Storm Documentation

Gustavo Niemeyer

Dec 14, 2021

Contents:

CHAPTER 1

Tutorial

1.1 Importing

Let's start by importing some names into the namespace.

```
>>> from storm.locals import *
```

1.2 Basic definition

Now we define a type with some properties describing the information we're about to map.

```
>>> class Person(object):
...     __storm_table__ = "person"
...     id = Int(primary=True)
...     name = Unicode()
```

Notice that this has no Storm-defined base class or constructor.

1.3 Creating a database and the store

We still don't have anyone to talk to, so let's define an in-memory SQLite database to play with, and a store using that database.

```
>>> database = create_database("sqlite:")
>>> store = Store(database)
```

Three databases are supported at the moment: SQLite, MySQL, and PostgreSQL. The parameter passed to `create_database()` is an URI, as follows:

```
# database = create_database(
#     "scheme://username:password@hostname:port/database_name")
```

The `scheme` may be `sqlite`, `mysql`, or `postgres`.

Now we have to create the table that will actually hold the data for our class.

```
>>> store.execute("CREATE TABLE person "
...               "(id INTEGER PRIMARY KEY, name VARCHAR)")
<storm.databases.sqlite.SQLiteResult object at 0x...>
```

We got a result back, but we don't care about it for now. We could also use `noresult=True` to avoid the result entirely.

1.4 Creating an object

Let's create an object of the defined class.

```
>>> joe = Person()
>>> joe.name = u"Joe Johnes"

>>> print(joe.id)
None
>>> print(joe.name)
Joe Johnes
```

So far this object has no connection to a database. Let's add it to the store we've created above.

```
>>> store.add(joe)
<...Person object at 0x...>

>>> print(joe.id)
None
>>> print(joe.name)
Joe Johnes
```

Notice that the object wasn't changed, even after being added to the store. That's because it wasn't flushed yet.

1.5 The store of an object

Once an object is added to a store, or retrieved from a store, its relation to that store is known. We can easily verify which store an object is bound.

```
>>> Store.of(joe) is store
True

>>> Store.of(Person()) is None
True
```

1.6 Finding an object

Now, what would happen if we actually asked the store to give us the person named "Joe Johnes"?

```
>>> person = store.find(Person, Person.name == u"Joe Johnes").one()

>>> print(person.id)
1
>>> print(person.name)
Joe Johnes
```

The person is there! Yeah, ok, you were expecting it. :-)

We can also retrieve the object using its primary key.

```
>>> print(store.get(Person, 1).name)
Joe Johnes
```

1.7 Caching behavior

One interesting thing is that this person is actually Joe, right? We've just added this object, so there's only one Joe, why would there be two different objects? There isn't.

```
>>> person is joe
True
```

What's going on behind the scenes is that each store has an object cache. When an object is linked to a store, it will be cached by the store for as long as there's a reference to the object somewhere, or while the object is dirty (has unflushed changes).

Storm ensures that at least a certain number of recently used objects will stay in memory inside the transaction, so that frequently used objects are not retrieved from the database too often.

1.8 Flushing

When we tried to find Joe in the database for the first time, we've noticed that the `id` property was magically assigned. This happened because the object was flushed implicitly so that the operation would affect any pending changes as well.

Flushes may also happen explicitly.

```
>>> mary = Person()
>>> mary.name = u"Mary Margaret"
>>> store.add(mary)
<...Person object at 0x...>

>>> print(mary.id)
None
>>> print(mary.name)
Mary Margaret

>>> store.flush()
>>> print(mary.id)
2
>>> print(mary.name)
Mary Margaret
```

1.9 Changing objects with the Store

Besides changing objects as usual, we can also benefit from the fact that objects are tied to a database to change them using expressions.

```
>>> store.find(
...     Person, Person.name == u"Mary Margaret").set(name=u"Mary Maggie")
>>> print(mary.name)
Mary Maggie
```

This operation will touch every matching object in the database, and also objects that are alive in memory.

1.10 Committing

Everything we've done so far is inside a transaction. At this point, we can either make these changes and any pending uncommitted changes persistent by committing them, or we can undo everything by rolling them back.

We'll commit them, with something as simple as

```
>>> store.commit()
```

That was straightforward. Everything is still the way it was, but now changes are there "for real".

1.11 Rolling back

Aborting changes is very straightforward as well.

```
>>> joe.name = u"Tom Thomas"
```

Let's see if these changes are really being considered by Storm and by the database.

```
>>> person = store.find(Person, Person.name == u"Tom Thomas").one()
>>> person is joe
True
```

Yes, they are. Now, for the magic step (suspense music, please).

```
>>> store.rollback()
```

Erm.. nothing happened?

Actually, something happened.. with Joe. He's back!

```
>>> print(joe.id)
1
>>> print(joe.name)
Joe Johnes
```

1.12 Constructors

So, we've been working for too long with people only. Let's introduce a new kind of data in our model: companies. For the company, we'll use a constructor, just for the fun of it. It will be the simplest company class you've ever seen:

```
>>> class Company(object):
...     __storm_table__ = "company"
...     id = Int(primary=True)
...     name = Unicode()
...
...     def __init__(self, name):
...         self.name = name
```

Notice that the constructor parameter isn't optional. It could be optional, if we wanted, but our companies always have names.

Let's add the table for it.

```
>>> store.execute(
...     "CREATE TABLE company (id INTEGER PRIMARY KEY, name VARCHAR)",
...     noresult=True)
```

Then, create a new company.

```
>>> circus = Company(u"Circus Inc.")

>>> print(circus.id)
None
>>> print(circus.name)
Circus Inc.
```

The `id` is still undefined because we haven't flushed it. In fact, we haven't even **added** the company to the store. We'll do that soon. Watch out.

1.13 References and subclassing

Now we want to assign some employees to our company. Rather than redoing the Person definition, we'll keep it as it is, since it's general, and will create a new subclass of it for employees, which include one extra field: the company id.

```
>>> class Employee(Person):
...     __storm_table__ = "employee"
...     company_id = Int()
...     company = Reference(company_id, Company.id)
...
...     def __init__(self, name):
...         self.name = name
```

Pay attention to that definition for a moment. Notice that it doesn't define what's already in person, and introduces the `company_id`, and a `company` property, which is a reference to another class. It also has a constructor, but which leaves the company alone.

As usual, we need a table. SQLite has no idea of what a foreign key is, so we'll not bother to define it.

```
>>> store.execute(
...     "CREATE TABLE employee "
...     "(id INTEGER PRIMARY KEY, name VARCHAR, company_id INTEGER)",
...     noresult=True)
```

Let's give life to Ben now.

```
>>> ben = store.add(Employee(u"Ben Bill"))

>>> print(ben.id)
None
>>> print(ben.name)
Ben Bill
>>> print(ben.company_id)
None
```

We can see that they were not flushed yet. Even then, we can say that Bill works on Circus.

```
>>> ben.company = circus

>>> print(ben.company_id)
None
>>> print(ben.company.name)
Circus Inc.
```

Of course, we still don't know the company id since it was not flushed to the database yet, and we didn't assign an id explicitly. Storm is keeping the relationship even then.

If whatever is pending is flushed to the database (implicitly or explicitly), objects will get their ids, and any references are updated as well (before being flushed!).

```
>>> store.flush()

>>> print(ben.company_id)
1
>>> print(ben.company.name)
Circus Inc.
```

They're both flushed to the database. Now, notice that the Circus company wasn't added to the store explicitly in any moment. Storm will do that automatically for referenced objects, for both objects (the referenced and the referencing one).

Let's create another company to check something. This time we'll flush the store just after adding it.

```
>>> sweets = store.add(Company(u"Sweets Inc."))
>>> store.flush()
>>> sweets.id
2
```

Nice, we've already got the id of the new company. So, what would happen if we changed **just the id** for Ben's company?

```
>>> ben.company_id = 2
>>> print(ben.company.name)
Sweets Inc.
>>> ben.company is sweets
True
```

Hah! **That** wasn't expected, was it? ;-)

Let's commit everything.

```
>>> store.commit()
```

1.14 Many-to-one reference sets

So, while our model says that employees work for a single company (we only design normal people here), companies may of course have multiple employees. We represent that in Storm using reference sets.

We won't define the company again. Instead, we'll add a new attribute to the class.

```
>>> Company.employees = ReferenceSet(Company.id, Employee.company_id)
```

Without any further work, we can already see which employees are working for a given company.

```
>>> sweets.employees.count()
1

>>> for employee in sweets.employees:
...     print(employee.id)
...     print(employee.name)
...     print(employee is ben)
...
1
Ben Bill
True
```

Let's create another employee, and add him to the company, rather than setting the company in the employee (it sounds better, at least).

```
>>> mike = store.add(Employee(u"Mike Mayer"))
>>> sweets.employees.add(mike)
```

That, of course, means that Mike's working for a company, and so it should be reflected elsewhere.

```
>>> mike.company_id
2

>>> mike.company is sweets
True
```

1.15 Many-to-many reference sets and composed keys

We want to represent accountants in our model as well. Companies have accountants, but accountants may also attend several companies, so we'll represent that using a many-to-many relationship.

Let's create a simple class to use with accountants, and the relationship class.

```
>>> class Accountant(Person):
...     __storm_table__ = "accountant"
...     def __init__(self, name):
...         self.name = name

>>> class CompanyAccountant(object):
...     __storm_table__ = "company_accountant"
...     __storm_primary__ = "company_id", "accountant_id"
...     company_id = Int()
...     accountant_id = Int()
```

Hey, we've just declared a class with a composed key!

Now, let's use it to declare the many-to-many relationship in the company. Once more, we'll just stick the new attribute in the existent object. It may easily be defined at class definition time. Later we'll see another way to do that as well.

```
>>> Company.accountants = ReferenceSet(Company.id,
...                                    CompanyAccountant.company_id,
...                                    CompanyAccountant.accountant_id,
...                                    Accountant.id)
```

Done! The order in which attributes were defined is important, but the logic should be pretty obvious.

We're missing some tables, at this point.

```
>>> store.execute(
...     "CREATE TABLE accountant (id INTEGER PRIMARY KEY, name VARCHAR)",
...     noresult=True)

>>> store.execute(
...     "CREATE TABLE company_accountant "
...     "(company_id INTEGER, accountant_id INTEGER,"
...     " PRIMARY KEY (company_id, accountant_id))",
...     noresult=True)
```

Let's give life to a couple of accountants, and register them in both companies.

```
>>> karl = Accountant(u"Karl Kent")
>>> frank = Accountant(u"Frank Fourt")

>>> sweets.accountants.add(karl)
>>> sweets.accountants.add(frank)

>>> circus.accountants.add(frank)
```

That's it! Really! Notice that we didn't even have to add them to the store, since it happens implicitly by linking to the other object which is already in the store, and that we didn't have to declare the relationship object, since that's known to the reference set.

We can now check them.

```
>>> sweets.accountants.count()
2

>>> circus.accountants.count()
1
```

Even though we didn't use the `CompanyAccountant` object explicitly, we can check it if we're really curious.

```
>>> store.get(CompanyAccountant, (sweets.id, frank.id))
<...CompanyAccountant object at 0x...>
```

Notice that we pass a tuple for the `get()` method, due to the composed key.

If we wanted to know for which companies accountants are working, we could easily define a reversed relationship:

```
>>> Accountant.companies = ReferenceSet(Accountant.id,
...                                     CompanyAccountant.accountant_id,
...                                     CompanyAccountant.company_id,
...                                     Company.id)
```

(continues on next page)

(continued from previous page)

```
>>> for name in sorted(company.name for company in frank.companies):
...     print(name)
Circus Inc.
Sweets Inc.

>>> for company in karl.companies:
...     print(company.name)
Sweets Inc.
```

1.16 Joins

Since we've got some nice data to play with, let's try to make a few interesting queries.

Let's start by checking which companies have at least one employee named Ben. We have at least two ways to do it.

First, with an implicit join.

```
>>> result = store.find(Company,
...                     Employee.company_id == Company.id,
...                     Employee.name.like(u"Ben %"))

>>> for company in result:
...     print(company.name)
Sweets Inc.
```

Then, we can also do an explicit join. This is interesting for mapping complex SQL joins to Storm queries.

```
>>> origin = [Company, Join(Employee, Employee.company_id == Company.id)]
>>> result = store.using(*origin).find(
...     Company, Employee.name.like(u"Ben %"))

>>> for company in result:
...     print(company.name)
Sweets Inc.
```

If we already had the company, and wanted to know which of his employees were named Ben, that'd have been easier.

```
>>> result = sweets.employees.find(Employee.name.like(u"Ben %"))

>>> for employee in result:
...     print(employee.name)
Ben Bill
```

1.17 Sub-selects

Suppose we want to find all accountants that aren't associated with a company. We can use a sub-select to get the data we want.

```
>>> laura = Accountant(u"Laura Montgomery")
>>> store.add(laura)
<...Accountant ...>
```

(continues on next page)

(continued from previous page)

```
>>> subselect = Select(CompanyAccountant.accountant_id, distinct=True)
>>> result = store.find(Accountant, Not(Accountant.id.is_in(subselect)))
>>> result.one() is laura
True
```

1.18 Ordering and limiting results

Ordering and limiting results obtained are certainly among the simplest and yet most wanted features for such tools, so we want to make them very easy to understand and use, of course.

A line of code is worth a thousand words, so here are a few examples that demonstrate how it works:

```
>>> garry = store.add(Employee(u"Garry Glare"))

>>> result = store.find(Employee)

>>> for employee in result.order_by(Employee.name):
...     print(employee.name)
Ben Bill
Garry Glare
Mike Mayer

>>> for employee in result.order_by(Desc(Employee.name)):
...     print(employee.name)
Mike Mayer
Garry Glare
Ben Bill

>>> for employee in result.order_by(Employee.name)[:2]:
...     print(employee.name)
Ben Bill
Garry Glare
```

1.19 Multiple types with one query

Sometimes, it may be interesting to retrieve more than one object involved in a given query. Imagine, for instance, that besides knowing which companies have an employee named Ben, we also want to know who is the employee. This may be achieved with a query like follows:

```
>>> result = store.find((Company, Employee),
...                     Employee.company_id == Company.id,
...                     Employee.name.like(u"Ben %"))

>>> for company, employee in result:
...     print(company.name)
...     print(employee.name)
Sweets Inc.
Ben Bill
```

1.20 The Storm base class

So far we've been defining our references and reference sets using classes and their properties. This has some advantages, like being easier to debug, but also has some disadvantages, such as requiring classes to be present in the local scope, which potentially leads to circular import issues.

To prevent that kind of situation, Storm supports defining these references using the stringified version of the class and property names. The only inconvenience of doing so is that all involved classes must inherit from the `Storm` base class.

Let's define some new classes to show that. To expose the point, we'll refer to a class before it's actually defined.

```
>>> class Country(Storm):
...     __storm_table__ = "country"
...     id = Int(primary=True)
...     name = Unicode()
...     currency_id = Int()
...     currency = Reference(currency_id, "Currency.id")

>>> class Currency(Storm):
...     __storm_table__ = "currency"
...     id = Int(primary=True)
...     symbol = Unicode()

>>> store.execute(
...     "CREATE TABLE country "
...     "(id INTEGER PRIMARY KEY, name VARCHAR, currency_id INTEGER)",
...     noresult=True)

>>> store.execute(
...     "CREATE TABLE currency (id INTEGER PRIMARY KEY, symbol VARCHAR)",
...     noresult=True)
```

Now, let's see if it works.

```
>>> real = store.add(Currency())
>>> real.id = 1
>>> real.symbol = u"BRL"

>>> brazil = store.add(Country())
>>> brazil.name = u"Brazil"
>>> brazil.currency_id = 1

>>> print(brazil.currency.symbol)
BRL
```

Questions!? ;-)

1.21 Loading hook

Storm allows classes to define a few different hooks are called to act when certain things happen. One of the interesting hooks available is the `__storm_loaded__` one.

Let's play with it. We'll define a temporary subclass of Person for that.

```
>>> class PersonWithHook(Person):
...     def __init__(self, name):
...         print("Creating %s" % name)
...         self.name = name
...
...     def __storm_loaded__(self):
...         print("Loaded %s" % self.name)

>>> earl = store.add(PersonWithHook(u"Earl Easton"))
Creating Earl Easton

>>> earl = store.find(PersonWithHook, name=u"Earl Easton").one()

>>> store.invalidate(earl)
>>> del earl
>>> import gc
>>> collected = gc.collect()

>>> earl = store.find(PersonWithHook, name=u"Earl Easton").one()
Loaded Earl Easton
```

Note that in the first find, nothing was called, since the object was still in memory and cached. Then, we invalidated the object from Storm's internal cache and ensured that it was out-of-memory by triggering a garbage collection. After that, the object had to be retrieved from the database again, and thus the hook was called (and not the constructor!).

1.22 Executing expressions

Storm also offers a way to execute expressions in a database-agnostic way, when that's necessary.

For instance:

```
>>> result = store.execute(Select(Person.name, Person.id == 1))
>>> (name,) = result.get_one()
>>> print(name)
Joe Johnes
```

This mechanism is used internally by Storm itself to implement the higher level features.

