a column in this row. This will also update the history of the value, if applicable.

> **Warning:** This method should not be used by user code; it is for types to interface with only.

Parameters

- **column** (*Column*) – The column to store.
- **value** (Any) – The value to store in the column.
- **track_history** (bool) – Should history be tracked? Only false if creating a row from a data source.

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

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

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

Parameters **include_attrs** (bool) – Should this include row_attrs?

Return type dict

classmethod generate_schema(*fp=None*)
Generates a Python class body that corresponds to the current DB schema.

Return type str

classmethod create(*cls*, ***, *if_not_exists=True*)
Creates a table with this schema in the database.

classmethod drop(*cls*, ***, *cascade=False*, *if_exists=True*)
Drops this table, or a table with the same name, from the database.

Parameters

- **cascade** (bool) – If this drop should cascade.
- **if_exists** (bool) – If we should only attempt to drop tables that exist.

classmethod get(*cls*, **conditions*)
Gets a single row of this table from the database.

> **Warning:** The resulting row will not be bound to a session.

Parameters **conditions** – The conditions to filter on.

Return type *Table*

Returns A new *Table* instance that was found in the database.

coroutine truncate(***, *cascade=False*)
Truncates this table.

Parameters **cascade** (bool) – If this truncation should cascade to other tables.

Changed in version 0.2.0: Moved from *TableMeta* to *Table* as a classmethod.

asyncqlio.orm.schema.table.**table_base**(*name='Table'*, *meta=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.metadata)

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

Parameters

- **name** (str) – The name of the new class to produce. By default, it is Table.
- **meta** (Optional[*TableMetadata*]) – 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*, *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** (str) – The name of the alias for this table.
- **table** (Type[*Table*]) – The *TableMeta* used to alias this table.

get_column(*column_name*)
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.

Return type *Column*

class asyncqlio.orm.schema.table.**PrimaryKey**(**cols*)
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.

index_name = None
The index name of this primary key, if any

asyncqlio.orm.schema.column

Classes for column objects.

Classes

AliasedColumn(alias_table, column)	Represents a column on an aliased table.
Column(type_, *[, primary_key, nullable, ...])	Represents a column in a table in a database.

class asyncqlio.orm.schema.column.**AliasedColumn**(*alias_table*, *column*)
Bases: object

Represents a column on an aliased table.

Parameters

- **alias_table** (*AliasedTable*) – The alias table this column is a member of.

- **column** (*Column*) – The Column object this aliased column proxies.

class asyncqlio.orm.schema.column.**Column**(*type_*, ***, *primary_key=False*, *nullable=False*, *default=<object object>*, *index=True*, *unique=False*, *foreign_key=None*, *table=None*)

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** (Union[*ColumnType*, Type[*ColumnType*]]) – The *ColumnType* that represents the type of this column.
- **primary_key** (bool) – Is this column the table's Primary Key (the unique identifier that identifies each row)?
- **nullable** (bool) – Can this column be NULL?
- **default** (Any) – The client-side default for this column. If no value is provided when inserting, this value will automatically be added to the insert query.
- **index** (bool) – Should this column be indexed?
- **unique** (bool) – Is this column unique?
- **foreign_key** (Optional[*ForeignKey*]) – 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.

indexed = None
If this Column is indexed.

unique = None
If this Column is unique.

foreign_key = None
The foreign key associated with this column.

table_name
The name of this column's table.

Return type str

autoincrement
Whether this column is set to autoincrement.

Return type bool

classmethod with_name(*name*, **args*, ***kwargs*)
Creates this column with a name already set.

Return type *Column*

get_ddl_sql()
Gets the DDL SQL for this column.

Return type str

generate_schema(*fp=None*)
Generates the library schema for this column.

Return type str

eq(*other*)
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.

Return type *Eq*

ne(*other*)
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.

Return type *NEq*

asc()
Returns the ascending sorter operator for this column.

Return type *AscSorter*

desc()
Returns the descending sorter operator for this column.

Return type *DescSorter*

set(*value*)
Sets this column in a bulk update.

Return type *ValueSetter*

incr(*value*)
Increments this column in a bulk update.

Return type *IncrementSetter*

decr(*value*)
Decrements this column in a bulk update.

Return type *DecrementSetter*

quoted_fullname_with_table(*table*)
Gets the quoted fullname with a table. This is used for columns with alias tables.

Parameters **table** (*TableMeta*) – The *Table* or *AliasedTable* to use.

Return type str

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

Return type *Column*

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

alias_name(*table=None*, *quoted=False*)
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** (bool) – Should the name be quoted?

Return type str

Returns A str representing the alias name.

asyncqlio.orm.schema.index

Represents

Classes

Index(*columns[, unique, table])	Represents an index in a table in a database.

class asyncqlio.orm.schema.index.**Index**(**columns*, *unique=False*, *table=None*)
Bases: object

Represents an index in a table in a database.

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

    # make an index on the name column
    # by specifying it as the column
    name_index = Index(name)
```

New in version 0.2.0.

Parameters

- **columns** (Union[*Column*, str]) – The *Column* objects that this index is on.
- **unique** (bool) – Is this index unique?
- **table** (Optional[*Table*]) – The *Table* for this index. Can be None if the index is a member of a table.

get_column_names()

Return type Generator[str, None, None]

Returns A generator that yields the names of the columns for this index.

table_name
The name of this index's table.

Return type str

quoted_name
Gets the quoted name for this Index.

This returns the column name in "index" format.

quoted_fullname
Gets the full quoted name for this index.

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

classmethod with_name(*name*, **args*, ***kwargs*)
Creates this column with a name and, optionally, table name already set.

Return type *Index*

get_ddl_sql()
Gets the DDL SQL for this index.

Return type str

generate_schema(*fp*)
Generates the library schema for this index.

Return type str

asyncqlio.orm.schema.relationship

Relationship objects.

Classes

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

class asyncqlio.orm.schema.relationship.**ForeignKey**(*foreign_column*)

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** (Union[*Column*, str]) – 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.

get_ddl_sql(*name=None*)

Generates SQL to add this foreign key as a named constraint.

Parameters **name** (Optional[str]) –

Return type str

generate_schema(*fp=None*)

Generates a library schema for this foreign key.

Return type str

class asyncqlio.orm.schema.relationship.**Relationship**(*left*, *right*, *, *load='select'*, *use_iter=True*, *back_ref=None*, *table_alias=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)
```

