sqlbuilder Documentation

Release 0

Ivan Zakrevsky

Dec 24, 2017

Contents

1 Quick start **3**

2 Django integration **5**

3 Short manual for sqlbuilder.smartsql **7**
- 3.1 Table . . . 7
- 3.2 Field . . . 7
- 3.3 Table operators . . . 8
- 3.4 Condition operators . . . 9
- 3.5 Module sqlbuilder.smartsql.contrib.evaluate . . . 13
- 3.6 Module sqlbuilder.smartsql . . . 14
- 3.7 Query object . . . 14
  - 3.7.1 Subquery . . . 24
- 3.8 Implementation of execution . . . 25
- 3.9 Compilers . . . 26

4 Short manual for sqlbuilder.mini **27**

5 Indices and tables **31**

Python Module Index **33**

SmartSQL - lightweight Python sql builder, follows the KISS principle. Supports Python2 and Python3.

You can use SmartSQL separatelly, or with Django, or with super-lightweight Ascetic ORM, or with super-lightweight datamapper Openorm (miror) etc.

- Home Page: https://bitbucket.org/emacsway/sqlbuilder
- Docs: https://sqlbuilder.readthedocs.io/
- Browse source code (canonical repo): https://bitbucket.org/emacsway/sqlbuilder/src
- GitHub mirror: https://github.com/emacsway/sqlbuilder
- Get source code (canonical repo): `hg clone https://bitbucket.org/emacsway/sqlbuilder`
- Get source code (mirror): `git clone https://github.com/emacsway/sqlbuilder.git`
- PyPI: https://pypi.python.org/pypi/sqlbuilder

LICENSE:

- License is BSD

CHAPTER 1

Quick start

```
>>> from sqlbuilder.smartsql import Q, T, compile
>>> compile(Q().tables(
...     (T.book & T.author).on(T.book.author_id == T.author.id)
... ).fields(
...     T.book.title, T.author.first_name, T.author.last_name
... ).where(
...     (T.author.first_name != 'Tom') & (T.author.last_name != 'Smith')
... )[20:30])
('SELECT "book"."title", "author"."first_name", "author"."last_name" FROM "book"␣
↪INNER JOIN "author" ON ("book"."author_id" = "author"."id") WHERE "author"."first_
↪name" <> %s AND "author"."last_name" <> %s LIMIT %s OFFSET %s', ['Tom', 'Smith', 10,
↪ 20])
```

CHAPTER 2

Django integration

Simple add “django_sqlbuilder” to your INSTALLED_APPS.

```
>>> object_list = Book.s.q.tables(
...     (Book.s & Author.s).on(Book.s.author == Author.s.pk)
... ).where(
...     (Author.s.first_name != 'James') & (Author.s.last_name != 'Joyce')
... )[:10]
```

Contents:

Table of Contents

- *Lightweight Python SQLBuilder*
- *Quick start*
- *Django integration*
- *Short manual for sqlbuilder.smartsql*
  - *Table*
  - *Field*
  - *Table operators*
  - *Condition operators*
  - *Module sqlbuilder.smartsql.contrib.evaluate*
  - *Module sqlbuilder.smartsql*
  - *Query object*
    - *Subquery*
  - *Implementation of execution*

    - *Compilers*
- *Short manual for sqlbuilder.mini*
- *Indices and tables*

CHAPTER 3

Short manual for sqlbuilder.smartsql

3.1 Table

```
>>> from sqlbuilder.smartsql import T, Q

>>> T.book
<Table: "book", []>

>>> T.book.as_('a')
<TableAlias: "a", []>

>>> T.book__a  # Same as T.book.as_('a')
<TableAlias: "a", []>
```

Compiling instance of TableAlias depends on context of usage:

```
>>> ta = T.book.as_('a')
>>> ta
<TableAlias: "a", []>

>>> Q().tables(ta).fields(ta.id, ta.status).where(ta.status.in_(('new', 'approved')))
<Query: SELECT "a"."id", "a"."status" FROM "book" AS "a" WHERE "a"."status" IN (%s,
↪%s), ['new', 'approved']>
```

3.2 Field

Get field as table attribute:

```
>>> from sqlbuilder.smartsql import T, F, Q

>>> T.book.title
<Field: "book"."title", []>
```

```
>>> T.book.title.as_('a')
<Alias: "a", []>

>>> T.book.title__a  # Same as T.book.title.as_('a')
<Alias: "a", []>
```

What if field name is reserved word in table namespace?

```
>>> T.book.natural  # Will returns a bound method
<bound method Table.<lambda> of <Table: "book", []>>

>>> T.book['natural']  # Will returns Field()
<Field: "book"."natural", []>

>>> T.book.f.natural  # Also
<Field: "book"."natural", []>

>>> T.book.f['natural']  # Also
<Field: "book"."natural", []>

>>> T.book.f('natural')  # Also
<Field: "book"."natural", []>

>>> F('natural', T.book)  # Also
<Field: "book"."natural", []>
```

Get field as attribute of F class (Legacy way):

```
>>> F.book__title  # Same as T.book.title
<Field: "book"."title", []>

>>> F.book__title.as_('a')  # Same as T.book.title.as_('a')
<Alias: "a", []>

>>> F.book__title__a  # Same as T.book.title.as_('a')
<Alias: "a", []>
```

Compiling instance of Alias depends on context of usage:

```
>>> al = T.book.status.as_('a')
>>> al
<Alias: "a", []>

>>> Q().tables(T.book).fields(T.book.id, al).where(al.in_(('new', 'approved')))
<Query: SELECT "book"."id", "book"."status" AS "a" FROM "book" WHERE "a" IN (%s, %s),␣
↪['new', 'approved']>
```

3.3 Table operators

```
>>> (T.book & T.author).on(T.book.author_id == T.author.id)
<TableJoin: "book" INNER JOIN "author" ON ("book"."author_id" = "author"."id"), []>

>>> (T.book + T.author).on(T.book.author_id == T.author.id)
<TableJoin: "book" LEFT OUTER JOIN "author" ON ("book"."author_id" = "author"."id"),␣
↪[]>
```

```
>>> (T.book - T.author).on(T.book.author_id == T.author.id)
<TableJoin: "book" RIGHT OUTER JOIN "author" ON ("book"."author_id" = "author"."id"),␣
↪[]>

>>> (T.book | T.author).on(T.book.author_id == T.author.id)
<TableJoin: "book" FULL OUTER JOIN "author" ON ("book"."author_id" = "author"."id"),␣
↪[]>

>>> (T.book * T.author).on(T.book.author_id == T.author.id)
<TableJoin: "book" CROSS JOIN "author" ON ("book"."author_id" = "author"."id"), []>
```