1.23 Auto-reloading values

Storm offers some special values that may be assigned to attributes under its control. One of these values is *AutoReload*. When used, it will make the object automatically reload the value from the database when touched. Even primary keys may benefit from its use, as shown below.

```
>>> from storm.locals import AutoReload

>>> ruy = store.add(Person())
>>> ruy.name = u"Ruy"
>>> print(ruy.id)
None

>>> ruy.id = AutoReload
```

(continues on next page)

(continued from previous page)

```
>>> print(ruy.id)
4
```

This may be set as the default value for any attribute, making the object be automatically flushed if necessary.

1.24 Expression values

Besides auto-reloading, it's also possible to assign what we call a "lazy expression" to an attribute. Such expressions are flushed to the database when the attribute is accessed, or when the object is flushed to the database (INSERT/UPDATE time).

For instance:

```
>>> ruy.name = SQL(
...     "(SELECT name || ? FROM person WHERE id=4)", (" Ritcher",))
>>> print(ruy.name)
Ruy Ritcher
```

Notice that this is just an example of what **may** be done. There's no need to write SQL statements this way, if you don't want to. You may also use class-based SQL expressions provided in Storm, or even not use lazy expressions at all.

1.25 Aliases

So now let's say that we want to find every pair of people that work for the same company. I have no idea about why one would *want* to do that, but that's a good case for us to exercise aliases.

First, we create an alias for the *Employee* class.

```
>>> from storm.info import ClassAlias
>>> AnotherEmployee = ClassAlias(Employee)
```

Nice, isn't it?

Now we can easily make the query we want, in a straightforward way:

```
>>> result = store.find((Employee, AnotherEmployee),
...                     Employee.company_id == AnotherEmployee.company_id,
...                     Employee.id > AnotherEmployee.id)

>>> for employee1, employee2 in result:
...     print(employee1.name)
...     print(employee2.name)
Mike Mayer
Ben Bill
```

Woah! Mike and Ben work for the same company!

(Quiz for the attentive reader: why is *greater than* being used in the query above?)

1.26 Debugging

Sometimes you just need to see which statements Storm is executing. A debug tracer built on top of Storm's tracing system can be used to see what's going on under the hood. A tracer is an object that gets notified when interesting events occur, such as when Storm executes a statement. A function to enable and disable statement tracing is provided. Statements are logged to sys.stderr by default, but a custom stream may also be used.

```
>>> import sys
>>> from storm.tracer import debug

>>> debug(True, stream=sys.stdout)
>>> result = store.find((Employee, AnotherEmployee),
...                     Employee.company_id == AnotherEmployee.company_id,
...                     Employee.id > AnotherEmployee.id)
>>> list(result)
[...] EXECUTE: ...'SELECT employee.company_id, employee.id, employee.name, "...".
↪company_id, "...".id, "...".name FROM employee, employee AS "..." WHERE employee.
↪company_id = "...".company_id AND employee.id > "...".id', ()
[...] DONE
[(<...Employee object at ...>, <...Employee object at ...>)]

>>> debug(False)
>>> list(result)
[(<...Employee object at ...>, <...Employee object at ...>)]
```

1.27 Much more!

There's a lot more about Storm to be shown. This tutorial is just a way to get initiated on some of the concepts. If your questions are not answered somewhere else, feel free to ask them in the mailing list.

CHAPTER 2

Infoheritance

Storm doesn't support classes that have columns in multiple tables. This makes using inheritance rather difficult. The infoheritance pattern described here provides a way to get the benefits of inheritance without running into the problems Storm has with multi-table classes.

2.1 Defining a sample model

Let's consider an inheritance hierarchy to migrate to Storm.

```
class Person(object):

    def __init__(self, name):
        self.name = name


class SecretAgent(Person):

    def __init__(self, name, passcode):
        super(SecretAgent, self).__init__(name)
        self.passcode = passcode


class Teacher(Person):

    def __init__(self, name, school):
        super(Employee, self).__init__(name):
        self.school = school
```

We want to use three tables to store data for these objects: `person`, `secret_agent` and `teacher`. We can't simply convert instance attributes to Storm properties and add `__storm_table__` definitions because a single object may not have columns that come from more than one table. We can't have `Teacher` getting its `name` column from the `person` table and its `school` column from the `teacher` table, for example.

2.2 The infoheritance pattern

The infoheritance pattern uses composition instead of inheritance to work around the multiple table limitation. A base Storm class is used to represent all objects in the hierarchy. Each instance of this base class has an info property that yields an instance of a specific info class. An info class provides the additional data and behaviour you'd normally implement in a subclass. Following is the design from above converted to use the pattern.

```
>>> from storm.locals import Storm, Store, Int, Unicode, Reference

>>> person_info_types = {}

>>> def register_person_info_type(info_type, info_class):
...     existing_info_class = person_info_types.get(info_type)
...     if existing_info_class is not None:
...         raise RuntimeError("%r has the same info_type of %r" %
...                            (info_class, existing_info_class))
...     person_info_types[info_type] = info_class
...     info_class.info_type = info_type


>>> class Person(Storm):
...
...     __storm_table__ = "person"
...
...     id = Int(allow_none=False, primary=True)
...     name = Unicode(allow_none=False)
...     info_type = Int(allow_none=False)
...     _info = None
...
...     def __init__(self, store, name, info_class, **kwargs):
...         self.name = name
...         self.info_type = info_class.info_type
...         store.add(self)
...         self._info = info_class(self, **kwargs)
...
...     @property
...     def info(self):
...         if self._info is not None:
...             return self._info
...         assert self.id is not None
...         info_class = person_info_types[self.info_type]
...         if not hasattr(info_class, "__storm_table__"):
...             info = info_class.__new__(info_class)
...             info.person = self
...         else:
...             info = Store.of(self).get(info_class, self.id)
...         self._info = info
...         return info


>>> class PersonInfo(object):
...
...     def __init__(self, person):
...         self.person = person


>>> class StoredPersonInfo(PersonInfo):
```

(continues on next page)

(continued from previous page)

```
...
...     person_id = Int(allow_none=False, primary=True)
...     person = Reference(person_id, Person.id)


>>> class SecretAgent(StoredPersonInfo):
...
...     __storm_table__ = "secret_agent"
...
...     passcode = Unicode(allow_none=False)
...
...     def __init__(self, person, passcode=None):
...         super(SecretAgent, self).__init__(person)
...         self.passcode = passcode


>>> class Teacher(StoredPersonInfo):
...
...     __storm_table__ = "teacher"
...
...     school = Unicode(allow_none=False)
...
...     def __init__(self, person, school=None):
...         super(Teacher, self).__init__(person)
...         self.school = school
```

The pattern works by having a base class, `Person`, keep a reference to an info class, `PersonInfo`. Info classes need to be registered so that `Person` can discover them and load them when necessary. Note that info types have the same ID as their parent object. This isn't strictly necessary, but it makes certain things easy, such as being able to look up info objects directly by ID when given a person object. `Person` objects are required to be in a store to ensure that an ID is available and can used by the info class.

2.3 Registering info classes

Let's register our info classes. Each class must be registered with a unique info type key. This key is stored in the database, so be sure to use a stable value.

```
>>> register_person_info_type(1, SecretAgent)
>>> register_person_info_type(2, Teacher)
```

Let's create a database to store person objects before we continue.

```
>>> from storm.locals import create_database

>>> database = create_database("sqlite:")
>>> store = Store(database)
>>> result = store.execute("""
...     CREATE TABLE person (
...         id INTEGER PRIMARY KEY,
...         info_type INTEGER NOT NULL,
...         name TEXT NOT NULL)
... """)
>>> result = store.execute("""
...     CREATE TABLE secret_agent (
```

(continues on next page)

(continued from previous page)

```
...         person_id INTEGER PRIMARY KEY,
...         passcode TEXT NOT NULL)
... """)
>>> result = store.execute("""
...     CREATE TABLE teacher (
...         person_id INTEGER PRIMARY KEY,
...         school TEXT NOT NULL)
... """)
```

2.4 Creating info classes

We can easily create person objects now.

```
>>> secret_agent = Person(store, u"Dick Tracy",
...                        SecretAgent, passcode=u"secret!")
>>> teacher = Person(store, u"Mrs. Cohen",
...                   Teacher, school=u"Cameron Elementary School")
>>> store.commit()
```

And we can easily find them again.

```
>>> del secret_agent
>>> del teacher
>>> store.rollback()

>>> [type(person.info)
...  for person in store.find(Person).order_by(Person.name)]
[<class '...SecretAgent'>, <class '...Teacher'>]
```

2.5 Retrieving info classes

Now that we have our basic hierarchy in place we're going to want to retrieve objects by info type. Let's implement a function to make finding `Persons` easier.

```
>>> def get_persons(store, info_classes=None):
...     where = []
...     if info_classes:
...         info_types = [
...             info_class.info_type for info_class in info_classes]
...         where = [Person.info_type.is_in(info_types)]
...     result = store.find(Person, *where)
...     result.order_by(Person.name)
...     return result

>>> secret_agent = get_persons(store, info_classes=[SecretAgent]).one()
>>> print(secret_agent.name)
Dick Tracy
>>> print(secret_agent.info.passcode)
secret!

>>> teacher = get_persons(store, info_classes=[Teacher]).one()
```

(continues on next page)

(continued from previous page)

```
>>> print(teacher.name)
Mrs. Cohen
>>> print(teacher.info.school)
Cameron Elementary School
```

Great, we can easily find different kinds of `Persons`.

2.6 In-memory info objects

This design also allows for in-memory info objects. Let's add one to our hierarchy.

```
>>> class Ghost(PersonInfo):
...
...     friendly = True

>>> register_person_info_type(3, Ghost)
```

We create and load in-memory objects the same way we do stored ones.

```
>>> ghost = Person(store, u"Casper", Ghost)
>>> store.commit()
>>> del ghost
>>> store.rollback()

>>> ghost = get_persons(store, info_classes=[Ghost]).one()
>>> print(ghost.name)
Casper
>>> print(ghost.info.friendly)
True
```

This pattern is very handy when using Storm with code that would naturally be implemented using inheritance.

CHAPTER 3

Zope integration

The `storm.zope` package contains the ZStorm utility which provides seamless integration between Storm and Zope 3's transaction system. Setting up ZStorm is quite easy. In most cases, you want to include `storm/zope/configure.zcml` in your application, which you would normally do in ZCML as follows:

```
<include package="storm.zope" />
```

For the purposes of this doctest we'll register ZStorm manually.

```
>>> from zope.component import provideUtility, getUtility
>>> import transaction
>>> from storm.zope.interfaces import IZStorm
>>> from storm.zope.zstorm import global_zstorm
```

```
>>> provideUtility(global_zstorm, IZStorm)
>>> zstorm = getUtility(IZStorm)
>>> zstorm
<storm.zope.zstorm.ZStorm object at ...>
```

Awesome, now that the utility is in place we can start to use it!

3.1 Getting stores

The ZStorm utility allows us work with named stores.

```
>>> zstorm.set_default_uri("test", "sqlite:")
```

Setting a default URI for stores isn't strictly required. We could pass it as the second argument to `zstorm.get`. Providing a default URI makes it possible to use `zstorm.get` more easily; this is especially handy when multiple threads are used as we'll see further on.

```
>>> store = zstorm.get("test")
>>> store
<storm.store.Store object at ...>
```

ZStorm has automatically created a store instance for us. If we ask for a store by name again, we should get the same instance.

```
>>> same_store = zstorm.get("test")
>>> same_store is store
True
```

The stores provided by ZStorm are per-thread. If we ask for the named store in a different thread we should get a different instance.

```
>>> import threading
```

```
>>> thread_store = []
>>> def get_thread_store():
...     thread_store.append(zstorm.get("test"))
```

```
>>> thread = threading.Thread(target=get_thread_store)
>>> thread.start()
>>> thread.join()
>>> thread_store != [store]
True
```

Great! ZStorm abstracts away the process of creating and managing named stores. Let's move on and use the stores with Zope's transaction system.

3.2 Committing transactions

The primary purpose of ZStorm is to integrate with Zope's transaction system. Let's create a schema so we can play with some real data and see how it works.

```
>>> result = store.execute("""
...     CREATE TABLE person (
...         id INTEGER PRIMARY KEY,
...         name TEXT)
... """)
>>> store.commit()
```

We'll need a `Person` class to use with this database.

```
>>> from storm.locals import Storm, Int, Unicode
```

```
>>> class Person(Storm):
...
...     __storm_table__ = "person"
...
...     id = Int(primary=True)
...     name = Unicode()
...
...     def __init__(self, name):
...         self.name = name
```

Great! Let's try it out.

```
>>> person = Person(u"John Doe")
>>> store.add(person)
<...Person object at ...>
>>> transaction.commit()
```

Notice that we're not using `store.commit` directly; we're using Zope's transaction system. Let's make sure it worked.

```
>>> store.rollback()
>>> same_person = store.find(Person).one()
>>> same_person is person
True
```

Awesome!

3.3 Aborting transactions

Let's make sure aborting transactions works, too.

```
>>> store.add(Person(u"Imposter!"))
<...Person object at ...>
```

At this point a `store.find` should return the new object.

```
>>> for name in sorted(person.name for person in store.find(Person)):
...     print(name)
Imposter!
John Doe
```

All this means is that the data has been flushed to the database; it's still not committed. If we abort the transaction the new `Person` object should disappear.

```
>>> transaction.abort()
>>> for person in store.find(Person):
...     print(person.name)
John Doe
```

Excellent! As you can see, ZStorm makes working with SQL databases and Zope 3 very natural.

3.4 ZCML

In the examples above we setup our stores manually. In many cases, setting up named stores via ZCML directives is more desirable. Add a stanza similar to the following to your ZCML configuration to setup a named store.

```
<store name="test" uri="sqlite:" />
```

With that in place `getUtility(IZStorm).get("test")` will return the store named "test".

3.5 Security Wrappers

Storm knows how to deal with “wrapped” objects – the identity of any Storm-managed object does not need to be the same as the original object, by way of the “object info” system. As long as the object info can be retrieved from the wrapped objects, things work fine.

To interoperate with the Zope security wrapper system, storm.zope tells Zope to exposes certain Storm-internal attributes which appear on Storm-managed objects.

```
>>> from storm.info import get_obj_info, ObjectInfo
>>> from zope.security.checker import ProxyFactory
>>> from pprint import pprint
```

```
>>> person = store.find(Person).one()
>>> type(get_obj_info(person)) is ObjectInfo
True
>>> type(get_obj_info(ProxyFactory(person))) is ObjectInfo
True
```

Security-wrapped result sets can be used in the same way as unwrapped ones.

```
>>> from zope.component.testing import (
...     setUp,
...     tearDown,
...     )
>>> from zope.configuration import xmlconfig
>>> from zope.security.protectclass import protectName
>>> import storm.zope
```

```
>>> setUp()
>>> _ = xmlconfig.file("configure.zcml", package=storm.zope)
>>> protectName(Person, "name", "zope.Public")
```

```
>>> another_person = Person(u"Jane Doe")
>>> store.add(another_person)
<...Person object at ...>
>>> result = ProxyFactory(store.find(Person).order_by(Person.name))
>>> for person in result:
...     print(person.name)
Jane Doe
John Doe
>>> print(result[0].name)
Jane Doe
>>> for person in result[:1]:
...     print(person.name)
Jane Doe
>>> another_person in result
True
>>> result.is_empty()
False
>>> result.any()
<...Person object at ...>
>>> print(result.first().name)
Jane Doe
>>> print(result.last().name)
John Doe
```

(continues on next page)

(continued from previous page)

```
>>> print(result.count())
2
```

Check `list()` as well as ordinary iteration: on Python 3, this tries to call `__len__` first (which doesn't exist, but is nevertheless allowed by the security wrapper).

```
>>> for person in list(result):
...     print(person.name)
Jane Doe
John Doe
```

```
>>> result = ProxyFactory(
...     store.find(Person, Person.name.startswith(u"John")))
>>> print(result.one().name)
John Doe
```

Security-wrapped reference sets work too.

```
>>> _ = store.execute("""
...     CREATE TABLE team (
...         id INTEGER PRIMARY KEY,
...         name TEXT)
... """)
>>> _ = store.execute("""
...     CREATE TABLE teammembership (
...         id INTEGER PRIMARY KEY,
...         person INTEGER NOT NULL REFERENCES person,
...         team INTEGER NOT NULL REFERENCES team)
... """)
>>> store.commit()
```

```
>>> from storm.locals import Reference, ReferenceSet, Store
```

```
>>> class TeamMembership(Storm):
...
...     __storm_table__ = "teammembership"
...
...     id = Int(primary=True)
...
...     person_id = Int(name="person", allow_none=False)
...     person = Reference(person_id, "Person.id")
...
...     team_id = Int(name="team", allow_none=False)
...     team = Reference(team_id, "Team.id")
...
...     def __init__(self, person, team):
...         self.person = person
...         self.team = team
```

```
>>> class Team(Storm):
...
...     __storm_table__ = "team"
...
...     id = Int(primary=True)
...     name = Unicode()
```

(continues on next page)

(continued from previous page)

```
...
...     def __init__(self, name):
...         self.name = name
...
...     members = ReferenceSet(
...         "id", "TeamMembership.team_id",
...         "TeamMembership.person_id", "Person.id",
...         order_by="Person.name")
...
...     def addMember(self, person):
...         Store.of(self).add(TeamMembership(person, self))
```

```
>>> protectName(Team, "members", "zope.Public")
>>> protectName(Team, "addMember", "zope.Public")
```

```
>>> doe_family = Team(U"does")
>>> store.add(doe_family)
<...Team object at ...>
>>> doe_family = ProxyFactory(doe_family)
>>> doe_family.addMember(person)
>>> doe_family.addMember(another_person)
```

```
>>> for member in doe_family.members:
...     print(member.name)
Jane Doe
John Doe
>>> for person in doe_family.members[:1]:
...     print(person.name)
Jane Doe
>>> print(doe_family.members[0].name)
Jane Doe
```

```
>>> tearDown()
```

3.6 ResultSet interfaces

Query results provide `IResultSet` (or `ISQLObjectResultSet` if SQLObject's compatibility layer is used).

```
>>> from storm.zope.interfaces import IResultSet, ISQLObjectResultSet
>>> from storm.store import EmptyResultSet, ResultSet
>>> from storm.sqlobject import SQLObjectResultSet
>>> IResultSet.implementedBy(ResultSet)
True
>>> IResultSet.implementedBy(EmptyResultSet)
True
```

```
>>> ISQLObjectResultSet.implementedBy(SQLObjectResultSet)
True
```

CHAPTER 4

API

4.1 Locals

The following names are re-exported from `storm.locals` for convenience:

- *storm.base.Storm*
- *storm.database.create_database()*
- *storm.exceptions.StormError*
- *storm.expr.And*
- *storm.expr.Asc*
- *storm.expr.Count*
- *storm.expr.Delete*
- *storm.expr.Desc*
- *storm.expr.In*
- *storm.expr.Insert*
- *storm.expr.Join*
- *storm.expr.Like*
- *storm.expr.Max*
- *storm.expr.Min*
- *storm.expr.Not*
- *storm.expr.Or*
- *storm.expr.SQL*
- *storm.expr.Select*
- *storm.expr.Update*
- *storm.info.ClassAlias*
- *storm.properties.Bool*
- *storm.properties.Bytes*
- *storm.properties.Date*
- *storm.properties.DateTime*
- *storm.properties.Decimal*
- *storm.properties.Enum*
- *storm.properties.Float*
- *storm.properties.Int*

- *storm.properties.JSON*
- *storm.properties.List*
- *storm.properties.Pickle*
- *storm.properties.Time*
- *storm.properties.TimeDelta*
- *storm.properties.UUID*
- *storm.properties.Unicode*
- *storm.references.Proxy*
- *storm.references.Reference*
- *storm.references.ReferenceSet*
- *storm.store.AutoReload*
- *storm.store.Store*
- *storm.xid.Xid*

4.2 Store

The Store interface to a database.