(continues on next page)

(continued from previous page)

```
    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** (Union[*Column*, str]) – The left-hand column (the Column on this table) in this relationship.
- **right** (Union[*Column*, str]) – The right-hand column (the Column on the foreign table) in this relationship.
- **load** (str) – 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** (bool) – 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** (Optional[str]) – 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** (Optional[str]) – 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.

Return type *Column*

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

Return type Tuple[*Column*, *Column*]

get_instance(*row*, *session*)
Gets a new "relationship" instance.

generate_schema(*fp=None*)
Generates the library schema for this relationship.

Return type str

class asyncqlio.orm.schema.relationship.**BaseLoadedRelationship**(*rel*, *row*, *session*)
Bases: object

Provides some common methods for specific relationship type subclasses.

Parameters

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

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

coroutine add(*self*, *row*)
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** (*Table*) – The TableRow object to add to this relationship.

coroutine remove(*self*, *row*)
Removes a row from this query.

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

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

class asyncqlio.orm.schema.relationship.**SelectLoadedRelationship**(*rel*, *row*, *session*)

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

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

Parameters

- **rel** (*Relationship*) – The *Relationship* that lies underneath this object.
- **row** (*Table*) – 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):
    ...
```

Return type *SelectQuery*

coroutine add(*self*, *row*)

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** (*Table*) – The TableRow object to add to this relationship.

coroutine remove(*self*, *row*)

Removes a row from this query.

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

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

set_rows(*rows*)

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*, *row*, *session*)

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

Represents a join-loaded one to many relationship.

Parameters

- **rel** (*Relationship*) – The *Relationship* that lies underneath this object.

- **row** (*Table*) – The TableRow this is being loaded from.
- **session** – The *Session* this object is attached to.

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

coroutine add(*self*, *row*)
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** (*Table*) – The TableRow object to add to this relationship.

coroutine remove(*self*, *row*)
Removes a row from this query.

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

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

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

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

Parameters

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

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

add(*row*)
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** (*Table*) – The TableRow object to add to this relationship.

remove(*row*)
Removes a row from this query.

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

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

coroutine set (*self*, *row*)
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** (*Table*) – The TableRow to set.

asyncqlio.orm.schema.types

Column types.

Classes

BigInt()	Represents a BIGINT type.
BigSerial()	Represents a BIGSERIAL type.
Boolean()	Represents a BOOL type.
ColumnType()	Implements some underlying mechanisms for a *Column*.
Integer()	Represents an INTEGER type.
Numeric([precision, scale, asdecimal])	Represents a NUMERIC type.
Real()	Represents a REAL type.
Serial()	Represents a SERIAL type.
SmallInt()	Represents a SMALLINT type.
SmallSerial()	Represents a SMALLSERIAL type.
String([size])	Represents a VARCHAR() type.
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()

Return type `str`

Returns The str SQL name of this type.

schema()

Return type `str`

Returns The library schema of this object.

If this is not defined, any arguments given to the type will not persist across schema generation.

validate_set(*row*, *value*)
Validates that the item being set is valid. This is called by the default `on_set`.

Parameters

- **row** (*Table*) – The row being set.
- **value** (`Any`) – The value to set.

Return type `bool`

Returns A bool indicating if this is valid or not.

store_value(*row*, *value*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** (*Table*) – The row to store in.
- **value** (Any) – The value to store in the row.

on_set(*row*, *value*)
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** (*Table*) – The row this value is being set on.
- **value** (Any) – The value being set.

Return type Any

on_get(*row*)
Called when a value is retrieved from this column.

Parameters **row** (*Table*) – The row that is being retrieved.

Return type Any

Returns The value of the row's internal storage.

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

Return type *ColumnType*

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

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

Return type *In*

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

Represents a VARCHAR() type.

size = None
The max size of this String.

sql()

Returns The str SQL name of this type.

schema()

Returns The library schema of this object.

If this is not defined, any arguments given to the type will not persist across schema generation.

validate_set(*row*, *value*)
Validates that the item being set is valid. This is called by the default on_set.

Parameters

- **row** – The row being set.
- **value** (Any) – The value to set.

Returns A bool indicating if this is valid or not.

like(*other*)
Returns a LIKE operator, checking if this column is LIKE another string.

Parameters **other** (str) – The other string to check.

Return type *Like*

ilike(*other*)
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** (str) – The other string to check.

Return type Union[*ILike*, *HackyILike*]

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

Return type *ColumnType*

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

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

Return type *In*

on_get(*row*)
Called when a value is retrieved from this column.

Parameters **row** (*Table*) – The row that is being retrieved.

Return type Any

Returns The value of the row's internal storage.

on_set(*row*, *value*)
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** (*Table*) – The row this value is being set on.
- **value** (Any) – The value being set.

Return type Any

store_value(*row*, *value*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** (*Table*) – The row to store in.
- **value** (Any) – 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.

sql()

Returns The str SQL name of this type.

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

Return type *ColumnType*

ilike(*other*)
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** (str) – The other string to check.

Return type Union[*ILike*, *HackyILike*]

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

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

Return type *In*

like(*other*)
Returns a LIKE operator, checking if this column is LIKE another string.

Parameters **other** (str) – The other string to check.

Return type *Like*

on_get(*row*)
Called when a value is retrieved from this column.

Parameters **row** (*Table*) – The row that is being retrieved.

Return type Any

Returns The value of the row's internal storage.

on_set(*row*, *value*)
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** (*Table*) – The row this value is being set on.
- **value** (Any) – The value being set.

Return type Any

schema()

Returns The library schema of this object.

If this is not defined, any arguments given to the type will not persist across schema generation.

store_value(*row*, *value*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** (*Table*) – The row to store in.
- **value** (Any) – The value to store in the row.

validate_set(*row*, *value*)
Validates that the item being set is valid. This is called by the default `on_set`.

Parameters

- **row** – The row being set.
- **value** (Any) – The value to set.

Returns A bool indicating if this is valid or not.

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

Represents a BOOL type.

sql()

Returns The str SQL name of this type.

validate_set(*row*, *value*)
Validates that the item being set is valid. This is called by the default `on_set`.

Parameters

- **row** (*Table*) – The row being set.
- **value** (Any) – The value to set.

Returns A bool indicating if this is valid or not.

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

Return type *ColumnType*

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

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

Return type *In*

on_get(*row*)
Called when a value is retrieved from this column.

Parameters **row** (*Table*) – The row that is being retrieved.