Nested join is also supported:

```
>>> t1, t2, t3, t4 = T.t1, T.t2, T.t3, T.t4

>>> t1 + (t2 * t3 * t4)().on((t2.a == t1.a) & (t3.b == t1.b) & (t4.c == t1.c))
<TableJoin: "t1" LEFT OUTER JOIN ("t2" CROSS JOIN "t3" CROSS JOIN "t4") ON ("t2"."a"␣
↪= "t1"."a" AND "t3"."b" = "t1"."b" AND "t4"."c" = "t1"."c"), []>

>>> t1 + (t2 + t3).on((t2.b == t3.b) | t2.b.is_(None))()
<TableJoin: "t1" LEFT OUTER JOIN ("t2" LEFT OUTER JOIN "t3" ON ("t2"."b" = "t3"."b"␣
↪OR "t2"."b" IS NULL)), []>

>>> (t1 + t2.on(t1.a == t2.a))() + t3.on((t2.b == t3.b) | t2.b.is_(None))
<TableJoin: ("t1" LEFT OUTER JOIN "t2" ON ("t1"."a" = "t2"."a")) LEFT OUTER JOIN "t3"␣
↪ON ("t2"."b" = "t3"."b" OR "t2"."b" IS NULL), []>
```

3.4 Condition operators

```
>>> from sqlbuilder.smartsql import T, P
>>> tb = T.author

>>> tb.name == 'Tom'
<Eq: "author"."name" = %s, ['Tom']>

>>> tb.name != 'Tom'
<Ne: "author"."name" <> %s, ['Tom']>

>>> tb.counter + 1
<Add: "author"."counter" + %s, [1]>

>>> 1 + tb.counter
<Add: %s + "author"."counter", [1]>

>>> tb.counter - 1
<Sub: "author"."counter" - %s, [1]>

>>> 10 - tb.counter
<Sub: %s - "author"."counter", [10]>

>>> tb.counter * 2
<Mul: "author"."counter" * %s, [2]>

>>> 2 * tb.counter
```

```
<Mul: %s * "author"."counter", [2]>

>>> tb.counter / 2
<Div: "author"."counter" / %s, [2]>

>>> 10 / tb.counter
<Div: %s / "author"."counter", [10]>

>>> tb.is_staff & tb.is_admin
<And: "author"."is_staff" AND "author"."is_admin", []>

>>> tb.is_staff | tb.is_admin
<Or: "author"."is_staff" OR "author"."is_admin", []>

>>> tb.counter > 10
<Gt: "author"."counter" > %s, [10]>

>>> 10 > tb.counter
<Lt: "author"."counter" < %s, [10]>

>>> tb.counter >= 10
<Ge: "author"."counter" >= %s, [10]>

>>> 10 >= tb.counter
<Le: "author"."counter" <= %s, [10]>

>>> tb.counter < 10
<Lt: "author"."counter" < %s, [10]>

>>> 10 < tb.counter
<Gt: "author"."counter" > %s, [10]>

>>> tb.counter <= 10
<Le: "author"."counter" <= %s, [10]>

>>> 10 <= tb.counter
<Ge: "author"."counter" >= %s, [10]>

>>> tb.mask << 1
<LShift: "author"."mask" << %s, [1]>

>>> tb.mask >> 1
<RShift: "author"."mask" >> %s, [1]>

>>> tb.is_staff.is_(True)
<Is: "author"."is_staff" IS %s, [True]>

>>> tb.is_staff.is_not(True)
<IsNot: "author"."is_staff" IS NOT %s, [True]>

>>> tb.status.in_(('new', 'approved'))
<In: "author"."status" IN (%s, %s), ['new', 'approved']>

>>> tb.status.not_in(('new', 'approved'))
<NotIn: "author"."status" NOT IN (%s, %s), ['new', 'approved']>


>>> tb.last_name.like('mi')
```

```
<Like: "author"."last_name" LIKE %s, ['mi']>

>>> tb.last_name.ilike('mi')
<Ilike: "author"."last_name" ILIKE %s, ['mi']>

>>> P('mi').like(tb.last_name)
<Like: %s LIKE "author"."last_name", ['mi']>

>>> tb.last_name.rlike('mi')
<Like: %s LIKE "author"."last_name", ['mi']>

>>> tb.last_name.rilike('mi')
<Ilike: %s ILIKE "author"."last_name", ['mi']>

>>> tb.last_name.startswith('Sm')
<Like: "author"."last_name" LIKE REPLACE(REPLACE(REPLACE(%s, '!', '!!'), '_', '!_'), '
↪%%', '!%%') || '%%' ESCAPE '!', ['Sm']>

>>> tb.last_name.istartswith('Sm')
<Ilike: "author"."last_name" ILIKE REPLACE(REPLACE(REPLACE(%s, '!', '!!'), '_', '!_'),
↪ '%%', '!%%') || '%%' ESCAPE '!', ['Sm']>

>>> tb.last_name.contains('mi')
<Like: "author"."last_name" LIKE '%%' || REPLACE(REPLACE(REPLACE(%s, '!', '!!'), '_',
↪'!_'), '%%', '!%%') || '%%' ESCAPE '!', ['mi']>

>>> tb.last_name.icontains('mi')
<Ilike: "author"."last_name" ILIKE '%%' || REPLACE(REPLACE(REPLACE(%s, '!', '!!'), '_
↪', '!_'), '%%', '!%%') || '%%' ESCAPE '!', ['mi']>

>>> tb.last_name.endswith('th')
<Like: "author"."last_name" LIKE '%%' || REPLACE(REPLACE(REPLACE(%s, '!', '!!'), '_',
↪'!_'), '%%', '!%%') ESCAPE '!', ['th']>

>>> tb.last_name.iendswith('th')
<Ilike: "author"."last_name" ILIKE '%%' || REPLACE(REPLACE(REPLACE(%s, '!', '!!'), '_
↪', '!_'), '%%', '!%%') ESCAPE '!', ['th']>

>>> tb.last_name.rstartswith('Sm')
<Like: %s LIKE REPLACE(REPLACE(REPLACE("author"."last_name", '!', '!!'), '_', '!_'), '
↪%%', '!%%') || '%%' ESCAPE '!', ['Sm']>

>>> tb.last_name.ristartswith('Sm')
<Ilike: %s ILIKE REPLACE(REPLACE(REPLACE("author"."last_name", '!', '!!'), '_', '!_'),
↪ '%%', '!%%') || '%%' ESCAPE '!', ['Sm']>

>>> tb.last_name.rcontains('mi')
<Like: %s LIKE '%%' || REPLACE(REPLACE(REPLACE("author"."last_name", '!', '!!'), '_',
↪'!_'), '%%', '!%%') || '%%' ESCAPE '!', ['mi']>

>>> tb.last_name.ricontains('mi')
<Ilike: %s ILIKE '%%' || REPLACE(REPLACE(REPLACE("author"."last_name", '!', '!!'), '_
↪', '!_'), '%%', '!%%') || '%%' ESCAPE '!', ['mi']>

>>> tb.last_name.rendswith('th')
<Like: %s LIKE '%%' || REPLACE(REPLACE(REPLACE("author"."last_name", '!', '!!'), '_',
↪'!_'), '%%', '!%%') ESCAPE '!', ['th']>
```

```
>>> tb.last_name.riendswith('th')
<Ilike: %s ILIKE '%%' || REPLACE(REPLACE(REPLACE("author"."last_name", '!', '!!'), '_
↪', '!_'), '%%', '!%%') ESCAPE '!', ['th']>


>>> +tb.counter
<Pos: +"author"."counter", []>

>>> -tb.counter
<Neg: -"author"."counter", []>

>>> ~tb.counter
<Not: NOT "author"."counter", []>

>>> tb.name.distinct()
<Distinct: DISTINCT "author"."name", []>

>>> tb.counter ** 2
<Callable: POW("author"."counter", %s), [2]>

>>> 2 ** tb.counter
<Callable: POW(%s, "author"."counter"), [2]>

>>> tb.counter % 2
<Callable: MOD("author"."counter", %s), [2]>

>>> 2 % tb.counter
<Callable: MOD(%s, "author"."counter"), [2]>

>>> abs(tb.counter)
<Callable: ABS("author"."counter"), []>

>>> tb.counter.count()
<Callable: COUNT("author"."counter"), []>

>>> tb.age.between(20, 30)
<Between: "author"."age" BETWEEN %s AND %s, [20, 30]>

>>> tb.age[20:30]
<Between: "author"."age" BETWEEN %s AND %s, [20, 30]>

>>> tb.age[20]
<Eq: "author"."age" = %s, [20]>

>>> tb.name.concat(' staff', ' admin')
<Concat: "author"."name" || %s || %s, [' staff', ' admin']>

>>> tb.name.concat_ws(' ', 'staff', 'admin')
<Concat: concat_ws(%s, "author"."name", %s, %s), [' ', 'staff', 'admin']>

>>> tb.name.op('MY_EXTRA_OPERATOR')(10)
<Binary: "author"."name" MY_EXTRA_OPERATOR %s, [10]>

>>> tb.name.rop('MY_EXTRA_OPERATOR')(10)
<Binary: %s MY_EXTRA_OPERATOR "author"."name", [10]>

>>> tb.name.asc()
<Asc: "author"."name" ASC, []>
```

```
>>> tb.name.desc()
<Desc: "author"."name" DESC, []>

>>> ((tb.age > 25) | (tb.answers > 10)) & (tb.is_staff | tb.is_admin)
<And: ("author"."age" > %s OR "author"."answers" > %s) AND ("author"."is_staff" OR
↪"author"."is_admin"), [25, 10]>

>>> (T.author.first_name != 'Tom') & (T.author.last_name.in_(('Smith', 'Johnson')))
<Binary: "author"."first_name" <> %s AND "author"."last_name" IN (%s, %s), ['Tom',
↪'Smith', 'Johnson']>

>>> (T.author.first_name != 'Tom') | (T.author.last_name.in_(('Smith', 'Johnson')))
<Binary: "author"."first_name" <> %s OR "author"."last_name" IN (%s, %s), ['Tom',
↪'Smith', 'Johnson']>
```