This module contains the highest-level ORM interface in Storm.

class storm.store.**Store**(*database*, *cache=None*)

Bases: `object`

The Storm Store.

This is the highest-level interface to a database. It manages transactions with *commit* and *rollback*, caching, high-level querying with *find*, and more.

Note that Store objects are not threadsafe. You should create one Store per thread in your application, passing them the same backend `Database` object.

Parameters

- **database** – The *storm.database.Database* instance to use.
- **cache** – The cache to use. Defaults to a `Cache` instance.

get_database()

Return this Store's Database object.

static of(*obj*)

Get the Store that the object is associated with.

If the given object has not yet been associated with a store, return None.

execute(*statement*, *params=None*, *noresult=False*)

Execute a basic query.

This is just like *storm.database.Connection.execute*, except that a flush is performed first.

close()

Close the connection.

`begin`(*xid*)
Start a new two-phase transaction.

Parameters **`xid`** – A *`Xid`* instance holding identification data for the new transaction.

`prepare`()
Prepare a two-phase transaction for the final commit.

Note It must be called inside a two-phase transaction started with *`begin`*.

`commit`()
Commit all changes to the database.

This invalidates the cache, so all live objects will have data reloaded next time they are touched.

`rollback`()
Roll back all outstanding changes, reverting to database state.

`get`(*cls*, *key*)
Get object of type cls with the given primary key from the database.

If the object is alive the database won't be touched.

Parameters

- **`cls`** – Class of the object to be retrieved.
- **`key`** – Primary key of object. May be a tuple for composed keys.

Returns The object found with the given primary key, or None if no object is found.

`find`(*cls_spec*, **args*, ***kwargs*)
Perform a query.

Some examples:

```
store.find(Person, Person.name == u"Joe") --> all Persons named Joe
store.find(Person, name=u"Joe") --> same
store.find((Company, Person), Person.company_id == Company.id) -->
    iterator of tuples of Company and Person instances which are
    associated via the company_id -> Company relation.
```

Parameters

- **`cls_spec`** – The class or tuple of classes whose associated tables will be queried.
- **`args`** – Instances of `Expr`.
- **`kwargs`** – Mapping of simple column names to values or expressions to query for.

Returns A *`ResultSet`* of instances `cls_spec`. If `cls_spec` was a tuple, then an iterator of tuples of such instances.

`using`(**tables*)
Specify tables to use explicitly.

The *`find`* method generally does a good job at figuring out the tables to query by itself, but in some cases it's useful to specify them explicitly.

This is most often necessary when an explicit SQL join is required. An example follows:

```
join = LeftJoin(Person, Person.id == Company.person_id)
print(list(store.using(Company, join).find((Company, Person))))
```

The previous code snippet will produce an SQL statement somewhat similar to this, depending on your backend:

```
SELECT company.id, employee.company_id, employee.id
FROM company
LEFT JOIN employee ON employee.company_id = company.id;
```

Returns A `TableSet`, which has a `find` method similar to `Store.find`.

`add`(*obj*)
Add the given object to the store.

The object will be inserted into the database if it has not yet been added.

The `added` event will be fired on the object info's event system.

`remove`(*obj*)
Remove the given object from the store.

The associated row will be deleted from the database.

`reload`(*obj*)
Reload the given object.

The object will immediately have all of its data reset from the database. Any pending changes will be thrown away.

`autoreload`(*obj=None*)
Set an object or all objects to be reloaded automatically on access.

When a database-backed attribute of one of the objects is accessed, the object will be reloaded entirely from the database.

Parameters **`obj`** – If passed, only mark the given object for autoreload. Otherwise, all cached objects will be marked for autoreload.

`invalidate`(*obj=None*)
Set an object or all objects to be invalidated.

This prevents Storm from returning the cached object without first verifying that the object is still available in the database.

This should almost never be called by application code; it is only necessary if it is possible that an object has disappeared through some mechanism that Storm was unable to detect, like direct SQL statements within the current transaction that bypassed the ORM layer. The Store automatically invalidates all cached objects on transaction boundaries.

`reset`()
Reset this store, causing all future queries to return new objects.

Beware this method: it breaks the assumption that there will never be two objects in memory which represent the same database object.

This is useful if you've got in-memory changes to an object that you want to "throw out"; next time they're fetched the objects will be recreated, so in-memory modifications will not be in effect for future queries.

`add_flush_order`(*before*, *after*)
Explicitly specify the order of flushing two objects.

When the next database flush occurs, the order of data modification statements will be ensured.

Parameters

- **before** – The object to flush first.
- **after** – The object to flush after before.

remove_flush_order (*before*, *after*)

Cancel an explicit flush order specified with *add_flush_order*.

Parameters

- **before** – The before object previously specified in a call to *add_flush_order*.
- **after** – The after object previously specified in a call to *add_flush_order*.

flush ()

Flush all dirty objects in cache to database.

This method will first call the __storm_pre_flush__ hook of all dirty objects. If more objects become dirty as a result of executing code in the hooks, the hook is also called on them. The hook is only called once for each object.

It will then flush each dirty object to the database, that is, execute the SQL code to insert/delete/update them. After each object is flushed, the hook __storm_flushed__ is called on it, and if changes are made to the object it will get back to the dirty list, and be flushed again.

Note that Storm will flush objects for you automatically, so you'll only need to call this method explicitly in very rare cases where normal flushing times are insufficient, such as when you want to make sure a database trigger gets run at a particular time.

block_implicit_flushes ()

Block implicit flushes from operations like execute().

unblock_implicit_flushes ()

Unblock implicit flushes from operations like execute().

block_access ()

Block access to the underlying database connection.

unblock_access ()

Unblock access to the underlying database connection.

class storm.store.**EmptyResultSet** (*ordered=False*)

Bases: object

An object that looks like a *ResultSet* but represents no rows.

This is convenient for application developers who want to provide a method which is guaranteed to return a *ResultSet*-like object but which, in certain cases, knows there is no point in querying the database. For example:

```
def get_people(self, ids):
    if not ids:
        return EmptyResultSet()
    return store.find(People, People.id.is_in(ids))
```

The methods on EmptyResultSet (one, config, union, etc) are meant to emulate a *ResultSet* which has matched no rows.

get_select_expr (**columns*)

Get a Select expression to retrieve only the specified columns.

Parameters columns – One or more *storm.expr.Column* objects whose values will be fetched.

Raises ***FeatureError*** – Raised if no columns are specified.

Returns A `Select` expression configured to use the query parameters specified for this result set. The result of the select will always be an empty set of rows.

class `storm.store.`**`block_access`**(**args*)

Bases: `object`

Context manager blocks database access by one or more *`Store`*s in the managed scope.

class `storm.store.`**`ResultSet`**(*store*, *find_spec*, *where=Undef*, *tables=Undef*, *select=Undef*)

Bases: `object`

The representation of the results of a query.

Note that having an instance of this class does not indicate that a database query has necessarily been made. Database queries are put off until absolutely necessary.

Generally these should not be constructed directly, but instead retrieved from calls to *`Store.find`*.

`copy`()

Return a copy of this ResultSet object, with the same configuration.

`config`(*distinct=None*, *offset=None*, *limit=None*)

Configure this result object in-place. All parameters are optional.

Parameters

- **`distinct`** – If True, enables usage of the DISTINCT keyword in the query. If a tuple or list of columns, inserts a DISTINCT ON (only supported by PostgreSQL).
- **`offset`** – Offset where results will start to be retrieved from the result set.
- **`limit`** – Limit the number of objects retrieved from the result set.

Returns self (not a copy).

`is_empty`()

Return `True` if this result set doesn't contain any results.

`any`()

Return a single item from the result set.

Returns An arbitrary object or `None` if one isn't available.

See *`one`*, *`first`*, and *`last`*.

`first`()

Return the first item from an ordered result set.

Raises ***`UnorderedError`*** – Raised if the result set isn't ordered.

Returns The first object or `None` if one isn't available.

See *`last`*, *`one`*, and *`any`*.

`last`()

Return the last item from an ordered result set.

Raises

- ***`FeatureError`*** – Raised if the result set has a `LIMIT` set.
- ***`UnorderedError`*** – Raised if the result set isn't ordered.

Returns The last object or `None` if one isn't available.

See *`first`*, *`one`*, and *`any`*.

one()
: Return one item from a result set containing at most one item.

    Raises ***NotOneError*** – Raised if the result set contains more than one item.

    Returns The object or `None` if one isn't available.

    See *first*, *last*, and *any*.

order_by(**args*)
: Specify the ordering of the results.

    The query will be modified appropriately with an ORDER BY clause.

    Ascending and descending order can be specified by wrapping the columns in `Asc` and `Desc`.

    Parameters **args** – One or more *storm.expr.Column* objects.

remove()
: Remove all rows represented by this ResultSet from the database.

    This is done efficiently with a DELETE statement, so objects are not actually loaded into Python.

group_by(**expr*)
: Group this ResultSet by the given expressions.

    Parameters **expr** – The expressions used in the GROUP BY statement.

    Returns self (not a copy).

having(**expr*)
: Filter result previously grouped by.

    Parameters **expr** – Instances of `Expr`.

    Returns self (not a copy).

count(*expr=Undef, distinct=False*)
: Get the number of objects represented by this ResultSet.

max(*expr*)
: Get the highest value from an expression.

min(*expr*)
: Get the lowest value from an expression.

avg(*expr*)
: Get the average value from an expression.

sum(*expr*)
: Get the sum of all values in an expression.

get_select_expr(**columns*)
: Get a `Select` expression to retrieve only the specified columns.

    Parameters **columns** – One or more *storm.expr.Column* objects whose values will be fetched.

    Raises ***FeatureError*** – Raised if no columns are specified or if this result is a set expression such as a union.

    Returns A `Select` expression configured to use the query parameters specified for this result set, and also limited to only retrieving data for the specified columns.

values(**columns*)

Retrieve only the specified columns.

This does not load full objects from the database into Python.

Parameters **columns** – One or more `storm.expr.Column` objects whose values will be fetched.

Raises **FeatureError** – Raised if no columns are specified or if this result is a set expression such as a union.

Returns An iterator of tuples of the values for each column from each matching row in the database.

set(**args*, ***kwargs*)

Update objects in the result set with the given arguments.

This method will update all objects in the current result set to match expressions given as equalities or keyword arguments. These objects may still be in the database (an UPDATE is issued) or may be cached.

For instance, `result.set(Class.attr1 == 1, attr2=2)` will set `attr1` to 1 and `attr2` to 2, on all matching objects.

cached()

Return matching objects from the cache for the current query.

find(**args*, ***kwargs*)

Perform a query on objects within this result set.

This is analogous to `Store.find`, although it doesn't take a `cls_spec` argument, instead using the same tables as the existing result set, and restricts the results to those in this set.

Parameters

- **args** – Instances of `Expr`.
- **kwargs** – Mapping of simple column names to values or expressions to query for.

Returns A `ResultSet` of matching instances.

union(*other*, *all=False*)

Get the `Union` of this result set and another.

Parameters **all** – If True, include duplicates.

difference(*other*, *all=False*)

Get the difference, using `Except`, of this result set and another.

Parameters **all** – If True, include duplicates.

intersection(*other*, *all=False*)

Get the `Intersection` of this result set and another.

Parameters **all** – If True, include duplicates.

class storm.store.**TableSet**(*store*, *tables*)

Bases: `object`

The representation of a set of tables which can be queried at once.

This will typically be constructed by a call to `Store.using`.

find(*cls_spec*, **args*, ***kwargs*)

Perform a query on the previously specified tables.

This is identical to *Store.find* except that the tables are explicitly specified instead of relying on inference.

Returns A *ResultSet*.

`storm.store.`**`AutoReload`**

A marker for reloading a single value.

Often this will be used to specify that a specific attribute should be loaded from the database on the next access, like so:

```
storm_object.property = AutoReload
```

On the next access to `storm_object.property`, the value will be loaded from the database.

It is also often used as a default value for a property:

```
class Person(object):
    __storm_table__ = "person"
    id = Int(allow_none=False, default=AutoReload)

person = store.add(Person)
person.id # gets the attribute from the database.
```

4.3 Defining tables and columns

4.3.1 Base

`class` `storm.base.`**`Storm`**

Bases: `object`

An optional base class for objects stored in a Storm Store.

It causes your subclasses to be associated with a Storm PropertyRegistry. It is necessary to use this if you want to specify References with strings.

4.3.2 Properties

`class` `storm.properties.`**`Property`**`(`*name=None*, *primary=False*, *variable_class=<class 'storm.variables.Variable'>*, *variable_kwargs={}*`)`

Bases: `object`

A property representing a database column.

Properties can be set as attributes of classes that have a `__storm_table__`, and can then be used like ordinary Python properties on instances of the class, corresponding to database columns.

Parameters

- **`name`** – The name of this property.
- **`primary`** – A boolean indicating whether this property is a primary key.
- **`variable_class`** – The type of *storm.variables.Variable* corresponding to this property.
- **`variable_kwargs`** – Dictionary of keyword arguments to be passed when constructing the underlying variable.

class storm.properties.**SimpleProperty**(*name=None*, *primary=False*, ***kwargs*)
Bases: *storm.properties.Property*

Parameters

- **name** – The name of this property.
- **primary** – A boolean indicating whether this property is a primary key.
- **default** – The initial value of this variable. The default behavior is for the value to stay undefined until it is set with `set`.
- **default_factory** – If specified, this will immediately be called to get the initial value.
- **allow_none** – A boolean indicating whether None should be allowed to be set as the value of this variable.
- **validator** – Validation function called whenever trying to set the variable to a non-db value. The function should look like validator(object, attr, value), where the first and second arguments are the result of validator_object_factory() (or None, if this parameter isn't provided) and the value of validator_attribute, respectively. When called, the function should raise an error if the value is unacceptable, or return the value to be used in place of the original value otherwise.
- **kwargs** – Other keyword arguments passed through when constructing the underlying variable.

class storm.properties.**Bool**(*name=None*, *primary=False*, ***kwargs*)
Bases: *storm.properties.SimpleProperty*

Boolean property.

This accepts integer, `float`, or `decimal.Decimal` values, and stores them as booleans.

Parameters

- **name** – The name of this property.
- **primary** – A boolean indicating whether this property is a primary key.
- **default** – The initial value of this variable. The default behavior is for the value to stay undefined until it is set with `set`.
- **default_factory** – If specified, this will immediately be called to get the initial value.
- **allow_none** – A boolean indicating whether None should be allowed to be set as the value of this variable.
- **validator** – Validation function called whenever trying to set the variable to a non-db value. The function should look like validator(object, attr, value), where the first and second arguments are the result of validator_object_factory() (or None, if this parameter isn't provided) and the value of validator_attribute, respectively. When called, the function should raise an error if the value is unacceptable, or return the value to be used in place of the original value otherwise.
- **kwargs** – Other keyword arguments passed through when constructing the underlying variable.

variable_class
alias of *storm.variables.BoolVariable*

class storm.properties.**Int**(*name=None*, *primary=False*, ***kwargs*)
Bases: *storm.properties.SimpleProperty*

Integer property.