Return type Any

Returns The value of the row's internal storage.

on_set(*row*, *value*)
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** (*Table*) – The row this value is being set on.
- **value** (Any) – The value being set.

Return type Any

schema()

Return type str

Returns The library schema of this object.

If this is not defined, any arguments given to the type will not persist across schema generation.

store_value(*row*, *value*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** (*Table*) – The row to store in.
- **value** (Any) – 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)

sql()

Returns The str SQL name of this type.

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

on_set(*row*, *value*)
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** (Any) – The value being set.

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

Return type *ColumnType*

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

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

Return type *In*

on_get(*row*)
Called when a value is retrieved from this column.

Parameters **row** (*Table*) – The row that is being retrieved.

Return type Any

Returns The value of the row's internal storage.

schema()

Return type str

Returns The library schema of this object.

If this is not defined, any arguments given to the type will not persist across schema generation.

store_value(*row*, *value*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** (*Table*) – The row to store in.
- **value** (Any) – The value to store in the row.

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

Represents a SMALLINT type.

sql()

Returns The str SQL name of this type.

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

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

Return type *ColumnType*

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

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

Return type *In*

on_get(*row*)
Called when a value is retrieved from this column.

Parameters **row** (*Table*) – The row that is being retrieved.

Return type Any

Returns The value of the row's internal storage.

on_set(*row*, *value*)
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** (Any) – The value being set.

schema()

Return type str

Returns The library schema of this object.

If this is not defined, any arguments given to the type will not persist across schema generation.

store_value(*row*, *value*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** (*Table*) – The row to store in.
- **value** (Any) – The value to store in the row.

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

Represents a BIGINT type.

sql()

Returns The str SQL name of this type.

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

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

Return type *ColumnType*

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

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

Return type *In*

on_get(*row*)
Called when a value is retrieved from this column.

Parameters **row** (*Table*) – The row that is being retrieved.

Return type Any

Returns The value of the row's internal storage.

on_set(*row*, *value*)
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** (Any) – The value being set.

schema()

Return type str

Returns The library schema of this object.

If this is not defined, any arguments given to the type will not persist across schema generation.

store_value(*row*, *value*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** (*Table*) – The row to store in.
- **value** (Any) – The value to store in the row.

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

Represents a SERIAL type.

This type does not exist in SQLite; integer primary keys autoincrement already.

sql()

Returns The str SQL name of this type.

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

Return type *ColumnType*

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

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

Return type *In*

on_get(*row*)
Called when a value is retrieved from this column.

Parameters **row** (*Table*) – The row that is being retrieved.

Return type Any

Returns The value of the row's internal storage.

on_set(*row*, *value*)
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** (Any) – The value being set.

schema()

Return type str

Returns The library schema of this object.

If this is not defined, any arguments given to the type will not persist across schema generation.

store_value(*row*, *value*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** (*Table*) – The row to store in.
- **value** (Any) – The value to store in the row.

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

class asyncqlio.orm.schema.types.**BigSerial**
Bases: *asyncqlio.orm.schema.types.Serial*, *asyncqlio.orm.schema.types.BigInt*

Represents a BIGSERIAL type.

sql()

Returns The str SQL name of this type.

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

Return type *ColumnType*

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

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

Return type *In*

on_get(*row*)
Called when a value is retrieved from this column.

Parameters **row** (*Table*) – The row that is being retrieved.

Return type Any

Returns The value of the row's internal storage.

on_set(*row*, *value*)
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** (Any) – The value being set.

schema()

Return type str

Returns The library schema of this object.

If this is not defined, any arguments given to the type will not persist across schema generation.

store_value(*row*, *value*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** (*Table*) – The row to store in.
- **value** (Any) – The value to store in the row.

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

class asyncqlio.orm.schema.types.**SmallSerial**
Bases: *asyncqlio.orm.schema.types.Serial*, *asyncqlio.orm.schema.types.SmallInt*

Represents a SMALLSERIAL type.

sql()

Returns The str SQL name of this type.

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

Return type *ColumnType*

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

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

Return type *In*

on_get(*row*)
Called when a value is retrieved from this column.

Parameters **row** (*Table*) – The row that is being retrieved.

Return type Any

Returns The value of the row's internal storage.

on_set(*row*, *value*)
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** (Any) – The value being set.

schema()

Return type str

Returns The library schema of this object.

If this is not defined, any arguments given to the type will not persist across schema generation.

store_value(*row*, *value*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** (*Table*) – The row to store in.
- **value** (Any) – The value to store in the row.

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

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

Represents a REAL type.

sql()

Returns The str SQL name of this type.

validate_set(*row*, *value*)
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.

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

Return type *ColumnType*

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

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

Return type *In*

on_get(*row*)
Called when a value is retrieved from this column.

Parameters **row** (*Table*) – The row that is being retrieved.

Return type Any

Returns The value of the row's internal storage.

on_set(*row*, *value*)
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** (*Table*) – The row this value is being set on.
- **value** (Any) – The value being set.

Return type Any

schema()

Return type str

Returns The library schema of this object.

If this is not defined, any arguments given to the type will not persist across schema generation.

store_value(*row*, *value*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** (*Table*) – The row to store in.
- **value** (Any) – The value to store in the row.

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

Represents a TIMESTAMP type.

sql()

Returns The str SQL name of this type.

validate_set(*row*, *value*)
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.

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

Return type *ColumnType*

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

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

Return type *In*

on_get(*row*)
Called when a value is retrieved from this column.

Parameters **row** (*Table*) – The row that is being retrieved.

Return type Any

Returns The value of the row's internal storage.

on_set(*row*, *value*)
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** (*Table*) – The row this value is being set on.
- **value** (Any) – The value being set.

Return type Any

schema()

Return type str

Returns The library schema of this object.

If this is not defined, any arguments given to the type will not persist across schema generation.

store_value(*row*, *value*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** (*Table*) – The row to store in.
- **value** (Any) – The value to store in the row.

class asyncqlio.orm.schema.types.**Numeric**(*precision=10*, *scale=0*, *asdecimal=True*)
Bases: *asyncqlio.orm.schema.types.ColumnType*

Represents a NUMERIC type.

Note: *Numeric* type columns are similar to *Real* objects, except Numeric represents an explicit known value (DECIMAL, NUMERIC, DOUBLE, etc) and not a floating point type like REAL, or FLOAT. Use *Real* for floating points.