3.5 Module sqlbuilder.smartsql.contrib.evaluate

Unfortunately, Python supports limited list of operators compared to PostgreSQL. Many operators like `@>`, `&>`, `-|-`, `@-@` and so on are not supported by Python.

You can use method `Expr.op()` or class `Binary` to solve this problem, for example:

```
>>> T.user.age.op('<@')(func.int8range(25, 30))
<Binary: "user"."age" <@ INT8RANGE(%s, %s), [25, 30]>

>>> Binary(T.user.age, '<@', func.int8range(25, 30))
<Binary: "user"."age" <@ INT8RANGE(%s, %s), [25, 30]>
```

But this solution has a lack of readability. So, sqlbuilder provides module *sqlbuilder.smartsql.contrib.evaluate*, that allows you to mix SQL operators (like `@>`, `&>`, `-|-`, `@-@` etc.) and python expressions. In other words, you can use SQL operators with Python expressions.

For example:

```
>>> from sqlbuilder.smartsql.contrib.evaluate import e
>>> e("T.user.age <@ func.int4range(25, 30)")
<Binary: "user"."age" <@ INT4RANGE(%s, %s), [25, 30]>
```

or with kwargs:

```
>>> from sqlbuilder.smartsql.contrib.evaluate import e
>>> required_range = func.int8range(25, 30)
>>> e("T.user.age <@ required_range", required_range=required_range)
<Binary: "user"."age" <@ INT4RANGE(%s, %s), [25, 30]>
```

or with context object:

```
>>> from sqlbuilder.smartsql.contrib.evaluate import e
>>> required_range = func.int8range(25, 30)
>>> e("T.user.age <@ required_range", locals())
<Binary: "user"."age" <@ INT4RANGE(%s, %s), [25, 30]>
```

You can pre-compile expression, similar re.compile(), to avoid parsing of it each time:

```
>>> from sqlbuilder.smartsql.contrib.evaluate import compile, e
>>> required_range = func.int8range(25, 30)
>>> compiled_expr = compile("""T.user.age <@ required_range""")
>>> e(compiled_expr, {'required_range': required_range})
<Binary: "user"."age" <@ INT4RANGE(%s, %s), [25, 30]>
```

Btw, you can even pre-compile expression into sql-string to achieve the fastest result:

```
>>> from sqlbuilder.smartsql import *
>>> from sqlbuilder.smartsql.contrib.evaluate import e
>>> sql, params = compile(e("T.user.age <@ func.int4range('%(min)s', '%(max)s')"))
>>> sql = sql.replace('%%', '%%%%') % tuple(params)
>>> sql
u'"user"."age" <@ INT4RANGE(%(min)s, %(max)s)'
```

More complex example:

```
>>> from sqlbuilder.smartsql import *
>>> from sqlbuilder.smartsql.contrib.evaluate import e
>>> required_range = func.int8range(25, 30)
>>> e("T.user.age <@ required_range AND NOT(T.user.is_staff OR T.user.is_admin)",␣
↪locals())
<Binary: "user"."age" <@ INT8RANGE(%s, %s) AND NOT ("user"."is_staff" OR "user"."is_
↪admin"), [25, 30]>
```

Note: Module *sqlbuilder.smartsql.contrib.evaluate* uses operator precedence similar PostgreSQL precedence, not Python precedence.