This accepts integer, `float`, or `decimal.Decimal` values, and stores them as integers.

Parameters

- **`name`** – The name of this property.
- **`primary`** – A boolean indicating whether this property is a primary key.
- **`default`** – The initial value of this variable. The default behavior is for the value to stay undefined until it is set with `set`.
- **`default_factory`** – If specified, this will immediately be called to get the initial value.
- **`allow_none`** – A boolean indicating whether None should be allowed to be set as the value of this variable.
- **`validator`** – Validation function called whenever trying to set the variable to a non-db value. The function should look like validator(object, attr, value), where the first and second arguments are the result of validator_object_factory() (or None, if this parameter isn't provided) and the value of validator_attribute, respectively. When called, the function should raise an error if the value is unacceptable, or return the value to be used in place of the original value otherwise.
- **`kwargs`** – Other keyword arguments passed through when constructing the underlying variable.

`variable_class`
alias of *`storm.variables.IntVariable`*

`class` `storm.properties.`**`Float`**(*name=None*, *primary=False*, ***kwargs*)
Bases: *`storm.properties.SimpleProperty`*

Float property.

This accepts integer, `float`, or `decimal.Decimal` values, and stores them as floating-point values.

Parameters

- **`name`** – The name of this property.
- **`primary`** – A boolean indicating whether this property is a primary key.
- **`default`** – The initial value of this variable. The default behavior is for the value to stay undefined until it is set with `set`.
- **`default_factory`** – If specified, this will immediately be called to get the initial value.
- **`allow_none`** – A boolean indicating whether None should be allowed to be set as the value of this variable.
- **`validator`** – Validation function called whenever trying to set the variable to a non-db value. The function should look like validator(object, attr, value), where the first and second arguments are the result of validator_object_factory() (or None, if this parameter isn't provided) and the value of validator_attribute, respectively. When called, the function should raise an error if the value is unacceptable, or return the value to be used in place of the original value otherwise.
- **`kwargs`** – Other keyword arguments passed through when constructing the underlying variable.

`variable_class`
alias of *`storm.variables.FloatVariable`*

`class` `storm.properties.`**`Decimal`**(*name=None*, *primary=False*, ***kwargs*)
Bases: *`storm.properties.SimpleProperty`*

Decimal property.

This accepts integer or `decimal.Decimal` values, and stores them as text strings containing their decimal representation.

Parameters

- **`name`** – The name of this property.
- **`primary`** – A boolean indicating whether this property is a primary key.
- **`default`** – The initial value of this variable. The default behavior is for the value to stay undefined until it is set with `set`.
- **`default_factory`** – If specified, this will immediately be called to get the initial value.
- **`allow_none`** – A boolean indicating whether None should be allowed to be set as the value of this variable.
- **`validator`** – Validation function called whenever trying to set the variable to a non-db value. The function should look like validator(object, attr, value), where the first and second arguments are the result of validator_object_factory() (or None, if this parameter isn't provided) and the value of validator_attribute, respectively. When called, the function should raise an error if the value is unacceptable, or return the value to be used in place of the original value otherwise.
- **`kwargs`** – Other keyword arguments passed through when constructing the underlying variable.

`variable_class`

alias of *`storm.variables.DecimalVariable`*

class `storm.properties.`**`Bytes`**(*name=None*, *primary=False*, ***kwargs*)

Bases: *`storm.properties.SimpleProperty`*

Bytes property.

This accepts `bytes`, `buffer` (Python 2), or `memoryview` (Python 3) objects, and stores them as byte strings.

Deprecated aliases: `Chars`, *`RawStr`*.

Parameters

- **`name`** – The name of this property.
- **`primary`** – A boolean indicating whether this property is a primary key.
- **`default`** – The initial value of this variable. The default behavior is for the value to stay undefined until it is set with `set`.
- **`default_factory`** – If specified, this will immediately be called to get the initial value.
- **`allow_none`** – A boolean indicating whether None should be allowed to be set as the value of this variable.
- **`validator`** – Validation function called whenever trying to set the variable to a non-db value. The function should look like validator(object, attr, value), where the first and second arguments are the result of validator_object_factory() (or None, if this parameter isn't provided) and the value of validator_attribute, respectively. When called, the function should raise an error if the value is unacceptable, or return the value to be used in place of the original value otherwise.
- **`kwargs`** – Other keyword arguments passed through when constructing the underlying variable.

variable_class
alias of *storm.variables.BytesVariable*

storm.properties.**RawStr**
alias of *storm.properties.Bytes*

class storm.properties.**Unicode**(*name=None*, *primary=False*, ***kwargs*)
Bases: *storm.properties.SimpleProperty*

Unicode property.

This accepts `unicode` (Python 2) or `str` (Python 3) objects, and stores them as text strings. Note that it does not accept `str` objects on Python 2.

Parameters

- **name** – The name of this property.
- **primary** – A boolean indicating whether this property is a primary key.
- **default** – The initial value of this variable. The default behavior is for the value to stay undefined until it is set with `set`.
- **default_factory** – If specified, this will immediately be called to get the initial value.
- **allow_none** – A boolean indicating whether None should be allowed to be set as the value of this variable.
- **validator** – Validation function called whenever trying to set the variable to a non-db value. The function should look like validator(object, attr, value), where the first and second arguments are the result of validator_object_factory() (or None, if this parameter isn't provided) and the value of validator_attribute, respectively. When called, the function should raise an error if the value is unacceptable, or return the value to be used in place of the original value otherwise.
- **kwargs** – Other keyword arguments passed through when constructing the underlying variable.

variable_class
alias of *storm.variables.UnicodeVariable*

class storm.properties.**DateTime**(*name=None*, *primary=False*, ***kwargs*)
Bases: *storm.properties.SimpleProperty*

Date and time property.

This accepts aware `datetime.datetime` objects and stores them as timestamps; it also accepts integer or `float` objects, converting them using `datetime.utcfromtimestamp`. Note that it does not accept naive `datetime.datetime` objects (those that do not have timezone information).

Parameters

- **name** – The name of this property.
- **primary** – A boolean indicating whether this property is a primary key.
- **default** – The initial value of this variable. The default behavior is for the value to stay undefined until it is set with `set`.
- **default_factory** – If specified, this will immediately be called to get the initial value.
- **allow_none** – A boolean indicating whether None should be allowed to be set as the value of this variable.

- **`validator`** – Validation function called whenever trying to set the variable to a non-db value. The function should look like validator(object, attr, value), where the first and second arguments are the result of validator_object_factory() (or None, if this parameter isn't provided) and the value of validator_attribute, respectively. When called, the function should raise an error if the value is unacceptable, or return the value to be used in place of the original value otherwise.
- **`kwargs`** – Other keyword arguments passed through when constructing the underlying variable.

`variable_class`
alias of *`storm.variables.DateTimeVariable`*

class `storm.properties.`**`Date`**(*name=None*, *primary=False*, ***kwargs*)
Bases: *`storm.properties.SimpleProperty`*

Date property.

This accepts `datetime.date` objects and stores them as datestamps; it also accepts `datetime.datetime` objects, converting them using `datetime.datetime.date`.

Parameters

- **`name`** – The name of this property.
- **`primary`** – A boolean indicating whether this property is a primary key.
- **`default`** – The initial value of this variable. The default behavior is for the value to stay undefined until it is set with `set`.
- **`default_factory`** – If specified, this will immediately be called to get the initial value.
- **`allow_none`** – A boolean indicating whether None should be allowed to be set as the value of this variable.
- **`validator`** – Validation function called whenever trying to set the variable to a non-db value. The function should look like validator(object, attr, value), where the first and second arguments are the result of validator_object_factory() (or None, if this parameter isn't provided) and the value of validator_attribute, respectively. When called, the function should raise an error if the value is unacceptable, or return the value to be used in place of the original value otherwise.
- **`kwargs`** – Other keyword arguments passed through when constructing the underlying variable.

`variable_class`
alias of *`storm.variables.DateVariable`*

class `storm.properties.`**`Time`**(*name=None*, *primary=False*, ***kwargs*)
Bases: *`storm.properties.SimpleProperty`*

Time property.

This accepts `datetime.time` objects and stores them as datestamps; it also accepts `datetime.datetime` objects, converting them using `datetime.datetime.time`.

Parameters

- **`name`** – The name of this property.
- **`primary`** – A boolean indicating whether this property is a primary key.
- **`default`** – The initial value of this variable. The default behavior is for the value to stay undefined until it is set with `set`.

- **`default_factory`** – If specified, this will immediately be called to get the initial value.
- **`allow_none`** – A boolean indicating whether None should be allowed to be set as the value of this variable.
- **`validator`** – Validation function called whenever trying to set the variable to a non-db value. The function should look like validator(object, attr, value), where the first and second arguments are the result of validator_object_factory() (or None, if this parameter isn't provided) and the value of validator_attribute, respectively. When called, the function should raise an error if the value is unacceptable, or return the value to be used in place of the original value otherwise.
- **`kwargs`** – Other keyword arguments passed through when constructing the underlying variable.

`variable_class`
alias of *`storm.variables.TimeVariable`*

class `storm.properties.`**`TimeDelta`**(*name=None*, *primary=False*, ***kwargs*)
Bases: *`storm.properties.SimpleProperty`*

Time delta property.

This accepts `datetime.timedelta` objects and stores them as time intervals.

Parameters

- **`name`** – The name of this property.
- **`primary`** – A boolean indicating whether this property is a primary key.
- **`default`** – The initial value of this variable. The default behavior is for the value to stay undefined until it is set with `set`.
- **`default_factory`** – If specified, this will immediately be called to get the initial value.
- **`allow_none`** – A boolean indicating whether None should be allowed to be set as the value of this variable.
- **`validator`** – Validation function called whenever trying to set the variable to a non-db value. The function should look like validator(object, attr, value), where the first and second arguments are the result of validator_object_factory() (or None, if this parameter isn't provided) and the value of validator_attribute, respectively. When called, the function should raise an error if the value is unacceptable, or return the value to be used in place of the original value otherwise.
- **`kwargs`** – Other keyword arguments passed through when constructing the underlying variable.

`variable_class`
alias of *`storm.variables.TimeDeltaVariable`*

class `storm.properties.`**`UUID`**(*name=None*, *primary=False*, ***kwargs*)
Bases: *`storm.properties.SimpleProperty`*

UUID property.

This accepts `uuid.UUID` objects and stores them as their text representation.

Parameters

- **`name`** – The name of this property.
- **`primary`** – A boolean indicating whether this property is a primary key.

- **`default`** – The initial value of this variable. The default behavior is for the value to stay undefined until it is set with `set`.
- **`default_factory`** – If specified, this will immediately be called to get the initial value.
- **`allow_none`** – A boolean indicating whether None should be allowed to be set as the value of this variable.
- **`validator`** – Validation function called whenever trying to set the variable to a non-db value. The function should look like validator(object, attr, value), where the first and second arguments are the result of validator_object_factory() (or None, if this parameter isn't provided) and the value of validator_attribute, respectively. When called, the function should raise an error if the value is unacceptable, or return the value to be used in place of the original value otherwise.
- **`kwargs`** – Other keyword arguments passed through when constructing the underlying variable.

`variable_class`
alias of *`storm.variables.UUIDVariable`*

`class` `storm.properties.`**`Pickle`**(*name=None*, *primary=False*, ***kwargs*)
Bases: *`storm.properties.SimpleProperty`*

Pickle property.

This accepts any object that can be serialized using `pickle`, and stores it as a byte string containing its pickled representation.

Parameters

- **`name`** – The name of this property.
- **`primary`** – A boolean indicating whether this property is a primary key.
- **`default`** – The initial value of this variable. The default behavior is for the value to stay undefined until it is set with `set`.
- **`default_factory`** – If specified, this will immediately be called to get the initial value.
- **`allow_none`** – A boolean indicating whether None should be allowed to be set as the value of this variable.
- **`validator`** – Validation function called whenever trying to set the variable to a non-db value. The function should look like validator(object, attr, value), where the first and second arguments are the result of validator_object_factory() (or None, if this parameter isn't provided) and the value of validator_attribute, respectively. When called, the function should raise an error if the value is unacceptable, or return the value to be used in place of the original value otherwise.
- **`kwargs`** – Other keyword arguments passed through when constructing the underlying variable.

`variable_class`
alias of *`storm.variables.PickleVariable`*

`class` `storm.properties.`**`JSON`**(*name=None*, *primary=False*, ***kwargs*)
Bases: *`storm.properties.SimpleProperty`*

JSON property.

This accepts any object that can be serialized using `json`, and stores it as a text string containing its JSON representation.

Parameters

- **`name`** – The name of this property.
- **`primary`** – A boolean indicating whether this property is a primary key.
- **`default`** – The initial value of this variable. The default behavior is for the value to stay undefined until it is set with `set`.
- **`default_factory`** – If specified, this will immediately be called to get the initial value.
- **`allow_none`** – A boolean indicating whether None should be allowed to be set as the value of this variable.
- **`validator`** – Validation function called whenever trying to set the variable to a non-db value. The function should look like validator(object, attr, value), where the first and second arguments are the result of validator_object_factory() (or None, if this parameter isn't provided) and the value of validator_attribute, respectively. When called, the function should raise an error if the value is unacceptable, or return the value to be used in place of the original value otherwise.
- **`kwargs`** – Other keyword arguments passed through when constructing the underlying variable.

`variable_class`
alias of *`storm.variables.JSONVariable`*

`class` `storm.properties.`**`List`**(*name=None*, ***kwargs*)
Bases: *`storm.properties.SimpleProperty`*

List property.

This accepts iterable objects and stores them as a list where each element is an object of the given value type.

Parameters

- **`name`** – The name of this property.
- **`type`** – An instance of *`Property`* defining the type of each element of this list.
- **`default_factory`** – If specified, this will immediately be called to get the initial value.
- **`validator`** – Validation function called whenever trying to set the variable to a non-db value. The function should look like validator(object, attr, value), where the first and second arguments are the result of validator_object_factory() (or None, if this parameter isn't provided) and the value of validator_attribute, respectively. When called, the function should raise an error if the value is unacceptable, or return the value to be used in place of the original value otherwise.
- **`kwargs`** – Other keyword arguments passed through when constructing the underlying variable.

`variable_class`
alias of *`storm.variables.ListVariable`*

`class` `storm.properties.`**`Enum`**(*name=None*, *primary=False*, ***kwargs*)
Bases: *`storm.properties.SimpleProperty`*

Enumeration property, allowing used values to differ from stored ones.

For instance:

```
class Class(Storm):
    prop = Enum(map={"one": 1, "two": 2})

obj.prop = "one"
assert obj.prop == "one"

obj.prop = 1 # Raises error.
```

Another example:

```
class Class(Storm):
    prop = Enum(map={"one": 1, "two": 2}, set_map={"um": 1})

obj.prop = "um"
assert obj.prop is "one"

obj.prop = "one" # Raises error.
```

variable_class
alias of *storm.variables.EnumVariable*

class storm.properties.**PropertyRegistry**
Bases: `object`

An object which remembers the Storm properties specified on classes, and is able to translate names to these properties.

get (*name*, *namespace=None*)
Translate a property name path to the actual property.

This method accepts a property name like `"id"` or `"Class.id"` or `"module.path.Class.id"`, and tries to find a unique class/property with the given name.

When the `namespace` argument is given, the registry will be able to disambiguate names by choosing the one that is closer to the given namespace. For instance `get("Class.id", "a.b.c")` will choose `a.Class.id` rather than `d.Class.id`.

add_class (*cls*)
Register properties of `cls` so that they may be found by *get()*.

add_property (*cls*, *prop*, *attr_name*)
Register property of `cls` so that it may be found by *get()*.

clear ()
Clean up all properties in the registry.

Used by tests.

4.3.3 References

class storm.references.**Reference** (*local_key*, *remote_key*, *on_remote=False*)
Bases: `object`

Descriptor for one-to-one relationships.

This is typically used when the class that it is being defined on has a foreign key onto another table:

```
class OtherGuy(object):
    ...
    id = Int()

class MyGuy(object):
    ...
    other_guy_id = Int()
    other_guy = Reference(other_guy_id, OtherGuy.id)
```

but can also be used for backwards references, where OtherGuy's table has a foreign key onto the class that you want this property on:

```
class OtherGuy(object):
    ...
    my_guy_id = Int() # in the database, a foreign key to my_guy.id

class MyGuy(object):
    ...
    id = Int()
    other_guy = Reference(id, OtherGuy.my_guy_id, on_remote=True)
```

In both cases, `MyGuy().other_guy` will resolve to the `OtherGuy` instance which is linked to it. In the first case, it will be the `OtherGuy` instance whose `id` is equivalent to the `MyGuy`'s `other_guy_id`; in the second, it'll be the `OtherGuy` instance whose `my_guy_id` is equivalent to the `MyGuy`'s `id`.

Assigning to the property, for example with C{MyGuy().other_guy = OtherGuy()}, will link the objects and update either `MyGuy.other_guy_id` or `OtherGuy.my_guy_id` accordingly.

String references may be used in place of *`storm.expr.Column`* objects throughout, and are resolved to columns using `PropertyResolver`.

Create a Reference property.

Parameters

- **`local_key`** – The sibling column which is the foreign key onto `remote_key`. (unless `on_remote` is passed; see below).
- **`remote_key`** – The column on the referred-to object which will have the same value as that for `local_key` when resolved on an instance.
- **`on_remote`** – If specified, then the reference is backwards: It is the `remote_key` which is a foreign key onto `local_key`.

class `storm.references.`**`ReferenceSet`**(*local_key1*, *remote_key1*, *remote_key2=None*, *local_key2=None*, *order_by=None*)

Bases: `object`

Descriptor for many-to-one and many-to-many reference sets.