Parameters

- **precision** (int) – Total number of digits stored, excluding the the decimal.
- **scale** (int) – Number of digits to be stored after the decimal point.
- **asdecimal** (bool) – Set value as Python decimal.Decimal, floats are used when false.

sql()

Returns The str SQL name of this type.

schema()

Returns The library schema of this object.

If this is not defined, any arguments given to the type will not persist across schema generation.

on_get(*row*)
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*, *value*)
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** (Any) – The value being set.

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

Return type *ColumnType*

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

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

Return type *In*

store_value(*row*, *value*)
Stores a value in the row's storage table.

This is for internal usage only.

Parameters

- **row** (*Table*) – The row to store in.
- **value** (Any) – The value to store in the row.

validate_set(*row*, *value*)
Validates that the item being set is valid. This is called by the default `on_set`.

Parameters

- **row** (*Table*) – The row being set.
- **value** (Any) – The value to set.

Return type bool

Returns A bool indicating if this is valid or not.

asyncqlio.orm.schema.decorators

Decorator helpers for tables.

Functions

enforce_bound(func)	Enforces that a method on a *Table* cannot be used before the table is bound to database via :meth:.DatabaseInterface.bind_tables.

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

Enforces that a method on a *Table* cannot be used before the table is bound to database via :meth:.DatabaseInterface.bind_tables.

New in version 0.2.0.

asyncqlio.orm.ddl

Data Domain Language helpers.

asyncqlio.orm.query

Classes for query objects.

Classes

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

class asyncqlio.orm.query.**BaseQuery**(*sess*)

Bases: asyncqlio.meta.AsyncABC

A base query object.

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

generate_sql()

Generates the SQL for this query. :rtype: Tuple[str, Mapping[str, Any]] :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*)

Bases: collections.abc.AsyncIterator

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

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

coroutine flatten(*self*)
Flattens this query into a single list.

Return type List[*Table*]

class asyncqlio.orm.query.**SelectQuery**(*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()
Generates the SQL for this query.

Return type Tuple[str, dict]

map_columns(*results*)
Maps columns in a result row to a *Table* instance object.

Parameters **results** (Mapping[str, Any]) – A single row of results from the query cursor.

Return type *Table*

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

map_many(**rows*)
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*)
Sets the table this query is selecting from.

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

Return type *SelectQuery*

Returns This query.

where(**conditions*)
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** (*BaseOperator*) – The conditions to use for this WHERE clause.

Return type *SelectQuery*

Returns This query.

limit(*row_limit*)
Sets a limit of the number of rows that can be returned from this query.

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

Return type *SelectQuery*

Returns This query.

offset(*offset*)
Sets the offset of rows to start returning results from/

Parameters **offset** (*int*) – The row offset.

Return type *SelectQuery*

Returns This query.

order_by(**col*, *sort_order='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*)
Sets the table to query on.

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

Return type *SelectQuery*

Returns This query.

add_condition(*condition*)
Adds a condition to the query/

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

Return type *SelectQuery*

Returns This query.

coroutine all(*self*)
Gets all results that match from this query.

Return type *ResultGenerator*

Returns A *ResultGenerator* that can be iterated over.

coroutine first(*self*)
Gets the first result that matches from this query.

Return type *Table*

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

coroutine run()
Runs this query.

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

Represents an INSERT query.

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

rows(**rows*)
Adds a set of rows to the query.

Parameters **rows** (*Table*) – The rows to insert.

Return type *InsertQuery*

Returns This query.

add_row(*row*)
Adds a row to this query, allowing it to be executed later.

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

Return type *InsertQuery*

Returns This query.

on_conflict(**columns*)
Get an *UpsertQuery* to react upon a conflict.

Parameters **columns** (*Column*) – The *Column* objects upon which to check for a conflict.

Return type *UpsertQuery*

generate_sql()
Generates the SQL statements for this insert query.

Return type List[Tuple[str, tuple]]

Returns A list of two-item tuples to execute: - The SQL query+params to emit to actually insert the row

coroutine run(*self*)
Runs this query.

Return type List[*Table*]

Returns A list of inserted `md_table.Table`.

class asyncqlio.orm.query.**UpsertQuery**(*sess*, **columns*, *rows*)
Bases: *asyncqlio.orm.query.InsertQuery*

Represents an UPSERT query.

New in version 0.2.0.

Parameters

- **sess** (*Session*) – The *Session* this query is attached to.
- **column** – The *Column* objects on which the conflict might happen.
- **rows** (*Table*) – The *Table* objects that are to be added.

on_conflict(**columns*)
Add more conflict columns to this query.

Parameters **column** – The *Column* objects upon which to check for a conflict.

Return type *UpsertQuery*

update(**cols*)
Used to specify which *Column* objects to update on a conflict.

Parameters **cols** (*Column*) – The *Column* objects to update.

Return type *UpsertQuery*

nothing()
Specify that this query should do nothing if there's a conflict.

This is the default behavior.

Return type *UpsertQuery*

generate_sql()
Generates the SQL statements for this upsert query.

Return type List[Tuple[str, tuple]]

Returns A list of two-item tuples: - The SQL query to use - The params to use with the query

add_row(*row*)
Adds a row to this query, allowing it to be executed later.

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

Return type *InsertQuery*

Returns This query.

rows(**rows*)
Adds a set of rows to the query.

Parameters **rows** (*Table*) – The rows to insert.

Return type *InsertQuery*

Returns This query.

coroutine run(*self*)
Runs this query.

Return type List[*Table*]

Returns A list of inserted `md_table.Table`.

class asyncqlio.orm.query.**BulkQuery**(*sess*)
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*)
Sets the table for this query.

where(**conditions*)
Sets the conditions for this query.

set_table(*table*)
Sets a table on this query.

add_condition(*condition*)
Adds a condition to this query.

generate_sql()
Generates the SQL for this query. :rtype: `Tuple[str, Mapping[str, Any]]` :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*)
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=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*)
Adds a condition to this query.

coroutine run()
Runs this query.

set_table(*table*)
Sets a table on this query.

table(*table*)
Sets the table for this query.

where(**conditions*)
Sets the conditions for this query.