Also note, that `Power` has left association similar PostgreSQL, in contrast Python has right association:

```
$ python
>>> 2 ** 3 ** 5
14134776518227074636666380005943348126619871175004951664972849610340958208L
```

```
$ psql
postgres=# SELECT 2 ^ 3 ^ 5;
 ?column?
----------
    32768
(1 row)
```

Real operator precedence and association directions you can see in source code of the module.

3.6 Module sqlbuilder.smartsql

3.7 Query object

class sqlbuilder.smartsql.**Query**

Query builder class

result

Instance of *Result*. See *Implementation of execution*.

__init__ ([*tables=None*, *result=None*])

Parameters

- **tables** (*Table, TableAlias, TableJoin, or None*) – Tables for FROM clause of SQL query
- **result** (*Result or None*) – Object with implementation of execution

Building methods:

distinct (**args*, ***opts*)

Builds DISTINCT [ON (...)] clause. This method has interface similar to *fields()*.

- Adds expressions if arguments exist.
- Sets expressions if exists single argument of list/tuple type.
- Gets expressions without arguments.
- Resets expressions with `reset=True` keyword argument.

Also can be used sole argument of boolean type:

- `True` to apply DISTINCT clause
- `False` to reset DISTINCT clause

Returns copy of self if arguments exist, else current expression list.

Return type *Query* or ExprList

Example of usage:

```
>>> # Sole argument of boolean type
>>> from sqlbuilder.smartsql import Q, T
>>> q = Q().fields('*').tables(T.author)
>>> q
<Query: SELECT * FROM "author", []>
>>> bool(q.distinct())
False
>>> q = q.distinct(True)
>>> q
<Query: SELECT DISTINCT * FROM "author", []>
>>> bool(q.distinct())
True
>>> q.distinct(False)
<Query: SELECT * FROM "author", []>
>>> q
<Query: SELECT DISTINCT * FROM "author", []>

>>> # Expression list
>>> from sqlbuilder.smartsql import Q, T
>>> q = Q().tables(T.author).fields(T.author.first_name, T.author.last_name,␣
↪T.author.age)
>>> q
<Query: SELECT "author"."first_name", "author"."last_name", "author"."age"␣
↪FROM "author", []>

>>> # Add expressions:
>>> q = q.distinct(T.author.first_name, T.author.last_name)
>>> q
<Query: SELECT DISTINCT ON ("author"."first_name", "author"."last_name")
↪"author"."first_name", "author"."last_name", "author"."age" FROM "author",␣
↪[]>
```

```
>>> q = q.distinct(T.author.age)
>>> q
<Query: SELECT DISTINCT ON ("author"."first_name", "author"."last_name",
↪"author"."age") "author"."first_name", "author"."last_name", "author"."age"␣
↪FROM "author", []>

>>> # Get expressions:
>>> q.distinct()
<ExprList: "author"."first_name", "author"."last_name", "author"."age", []>

>>> # Set new expressions list:
>>> q = q.distinct([T.author.id, T.author.status])
>>> q
<Query: SELECT DISTINCT ON ("author"."id", "author"."status") "author"."first_
↪name", "author"."last_name", "author"."age" FROM "author", []>

>>> # Reset expressions:
>>> q.distinct([])
<Query: SELECT "author"."first_name", "author"."last_name", "author"."age"␣
↪FROM "author", []>
>>> q.distinct(reset=True)
<Query: SELECT "author"."first_name", "author"."last_name", "author"."age"␣
↪FROM "author", []>
```

fields(**args*, ***opts*)

Builds SELECT clause.

- Adds fields if arguments exist.
- Sets fields if exists single argument of list/tuple type.
- Gets fields without arguments.
- Resets fields with `reset=True` keyword argument.

Returns copy of self if arguments exist, else current field list.

Return type *Query* or FieldList

Example of usage:

```
>>> from sqlbuilder.smartsql import *
>>> q = Q().tables(T.author)

>>> # Add fields:
>>> q = q.fields(T.author.first_name, T.author.last_name)
>>> q
<Query: SELECT "author"."first_name", "author"."last_name" FROM "author", []>
>>> q = q.fields(T.author.age)
>>> q
<Query: SELECT "author"."first_name", "author"."last_name", "author"."age"␣
↪FROM "author", []>

>>> # Get fields:
>>> q.fields()
<FieldList: "author"."first_name", "author"."last_name", "author"."age", []>

>>> # Set new fields list:
>>> q = q.fields([T.author.id, T.author.status])
```

```
>>> q
<Query: SELECT "author"."id", "author"."status" FROM "author", []>

>>> # Reset fields:
>>> q = q.fields([])
>>> q
<Query: SELECT  FROM "author", []>

>>> # Another way to reset fields:
>>> q = q.fields(reset=True)
>>> q
<Query: SELECT  FROM "author", []>
```

tables(*tables=None*)

Builds FROM clause.

Parameters **tables** (*None, Table or TableJoin*) – Can be None, Table or TableJoin instance

Returns copied self with new tables if `tables` argument is not None, else current tables.

Return type TableJoin or *Query*

Example of usage:

```
>>> from sqlbuilder.smartsql import T, Q
>>> q = Q().tables(T.author).fields('*')
>>> q
<Query: SELECT * FROM "author", []>
>>> q = q.tables(T.author.as_('author_alias'))
>>> q
<Query: SELECT * FROM "author" AS "author_alias", []>
>>> q.tables()
<TableJoin: "author" AS "author_alias", []>
>>> q = q.tables((q.tables() + T.book).on(T.book.author_id == T.author.as_(
↪'author_alias').id))
>>> q
<Query: SELECT * FROM "author" AS "author_alias" LEFT OUTER JOIN "book" ON (
↪"book"."author_id" = "author_alias"."id"), []>
```

where(*cond*[, *op=operator.and_*])

Builds WHERE clause.

- Adds new criterias using the `op` operator, if `op` is not None.
- Sets new criterias if `op` is None.