This is typically used when another class has a foreign key onto the class being defined, either directly (the many-to-one case) or via an intermediate table (the many-to-many case). For instance:

```
class Person(Storm):
    ...
    id = Int(primary=True)
    email_addresses = ReferenceSet("id", "EmailAddress.owner_id")

class EmailAddress(Storm):
    ...
```

(continues on next page)

(continued from previous page)

```
    owner_id = Int(name="owner", allow_none=False)
    owner = Reference(owner_id, "Person.id")

class TeamMembership(Storm):
    ...
    person_id = Int(name="person", allow_none=False)
    person = Reference(person_id, "Person.id")
    team_id = Int(name="team", allow_none=False)
    team = Reference(team_id, "Team.id")

class Team(Storm):
    ...
    id = Int(primary=True)
    members = ReferenceSet(
        "id", "TeamMembership.team_id",
        "TeamMembership.person_id", "Person.id",
        order_by="Person.name")
```

In this case, `Person().email_addresses` resolves to a `BoundReferenceSet` of all the email addresses linked to that person (a many-to-one relationship), while `Team().members` resolves to a `BoundIndirectReferenceSet` of all the members of that team (a many-to-many relationship). These can be used in a somewhat similar way to *`ResultSet`* objects.

String references may be used in place of *`storm.expr.Column`* objects throughout, and are resolved to columns using `PropertyResolver`.

Parameters

- **`local_key1`** – The sibling column which has the same value as that for `remote_key1` when resolved on an instance.
- **`remote_key1`** – The column on the referring object (in the case of a many-to-one relation) or on the intermediate table (in the case of a many-to-many relation) which is the foreign key onto `local_key1`.
- **`remote_key2`** – In the case of a many-to-many relation, the column on the intermediate table which is the foreign key onto `local_key2`.
- **`local_key2`** – In the case of a many-to-many relation, the column on the referred-to object which has the same value as `remote_key2` when resolved on an instance.
- **`order_by`** – If not `None`, order the resolved `BoundReferenceSet` or `BoundIndirectReferenceSet` by these columns, as in *`storm.store.ResultSet.order_by`*.

class `storm.references.`**`Proxy`**(*reference*, *remote_prop*)

Bases: *`storm.expr.ComparableExpr`*

Proxy exposes a referred object's column as a local column.

For example:

```
class Foo(object):
    bar_id = Int()
    bar = Reference(bar_id, Bar.id)
    bar_title = Proxy(bar, Bar.title)
```

For most uses, `Foo.bar_title` should behave as if it were a native property of `Foo`.

class RemoteProp

Bases: `object`

This descriptor will resolve and set the _remote_prop attribute when it's first used. It avoids having a test at every single place where the attribute is touched.

4.3.4 Variables

class `storm.variables.`**`LazyValue`**

Bases: `object`

Marker to be used as a base class on lazily evaluated values.

`storm.variables.`**`VariableFactory`**

alias of `functools.partial`

class `storm.variables.`**`Variable`**(*value=Undef*, *value_factory=Undef*, *from_db=False*, *allow_none=True*, *column=None*, *event=None*, *validator=None*, *validator_object_factory=None*, *validator_attribute=None*)

Bases: `object`

Basic representation of a database value in Python.

Variables

- **`column`** – The column this variable represents.
- ***`event`*** – The event system on which to broadcast events. If None, no events will be emitted.

Parameters

- **`value`** – The initial value of this variable. The default behavior is for the value to stay undefined until it is set with *`set`*.
- **`value_factory`** – If specified, this will immediately be called to get the initial value.
- **`from_db`** – A boolean value indicating where the initial value comes from, if `value` or `value_factory` are specified.
- **`allow_none`** – A boolean indicating whether None should be allowed to be set as the value of this variable.
- **`validator`** – Validation function called whenever trying to set the variable to a non-db value. The function should look like validator(object, attr, value), where the first and second arguments are the result of validator_object_factory() (or None, if this parameter isn't provided) and the value of validator_attribute, respectively. When called, the function should raise an error if the value is unacceptable, or return the value to be used in place of the original value otherwise.
- **`column`** (*`storm.expr.Column`*) – The column that this variable represents. It's used for reporting better error messages.
- **`event`** (*`storm.event.EventSystem`*) – The event system to broadcast messages with. If not specified, then no events will be broadcast.

`get_lazy`(*default=None*)

Get the current *`LazyValue`* without resolving its value.

Parameters **`default`** – If no *`LazyValue`* was previously specified, return this value. Defaults to None.

get(*default=None*, *to_db=False*)
Get the value, resolving it from a *LazyValue* if necessary.

If the current value is an instance of *LazyValue*, then the `resolve-lazy-value` event will be emitted, to give third parties the chance to resolve the lazy value to a real value.

Parameters

- **default** – Returned if no value has been set.
- **to_db** – A boolean flag indicating whether this value is destined for the database.

set(*value*, *from_db=False*)
Set a new value.

Generally this will be called when an attribute was set in Python, or data is being loaded from the database.

If the value is different from the previous value (or it is a *LazyValue*), then the `changed` event will be emitted.

Parameters

- **value** – The value to set. If this is an instance of *LazyValue*, then later calls to *get* will try to resolve the value.
- **from_db** – A boolean indicating whether this value has come from the database.

delete()
Delete the internal value.

If there was a value set, then emit the `changed` event.

is_defined()
Check whether there is currently a value.

Returns boolean indicating whether there is currently a value for this variable. Note that if a *LazyValue* was previously set, this returns False; it only returns True if there is currently a real value set.

has_changed()
Check whether the value has changed.

Returns boolean indicating whether the value has changed since the last call to *checkpoint*.

get_state()
Get the internal state of this object.

Returns A value which can later be passed to *set_state*.

set_state(*state*)
Set the internal state of this object.

Parameters **state** – A result from a previous call to *get_state*. The internal state of this variable will be set to the state of the variable which get_state was called on.

checkpoint()
"Checkpoint" the internal state.

See *has_changed*.

copy()
Make a new copy of this Variable with the same internal state.

parse_get(*value*, *to_db*)
Convert the internal value to an external value.

Get a representation of this value either for Python or for the database. This method is only intended to be overridden in subclasses, not called from external code.

Parameters

- **value** – The value to be converted.
- **to_db** – Whether or not this value is destined for the database.

parse_set(*value*, *from_db*)

Convert an external value to an internal value.

A value is being set either from Python code or from the database. Parse it into its internal representation. This method is only intended to be overridden in subclasses, not called from external code.

Parameters

- **value** – The value, either from Python code setting an attribute or from a column in a database.
- **from_db** – A boolean flag indicating whether this value is from the database.

class storm.variables.**BoolVariable**(*value=Undef*, *value_factory=Undef*, *from_db=False*, *allow_none=True*, *column=None*, *event=None*, *validator=None*, *validator_object_factory=None*, *validator_attribute=None*)

Bases: *storm.variables.Variable*

Parameters

- **value** – The initial value of this variable. The default behavior is for the value to stay undefined until it is set with `set`.
- **value_factory** – If specified, this will immediately be called to get the initial value.
- **from_db** – A boolean value indicating where the initial value comes from, if `value` or `value_factory` are specified.
- **allow_none** – A boolean indicating whether None should be allowed to be set as the value of this variable.
- **validator** – Validation function called whenever trying to set the variable to a non-db value. The function should look like validator(object, attr, value), where the first and second arguments are the result of validator_object_factory() (or None, if this parameter isn't provided) and the value of validator_attribute, respectively. When called, the function should raise an error if the value is unacceptable, or return the value to be used in place of the original value otherwise.
- **column** (*storm.expr.Column*) – The column that this variable represents. It's used for reporting better error messages.
- **event** (*storm.event.EventSystem*) – The event system to broadcast messages with. If not specified, then no events will be broadcast.

parse_set(*value*, *from_db*)

Convert an external value to an internal value.

A value is being set either from Python code or from the database. Parse it into its internal representation. This method is only intended to be overridden in subclasses, not called from external code.

Parameters

- **value** – The value, either from Python code setting an attribute or from a column in a database.

- **from_db** – A boolean flag indicating whether this value is from the database.

class storm.variables.**IntVariable**(*value=Undef*, *value_factory=Undef*, *from_db=False*, *allow_none=True*, *column=None*, *event=None*, *validator=None*, *validator_object_factory=None*, *validator_attribute=None*)

Bases: *storm.variables.Variable*

Parameters

- **value** – The initial value of this variable. The default behavior is for the value to stay undefined until it is set with `set`.
- **value_factory** – If specified, this will immediately be called to get the initial value.
- **from_db** – A boolean value indicating where the initial value comes from, if `value` or `value_factory` are specified.
- **allow_none** – A boolean indicating whether None should be allowed to be set as the value of this variable.
- **validator** – Validation function called whenever trying to set the variable to a non-db value. The function should look like validator(object, attr, value), where the first and second arguments are the result of validator_object_factory() (or None, if this parameter isn't provided) and the value of validator_attribute, respectively. When called, the function should raise an error if the value is unacceptable, or return the value to be used in place of the original value otherwise.
- **column** (*storm.expr.Column*) – The column that this variable represents. It's used for reporting better error messages.
- **event** (*storm.event.EventSystem*) – The event system to broadcast messages with. If not specified, then no events will be broadcast.

parse_set(*value*, *from_db*)

Convert an external value to an internal value.

A value is being set either from Python code or from the database. Parse it into its internal representation. This method is only intended to be overridden in subclasses, not called from external code.

Parameters

- **value** – The value, either from Python code setting an attribute or from a column in a database.
- **from_db** – A boolean flag indicating whether this value is from the database.

class storm.variables.**FloatVariable**(*value=Undef*, *value_factory=Undef*, *from_db=False*, *allow_none=True*, *column=None*, *event=None*, *validator=None*, *validator_object_factory=None*, *validator_attribute=None*)

Bases: *storm.variables.Variable*

Parameters

- **value** – The initial value of this variable. The default behavior is for the value to stay undefined until it is set with `set`.
- **value_factory** – If specified, this will immediately be called to get the initial value.
- **from_db** – A boolean value indicating where the initial value comes from, if `value` or `value_factory` are specified.

- **allow_none** – A boolean indicating whether None should be allowed to be set as the value of this variable.
- **validator** – Validation function called whenever trying to set the variable to a non-db value. The function should look like validator(object, attr, value), where the first and second arguments are the result of validator_object_factory() (or None, if this parameter isn't provided) and the value of validator_attribute, respectively. When called, the function should raise an error if the value is unacceptable, or return the value to be used in place of the original value otherwise.
- **column** (*storm.expr.Column*) – The column that this variable represents. It's used for reporting better error messages.
- **event** (*storm.event.EventSystem*) – The event system to broadcast messages with. If not specified, then no events will be broadcast.

parse_set (*value*, *from_db*)

Convert an external value to an internal value.

A value is being set either from Python code or from the database. Parse it into its internal representation. This method is only intended to be overridden in subclasses, not called from external code.

Parameters

- **value** – The value, either from Python code setting an attribute or from a column in a database.
- **from_db** – A boolean flag indicating whether this value is from the database.

class storm.variables.**DecimalVariable** (*value=Undef*, *value_factory=Undef*, *from_db=False*, *allow_none=True*, *column=None*, *event=None*, *validator=None*, *validator_object_factory=None*, *validator_attribute=None*)

Bases: *storm.variables.Variable*

Parameters

- **value** – The initial value of this variable. The default behavior is for the value to stay undefined until it is set with `set`.
- **value_factory** – If specified, this will immediately be called to get the initial value.
- **from_db** – A boolean value indicating where the initial value comes from, if `value` or `value_factory` are specified.
- **allow_none** – A boolean indicating whether None should be allowed to be set as the value of this variable.
- **validator** – Validation function called whenever trying to set the variable to a non-db value. The function should look like validator(object, attr, value), where the first and second arguments are the result of validator_object_factory() (or None, if this parameter isn't provided) and the value of validator_attribute, respectively. When called, the function should raise an error if the value is unacceptable, or return the value to be used in place of the original value otherwise.
- **column** (*storm.expr.Column*) – The column that this variable represents. It's used for reporting better error messages.
- **event** (*storm.event.EventSystem*) – The event system to broadcast messages with. If not specified, then no events will be broadcast.

static parse_set (*value*, *from_db*)

Convert an external value to an internal value.

A value is being set either from Python code or from the database. Parse it into its internal representation. This method is only intended to be overridden in subclasses, not called from external code.

Parameters

- **value** – The value, either from Python code setting an attribute or from a column in a database.
- **from_db** – A boolean flag indicating whether this value is from the database.

static parse_get(*value*, *to_db*)
Convert the internal value to an external value.

Get a representation of this value either for Python or for the database. This method is only intended to be overridden in subclasses, not called from external code.

Parameters

- **value** – The value to be converted.
- **to_db** – Whether or not this value is destined for the database.

class storm.variables.**BytesVariable**(*value=Undef*, *value_factory=Undef*, *from_db=False*, *allow_none=True*, *column=None*, *event=None*, *validator=None*, *validator_object_factory=None*, *validator_attribute=None*)
Bases: *storm.variables.Variable*

Parameters

- **value** – The initial value of this variable. The default behavior is for the value to stay undefined until it is set with `set`.
- **value_factory** – If specified, this will immediately be called to get the initial value.
- **from_db** – A boolean value indicating where the initial value comes from, if `value` or `value_factory` are specified.
- **allow_none** – A boolean indicating whether None should be allowed to be set as the value of this variable.
- **validator** – Validation function called whenever trying to set the variable to a non-db value. The function should look like validator(object, attr, value), where the first and second arguments are the result of validator_object_factory() (or None, if this parameter isn't provided) and the value of validator_attribute, respectively. When called, the function should raise an error if the value is unacceptable, or return the value to be used in place of the original value otherwise.
- **column** (*storm.expr.Column*) – The column that this variable represents. It's used for reporting better error messages.
- **event** (*storm.event.EventSystem*) – The event system to broadcast messages with. If not specified, then no events will be broadcast.

parse_set(*value*, *from_db*)
Convert an external value to an internal value.

A value is being set either from Python code or from the database. Parse it into its internal representation. This method is only intended to be overridden in subclasses, not called from external code.

Parameters

- **value** – The value, either from Python code setting an attribute or from a column in a database.

- **from_db** – A boolean flag indicating whether this value is from the database.

storm.variables.**RawStrVariable**

alias of *storm.variables.BytesVariable*

class storm.variables.**UnicodeVariable**(*value=Undef*, *value_factory=Undef*, *from_db=False*, *allow_none=True*, *column=None*, *event=None*, *validator=None*, *validator_object_factory=None*, *validator_attribute=None*)

Bases: *storm.variables.Variable*

Parameters

- **value** – The initial value of this variable. The default behavior is for the value to stay undefined until it is set with set.
- **value_factory** – If specified, this will immediately be called to get the initial value.
- **from_db** – A boolean value indicating where the initial value comes from, if value or value_factory are specified.
- **allow_none** – A boolean indicating whether None should be allowed to be set as the value of this variable.
- **validator** – Validation function called whenever trying to set the variable to a non-db value. The function should look like validator(object, attr, value), where the first and second arguments are the result of validator_object_factory() (or None, if this parameter isn't provided) and the value of validator_attribute, respectively. When called, the function should raise an error if the value is unacceptable, or return the value to be used in place of the original value otherwise.
- **column** (*storm.expr.Column*) – The column that this variable represents. It's used for reporting better error messages.
- **event** (*storm.event.EventSystem*) – The event system to broadcast messages with. If not specified, then no events will be broadcast.

parse_set(*value*, *from_db*)

Convert an external value to an internal value.

A value is being set either from Python code or from the database. Parse it into its internal representation. This method is only intended to be overridden in subclasses, not called from external code.

Parameters

- **value** – The value, either from Python code setting an attribute or from a column in a database.
- **from_db** – A boolean flag indicating whether this value is from the database.

class storm.variables.**DateTimeVariable**(**args*, ***kwargs*)

Bases: *storm.variables.Variable*

parse_set(*value*, *from_db*)

Convert an external value to an internal value.

A value is being set either from Python code or from the database. Parse it into its internal representation. This method is only intended to be overridden in subclasses, not called from external code.

Parameters

- **value** – The value, either from Python code setting an attribute or from a column in a database.
- **from_db** – A boolean flag indicating whether this value is from the database.

class storm.variables.**DateVariable**(*value=Undef*, *value_factory=Undef*, *from_db=False*, *allow_none=True*, *column=None*, *event=None*, *validator=None*, *validator_object_factory=None*, *validator_attribute=None*)

Bases: `storm.variables.Variable`

Parameters

- **value** – The initial value of this variable. The default behavior is for the value to stay undefined until it is set with `set`.
- **value_factory** – If specified, this will immediately be called to get the initial value.
- **from_db** – A boolean value indicating where the initial value comes from, if `value` or `value_factory` are specified.
- **allow_none** – A boolean indicating whether None should be allowed to be set as the value of this variable.
- **validator** – Validation function called whenever trying to set the variable to a non-db value. The function should look like validator(object, attr, value), where the first and second arguments are the result of validator_object_factory() (or None, if this parameter isn't provided) and the value of validator_attribute, respectively. When called, the function should raise an error if the value is unacceptable, or return the value to be used in place of the original value otherwise.
- **column** (`storm.expr.Column`) – The column that this variable represents. It's used for reporting better error messages.
- **event** (`storm.event.EventSystem`) – The event system to broadcast messages with. If not specified, then no events will be broadcast.

parse_set(*value*, *from_db*)

Convert an external value to an internal value.