class asyncqlio.orm.query.**BulkDeleteQuery**(*sess*)
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()
```

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

add_condition(*condition*)
Adds a condition to this query.

coroutine run()
Runs this query.

set_table(*table*)
Sets a table on this query.

table(*table*)
Sets the table for this query.

where(**conditions*)
Sets the conditions for this query.

class asyncqlio.orm.query.**RowUpdateQuery**(*sess*)
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.

rows(**rows*)
Adds a set of rows to the query.

Parameters **rows** (*Table*) – The rows to insert.

Return type *RowUpdateQuery*

Returns This query.

add_row(*row*)
Adds a row to this query, allowing it to be executed later.

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

Return type *RowUpdateQuery*

Returns This query.

generate_sql()
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

Return type List[Tuple[str, tuple]]

coroutine run()
Executes this query.

class asyncqlio.orm.query.**RowDeleteQuery**(*sess*)
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*)
Adds a set of rows to the query.

Parameters **rows** (*Table*) – The rows to insert.

Return type *RowDeleteQuery*

Returns This query.

coroutine run()
Runs this query.

add_row(*row*)
Adds a row to this query.

Parameters **row** (*Table*) – The *Table* instance

generate_sql()
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

Return type List[Tuple[str, tuple]]

asyncqlio.orm.session

Classes for session objects.

Functions

enforce_open(func)

Classes

Session(bind, **kwargs)	Sessions act as a temporary window into the database.
SessionBase(bind, **kwargs)	A superclass for session-like objects.
SessionState	An enumeration.

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

An enumeration.

class asyncqlio.orm.session.**SessionBase**(*bind*, ***kwargs*)
Bases: object

A superclass for session-like objects.

Parameters **bind** (*DatabaseInterface*) – 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.

close(*, *has_error=False*)
Closes the current session.

Parameters **has_error** (bool) – If this session had an error. Internal usage only.

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

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

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

Return type *SessionBase*

cursor(*sql*, *params=None*)
Executes SQL inside the current session, and returns a new BaseResultSet.

Parameters

- **sql** (str) – The SQL to execute.
- **params** (Union[Mapping[str, Any], Iterable[Any], None]) – The parameters to use inside the query.

Return type *BaseResultSet*

execute(*sql*, *params=None*)
Executes SQL inside the current session.

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

Parameters

- **sql** (str) – The SQL to execute.
- **params** (Union[Mapping[str, Any], Iterable[Any], None]) – The parameters to use inside the query.

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

Return type Mapping[~KT, +VT_co]

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

Parameters **checkpoint** (Optional[str]) – The checkpoint to roll back to, if applicable.

Return type *SessionBase*

coroutine start(*self*)
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()  # or get_ddl_session etc
await sess.start()
```

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

Return type *SessionBase*

class asyncqlio.orm.session.**Session**(*bind*, ***kwargs*)
Bases: *asyncqlio.orm.session.SessionBase*

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** (*DatabaseInterface*) – The *DatabaseInterface* instance we are bound to.

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

Return type *SelectQuery*

Returns A new *SelectQuery*.

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

Return type *InsertQuery*

Returns A new *InsertQuery*.

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

Return type *BulkUpdateQuery*

Returns A new *BulkUpdateQuery*.

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

Return type *BulkDeleteQuery*

Returns A new *BulkDeleteQuery*.

coroutine add(*self*, *row*)
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** (*Table*) – The *Table* instance object to add to the transaction.

Return type *Table*

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

close(*, *has_error=False*)
Closes the current session.

Parameters **has_error** (bool) – If this session had an error. Internal usage only.

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

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

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

Return type *SessionBase*

cursor(*sql*, *params=None*)
Executes SQL inside the current session, and returns a new `BaseResultSet`.

Parameters

- **sql** (str) – The SQL to execute.
- **params** (Union[Mapping[str, Any], Iterable[Any], None]) – The parameters to use inside the query.

Return type *BaseResultSet*

delete_now(*row*)
Deletes a row NOW.

Return type *Table*

execute(*sql*, *params=None*)
Executes SQL inside the current session.

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

Parameters

- **sql** (str) – The SQL to execute.
- **params** (Union[Mapping[str, Any], Iterable[Any], None]) – The parameters to use inside the query.

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

Return type Mapping[~KT, +VT_co]

insert_now(*row*)
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** (*Table*) – The *Table* instance to insert.

Return type Any

Returns The row, with primary key included.

coroutine merge(*self*, *row*)
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** (*Table*) – The *Table* instance to merge.

Return type *Table*

Returns The *Table* instance once updated.

coroutine remove(*self*, *row*)
Removes a row from the database.

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

Return type *Table*

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

Parameters **checkpoint** (Optional[str]) – The checkpoint to roll back to, if applicable.

Return type *SessionBase*

coroutine run_delete_query(*self*, *query*)
Executes a delete query.

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

coroutine run_insert_query(*self*, *query*)
Executes an insert query.

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

Returns The list of rows that were inserted.

coroutine run_select_query(*self*, *query*)
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** (*SelectQuery*) – The *SelectQuery* to use.

Returns A `_ResultGenerator` for this query.

coroutine run_update_query(*self*, *query*)
Executes an update query.

Parameters **query** (*BaseQuery*) – The *RowUpdateQuery* or *BulkUpdateQuery* to execute.