Parameters

- **cond** (*Expr*) – Selection criterias
- **op** – Attribute of `operator` module or None, `operator.and_` by default

Returns copy of self with new criteria

Return type *Query*

Example of usage:

```
>>> import operator
>>> from sqlbuilder.smartsql import T, Q
>>> q = Q().tables(T.author).fields('*')
>>> q
<Query: SELECT * FROM "author", []>

>>> # Add conditions
>>> q = q.where(T.author.is_staff.is_(True))
>>> q
<Query: SELECT * FROM "author" WHERE "author"."is_staff" IS %s, [True]>
>>> q = q.where(T.author.first_name == 'John')
>>> q
<Query: SELECT * FROM "author" WHERE "author"."is_staff" IS %s AND "author".
↪"first_name" = %s, [True, 'John']>
>>> q = q.where(T.author.last_name == 'Smith', op=operator.or_)
>>> q
<Query: SELECT * FROM "author" WHERE "author"."is_staff" IS %s AND "author".
↪"first_name" = %s OR "author"."last_name" = %s, [True, 'John', 'Smith']>

>>> # Set conditions
>>> q = q.where(T.author.last_name == 'Smith', op=None)
>>> q
<Query: SELECT * FROM "author" WHERE "author"."last_name" = %s, ['Smith']>
```

group_by(**args*, ***opts*)

Builds GROUP BY clause. This method has interface similar to *fields()*.

- Adds expressions if arguments exist.
- Sets expressions if exists single argument of list/tuple type.
- Gets expressions without arguments.
- Resets expressions with `reset=True` keyword argument.

Returns copy of self if arguments exist, else current expression list.

Return type *Query* or ExprList

Example of usage:

```
>>> from sqlbuilder.smartsql import T, Q
>>> q = Q().tables(T.author).fields('*')
>>> q
<Query: SELECT * FROM "author", []>

>>> # Add expressions:
>>> q = q.group_by(T.author.first_name, T.author.last_name)
>>> q
<Query: SELECT * FROM "author" GROUP BY "author"."first_name", "author"."last_
↪name", []>
>>> q = q.group_by(T.author.age)
>>> q
<Query: SELECT * FROM "author" GROUP BY "author"."first_name", "author"."last_
↪name", "author"."age", []>

>>> # Get expressions:
>>> q.group_by()
<ExprList: "author"."first_name", "author"."last_name", "author"."age", []>
```

```
>>> # Set new expressions list:
>>> q = q.group_by([T.author.id, T.author.status])
>>> q
<Query: SELECT * FROM "author" GROUP BY "author"."id", "author"."status", []>

>>> # Reset expressions:
>>> q = q.group_by([])
>>> q
<Query: SELECT * FROM "author", []>

>>> # Another way to reset expressions:
>>> q = q.group_by(reset=True)
>>> q
<Query: SELECT * FROM "author", []>
```

having(*cond*[, *op=operator.and_*])

Builds HAVING clause. This method has interface similar to *where()*.

- Adds new criterias using the `op` operator, if `op` is not None.
- Sets new criterias if `op` is None.

Parameters

- **cond** (*Expr*) – Selection criterias
- **op** – Attribute of `operator` module or None, `operator.and_` by default

Returns copy of self with new criteria

Return type *Query*

Example of usage:

```
>>> import operator
>>> from sqlbuilder.smartsql import T, Q
>>> q = Q().fields('*').tables(T.author).group_by(T.author.status)
>>> q
<Query: SELECT * FROM "author" GROUP BY "author"."status", []>

>>> # Add conditions
>>> q = q.having(T.author.is_staff.is_(True))
>>> q
<Query: SELECT * FROM "author" GROUP BY "author"."status" HAVING "author"."is_
↪staff" IS %s, [True]>
>>> q = q.having(T.author.first_name == 'John')
>>> q
<Query: SELECT * FROM "author" GROUP BY "author"."status" HAVING "author"."is_
↪staff" IS %s AND "author"."first_name" = %s, [True, 'John']>
>>> q = q.having(T.author.last_name == 'Smith', op=operator.or_)
>>> q
<Query: SELECT * FROM "author" GROUP BY "author"."status" HAVING "author"."is_
↪staff" IS %s AND "author"."first_name" = %s OR "author"."last_name" = %s,␣
↪[True, 'John', 'Smith']>

>>> # Set conditions
>>> q = q.having(T.author.last_name == 'Smith', op=None)
>>> q
<Query: SELECT * FROM "author" GROUP BY "author"."status" HAVING "author".
↪"last_name" = %s, ['Smith']>
```

order_by(**args*, ***opts*)

Builds ORDER BY clause. This method has interface similar to *fields()*.

- Adds expressions if arguments exist.
- Sets expressions if exists single argument of list/tuple type.
- Gets expressions without arguments.
- Resets expressions with `reset=True` keyword argument.

Returns copy of self if arguments exist, else current expression list.

Return type *Query* or ExprList

Example of usage:

```
>>> from sqlbuilder.smartsql import T, Q
>>> q = Q().tables(T.author).fields('*')
>>> q
<Query: SELECT * FROM "author", []>

>>> # Add expressions:
>>> q = q.order_by(T.author.first_name, T.author.last_name)
>>> q
<Query: SELECT * FROM "author" ORDER BY "author"."first_name" ASC, "author".
↪"last_name" ASC, []>
>>> q = q.order_by(T.author.age.desc())
>>> q

>>> # Get expressions:
>>> q.order_by()
<ExprList: "author"."first_name" ASC, "author"."last_name" ASC, "author"."age
↪" DESC, []>

>>> # Set new expressions list:
<Query: SELECT * FROM "author" ORDER BY "author"."first_name" ASC, "author".
↪"last_name" ASC, "author"."age" DESC, []>
>>> q = q.order_by([T.author.id.desc(), T.author.status])
>>> q
<Query: SELECT * FROM "author" ORDER BY "author"."id" DESC, "author"."status"␣
↪ASC, []>

>>> # Reset expressions:
>>> q = q.order_by([])
>>> q
<Query: SELECT * FROM "author", []>

>>> # Another way to reset expressions:
>>> q = q.order_by(reset=True)
>>> q
<Query: SELECT * FROM "author", []>
```

Executing methods:

select(**args*, ***opts*)

Example of usage:

```
>>> from sqlbuilder.smartsql import Q, T
>>> Q(T.author).fields('*').select(for_update=True)
('SELECT * FROM "author" FOR UPDATE', [])
```

count()

Example of usage:

```
>>> from sqlbuilder.smartsql import Q, T
>>> Q(T.author).fields('*').count()
('SELECT COUNT(1) AS "count_value" FROM (SELECT * FROM "author") AS "count_
↪list"', [])
```

insert([*key_values=None*, ***kw*])

Parameters

- **key_values** (*dict*) – Map of fields (or field names) to it's values.
- **kw** – Extra keyword arguments that will be passed to `Insert` instance.