A value is being set either from Python code or from the database. Parse it into its internal representation. This method is only intended to be overridden in subclasses, not called from external code.

Parameters

- **value** – The value, either from Python code setting an attribute or from a column in a database.
- **from_db** – A boolean flag indicating whether this value is from the database.

class storm.variables.**TimeVariable**(*value=Undef*, *value_factory=Undef*, *from_db=False*, *allow_none=True*, *column=None*, *event=None*, *validator=None*, *validator_object_factory=None*, *validator_attribute=None*)

Bases: `storm.variables.Variable`

Parameters

- **value** – The initial value of this variable. The default behavior is for the value to stay undefined until it is set with `set`.
- **value_factory** – If specified, this will immediately be called to get the initial value.
- **from_db** – A boolean value indicating where the initial value comes from, if `value` or `value_factory` are specified.
- **allow_none** – A boolean indicating whether None should be allowed to be set as the value of this variable.

- **validator** – Validation function called whenever trying to set the variable to a non-db value. The function should look like validator(object, attr, value), where the first and second arguments are the result of validator_object_factory() (or None, if this parameter isn't provided) and the value of validator_attribute, respectively. When called, the function should raise an error if the value is unacceptable, or return the value to be used in place of the original value otherwise.
- **column** (`storm.expr.Column`) – The column that this variable represents. It's used for reporting better error messages.
- **event** (`storm.event.EventSystem`) – The event system to broadcast messages with. If not specified, then no events will be broadcast.

parse_set (*value*, *from_db*)

Convert an external value to an internal value.

A value is being set either from Python code or from the database. Parse it into its internal representation. This method is only intended to be overridden in subclasses, not called from external code.

Parameters

- **value** – The value, either from Python code setting an attribute or from a column in a database.
- **from_db** – A boolean flag indicating whether this value is from the database.

class storm.variables.**TimeDeltaVariable** (*value=Undef*, *value_factory=Undef*, *from_db=False*, *allow_none=True*, *column=None*, *event=None*, *validator=None*, *validator_object_factory=None*, *validator_attribute=None*)

Bases: `storm.variables.Variable`

Parameters

- **value** – The initial value of this variable. The default behavior is for the value to stay undefined until it is set with `set`.
- **value_factory** – If specified, this will immediately be called to get the initial value.
- **from_db** – A boolean value indicating where the initial value comes from, if `value` or `value_factory` are specified.
- **allow_none** – A boolean indicating whether None should be allowed to be set as the value of this variable.
- **validator** – Validation function called whenever trying to set the variable to a non-db value. The function should look like validator(object, attr, value), where the first and second arguments are the result of validator_object_factory() (or None, if this parameter isn't provided) and the value of validator_attribute, respectively. When called, the function should raise an error if the value is unacceptable, or return the value to be used in place of the original value otherwise.
- **column** (`storm.expr.Column`) – The column that this variable represents. It's used for reporting better error messages.
- **event** (`storm.event.EventSystem`) – The event system to broadcast messages with. If not specified, then no events will be broadcast.

parse_set (*value*, *from_db*)

Convert an external value to an internal value.

A value is being set either from Python code or from the database. Parse it into its internal representation. This method is only intended to be overridden in subclasses, not called from external code.

Parameters

- **`value`** – The value, either from Python code setting an attribute or from a column in a database.
- **`from_db`** – A boolean flag indicating whether this value is from the database.

class `storm.variables.`**`UUIDVariable`**(*value=Undef*, *value_factory=Undef*, *from_db=False*, *allow_none=True*, *column=None*, *event=None*, *validator=None*, *validator_object_factory=None*, *validator_attribute=None*)

Bases: *`storm.variables.Variable`*

Parameters

- **`value`** – The initial value of this variable. The default behavior is for the value to stay undefined until it is set with `set`.
- **`value_factory`** – If specified, this will immediately be called to get the initial value.
- **`from_db`** – A boolean value indicating where the initial value comes from, if `value` or `value_factory` are specified.
- **`allow_none`** – A boolean indicating whether None should be allowed to be set as the value of this variable.
- **`validator`** – Validation function called whenever trying to set the variable to a non-db value. The function should look like validator(object, attr, value), where the first and second arguments are the result of validator_object_factory() (or None, if this parameter isn't provided) and the value of validator_attribute, respectively. When called, the function should raise an error if the value is unacceptable, or return the value to be used in place of the original value otherwise.
- **`column`** (*`storm.expr.Column`*) – The column that this variable represents. It's used for reporting better error messages.
- **`event`** (*`storm.event.EventSystem`*) – The event system to broadcast messages with. If not specified, then no events will be broadcast.

`parse_set`(*value*, *from_db*)

Convert an external value to an internal value.

A value is being set either from Python code or from the database. Parse it into its internal representation. This method is only intended to be overridden in subclasses, not called from external code.

Parameters

- **`value`** – The value, either from Python code setting an attribute or from a column in a database.
- **`from_db`** – A boolean flag indicating whether this value is from the database.

`parse_get`(*value*, *to_db*)

Convert the internal value to an external value.

Get a representation of this value either for Python or for the database. This method is only intended to be overridden in subclasses, not called from external code.

Parameters

- **`value`** – The value to be converted.

- **to_db** – Whether or not this value is destined for the database.

class storm.variables.**EnumVariable**(*get_map*, *set_map*, **args*, ***kwargs*)

Bases: storm.variables.Variable

parse_set(*value*, *from_db*)

Convert an external value to an internal value.

A value is being set either from Python code or from the database. Parse it into its internal representation. This method is only intended to be overridden in subclasses, not called from external code.

Parameters

- **value** – The value, either from Python code setting an attribute or from a column in a database.
- **from_db** – A boolean flag indicating whether this value is from the database.

parse_get(*value*, *to_db*)

Convert the internal value to an external value.

Get a representation of this value either for Python or for the database. This method is only intended to be overridden in subclasses, not called from external code.

Parameters

- **value** – The value to be converted.
- **to_db** – Whether or not this value is destined for the database.

class storm.variables.**PickleVariable**(**args*, ***kwargs*)

Bases: storm.variables.EncodedValueVariable

class storm.variables.**JSONVariable**(**args*, ***kwargs*)

Bases: storm.variables.EncodedValueVariable

class storm.variables.**ListVariable**(*item_factory*, **args*, ***kwargs*)

Bases: storm.variables.MutableValueVariable

parse_set(*value*, *from_db*)

Convert an external value to an internal value.

A value is being set either from Python code or from the database. Parse it into its internal representation. This method is only intended to be overridden in subclasses, not called from external code.

Parameters

- **value** – The value, either from Python code setting an attribute or from a column in a database.
- **from_db** – A boolean flag indicating whether this value is from the database.

parse_get(*value*, *to_db*)

Convert the internal value to an external value.

Get a representation of this value either for Python or for the database. This method is only intended to be overridden in subclasses, not called from external code.

Parameters

- **value** – The value to be converted.
- **to_db** – Whether or not this value is destined for the database.

get_state()

Get the internal state of this object.

Returns A value which can later be passed to *set_state*.

set_state(*state*)
Set the internal state of this object.

Parameters state – A result from a previous call to *get_state*. The internal state of this variable will be set to the state of the variable which get_state was called on.

4.3.5 SQLObject emulation

A SQLObject emulation layer for Storm.

SQLObjectBase is the central point of compatibility.

storm.sqlobject.**DESC**
alias of *storm.expr.Desc*

storm.sqlobject.**AND**
alias of *storm.expr.And*

storm.sqlobject.**OR**
alias of *storm.expr.Or*

storm.sqlobject.**NOT**
alias of *storm.expr.Not*

storm.sqlobject.**IN**
alias of *storm.expr.In*

storm.sqlobject.**LIKE**
alias of *storm.expr.Like*

storm.sqlobject.**SQLConstant**
alias of *storm.expr.SQL*

storm.sqlobject.**SQLObjectMoreThanOneResultError**
alias of *storm.exceptions.NotOneError*

exception storm.sqlobject.**SQLObjectNotFound**
Bases: *storm.exceptions.StormError*

class storm.sqlobject.**SQLObjectBase**(**args*, ***kwargs*)
Bases: *storm.base.Storm*

The root class of all SQLObject-emulating classes in your application.

The general strategy for using Storm's SQLObject emulation layer is to create an application-specific subclass of SQLObjectBase (probably named "SQLObject") that provides an implementation of _get_store to return an instance of *storm.store.Store*. It may even be implemented as returning a global `Store` instance. Then all database classes should subclass that class.

class storm.sqlobject.**SQLObjectResultSet**(*cls*, *clause=None*, *clauseTables=None*, *orderBy=None*, *limit=None*, *distinct=None*, *prejoins=None*, *prejoinClauseTables=None*, *selectAlso=None*, *by={}*, *prepared_result_set=None*, *slice=None*)
Bases: `object`

SQLObject-equivalent of the ResultSet class in Storm.

Storm handles joins in the Store interface, while SQLObject does that in the result one. To offer support for prejoins, we can't simply wrap our ResultSet instance, and instead have to postpone the actual find until the very last moment.

is_empty()
Return `True` if this result set doesn't contain any results.

class storm.sqlobject.**StringCol**(*dbName=None*, *notNull=False*, *default=Undef*, *alternateID=None*, *unique=<object object>*, *name=<object object>*, *alternateMethodName=None*, *length=<object object>*, *immutable=None*, *storm_validator=None*)
Bases: `storm.sqlobject.PropertyAdapter`, `storm.sqlobject.AutoUnicode`

class storm.sqlobject.**IntCol**(*dbName=None*, *notNull=False*, *default=Undef*, *alternateID=None*, *unique=<object object>*, *name=<object object>*, *alternateMethodName=None*, *length=<object object>*, *immutable=None*, *storm_validator=None*)
Bases: `storm.sqlobject.PropertyAdapter`, *`storm.properties.Int`*

class storm.sqlobject.**BoolCol**(*dbName=None*, *notNull=False*, *default=Undef*, *alternateID=None*, *unique=<object object>*, *name=<object object>*, *alternateMethodName=None*, *length=<object object>*, *immutable=None*, *storm_validator=None*)
Bases: `storm.sqlobject.PropertyAdapter`, *`storm.properties.Bool`*

class storm.sqlobject.**FloatCol**(*dbName=None*, *notNull=False*, *default=Undef*, *alternateID=None*, *unique=<object object>*, *name=<object object>*, *alternateMethodName=None*, *length=<object object>*, *immutable=None*, *storm_validator=None*)
Bases: `storm.sqlobject.PropertyAdapter`, *`storm.properties.Float`*

class storm.sqlobject.**UtcDateTimeCol**(*dbName=None*, *notNull=False*, *default=Undef*, *alternateID=None*, *unique=<object object>*, *name=<object object>*, *alternateMethodName=None*, *length=<object object>*, *immutable=None*, *storm_validator=None*)
Bases: `storm.sqlobject.PropertyAdapter`, *`storm.properties.DateTime`*

class storm.sqlobject.**DateCol**(*dbName=None*, *notNull=False*, *default=Undef*, *alternateID=None*, *unique=<object object>*, *name=<object object>*, *alternateMethodName=None*, *length=<object object>*, *immutable=None*, *storm_validator=None*)
Bases: `storm.sqlobject.PropertyAdapter`, *`storm.properties.Date`*

class storm.sqlobject.**IntervalCol**(*dbName=None*, *notNull=False*, *default=Undef*, *alternateID=None*, *unique=<object object>*, *name=<object object>*, *alternateMethodName=None*, *length=<object object>*, *immutable=None*, *storm_validator=None*)
Bases: `storm.sqlobject.PropertyAdapter`, *`storm.properties.TimeDelta`*

class storm.sqlobject.**SQLMultipleJoin**(*otherClass=None*, *joinColumn=None*, *intermediateTable=None*, *otherColumn=None*, *orderBy=None*, *prejoins=None*)
Bases: *`storm.references.ReferenceSet`*

storm.sqlobject.**SQLRelatedJoin**
alias of *`storm.sqlobject.SQLMultipleJoin`*

class storm.sqlobject.**SingleJoin**(*otherClass*, *joinColumn*, *prejoins=<object object>*)
Bases: *`storm.references.Reference`*

class storm.sqlobject.**CONTAINSSTRING**(*expr*, *string*)
Bases: *`storm.expr.Like`*

4.4 Expressions

class storm.expr.**Compile**(*parent=None*)

Bases: object

Compiler based on the concept of generic functions.

when(**types*)

Decorator to include a type handler in this compiler.

Use this as:

```
>>> @compile.when(TypeA, TypeB)
>>> def compile_type_a_or_b(compile, expr, state):
>>>     ...
>>>     return "THE COMPILED SQL STATEMENT"
```

add_reserved_words(*words*)

Include words to be considered reserved and thus escaped.

Reserved words are escaped during compilation when they're seen in a SQLToken expression.

create_child()

Create a new instance of *Compile* which inherits from this one.

This is most commonly used to customize a compiler for database-specific compilation strategies.

class storm.expr.**CompilePython**(*parent=None*)

Bases: *storm.expr.Compile*

class storm.expr.**State**

Bases: object

All the data necessary during compilation of an expression.

Variables

- **aliases** – Dict of *Column* instances to *Alias* instances, specifying how columns should be compiled as aliases in very specific situations. This is typically used to work around strange deficiencies in various databases.
- **auto_tables** – The list of all implicitly-used tables. e.g., in store.find(Foo, Foo.attr==Bar.id), the tables of Bar and Foo are implicitly used because columns in them are referenced. This is used when building tables.
- **join_tables** – If not None, when Join expressions are compiled, tables seen will be added to this set. This acts as a blacklist against auto_tables when compiling Joins, because the generated statements should not refer to the table twice.
- **context** – an instance of *Context*, specifying the context of the expression currently being compiled.
- **precedence** – Current precedence, automatically set and restored by the compiler. If an inner precedence is lower than an outer precedence, parenthesis around the inner expression are automatically emitted.

push(*attr, new_value=Undef*)

Set an attribute in a way that can later be reverted with *pop*.

pop()

Revert the topmost *push*.

class storm.expr.**Context**(*name*)

Bases: `object`

An object used to specify the nature of expected SQL expressions being compiled in a given context.

class storm.expr.**Expr**

Bases: *storm.variables.LazyValue*

class storm.expr.**ComparableExpr**

Bases: *storm.expr.Expr*, `storm.expr.Comparable`

class storm.expr.**BinaryExpr**(*expr1*, *expr2*)

Bases: *storm.expr.ComparableExpr*

class storm.expr.**CompoundExpr**(**exprs*)

Bases: *storm.expr.ComparableExpr*

storm.expr.**build_tables**(*compile*, *tables*, *default_tables*, *state*)

Compile provided tables.

Tables will be built from either `tables`, `state.auto_tables`, or `default_tables`. If `tables` is not `Undef`, it will be used. If `tables` is `Undef` and `state.auto_tables` is available, that's used instead. If neither `tables` nor `state.auto_tables` are available, `default_tables` is tried as a last resort. If none of them are available, `NoTableError` is raised.

class storm.expr.**Select**(*columns*, *where=Undef*, *tables=Undef*, *default_tables=Undef*, *order_by=Undef*, *group_by=Undef*, *limit=Undef*, *offset=Undef*, *distinct=False*, *having=Undef*)

Bases: *storm.expr.Expr*

class storm.expr.**Insert**(*map*, *table=Undef*, *default_table=Undef*, *primary_columns=Undef*, *primary_variables=Undef*, *values=Undef*)

Bases: *storm.expr.Expr*

Expression representing an insert statement.

Variables

- **map** – Dictionary mapping columns to values, or a sequence of columns for a bulk insert.
- **table** – Table where the row should be inserted.
- **default_table** – Table to use if no table is explicitly provided, and no tables may be inferred from provided columns.
- **primary_columns** – Tuple of columns forming the primary key of the table where the row will be inserted. This is a hint used by backends to process the insertion of rows.
- **primary_variables** – Tuple of variables with values for the primary key of the table where the row will be inserted. This is a hint used by backends to process the insertion of rows.
- ***values*** – Expression or sequence of tuples of values for bulk insertion.

class storm.expr.**Update**(*map*, *where=Undef*, *table=Undef*, *default_table=Undef*, *primary_columns=Undef*)

Bases: *storm.expr.Expr*

class storm.expr.**Delete**(*where=Undef*, *table=Undef*, *default_table=Undef*)

Bases: *storm.expr.Expr*

class storm.expr.**Column**(*name=Undef*, *table=Undef*, *primary=False*, *variable_factory=None*)

Bases: *storm.expr.ComparableExpr*

Representation of a column in some table.

Variables

- ***name*** – Column name.
- **table** – Column table (maybe another expression).
- **primary** – Integer representing the primary key position of this column, or 0 if it's not a primary key. May be provided as a bool.
- **variable_factory** – Factory producing `Variable` instances typed according to this column.

class `storm.expr.`**Alias**(*expr*, *name=Undef*)

Bases: *storm.expr.ComparableExpr*

A representation of "AS" alias clauses. e.g., SELECT foo AS bar.