coroutine start(*self*)
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()  # or get_ddl_session etc
await sess.start()
```

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

Return type *SessionBase*

coroutine truncate(*self*, *table*, *, *cascade=False*)
Truncates a table.

Parameters

- **table** (`Type`[*Table*]) – The table to truncate.
- **cascade** (`bool`) – If this truncate should cascade to other tables.

update_now(*row*)
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** (*Table*) – The *Table* instance to update.

Return type *Table*

Returns The *Table* instance that was updated.

asyncqlio.orm.inspection

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

Functions

get_pk(row[, as_tuple])	Gets the primary key for a Table row.
get_row_session(row)	Gets the *Session* associated with a `TableRow`.

`asyncqlio.orm.inspection.`**get_row_session**(*row*)
Gets the *Session* associated with a `TableRow`.

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

Return type *Session*

asyncqlio.orm.inspection.**get_pk**(*row*, *as_tuple=True*)

Gets the primary key for a Table row.

Parameters

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

asyncqlio.orm.operators

Classes for operators returned from queries.

Functions

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

Classes

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

class asyncqlio.orm.operators.**OperatorResponse**(*sql*, *parameters*)
Bases: object

A storage class for the generated SQL from an operator.

Parameters

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

asyncqlio.orm.operators.**requires_bop**(*func*)
A decorator that marks a magic method as requiring another BaseOperator.

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

Return type Callable[[*BaseOperator*, *BaseOperator*], Any]

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.

generate_sql(*emitter*)
Generates the SQL for an operator.

Parameters must be generated using the emitter callable.

Parameters **emitter** (Callable[[], Tuple[str, str]]) – A callable that can be used to generate param placeholders in a query.

Return type *OperatorResponse*

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*)
Bases: *asyncqlio.orm.operators.BaseOperator*

Represents an AND operator in a query.

This will join multiple other *BaseOperator* objects together.

generate_sql(*emitter*)
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.

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

Represents an OR operator in a query.

This will join multiple other *BaseOperator* objects together.

generate_sql(*emitter*)
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.

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.**Sorter**(**columns*)
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.

generate_sql(*emitter*)
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.

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.**AscSorter**(**columns*)
Bases: *asyncqlio.orm.operators.Sorter*

generate_sql(*emitter*)
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.

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.**DescSorter**(**columns*)
Bases: *asyncqlio.orm.operators.Sorter*

generate_sql(*emitter*)
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.

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.**ColumnValueMixin**(*column*, *value*)
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*, *value*)
Bases: *asyncqlio.orm.operators.BaseOperator*, *asyncqlio.orm.operators.ColumnValueMixin*

Represents a basic setting operation. Used for bulk queries.

set_operator

Return type str

Returns The "setting" operator to use when generating the SQL.

generate_sql(*emitter*)
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.

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.**ValueSetter**(*column*, *value*)
Bases: *asyncqlio.orm.operators.BasicSetter*

Represents a value setter (`col = 1`).

generate_sql(*emitter*)
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.

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.**IncrementSetter**(*column*, *value*)
Bases: *asyncqlio.orm.operators.BasicSetter*

Represents an increment setter. (`col = col + 1`)

generate_sql(*emitter*)
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.

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.**DecrementSetter**(*column*, *value*)
Bases: *asyncqlio.orm.operators.BasicSetter*

Represents a decrement setter.

generate_sql(*emitter*)
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.

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.**In**(*column*, *value*)
Bases: *asyncqlio.orm.operators.BaseOperator*, *asyncqlio.orm.operators.ColumnValueMixin*

generate_sql(*emitter*)
Generates the SQL for an operator.

Parameters must be generated using the emitter callable.

Parameters **emitter** (Callable[[str], str]) – A callable that can be used to generate param placeholders in a query.

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.**ComparisonOp**(*column*, *value*)
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.

generate_sql(*emitter*)
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.

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.**Eq**(*column*, *value*)
Bases: *asyncqlio.orm.operators.ComparisonOp*

Represents an equality operator.

generate_sql(*emitter*)
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.

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.**NEq**(*column*, *value*)
Bases: *asyncqlio.orm.operators.ComparisonOp*

Represents a non-equality operator.

generate_sql(*emitter*)
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.

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.**Lt**(*column*, *value*)
Bases: *asyncqlio.orm.operators.ComparisonOp*

Represents a less than operator.

generate_sql(*emitter*)
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.

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.**Gt**(*column*, *value*)
Bases: *asyncqlio.orm.operators.ComparisonOp*

Represents a more than operator.

generate_sql(*emitter*)
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.

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.**Lte**(*column*, *value*)
Bases: *asyncqlio.orm.operators.ComparisonOp*

Represents a less than or equals to operator.

generate_sql(*emitter*)
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.

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.**Gte**(*column*, *value*)
Bases: *asyncqlio.orm.operators.ComparisonOp*

Represents a more than or equals to operator.

generate_sql(*emitter*)
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.

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.**Like**(*column*, *value*)
Bases: *asyncqlio.orm.operators.ComparisonOp*

Represents a LIKE operator.

generate_sql(*emitter*)
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.

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.**ILike**(*column*, *value*)
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.

generate_sql(*emitter*)
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.

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.**HackyILike**(*column*, *value*)
Bases: *asyncqlio.orm.operators.BaseOperator*, *asyncqlio.orm.operators.ColumnValueMixin*

A "hacky" ILIKE operator for databases that do not support it.

generate_sql(*emitter*)
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.

Returns A *OperatorResponse* representing the result.

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

2.1.3 asyncqlio.backends

SQL driver backends for asyncqlio.

postgresql	PostgreSQL backends.
sqlite3	SQLite3 backends.
mysql	MySQL backends.