Example of usage:

```
>>> from sqlbuilder.smartsql import Q, T, func
>>> Q(T.stats).insert({
...     T.stats.object_type: 'author',
...     T.stats.object_id: 15,
...     T.stats.counter: 1
... }, on_duplicate_key_update={
...     T.stats.counter: T.stats.counter + func.VALUES(T.stats.counter)
... })
...
('INSERT INTO "stats" ("stats"."counter", "stats"."object_id", "stats".
↪"object_type") VALUES (%s, %s, %s) ON CONFLICT DO UPDATE SET "stats".
↪"counter" = "stats"."counter" + VALUES("stats"."counter")', [1, 15, 'author
↪'])

>>> # Keys of dict are strings
>>> Q(T.stats).insert({
...     'object_type': 'author',
...     'object_id': 15,
...     'counter': 1
... }, on_duplicate_key_update={
...     'counter': T.stats.counter + func.VALUES(T.stats.counter)
... })
...
('INSERT INTO "stats" ("object_type", "object_id", "counter") VALUES (%s, %s,
↪%s) ON CONFLICT DO UPDATE SET "counter" = "stats"."counter" + VALUES("stats
↪"."counter")', ['author', 15, 1])

>>> # Use "values" keyword argument
>>> Q().fields(
...     T.stats.object_type, T.stats.object_id, T.stats.counter
... ).tables(T.stats).insert(
...     values=('author', 15, 1),
...     on_duplicate_key_update={T.stats.counter: T.stats.counter + func.
↪VALUES(T.stats.counter)}
... )
...
('INSERT INTO "stats" ("stats"."object_type", "stats"."object_id", "stats".
↪"counter") VALUES %s, %s, %s ON CONFLICT DO UPDATE SET "stats"."counter" =
↪"stats"."counter" + VALUES("stats"."counter")', ['author', 15, 1])
```

```
>>> # Insert many
>>> Q().fields(
...     T.stats.object_type, T.stats.object_id, T.stats.counter
... ).tables(T.stats).insert(
...     values=(
...         ('author', 15, 1),
...         ('author', 16, 1),
...     ),
...     on_duplicate_key_update={T.stats.counter: T.stats.counter + func.
↪VALUES(T.stats.counter)}
... )
...
('INSERT INTO "stats" ("stats"."object_type", "stats"."object_id", "stats".
↪"counter") VALUES (%s, %s, %s), (%s, %s, %s) ON CONFLICT DO UPDATE SET
↪"stats"."counter" = "stats"."counter" + VALUES("stats"."counter")', ['author
↪', 15, 1, 'author', 16, 1])

>>> # Insert ignore
>>> Q().fields(
...     T.stats.object_type, T.stats.object_id, T.stats.counter
... ).tables(T.stats).insert(
...     values=('author', 15, 1),
...     ignore=True
... )
...
('INSERT INTO "stats" ("stats"."object_type", "stats"."object_id", "stats".
↪"counter") VALUES %s, %s, %s ON CONFLICT DO NOTHING', ['author', 15, 1])

>>> # INSERT ... SELECT Syntax
>>> Q().fields(
...     T.stats.object_type, T.stats.object_id, T.stats.counter
... ).tables(T.stats).insert(
...     values=Q().fields(
...         T.old_stats.object_type, T.old_stats.object_id, T.old_stats.
↪counter
...     ).tables(T.old_stats),
...     on_duplicate_key_update={
...         T.stats.counter: T.stats.counter + T.old_stats.counter,
...     }
... )
('INSERT INTO "stats" ("stats"."object_type", "stats"."object_id", "stats".
↪"counter") SELECT "old_stats"."object_type", "old_stats"."object_id", "old_
↪stats"."counter" FROM "old_stats" ON CONFLICT DO UPDATE SET "stats"."counter
↪" = "stats"."counter" + "old_stats"."counter"', [])
```

update(*[key_values=None, **kw]*)

Parameters

- **key_values** (*dict*) – Map of fields (or field names) to it's values.
- **kw** – Extra keyword arguments that will be passed to `Update` instance.

Example of usage:

```
>>> from sqlbuilder.smartsql import Q, T, func
>>> Q(T.author).where(T.author.id == 10).update({
...     T.author.first_name: 'John',
```

```
...     T.author.last_login: func.NOW()
... })
...
('UPDATE "author" SET "author"."last_login" = NOW(), "author"."first_name" =
↪%s WHERE "author"."id" = %s', ['John', 10])

>>> # Keys of dict are strings
>>> Q(T.author).where(T.author.id == 10).update({
...     'first_name': 'John',
...     'last_login': func.NOW()
... })
...
('UPDATE "author" SET "first_name" = %s, "last_login" = NOW() WHERE "author".
↪"id" = %s', ['John', 10])

>>> # Use "values" keyword argument
>>> Q(T.author).fields(
...     T.author.first_name, T.author.last_login
... ).where(T.author.id == 10).update(
...     values=('John', func.NOW())
... )
...
('UPDATE "author" SET "author"."first_name" = %s, "author"."last_login" =␣
↪NOW() WHERE "author"."id" = %s', ['John', 10])
```

delete(***kw*)

Parameters **kw** – Extra keyword arguments that will be passed to `Delete` instance.

Example of usage:

```
>>> from sqlbuilder.smartsql import Q, T
>>> Q(T.author).where(T.author.id == 10).delete()
('DELETE FROM "author" WHERE "author"."id" = %s', [10])
```

as_table(*alias*)

Returns current query as table reference.

Parameters **kw** (*str*) – alias name.

Example of usage:

```
>>> from sqlbuilder.smartsql import T, Q
>>> author_query_alias = Q(T.author).fields(T.author.id).where(
...     T.author.status == 'active'
... ).as_table('author_query_alias')
>>> Q().fields(T.book.id, T.book.title).tables(
...     (T.book & author_query_alias
... ).on(
...     T.book.author_id == author_query_alias.id
... ))
...
<Query: SELECT "book"."id", "book"."title" FROM "book" INNER JOIN (SELECT
↪"author"."id" FROM "author" WHERE "author"."status" = %s) AS "author_query_
↪alias" ON ("book"."author_id" = "author_query_alias"."id"), ['active']>
```

as_set([*all=False*])

Returns current query as set, to can be combined with other queries using the set operations union, intersection, and difference.

Parameters **all** (*bool*) – eliminates duplicate rows from result, if all is False.