Create alias of `expr` AS `name`.

If `name` is not given, then a name will automatically be generated.

class `storm.expr.`**FromExpr**

Bases: *storm.expr.Expr*

class `storm.expr.`**Table**(*name*)

Bases: *storm.expr.FromExpr*

class `storm.expr.`**JoinExpr**(*arg1*, *arg2=Undef*, *on=Undef*)

Bases: *storm.expr.FromExpr*

class `storm.expr.`**Join**(*arg1*, *arg2=Undef*, *on=Undef*)

Bases: *storm.expr.JoinExpr*

class `storm.expr.`**LeftJoin**(*arg1*, *arg2=Undef*, *on=Undef*)

Bases: *storm.expr.JoinExpr*

class `storm.expr.`**RightJoin**(*arg1*, *arg2=Undef*, *on=Undef*)

Bases: *storm.expr.JoinExpr*

class `storm.expr.`**NaturalJoin**(*arg1*, *arg2=Undef*, *on=Undef*)

Bases: *storm.expr.JoinExpr*

class `storm.expr.`**NaturalLeftJoin**(*arg1*, *arg2=Undef*, *on=Undef*)

Bases: *storm.expr.JoinExpr*

class `storm.expr.`**NaturalRightJoin**(*arg1*, *arg2=Undef*, *on=Undef*)

Bases: *storm.expr.JoinExpr*

class `storm.expr.`**Distinct**(*expr*)

Bases: *storm.expr.Expr*

Add the 'DISTINCT' prefix to an expression.

class `storm.expr.`**BinaryOper**(*expr1*, *expr2*)

Bases: *storm.expr.BinaryExpr*

class `storm.expr.`**NonAssocBinaryOper**(*expr1*, *expr2*)

Bases: *storm.expr.BinaryOper*

class `storm.expr.`**CompoundOper**(**exprs*)

Bases: *storm.expr.CompoundExpr*

class `storm.expr.`**Eq**(*expr1*, *expr2*)

Bases: *storm.expr.BinaryOper*

class storm.expr.**Ne**(*expr1*, *expr2*)
Bases: *storm.expr.BinaryOper*

class storm.expr.**Gt**(*expr1*, *expr2*)
Bases: *storm.expr.BinaryOper*

class storm.expr.**Ge**(*expr1*, *expr2*)
Bases: *storm.expr.BinaryOper*

class storm.expr.**Lt**(*expr1*, *expr2*)
Bases: *storm.expr.BinaryOper*

class storm.expr.**Le**(*expr1*, *expr2*)
Bases: *storm.expr.BinaryOper*

class storm.expr.**RShift**(*expr1*, *expr2*)
Bases: *storm.expr.BinaryOper*

class storm.expr.**LShift**(*expr1*, *expr2*)
Bases: *storm.expr.BinaryOper*

class storm.expr.**Like**(*expr1*, *expr2*, *escape=Undef*, *case_sensitive=None*)
Bases: *storm.expr.BinaryOper*

class storm.expr.**In**(*expr1*, *expr2*)
Bases: *storm.expr.BinaryOper*

class storm.expr.**Add**(**exprs*)
Bases: *storm.expr.CompoundOper*

class storm.expr.**Sub**(*expr1*, *expr2*)
Bases: *storm.expr.NonAssocBinaryOper*

class storm.expr.**Mul**(**exprs*)
Bases: *storm.expr.CompoundOper*

class storm.expr.**Div**(*expr1*, *expr2*)
Bases: *storm.expr.NonAssocBinaryOper*

class storm.expr.**Mod**(*expr1*, *expr2*)
Bases: *storm.expr.NonAssocBinaryOper*

class storm.expr.**And**(**exprs*)
Bases: *storm.expr.CompoundOper*

class storm.expr.**Or**(**exprs*)
Bases: *storm.expr.CompoundOper*

class storm.expr.**SetExpr**(**exprs*, ***kwargs*)
Bases: *storm.expr.Expr*

class storm.expr.**Union**(**exprs*, ***kwargs*)
Bases: *storm.expr.SetExpr*

class storm.expr.**Except**(**exprs*, ***kwargs*)
Bases: *storm.expr.SetExpr*

class storm.expr.**Intersect**(**exprs*, ***kwargs*)
Bases: *storm.expr.SetExpr*

class storm.expr.**FuncExpr**
Bases: *storm.expr.ComparableExpr*

class storm.expr.**Count**(*column=Undef*, *distinct=False*)
Bases: *storm.expr.FuncExpr*

class storm.expr.**Func**(*name*, **args*)
Bases: *storm.expr.FuncExpr*

name
str(object=") -> str str(bytes_or_buffer[, encoding[, errors]]) -> str

Create a new string object from the given object. If encoding or errors is specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler. Otherwise, returns the result of object.__str__() (if defined) or repr(object). encoding defaults to sys.getdefaultencoding(). errors defaults to 'strict'.

class storm.expr.**NamedFunc**(**args*)
Bases: *storm.expr.FuncExpr*

class storm.expr.**Max**(**args*)
Bases: *storm.expr.NamedFunc*

class storm.expr.**Min**(**args*)
Bases: *storm.expr.NamedFunc*

class storm.expr.**Avg**(**args*)
Bases: *storm.expr.NamedFunc*

class storm.expr.**Sum**(**args*)
Bases: *storm.expr.NamedFunc*

class storm.expr.**Lower**(**args*)
Bases: *storm.expr.NamedFunc*

class storm.expr.**Upper**(**args*)
Bases: *storm.expr.NamedFunc*

class storm.expr.**Coalesce**(**args*)
Bases: *storm.expr.NamedFunc*

class storm.expr.**Row**(**args*)
Bases: *storm.expr.NamedFunc*

class storm.expr.**Cast**(*column*, *type*)
Bases: *storm.expr.FuncExpr*

A representation of `CAST` clauses. e.g., `CAST(bar AS TEXT)`.

Create a cast of `column` as `type`.

storm.expr.**compile_cast**(*compile*, *cast*, *state*)
Compile *Cast* expressions.

class storm.expr.**PrefixExpr**(*expr*)
Bases: *storm.expr.Expr*

class storm.expr.**SuffixExpr**(*expr*)
Bases: *storm.expr.Expr*

class storm.expr.**Not**(*expr*)
Bases: *storm.expr.PrefixExpr*

class storm.expr.**Exists**(*expr*)
Bases: *storm.expr.PrefixExpr*

class storm.expr.**Neg**(*expr*)
Bases: *storm.expr.PrefixExpr*

class storm.expr.**Asc**(*expr*)
Bases: *storm.expr.SuffixExpr*

class storm.expr.**Desc**(*expr*)
Bases: *storm.expr.SuffixExpr*

class storm.expr.**SQLRaw**
Bases: str

Subtype to mark a string as something that shouldn't be compiled.

This is handled internally by the compiler.

class storm.expr.**SQLToken**
Bases: str

Marker for strings that should be considered as a single SQL token.

These strings will be quoted, when needed.

storm.expr.**is_safe_token**()
Matches zero or more characters at the beginning of the string.

class storm.expr.**SQL**(*expr*, *params=Undef*, *tables=Undef*)
Bases: *storm.expr.ComparableExpr*

class storm.expr.**Sequence**(*name*)
Bases: *storm.expr.Expr*

Expression representing auto-incrementing support from the database.

This should be translated into the *next* value of the named auto-incrementing sequence. There's no standard way to compile a sequence, since it's very database-dependent.

This may be used as follows:

```
class Class(object):
    (...)
    id = Int(default=Sequence("my_sequence_name"))
```

class storm.expr.**AutoTables**(*expr*, *tables*, *replace=False*)
Bases: *storm.expr.Expr*

This class will inject one or more entries in state.auto_tables.

If the constructor is passed replace=True, it will also discard any auto_table entries injected by compiling the given expression.

4.5 Databases

Basic database interfacing mechanisms for Storm.

This is the common code for database support; specific databases are supported in modules in `storm.databases`.

class storm.database.**Result**(*connection*, *raw_cursor*)
Bases: object

A representation of the results from a single SQL statement.

close()
: Close the underlying raw cursor, if it hasn't already been closed.

get_one()
: Fetch one result from the cursor.

  The result will be converted to an appropriate format via *from_database*.

  Raises ***DisconnectionError*** – Raised when the connection is lost. Reconnection happens automatically on rollback.

  Returns A converted row or None, if no data is left.

get_all()
: Fetch all results from the cursor.

  The results will be converted to an appropriate format via *from_database*.

  Raises ***DisconnectionError*** – Raised when the connection is lost. Reconnection happens automatically on rollback.

rowcount
: See PEP 249 for further details on rowcount.

  Returns the number of affected rows, or None if the database backend does not provide this information. Return value is undefined if all results have not yet been retrieved.

get_insert_identity(*primary_columns*, *primary_variables*)
: Get a query which will return the row that was just inserted.

  This must be overridden in database-specific subclasses.

  Return type *storm.expr.Expr*

static set_variable(*variable*, *value*)
: Set the given variable's value from the database.

static from_database(*row*)
: Convert a row fetched from the database to an agnostic format.

  This method is intended to be overridden in subclasses, but not called externally.

  If there are any peculiarities in the datatypes returned from a database backend, this method should be overridden in the backend subclass to convert them.

class storm.database.**Connection**(*database*, *event=None*)
: Bases: object

  A connection to a database.

  Variables

  - ***result_factory*** – A callable which takes this *Connection* and the backend cursor and returns an instance of *Result*.
  - **param_mark** – The dbapi paramstyle that the database backend expects.
  - **compile** – The compiler to use for connections of this type.

result_factory
: alias of *Result*

block_access()
: Block access to the connection.

Attempts to execute statements or commit a transaction will result in a `ConnectionBlockedError` exception. Rollbacks are permitted as that operation is often used in case of failures.

unblock_access()
Unblock access to the connection.

execute(*statement*, *params=None*, *noresult=False*)
Execute a statement with the given parameters.

Parameters

- **statement** (`Expr` or `str`) – The statement to execute. It will be compiled if necessary.
- **noresult** – If True, no result will be returned.

Raises

- ***ConnectionBlockedError*** – Raised if access to the connection has been blocked with *block_access*.
- ***DisconnectionError*** – Raised when the connection is lost. Reconnection happens automatically on rollback.

Returns The result of `self.result_factory`, or None if `noresult` is True.

close()
Close the connection if it is not already closed.

begin(*xid*)
Begin a two-phase transaction.

prepare()
Run the prepare phase of a two-phase transaction.

commit(*xid=None*)
Commit the connection.

Parameters xid – Optionally the `Xid` of a previously prepared transaction to commit. This form should be called outside of a transaction, and is intended for use in recovery.

Raises

- ***ConnectionBlockedError*** – Raised if access to the connection has been blocked with *block_access*.
- ***DisconnectionError*** – Raised when the connection is lost. Reconnection happens automatically on rollback.

recover()
Return a list of `Xid`s representing pending transactions.

rollback(*xid=None*)
Rollback the connection.

Parameters xid – Optionally the `Xid` of a previously prepared transaction to rollback. This form should be called outside of a transaction, and is intended for use in recovery.

static to_database(*params*)
Convert some parameters into values acceptable to a database backend.

It is acceptable to override this method in subclasses, but it is not intended to be used externally.

This delegates conversion to any `Variables` in the parameter list, and passes through all other values untouched.

build_raw_cursor()
Get a new dbapi cursor object.

It is acceptable to override this method in subclasses, but it is not intended to be called externally.

raw_execute(*statement*, *params=None*)
Execute a raw statement with the given parameters.

It's acceptable to override this method in subclasses, but it is not intended to be called externally.

If the global `DEBUG` is True, the statement will be printed to standard out.

Returns The dbapi cursor object, as fetched from *build_raw_cursor*.

is_disconnection_error(*exc*, *extra_disconnection_errors=()*)
Check whether an exception represents a database disconnection.

This should be overridden by backends to detect whichever exception values are used to represent this condition.

preset_primary_key(*primary_columns*, *primary_variables*)
Process primary variables before an insert happens.

This method may be overwritten by backends to implement custom changes in primary variables before an insert happens.

class storm.database.**Database**(*uri=None*)
Bases: `object`

A database that can be connected to.

This should be subclassed for individual database backends.

Variables ***connection_factory*** – A callable which will take this database and should return an instance of *Connection*.

connection_factory
alias of *Connection*

get_uri()
Return the URI object this database was created with.

connect(*event=None*)
Create a connection to the database.

It calls `self.connection_factory` to allow for ease of customization.

Parameters event – The event system to broadcast messages with. If not specified, then no events will be broadcast.

Returns An instance of *Connection*.

raw_connect()
Create a raw database connection.

This is used by *Connection* objects to connect to the database. It should be overriden in subclasses to do any database-specific connection setup.

Returns A DB-API connection object.

storm.database.**register_scheme**(*scheme*, *factory*)
Register a handler for a new database URI scheme.

Parameters

- **scheme** – the database URI scheme

- **factory** – a function taking a URI instance and returning a database.

storm.database.**create_database**(*uri*)

Create a database instance.

Parameters **uri** – An URI instance, or a string describing the URI. Some examples:

"sqlite:" An in memory sqlite database.

"sqlite:example.db" A SQLite database called example.db

"postgres:test" The database 'test' from the local postgres server.

"postgres://user:password@host/test" The database test on machine host with supplied user credentials, using postgres.

"anything:..." Where 'anything' has previously been registered with *register_scheme*.

4.5.1 PostgreSQL

class storm.databases.postgres.**Returning**(*expr*, *columns=None*)

Bases: *storm.expr.Expr*

Appends the "RETURNING <columns>" suffix to an INSERT or UPDATE.

Parameters

- **expr** – an Insert or Update expression.
- **columns** – The columns to return, if None then expr.primary_columns will be used.

This is only supported in PostgreSQL 8.2+.

class storm.databases.postgres.**Case**(*cases*, *expression=Undef*, *default=Undef*)

Bases: *storm.expr.Expr*

A CASE statement.

Params cases a list of tuples of (condition, result) or (value, result), if an expression is passed too.

Parameters

- **expression** – the expression to compare (if the simple form is used).
- **default** – an optional default condition if no other case matches.

class storm.databases.postgres.**currval**(*column*)

Bases: *storm.expr.FuncExpr*

storm.databases.postgres.**compile_currval**(*compile*, *expr*, *state*)

Compile a currval.

This is a bit involved because we have to get escaping right. Here are a few cases to keep in mind:

```
currval('thetable_thecolumn_seq')
currval('theschema.thetable_thecolumn_seq')
currval('"the schema".thetable_thecolumn_seq')
currval('theschema."the table_thecolumn_seq"')
currval('theschema."thetable_the column_seq"')
currval('"thetable_the column_seq"')
currval('"the schema"."the table_the column_seq"')
```

class `storm.databases.postgres.`**`PostgresResult`**(*connection*, *raw_cursor*)

Bases: *`storm.database.Result`*

`get_insert_identity`(*primary_key*, *primary_variables*)

Get a query which will return the row that was just inserted.

This must be overridden in database-specific subclasses.

Return type *`storm.expr.Expr`*

class `storm.databases.postgres.`**`PostgresConnection`**(*database*, *event=None*)

Bases: *`storm.database.Connection`*

`result_factory`

alias of *`PostgresResult`*

`execute`(*statement*, *params=None*, *noresult=False*)

Execute a statement with the given parameters.

This extends the `Connection.execute` method to add support for automatic retrieval of inserted primary keys to link in-memory objects with their specific rows.

`to_database`(*params*)

Like `Connection.to_database`, but this converts datetime types to strings, and bytes to `psycopg2.Binary` instances.

`is_disconnection_error`(*exc*, *extra_disconnection_errors=()*)

Check whether an exception represents a database disconnection.

This should be overridden by backends to detect whichever exception values are used to represent this condition.

class `storm.databases.postgres.`**`Postgres`**(*uri*)

Bases: *`storm.database.Database`*

`connection_factory`

alias of *`PostgresConnection`*

`raw_connect`()

Create a raw database connection.

This is used by `Connection` objects to connect to the database. It should be overriden in subclasses to do any database-specific connection setup.

Returns A DB-API connection object.

`storm.databases.postgres.`**`create_from_uri`**

alias of *`storm.databases.postgres.Postgres`*

`storm.databases.postgres.`**`make_dsn`**(*uri*)

Convert a URI object to a PostgreSQL DSN string.

class `storm.databases.postgres.`**`PostgresTimeoutTracer`**(*granularity=5*)

Bases: *`storm.tracer.TimeoutTracer`*

`set_statement_timeout`(*raw_cursor*, *remaining_time*)

Perform the timeout setup in the raw cursor.

The database should raise an error if the next statement takes more than the number of seconds provided in `remaining_time`.

Must be specialized in the backend.

`connection_raw_execute_error`(*connection*, *raw_cursor*, *statement*, *params*, *error*)
Raise `TimeoutError` if the given error was a timeout issue.