asyncqlio.backends.postgresql

PostgreSQL backends.

asyncpg	The asyncpg connector for PostgreSQL databases.

asyncqlio.backends.postgresql.asyncpg

The asyncpg connector for PostgreSQL databases.

Functions

get_param_query(sql, params)	Re-does a SQL query so that it uses asyncpg's special query format.

Classes

AsyncpgConnector(parsed)	A connector that uses the asyncpg library.
AsyncpgResultSet(cur)	
AsyncpgTransaction(conn)	A transaction that uses the asyncpg library.
CONNECTOR_TYPE	alias of *asyncqlio.backends.postgresql.asyncpg.AsyncpgConnector*

asyncqlio.backends.postgresql.asyncpg.**get_param_query**(*sql*, *params*)

Re-does a SQL query so that it uses asyncpg's special query format.

Parameters

- **sql** (str) – The SQL statement to use.
- **params** (dict) – The dict of parameters to use.

Return type Tuple[str, tuple]

Returns A two-item tuple of (new_query, arguments)

class asyncqlio.backends.postgresql.asyncpg.**AsyncpgResultSet**(*cur*)

Bases: *asyncqlio.backends.base.BaseResultSet*

keys

Return type Iterable[str]

Returns An iterable of keys that this query contained.

coroutine close()

Closes this result set.

coroutine fetch_many(*self*, *n*)

Fetches the **next N rows** in this query.

Parameters **n** (int) – The number of rows to fetch.

coroutine fetch_row()

Fetches the **next row** in this query.

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

coroutine flatten(*self*)
Flattens this ResultSet.

Return type List[DictRow]

Returns A list of DictRow objects.

class asyncqlio.backends.postgresql.asyncpg.**AsyncpgTransaction**(*conn*)
Bases: *asyncqlio.backends.base.BaseTransaction*

A transaction that uses the asyncpg library.

acquired_connection = None
The acquired connection from the connection pool.

transaction = None
The asyncpg internal transaction.

coroutine begin(***transaction_options*)
Begins the transaction.

coroutine close(*self*, *, *has_error=False*)
Called at the end of a transaction to cleanup. The connection will be released if there's no error; otherwise it will be closed.

Parameters **has_error** (bool) – If the transaction has an error.

coroutine commit()
Commits the transaction.

coroutine create_savepoint(*self*, *name*)
Creates a savepoint in the current transaction.

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

Parameters **name** (str) – The name of the savepoint to create.

coroutine cursor(*self*, *sql*, *params=None*)
Executes a SQL statement and returns a cursor to iterate over the rows of the result.

Return type *AsyncpgResultSet*

coroutine execute(*self*, *sql*, *params=None*)
Executes SQL inside the transaction.

Parameters

- **sql** (str) – The SQL to execute.
- **params** (Optional[Mapping[str, Any]]) – The parameters to excecute with.

coroutine release_savepoint(*self*, *name*)
Releases a savepoint in the current transaction.

Parameters **name** (str) – The name of the savepoint to release.

coroutine rollback(*self*, *checkpoint=None*)
Rolls back the transaction.

Parameters **checkpoint** (Optional[str]) – If provided, the checkpoint to rollback to. Otherwise, the entire transaction will be rolled back.

class asyncqlio.backends.postgresql.asyncpg.**AsyncpgConnector**(*parsed*)
Bases: asyncqlio.backends.base.BaseConnector

A connector that uses the asyncpg library.

pool = None
The asyncpg.pool.Pool connection pool.

emit_param(*name*)
Emits a parameter that can be used as a substitute during a query.

Parameters **name** (str) – The name of the parameter.

Return type str

Returns A string that represents the substitute to be placed in the query.

get_transaction()
Gets a new transaction object for this connection.

Return type *AsyncpgTransaction*

Returns A new *BaseTransaction* object attached to this connection.

coroutine close()
Closes the current Connector.

coroutine connect(*self*, *, *loop=None*)
Connects the current connector to the database server. This is called automatically by the :class:‘.DatabaseInterface

Return type BaseConnector

Returns The original BaseConnector instance.

coroutine get_db_server_version()
Gets the version of the DB server running.

asyncqlio.backends.postgresql.asyncpg.**CONNECTOR_TYPE**
alias of *asyncqlio.backends.postgresql.asyncpg.AsyncpgConnector*

Classes

PostgresqlDialect	The dialect for Postgres.

class asyncqlio.backends.postgresql.**PostgresqlDialect**
Bases: asyncqlio.backends.base.BaseDialect

The dialect for Postgres.

has_checkpoints
Returns True if this dialect can use transaction checkpoints.

has_serial
Returns True if this dialect can use the SERIAL datatype.

lastval_method
The last value method for a dialect. For example, in PostgreSQL this is LASTVAL();

has_returns
Returns True if this dialect has RETURNS.

has_ilike
Returns True if this dialect has ILIKE.

has_default
Returns True if this dialect has DEFAULT.

has_truncate
Returns TRUE if this dialect has TRUNCATE.

has_cascade
Returns True if this dialect has DROP TABLE ... CASCADE.

get_primary_key_index_name(*table_name*)
Get the name a dialect gives to a table's primary key index.

get_unique_column_index_name(*table_name*, *column_name*)
Get the name a dialect gives to a unique column index.

Parameters

- **table_name** – The name of the table to use.
- **column_name** – The name of the column to use.

get_column_sql(*table_name=None*, *, *emitter*)
Get a query to find information on all columns, optionally limiting by table.

Parameters

- **table_name** – The name of the table to use.
- **emitter** – The emitter to use.

get_index_sql(*table_name=None*, *, *emitter*)
Get a query to find information on all indexes, optionally limiting by table.

Parameters

- **table_name** – The name of the table to use.
- **emitter** – The emitter to use.

get_upsert_sql(*table_name*, *, *on_conflict_update=True*)
Get a formattable query and a set of required params to execute upsert-like functionality.

Parameters

- **table_name** – The name of the table to upsert into.
- **on_conflict_update** – If this is to update on conflict.

transform_rows_to_indexes(**rows*, *table_name=None*)
Transform appropriate database rows to Index objects.

Parameters **rows** – A list of `DictRow` objects returned from the database.

transform_columns_to_indexes(**rows*, *table_name*)
Transform appropriate database rows to Column objects.

Parameters

- **rows** (`DictRow`) – A list of `DictRow` objects returned from the database.
- **table_name** (`str`) – The name of the table being transformed.

asyncqlio.backends.sqlite3

SQLite3 backends.

sqlite3

Classes

Sqlite3Dialect	The dialect for SQLite3.

class asyncqlio.backends.sqlite3.**Sqlite3Dialect**
Bases: asyncqlio.backends.base.BaseDialect

The dialect for SQLite3.

has_checkpoints
Returns True if this dialect can use transaction checkpoints.

has_serial
Returns True if this dialect can use the SERIAL datatype.

lastval_method
The last value method for a dialect. For example, in PostgreSQL this is LASTVAL();

has_returns
Returns True if this dialect has RETURNS.

has_ilike
Returns True if this dialect has ILIKE.

has_default
Returns True if this dialect has DEFAULT.

has_truncate
Returns TRUE if this dialect has TRUNCATE.

has_cascade
Returns True if this dialect has DROP TABLE ... CASCADE.

get_primary_key_index_name(*table_name*)
Get the name a dialect gives to a table's primary key index.

get_unique_column_index_name(*table_name*, *column_name*)
Get the name a dialect gives to a unique column index.

Parameters

- **table_name** – The name of the table to use.
- **column_name** – The name of the column to use.

get_column_sql(*table_name=None*, *, *emitter*)
Get a query to find information on all columns, optionally limiting by table.

Parameters

- **table_name** – The name of the table to use.