Example of usage:

```
>>> from sqlbuilder.smartsql import T, Q
>>> q1 = Q(T.book1).fields(T.book1.id, T.book1.title).where(T.book1.author_id
↪== 10)
>>> q2 = Q(T.book2).fields(T.book2.id, T.book2.title).where(T.book2.author_id
↪== 10)
>>> q1.as_set() | q2
<Union: (SELECT "book1"."id", "book1"."title" FROM "book1" WHERE "book1".
↪"author_id" = %s) UNION (SELECT "book2"."id", "book2"."title" FROM "book2"
↪WHERE "book2"."author_id" = %s), [10, 10]>
>>> q1.as_set() | q2
<Union: (SELECT "book1"."id", "book1"."title" FROM "book1" WHERE "book1".
↪"author_id" = %s) UNION (SELECT "book2"."id", "book2"."title" FROM "book2"
↪WHERE "book2"."author_id" = %s), [10, 10]>
>>> q1.as_set() & q2
<Intersect: (SELECT "book1"."id", "book1"."title" FROM "book1" WHERE "book1".
↪"author_id" = %s) INTERSECT (SELECT "book2"."id", "book2"."title" FROM
↪"book2" WHERE "book2"."author_id" = %s), [10, 10]>
>>> q1.as_set() - q2
<Except: (SELECT "book1"."id", "book1"."title" FROM "book1" WHERE "book1".
↪"author_id" = %s) EXCEPT (SELECT "book2"."id", "book2"."title" FROM "book2"
↪WHERE "book2"."author_id" = %s), [10, 10]>
>>> q1.as_set(all=True) | q2
<Union: (SELECT "book1"."id", "book1"."title" FROM "book1" WHERE "book1".
↪"author_id" = %s) UNION ALL (SELECT "book2"."id", "book2"."title" FROM
↪"book2" WHERE "book2"."author_id" = %s), [10, 10]>
>>> q1.as_set(all=True) & q2
<Intersect: (SELECT "book1"."id", "book1"."title" FROM "book1" WHERE "book1".
↪"author_id" = %s) INTERSECT ALL (SELECT "book2"."id", "book2"."title" FROM
↪"book2" WHERE "book2"."author_id" = %s), [10, 10]>
>>> q1.as_set(all=True) - q2
<Except: (SELECT "book1"."id", "book1"."title" FROM "book1" WHERE "book1".
↪"author_id" = %s) EXCEPT ALL (SELECT "book2"."id", "book2"."title" FROM
↪"book2" WHERE "book2"."author_id" = %s), [10, 10]>
```

3.7.1 Subquery

Query extends `Expr`, so, it can be used as usual expression:

```
>>> from sqlbuilder.smartsql import Q, T

>>> # Subquery as condition
>>> sub_q = Q().fields(T.author.id).tables(T.author).where(T.author.status == 'active
↪')
>>> Q().fields(T.book.id).tables(T.book).where(T.book.author_id.in_(sub_q))
<Query: SELECT "book"."id" FROM "book" WHERE "book"."author_id" IN (SELECT "author".
↪"id" FROM "author" WHERE "author"."status" = %s), ['active']>

>>> # Subquery as field
>>> sub_q = Q().fields(
...     T.book.id.count().as_("book_count")
... ).tables(T.book).where(
...     T.book.pub_date > '2015-01-01'
... ).group_by(T.book.author_id)
```

```
...
>>> Q().fields(
...     T.author.id, sub_q.where(T.book.author_id == T.author.id)
... ).tables(T.author).where(
...     T.author.status == 'active'
... )
...
<Query: SELECT "author"."id", (SELECT COUNT("book"."id") AS "book_count" FROM "book"␣
↪WHERE "book"."pub_date" > %s AND "book"."author_id" = "author"."id" GROUP BY "book".
↪"author_id") FROM "author" WHERE "author"."status" = %s, ['2015-01-01', 'active']>

>>> # Subquery as alias
>>> alias = Q().fields(
...     T.book.id.count()
... ).tables(T.book).where(
...     (T.book.pub_date > '2015-01-01') &
...     (T.book.author_id == T.author.id)
... ).group_by(
...     T.book.author_id
... ).as_("book_count")
...
>>> Q().fields(
...     T.author.id, alias
... ).tables(T.author).where(
...     T.author.status == 'active'
... ).order_by(alias.desc())
...
<Query: SELECT "author"."id", (SELECT COUNT("book"."id") FROM "book" WHERE "book".
↪"pub_date" > %s AND "book"."author_id" = "author"."id" GROUP BY "book"."author_id")␣
↪AS "book_count" FROM "author" WHERE "author"."status" = %s ORDER BY "book_count"␣
↪DESC, ['2015-01-01', 'active']>
```

See also method *Query.as_table()*.

3.8 Implementation of execution

Class *Query* uses "Bridge pattern" for implementation of execution.

Module *sqlbuilder.smartsql* has default implementation in class *Result* for demonstration purposes, that does only one thing - returns a tuple with SQL string and parameters.

You can develop your own implementation, or, at least, specify what same compiler to use, for example:

```
>>> from sqlbuilder.smartsql import T, Q, Result
>>> from sqlbuilder.smartsql.dialects.mysql import compile as mysql_compile
>>> Q(result=Result(compile=mysql_compile)).fields(T.author.id, T.author.name).
↪tables(T.author).select()
('SELECT `author`.`id`, `author`.`name` FROM `author`', [])
```

See also examples of implementation in Django integration or Ascetic ORM integration

Instance of *Query* also delegates all unknown methods and properties to *Query.result*. Example:

```
>>> from sqlbuilder.smartsql import T, Q, Result
>>> from sqlbuilder.smartsql.dialects.mysql import compile as mysql_compile
>>> class CustomResult(Result):
...         custom_attr = 5
```

```
...         def custom_method(self, arg1, arg2):
...                 return (self._query, arg1, arg2)
...         def find_by_name(self, name):
...                 return self._query.where(T.author.name == name)
...
>>> q = Q(result=CustomResult(compile=mysql_compile)).fields(T.author.id, T.author.
↪name).tables(T.author)
>>> q.custom_attr
5
>>> q.custom_method(5, 10)
(<Query: SELECT "author"."id", "author"."name" FROM "author", []>, 5, 10)
>>> q.find_by_name('John')
<Query: SELECT "author"."id", "author"."name" FROM "author" WHERE "author"."name" =
↪%s, ['John']>
```

class sqlbuilder.smartsql.**Result**

Default implementation of execution for *Query* class.

compile

instance of Compiler, *sqlbuilder.smartsql.compile()* by default

_query

Current *Query* instance

__init__(*[compile=None]*)

Parameters **compile** (*Compiler or None*) – Compiler to compile SQL string

3.9 Compilers

There are three compilers for three dialects:

sqlbuilder.smartsql.**compile**(*expr*[, *state=None*])

It's a default compiler for PostgreSQL dialect, instance of Compiler. It also used for representation of expressions.

Parameters

- **expr** (*Expr*) – Expression to be compiled
- **state** (*State or None*) – Instance of State or None

Returns If state is None, then returns tuple with SQL string and list of parameters. Else returns None.