Must be specialized in the backend.

class `storm.databases.postgres.`**`JSONElement`**(*expr1*, *expr2*)
Bases: *`storm.expr.BinaryOper`*

Return an element of a JSON value (by index or field name).

class `storm.databases.postgres.`**`JSONTextElement`**(*expr1*, *expr2*)
Bases: *`storm.expr.BinaryOper`*

Return an element of a JSON value (by index or field name) as text.

class `storm.databases.postgres.`**`JSONVariable`**(**args*, ***kwargs*)
Bases: *`storm.variables.JSONVariable`*

class `storm.databases.postgres.`**`JSON`**(*name=None*, *primary=False*, ***kwargs*)
Bases: *`storm.properties.SimpleProperty`*

Parameters

- **`name`** – The name of this property.
- **`primary`** – A boolean indicating whether this property is a primary key.
- **`default`** – The initial value of this variable. The default behavior is for the value to stay undefined until it is set with `set`.
- **`default_factory`** – If specified, this will immediately be called to get the initial value.
- **`allow_none`** – A boolean indicating whether None should be allowed to be set as the value of this variable.
- **`validator`** – Validation function called whenever trying to set the variable to a non-db value. The function should look like validator(object, attr, value), where the first and second arguments are the result of validator_object_factory() (or None, if this parameter isn't provided) and the value of validator_attribute, respectively. When called, the function should raise an error if the value is unacceptable, or return the value to be used in place of the original value otherwise.
- **`kwargs`** – Other keyword arguments passed through when constructing the underlying variable.

`variable_class`
alias of *`JSONVariable`*

4.5.2 SQLite

class `storm.databases.sqlite.`**`SQLiteResult`**(*connection*, *raw_cursor*)
Bases: *`storm.database.Result`*

`get_insert_identity`(*primary_key*, *primary_variables*)
Get a query which will return the row that was just inserted.

This must be overridden in database-specific subclasses.

Return type *`storm.expr.Expr`*

`static set_variable`(*variable*, *value*)
Set the given variable's value from the database.

static from_database(*row*)
Convert SQLite-specific datatypes to "normal" Python types.

On Python 2, if there are any `buffer` instances in the row, convert them to bytes. On Python 3, BLOB types are converted to bytes, which is already what we want.

class storm.databases.sqlite.**SQLiteConnection**(*database*, *event=None*)
Bases: *storm.database.Connection*

result_factory
alias of *SQLiteResult*

static to_database(*params*)
Like `Connection.to_database`, but this also converts instances of `datetime` types to strings, and (on Python 2) bytes instances to `buffer` instances.

commit()
Commit the connection.

Parameters xid – Optionally the `Xid` of a previously prepared transaction to commit. This form should be called outside of a transaction, and is intended for use in recovery.

Raises

- ***ConnectionBlockedError*** – Raised if access to the connection has been blocked with `block_access`.
- ***DisconnectionError*** – Raised when the connection is lost. Reconnection happens automatically on rollback.

rollback()
Rollback the connection.

Parameters xid – Optionally the `Xid` of a previously prepared transaction to rollback. This form should be called outside of a transaction, and is intended for use in recovery.

raw_execute(*statement*, *params=None*, *_end=False*)
Execute a raw statement with the given parameters.

This method will automatically retry on locked database errors. This should be done by pysqlite, but it doesn't work with versions < 2.3.4, so we make sure the timeout is respected here.

class storm.databases.sqlite.**SQLite**(*uri*)
Bases: *storm.database.Database*

connection_factory
alias of *SQLiteConnection*

raw_connect()
Create a raw database connection.

This is used by `Connection` objects to connect to the database. It should be overriden in subclasses to do any database-specific connection setup.

Returns A DB-API connection object.

storm.databases.sqlite.**create_from_uri**
alias of *storm.databases.sqlite.SQLite*

4.5.3 Transaction identifiers

class storm.xid.**Xid**(*format_id*, *global_transaction_id*, *branch_qualifier*)
Bases: `object`

Represent a transaction identifier compliant with the XA specification.

4.6 Hooks and events

4.6.1 Event

class `storm.event.`**`EventSystem`**(*owner*)

Bases: `object`

A system for managing hooks that are called when events are emitted.

Hooks are callables that take the event system `owner` as their first argument, followed by the arguments passed when emitting the event, followed by any additional `data` arguments given when registering the hook.

Hooks registered for a given event `name` are stored without ordering: no particular call order may be assumed when an event is emitted.

Parameters **`owner`** – The object that owns this event system. It is passed as the first argument to each hook function.

`hook`(*name*, *callback*, **data*)

Register a hook.

Parameters

- **`name`** – The name of the event for which this hook should be called.
- **`callback`** – A callable which should be called when the event is emitted.
- **`data`** – Additional arguments to pass to the callable, after the `owner` and any arguments passed when emitting the event.

`unhook`(*name*, *callback*, **data*)

Unregister a hook.

This ignores attempts to unregister hooks that were not already registered.

Parameters

- **`name`** – The name of the event for which this hook should no longer be called.
- **`callback`** – The callable to unregister.
- **`data`** – Additional arguments that were passed when registering the callable.

`emit`(*name*, **args*)

Emit an event, calling any registered hooks.

Parameters

- **`name`** – The name of the event.
- **`args`** – Additional arguments to pass to hooks.

4.6.2 Tracer

class `storm.tracer.`**`TimeoutTracer`**(*granularity=5*)

Bases: `object`

Provide a timeout facility for connections to prevent rogue operations.

This tracer must be subclassed by backend-specific implementations that override `connection_raw_execute_error`, `set_statement_timeout` and `get_remaining_time` methods.

`connection_raw_execute`(*connection*, *raw_cursor*, *statement*, *params*)
Check timeout conditions before a statement is executed.

Parameters

- **`connection`** – The `Connection` to the database.
- **`raw_cursor`** – A cursor object, specific to the backend being used.
- **`statement`** – The SQL statement to execute.
- **`params`** – The parameters to use with `statement`.

Raises ***`TimeoutError`*** – Raised if there isn't enough time left to execute `statement`.

`connection_raw_execute_error`(*connection*, *raw_cursor*, *statement*, *params*, *error*)
Raise `TimeoutError` if the given error was a timeout issue.

Must be specialized in the backend.

`connection_commit`(*connection*, *xid=None*)
Reset `Connection._timeout_tracer_remaining_time`.

Parameters

- **`connection`** – The `Connection` to the database.
- **`xid`** – Optionally the `Xid` of a previously prepared transaction to commit.

`connection_rollback`(*connection*, *xid=None*)
Reset `Connection._timeout_tracer_remaining_time`.

Parameters

- **`connection`** – The `Connection` to the database.
- **`xid`** – Optionally the `Xid` of a previously prepared transaction to rollback.

`set_statement_timeout`(*raw_cursor*, *remaining_time*)
Perform the timeout setup in the raw cursor.

The database should raise an error if the next statement takes more than the number of seconds provided in `remaining_time`.

Must be specialized in the backend.

`get_remaining_time`()
Tells how much time the current context (HTTP request, etc) has.

Must be specialized with application logic.

Returns Number of seconds allowed for the next statement.

`class` `storm.tracer.`**`BaseStatementTracer`**
Bases: `object`

Storm tracer base class that does query interpolation.

`class` `storm.tracer.`**`TimelineTracer`**(*timeline_factory*, *prefix='SQL-'*)
Bases: *`storm.tracer.BaseStatementTracer`*

Storm tracer class to insert executed statements into a `Timeline`.

For more information on timelines see the module at https://pypi.org/project/timeline/.

The timeline to use is obtained by calling the timeline_factory supplied to the constructor. This simple function takes no parameters and returns a timeline to use. If it returns None, the tracer is bypassed.

Create a TimelineTracer.

Parameters

- **`timeline_factory`** – A factory function to produce the timeline to record a query against.
- **`prefix`** – A prefix to give the connection name when starting an action. Connection names are found by trying a getattr for 'name' on the connection object. If no name has been assigned, '<unknown>' is used instead.

4.7 Miscellaneous

4.7.1 Cache

class `storm.cache.`**`Cache`**(*size=1000*)

Bases: `object`

Prevents recently used objects from being deallocated.

This prevents recently used objects from being deallocated by Python even if the user isn't holding any strong references to it. It does that by holding strong references to the objects referenced by the last `N` `obj_info`s added to it (where `N` is the cache size).

`clear`()

Clear the entire cache at once.

`add`(*obj_info*)

Add `obj_info` as the most recent entry in the cache.

If the `obj_info` is already in the cache, it remains in the cache and has its order changed to become the most recent entry (IOW, will be the last to leave).

`remove`(*obj_info*)

Remove `obj_info` from the cache, if present.

Returns True if `obj_info` was cached, False otherwise.

`set_size`(*size*)

Set the maximum number of objects that may be held in this cache.

If the size is reduced, older `obj_info`s may be dropped from the cache to respect the new size.

`get_cached`()

Return an ordered list of the currently cached `obj_info`s.

The most recently added objects come first in the list.

class `storm.cache.`**`GenerationalCache`**(*size=1000*)

Bases: `object`

Generational replacement for Storm's LRU cache.

This cache approximates LRU without keeping exact track. Instead, it keeps a primary dict of "recently used" objects plus a similar, secondary dict of objects used in a previous timeframe.

When the "most recently used" dict reaches its size limit, it is demoted to secondary dict and a fresh primary dict is set up. The previous secondary dict is evicted in its entirety.

Use this to replace the LRU cache for sizes where LRU tracking overhead becomes too large (e.g. 100,000 objects) or the *StupidCache* when it eats up too much memory.

Create a generational cache with the given size limit.

The size limit applies not to the overall cache, but to the primary one only. When this reaches the size limit, the real number of cached objects will be somewhere between size and 2*size depending on how much overlap there is between the primary and secondary caches.

clear()
See `Cache.clear`.

Clears both the primary and the secondary caches.

add(*obj_info*)
See `Cache.add`.

remove(*obj_info*)
See `Cache.remove`.

set_size(*size*)
See `Cache.set_size`.

After calling this, the cache may still contain more than *size* objects, but no more than twice that number.

get_cached()
See `Cache.get_cached`.

The result is a loosely-ordered list. Any object in the primary generation comes before any object that is only in the secondary generation, but objects within a generation are not ordered and there is no indication of the boundary between the two.

Objects that are in both the primary and the secondary generation are listed only as part of the primary generation.

4.7.2 Exceptions

exception storm.exceptions.**StormError**
Bases: `Exception`

exception storm.exceptions.**CompileError**
Bases: `storm.exceptions.StormError`

exception storm.exceptions.**NoTableError**
Bases: `storm.exceptions.CompileError`

exception storm.exceptions.**ExprError**
Bases: `storm.exceptions.StormError`

exception storm.exceptions.**NoneError**
Bases: `storm.exceptions.StormError`

exception storm.exceptions.**PropertyPathError**
Bases: `storm.exceptions.StormError`

exception storm.exceptions.**ClassInfoError**
Bases: `storm.exceptions.StormError`

exception storm.exceptions.**URIError**
Bases: `storm.exceptions.StormError`

exception storm.exceptions.**ClosedError**
Bases: `storm.exceptions.StormError`

exception storm.exceptions.**FeatureError**
Bases: *storm.exceptions.StormError*

exception storm.exceptions.**DatabaseModuleError**
Bases: *storm.exceptions.StormError*

exception storm.exceptions.**StoreError**
Bases: *storm.exceptions.StormError*

exception storm.exceptions.**NoStoreError**
Bases: *storm.exceptions.StormError*

exception storm.exceptions.**WrongStoreError**
Bases: *storm.exceptions.StoreError*

exception storm.exceptions.**NotFlushedError**
Bases: *storm.exceptions.StoreError*

exception storm.exceptions.**OrderLoopError**
Bases: *storm.exceptions.StoreError*

exception storm.exceptions.**NotOneError**
Bases: *storm.exceptions.StoreError*

exception storm.exceptions.**UnorderedError**
Bases: *storm.exceptions.StoreError*

exception storm.exceptions.**LostObjectError**
Bases: *storm.exceptions.StoreError*

exception storm.exceptions.**Error**
Bases: *storm.exceptions.StormError*

exception storm.exceptions.**Warning**
Bases: *storm.exceptions.StormError*

exception storm.exceptions.**InterfaceError**
Bases: *storm.exceptions.Error*

exception storm.exceptions.**DatabaseError**
Bases: *storm.exceptions.Error*

exception storm.exceptions.**InternalError**
Bases: *storm.exceptions.DatabaseError*

exception storm.exceptions.**OperationalError**
Bases: *storm.exceptions.DatabaseError*

exception storm.exceptions.**ProgrammingError**
Bases: *storm.exceptions.DatabaseError*

exception storm.exceptions.**IntegrityError**
Bases: *storm.exceptions.DatabaseError*

exception storm.exceptions.**DataError**
Bases: *storm.exceptions.DatabaseError*

exception storm.exceptions.**NotSupportedError**
Bases: *storm.exceptions.DatabaseError*

exception storm.exceptions.**DisconnectionError**
Bases: *storm.exceptions.OperationalError*

exception `storm.exceptions.`**`TimeoutError`**(*statement*, *params*, *message=None*)

Bases: *`storm.exceptions.StormError`*

Raised by timeout tracers when remining time is over.

exception `storm.exceptions.`**`ConnectionBlockedError`**

Bases: *`storm.exceptions.StormError`*

Raised when an attempt is made to use a blocked connection.

`storm.exceptions.`**`wrap_exceptions`**(*database*)

Context manager that re-raises DB exceptions as StormError instances.

4.7.3 Info

class `storm.info.`**`ClassInfo`**(*cls*)

Bases: `dict`

Persistent Storm-related information of a class.

The following attributes are defined:

Variables

- **`table`** – Expression from where columns will be looked up.
- **`cls`** – Class which should be used to build objects.
- **`columns`** – Tuple of column properties found in the class.
- **`primary_key`** – Tuple of column properties used to form the primary key
- **`primary_key_pos`** – Position of primary_key items in the columns tuple.

class `storm.info.`**`ObjectInfo`**(*obj*)

Bases: `dict`

class `storm.info.`**`ClassAlias`**

Bases: `object`

Create a named alias for a Storm class for use in queries.

This is useful basically when the SQL ‘AS’ feature is desired in code using Storm queries.

ClassAliases which are explicitly named (i.e., when ‘name’ is passed) are cached for as long as the class exists, such that the alias returned from `ClassAlias(Foo, 'foo_alias')` will be the same object no matter how many times it’s called.

Parameters

- **`cls`** – The class to create the alias of.
- **`name`** – If provided, specify the name of the alias to create.

4.7.4 Testing

class `storm.testing.`**`CaptureTracer`**

Bases: *`storm.tracer.BaseStatementTracer`*, `fixtures.fixture.Fixture`

Trace SQL statements appending them to a `list`.

Example:

```
with CaptureTracer() as tracer:
    # Run queries
print(tracer.queries)  # Print the queries that have been run
```

Note This class requires the fixtures package to be available.

4.7.5 Timezone

This module offers extensions to the standard python 2.3+ datetime module.

class storm.tz.**tzutc**

Bases: `datetime.tzinfo`

utcoffset(*dt*)
datetime -> timedelta showing offset from UTC, negative values indicating West of UTC

dst(*dt*)
datetime -> DST offset as timedelta positive east of UTC.

tzname(*dt*)
datetime -> string name of time zone.

class storm.tz.**tzoffset**(*name*, *offset*)

Bases: `datetime.tzinfo`

utcoffset(*dt*)
datetime -> timedelta showing offset from UTC, negative values indicating West of UTC

dst(*dt*)
datetime -> DST offset as timedelta positive east of UTC.

tzname(*dt*)
datetime -> string name of time zone.

class storm.tz.**tzlocal**

Bases: `datetime.tzinfo`

utcoffset(*dt*)
datetime -> timedelta showing offset from UTC, negative values indicating West of UTC

dst(*dt*)
datetime -> DST offset as timedelta positive east of UTC.

tzname(*dt*)
datetime -> string name of time zone.

class storm.tz.**tzfile**(*fileobj*)

Bases: `datetime.tzinfo`

utcoffset(*dt*)
datetime -> timedelta showing offset from UTC, negative values indicating West of UTC

dst(*dt*)
datetime -> DST offset as timedelta positive east of UTC.

tzname(*dt*)
datetime -> string name of time zone.

class storm.tz.**tzrange**(*stdabbr*, *stdoffset=None*, *dstabbr=None*, *dstoffset=None*, *start=None*, *end=None*)

Bases: `datetime.tzinfo`

utcoffset(*dt*)
datetime -> timedelta showing offset from UTC, negative values indicating West of UTC

dst(*dt*)
datetime -> DST offset as timedelta positive east of UTC.

tzname(*dt*)
datetime -> string name of time zone.

class storm.tz.**tzstr**(*s*)
Bases: *storm.tz.tzrange*

4.7.6 URIs

class storm.uri.**URI**(*uri_str*)
Bases: object

A representation of a Uniform Resource Identifier (URI).

This is intended exclusively for database connection URIs.

Variables

- **username** – The username part of the URI, or None.
- **password** – The password part of the URI, or None.
- **host** – The host part of the URI, or None.
- **port** – The port part of the URI, or None.
- ***database*** – The part of the URI representing the database name, or None.

4.7.7 WSGI

Glue to wire a storm timeline tracer up to a WSGI app.

storm.wsgi.**make_app**(*app*)
Capture the per-request timeline object needed for Storm tracing.

To use firstly make your app and then wrap it with this *make_app*:

```
>>> app, find_timeline = make_app(app)
```

Then wrap the returned app with the `timeline` app (or anything that sets `environ['timeline.timeline']`):

```
>>> app = timeline.wsgi.make_app(app)
```

Finally install a timeline tracer to capture Storm queries:

```
>>> install_tracer(TimelineTracer(find_timeline))
```

Returns A wrapped WSGI app and a timeline factory function for use with *TimelineTracer*.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index