- **emitter** – The emitter to use.

get_index_sql(*table_name=None*, *, *emitter*)
Get a query to find information on all indexes, optionally limiting by table.

Parameters

- **table_name** – The name of the table to use.
- **emitter** – The emitter to use.

get_upsert_sql(*table_name*, *, *on_conflict_update=True*)
Get a formattable query and a set of required params to execute upsert-like functionality.

Parameters

- **table_name** – The name of the table to upsert into.
- **on_conflict_update** – If this is to update on conflict.

transform_rows_to_indexes(**rows*, *table_name*)
Transform appropriate database rows to Index objects.

Parameters **rows** – A list of `DictRow` objects returned from the database.

transform_columns_to_indexes(**rows*, *table_name*)
Transform appropriate database rows to Column objects.

Parameters

- **rows** (`DictRow`) – A list of `DictRow` objects returned from the database.
- **table_name** (`str`) – The name of the table being transformed.

asyncqlio.backends.mysql

MySQL backends.

aiomysql	The aiomysql connector for MySQL/MariaDB databases.

asyncqlio.backends.mysql.aiomysql

The aiomysql connector for MySQL/MariaDB databases.

Classes

AiomysqlConnector(dsn)	A connector that uses the aiomysql library.
AiomysqlResultSet(cursor)	Represents a result set returned by the MySQL database.
AiomysqlTransaction(connector)	Represents a transaction for aiomysql.
CONNECTOR_TYPE	alias of *asyncqlio.backends.mysql.aiomysql.AiomysqlConnector*

class asyncqlio.backends.mysql.aiomysql.**AiomysqlResultSet**(*cursor*)
Bases: *asyncqlio.backends.base.BaseResultSet*

Represents a result set returned by the MySQL database.

keys

Returns An iterable of keys that this query contained.

coroutine close()
Closes this result set.

coroutine fetch_all()
Fetches ALL the rows.

coroutine fetch_many(*self*, *n*)
Fetches the next N rows.

coroutine fetch_row(*self*)
Fetches the next row in this result set.

Return type Dict[Any, Any]

coroutine flatten(*self*)
Flattens this ResultSet.

Return type List[DictRow]

Returns A list of DictRow objects.

class asyncqlio.backends.mysql.aiomysql.**AiomysqlTransaction**(*connector*)
Bases: *asyncqlio.backends.base.BaseTransaction*

Represents a transaction for aiomysql.

connection = None
The current acquired connection for this transaction.

coroutine cursor(*self*, *sql*, *params=None*)
Returns a *AiomysqlResultSet* for the specified SQL.

Return type *AiomysqlResultSet*

coroutine begin()
Begins the current transaction.

coroutine close(*self*, *, *has_error=False*)
Closes the current connection.

coroutine commit()
Commits the current transaction.

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

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

Parameters **name** (str) – The name of the savepoint to create.

coroutine execute(*self*, *sql*, *params=None*)
Executes some SQL in the current transaction.

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

Parameters **name** (str) – The name of the savepoint to release.

coroutine rollback(*self*, *checkpoint=None*)
Rolls back the current transaction.

Parameters **checkpoint** (Optional[str]) – Ignored.

class asyncqlio.backends.mysql.aiomysql.**AiomysqlConnector**(*dsn*)
Bases: asyncqlio.backends.base.BaseConnector

A connector that uses the aiomysql library.

pool = None
The current connection pool for this connector.

get_transaction()
Gets a new transaction object.

Return type *BaseTransaction*

emit_param(*name*)
Emits a parameter that can be used as a substitute during a query.

Parameters **name** (str) – The name of the parameter.

Return type str

Returns A string that represents the substitute to be placed in the query.

coroutine close(*self*, *forcefully=False*)
Closes this connector.

coroutine connect(*self*, *, *loop=None*)
Connects this connector.

Return type *AiomysqlConnector*

coroutine get_db_server_version()
Gets the version of the DB server running.

asyncqlio.backends.mysql.aiomysql.**CONNECTOR_TYPE**
alias of *asyncqlio.backends.mysql.aiomysql.AiomysqlConnector*

Classes

MysqlDialect	The dialect for MySQL.

class asyncqlio.backends.mysql.**MysqlDialect**
Bases: asyncqlio.backends.base.BaseDialect

The dialect for MySQL.

has_checkpoints
Returns True if this dialect can use transaction checkpoints.

has_serial
Returns True if this dialect can use the SERIAL datatype.

lastval_method
The last value method for a dialect. For example, in PostgreSQL this is LASTVAL();

has_returns
Returns True if this dialect has RETURNS.

has_default
Returns True if this dialect has DEFAULT.

has_ilike
Returns True if this dialect has ILIKE.

has_truncate
Returns TRUE if this dialect has TRUNCATE.

has_cascade
Returns True if this dialect has DROP TABLE ... CASCADE.

get_primary_key_index_name(*table*)
Get the name a dialect gives to a table's primary key index.

get_unique_column_index_name(*table_name*, *column_name*)
Get the name a dialect gives to a unique column index.

Parameters

- **table_name** – The name of the table to use.
- **column_name** – The name of the column to use.

get_column_sql(*table_name=None*, *, *emitter*)
Get a query to find information on all columns, optionally limiting by table.

Parameters

- **table_name** – The name of the table to use.
- **emitter** – The emitter to use.

get_index_sql(*table_name=None*, *, *emitter*)
Get a query to find information on all indexes, optionally limiting by table.

Parameters

- **table_name** – The name of the table to use.
- **emitter** – The emitter to use.

get_upsert_sql(*table_name*, *, *on_conflict_update=True*)
Get a formattable query and a set of required params to execute upsert-like functionality.

Parameters

- **table_name** – The name of the table to upsert into.
- **on_conflict_update** – If this is to update on conflict.

transform_rows_to_indexes(**rows*, *table_name=None*)
Transform appropriate database rows to Index objects.

Parameters **rows** – A list of `DictRow` objects returned from the database.

transform_columns_to_indexes(**rows*, *table_name*)
Transform appropriate database rows to Column objects.

Parameters

- **rows** (`DictRow`) – A list of `DictRow` objects returned from the database.
- **table_name** (`str`) – The name of the table being transformed.

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.
UnsupportedOperationException	Raised when an operation that the database driver doesn't support is attempted.

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.

exception asyncqlio.exc.**UnsupportedOperationException**
Bases: *asyncqlio.exc.DatabaseException*

Raised when an operation that the database driver doesn't support is attempted.

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)	Makes a proxy object for magic methods.
proxy_to_getattr(*magic_methods)	Proxies a method to to __getattr__ when it would not be normally proxied.
typeproperty(func)	Marks a function as a type property.

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	
TypeProperty(fget)	A property on a type.

asyncqlio.meta.**make_proxy**(*name*)
Makes a proxy object for magic methods.

asyncqlio.meta.**proxy_to_getattr**(**magic_methods*)
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** (str) – The magic methods to proxy to getattr.

class asyncqlio.meta.**TypeProperty**(*fget*)
Bases: object

A property on a type.

Parameters **fget** – The function to call on getting the property.

asyncqlio.meta.**typeproperty**(*func*)
Marks a function as a type property.

Return type *TypeProperty*

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.

```
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

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W