Return type tuple or None

sqlbuilder.smartsql.dialects.mysql.**compile**(*expr*[, *state=None*])

Compiler for MySQL dialect.

sqlbuilder.smartsql.dialects.sqlite.**compile**(*expr*[, *state=None*])

Compiler for SQLite dialect.

CHAPTER 4

Short manual for sqlbuilder.mini

The package contains yet another, extremely lightweight sql builder - *sqlbuilder.mini*, especially for Raw-SQL fans. It's just a hierarchical list of SQL strings, no more. Such form of presentation allows modify query without syntax analysis.

You can use sqlparse library to parse SQL-string to hierarchical list, see, for example, module `sqlbuilder.mini.parser`. By the way, you can use this library to change SQL directly, - it has very nice API like DOM manipulation. You can also parse SQL to DOM or etree, to have navigation by XPath.

Also you can use pyparsing library to parse SQL-string to hierarchical list.

```
>>> from sqlbuilder.mini import P, Q, compile
>>> sql = [
...     'SELECT', [
...         'author.id', 'author.first_name', 'author.last_name'
...     ],
...     'FROM', [
...         'author', 'INNER JOIN', ['book as b', 'ON', 'b.author_id = author.id']
...     ],
...     'WHERE', [
...         'b.status', '==', P('new')
...     ],
...     'ORDER BY', [
...         'author.first_name', 'author.last_name'
...     ]
... ]

>>> # Let change query
>>> sql[sql.index('SELECT') + 1].append('author.age')

>>> compile(sql)
('SELECT author.id, author.first_name, author.last_name, author.age FROM author INNER␣
↪JOIN book as b ON b.author_id = author.id WHERE b.status == %s ORDER BY author.
↪first_name, author.last_name', ['new'])
```

To facilitate navigation and change SQL, there is helper `sqlbuilder.mini.Q`:

```
>>> sql = [
...     'SELECT', [
...         'author.id', 'author.first_name', 'author.last_name'
...     ],
...     'FROM', [
...         'author', 'INNER JOIN', [
...             '(', 'SELECT', [
...                 'book.title'
...             ],
...             'FROM', [
...                 'book'
...             ],
...             ')', 'AS b', 'ON', 'b.author_id = author.id'
...         ],
...     ],
...     'WHERE', [
...         'b.status', '==', P('new')
...     ],
...     'ORDER BY', [
...         'author.first_name', 'author.last_name'
...     ]
... ]

>>> sql = Q(sql)
>>> sql.prepend_child(
...     ['FROM', 'INNER JOIN', 'SELECT'],  # path
...     ['book.id', 'book.pages']  # values to append
... )
>>> sql.append_child(
...     ['FROM', 'INNER JOIN', 'SELECT'],
...     ['book.date']
... )
>>> sql.insert_after(
...     ['FROM', 'INNER JOIN', (list, 1), ],
...     ['WHERE', ['b.pages', '>', P(100)]]
... )
>>> sql.insert_before(
...     ['FROM', 'INNER JOIN', 'WHERE', 'b.pages'],
...     ['b.pages', '<', P(500), 'AND']
... )
>>> compile(sql)
('SELECT author.id, author.first_name, author.last_name FROM author INNER JOIN (␣
↪SELECT book.id, book.pages, book.title, book.date FROM book WHERE b.pages < %s AND␣
↪b.pages > %s ) AS b ON b.author_id = author.id WHERE b.status == %s ORDER BY author.
↪first_name, author.last_name', [500, 100, 'new'])
```

As step of path can be used:

- Exact string: [‘FROM’, ‘INNER JOIN’, ‘SELECT’]
- Callable object, that returns index: [‘FROM’, ‘INNER JOIN’, (lambda i, item, collection: item == ‘SELECT’)]
- Index as integer (from zero): [‘FROM’, ‘INNER JOIN’, 1]
- Slice: [‘FROM’, ‘INNER JOIN’, slice(2, 5)]
- Each item: [‘FROM’, enumerate, ‘SELECT’]
- Compiled Regular Expression Objects: [‘FROM’, ‘INNER JOIN’, re.compile(“^SELECT$”)]

Also it’s possible combine rules.

Filter result by each next rules: [‘FROM’, ‘INNER JOIN’, (‘SELECT’, 0)] is egual to [‘FROM’, ‘INNER JOIN’, Filter(Exact(‘SELECT’), Index(0))]

Union results matched by any rules: [‘FROM’, ‘INNER JOIN’, Union(Exact(‘SELECT’), Index(1))].

Intersect results matched by all rules: [‘FROM’, ‘INNER JOIN’, Intersect(Exact(‘SELECT’), Index(1))].

The difference between Filter and Union in that, Filter handles only collection of matched elements by previous rules, and Union - a full collection.

P.S.: See also article about SQLBuilder in English and in Russian.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

sqlbuilder.mini, 26
sqlbuilder.smartsql, 14
sqlbuilder.smartsql.contrib.evaluate, 13

Index

Symbols

__init__() (sqlbuilder.smartsql.Query method), 14
__init__() (sqlbuilder.smartsql.Result method), 26
_query (sqlbuilder.smartsql.Result attribute), 26

A

as_set() (sqlbuilder.smartsql.Query method), 23
as_table() (sqlbuilder.smartsql.Query method), 23

C

compile (sqlbuilder.smartsql.Result attribute), 26
count() (sqlbuilder.smartsql.Query method), 21

D

delete() (sqlbuilder.smartsql.Query method), 23
distinct() (sqlbuilder.smartsql.Query method), 15

F

fields() (sqlbuilder.smartsql.Query method), 16

G

group_by() (sqlbuilder.smartsql.Query method), 18

H

having() (sqlbuilder.smartsql.Query method), 19

I

insert() (sqlbuilder.smartsql.Query method), 21

O

order_by() (sqlbuilder.smartsql.Query method), 20

Q

Query (class in sqlbuilder.smartsql), 14

R

Result (class in sqlbuilder.smartsql), 26
result (sqlbuilder.smartsql.Query attribute), 14

S

select() (sqlbuilder.smartsql.Query method), 20
sqlbuilder.mini (module), 26
sqlbuilder.smartsql (module), 14
sqlbuilder.smartsql.compile() (in module sqlbuilder.smartsql), 26
sqlbuilder.smartsql.contrib.evaluate (module), 13
sqlbuilder.smartsql.dialects.mysql.compile() (in module sqlbuilder.smartsql), 26
sqlbuilder.smartsql.dialects.sqlite.compile() (in module sqlbuilder.smartsql), 26

T

tables() (sqlbuilder.smartsql.Query method), 17

U

update() (sqlbuilder.smartsql.Query method), 22

W

where() (sqlbuilder.smartsql.Query method), 17