Performing raw SQL queries

Django gives you two ways of performing raw SQL queries: you can use `Manager.raw()` to perform raw queries and return model instances, or you can avoid the model layer entirely and execute custom SQL directly.

Explore the ORM before using raw SQL!: The Django ORM provides many tools to express queries without writing raw SQL. For example:

- The QuerySet API is extensive.
- You can `annotate` and aggregate using many built-in database functions. Beyond those, you can create custom query expressions.

Before using raw SQL, explore the ORM. Ask on one of the support channels to see if the ORM supports your use case.

Warning: You should be very careful whenever you write raw SQL. Every time you use it, you should properly escape any parameters that the user can control by using `params` in order to protect against SQL injection attacks. Please read more about SQL injection protection.

Performing raw queries

The `raw()` manager method can be used to perform raw SQL queries that return model instances:

```
Manager.raw(raw_query, params=(), translations=None)
```

This method takes a raw SQL query, executes it, and returns a `django.db.models.query.RawQuerySet` instance. This `RawQuerySet` instance can be iterated over like a normal `QuerySet` to provide object instances.

This is best illustrated with an example. Suppose you have the following model:

```
class Person(models.Model):
    first_name = models.CharField(...)
    last_name = models.CharField(...)
    birth_date = models.DateField(...)
```

You could then execute custom SQL like so:

```
>>> for p in Person.objects.raw("SELECT * FROM myapp_person"):
...     print(p)
...
John Smith
Jane Jones
```

This example isn't very exciting – it's exactly the same as running `Person.objects.all()`. However, `raw()` has a bunch of other options that make it very powerful.

Model table names: Where did the name of the `Person` table come from in that example?

By default, Django figures out a database table name by joining the model's "app label" – the name you used in `manage.py startapp` – to the model's class name, with an underscore between them. In the example we've assumed that the `Person` model lives in an app named `myapp`, so its table would be `myapp_person`.

For more details check out the documentation for the `db_table` option, which also lets you manually set the database table name.

Warning: No checking is done on the SQL statement that is passed in to `.raw()`. Django expects that the statement will return a set of rows from the database, but does nothing to enforce that. If the query does not return rows, a (possibly cryptic) error will result.

Warning: If you are performing queries on MySQL, note that MySQL's silent type coercion may cause unexpected results when mixing types. If you query on a string type column, but with an integer value, MySQL will coerce the types of all values in the table to an integer before performing the comparison. For example, if your table contains the values `'abc'`, `'def'` and you query for `WHERE mycolumn=0`, both rows will match. To prevent this, perform the correct typecasting before using the value in a query.

Mapping query fields to model fields

`raw()` automatically maps fields in the query to fields on the model.

The order of fields in your query doesn't matter. In other words, both of the following queries work identically:

```
>>> Person.objects.raw("SELECT id, first_name, last_name, birth_date FROM myapp_person")
>>> Person.objects.raw("SELECT last_name, birth_date, first_name, id FROM myapp_person")
```

Matching is done by name. This means that you can use SQL's `AS` clauses to map fields in the query to model fields. So if you had some other table that had `Person` data in it, you could easily map it into `Person` instances:

```
>>> Person.objects.raw(
...     """
...     SELECT first AS first_name,
...            last AS last_name,
...            bd AS birth_date,
...            pk AS id,
...     FROM some_other_table
...     """
... )
```

As long as the names match, the model instances will be created correctly.

Alternatively, you can map fields in the query to model fields using the `translations` argument to `raw()`. This is a dictionary mapping names of fields in the query to names of fields on the model. For example, the above query could also

be written:

```
>>> name_map = {"first": "first_name", "last": "last_name", "bd": "birth_date", "pk": "id"}
>>> Person.objects.raw("SELECT * FROM some_other_table", translations=name_map)
```

Index lookups

`raw()` supports indexing, so if you need only the first result you can write:

```
>>> first_person = Person.objects.raw("SELECT * FROM myapp_person")[0]
```

However, the indexing and slicing are not performed at the database level. If you have a large number of `Person` objects in your database, it is more efficient to limit the query at the SQL level:

```
>>> first_person = Person.objects.raw("SELECT * FROM myapp_person LIMIT 1")[0]
```

Deferring model fields

Fields may also be left out:

```
>>> people = Person.objects.raw("SELECT id, first_name FROM myapp_person")
```

The `Person` objects returned by this query will be deferred model instances (see `defer()`). This means that the fields that are omitted from the query will be loaded on demand. For example:

```
>>> for p in Person.objects.raw("SELECT id, first_name FROM myapp_person"):
...     print(
...         p.first_name,  # This will be retrieved by the original query
...         p.last_name,  # This will be retrieved on demand
...     )
...
John Smith
Jane Jones
```

From outward appearances, this looks like the query has retrieved both the first name and last name. However, this example actually issued 3 queries. Only the first names were retrieved by the `raw()` query – the last names were both retrieved on demand when they were printed.

There is only one field that you can't leave out - the primary key field. Django uses the primary key to identify model instances, so it must always be included in a raw query. A `FieldDoesNotExist` exception will be raised if you forget to include the primary key.

Adding annotations

You can also execute queries containing fields that aren't defined on the model. For example, we could use PostgreSQL's age() function to get a list of people with their ages calculated by the database:

```
>>> people = Person.objects.raw("SELECT *, age(birth_date) AS age FROM myapp_person")
>>> for p in people:
...     print("%s is %s." % (p.first_name, p.age))
...
John is 37.
Jane is 42.
...
```

You can often avoid using raw SQL to compute annotations by instead using a Func() expression.

Passing parameters into `raw()`

If you need to perform parameterized queries, you can use the `params` argument to `raw()`:

```
>>> lname = "Doe"
>>> Person.objects.raw("SELECT * FROM myapp_person WHERE last_name = %s", [lname])
```

`params` is a list or dictionary of parameters. You'll use `%s` placeholders in the query string for a list, or `%(key)s` placeholders for a dictionary (where `key` is replaced by a dictionary key), regardless of your database engine. Such placeholders will be replaced with parameters from the `params` argument.

Note: Dictionary params are not supported with the SQLite backend; with this backend, you must pass parameters as a list.

Warning: Do not use string formatting on raw queries or quote placeholders in your SQL strings!

It's tempting to write the above query as:

```
>>> query = "SELECT * FROM myapp_person WHERE last_name = %s" % lname
>>> Person.objects.raw(query)
```

You might also think you should write your query like this (with quotes around `%s`):

```
>>> query = "SELECT * FROM myapp_person WHERE last_name = '%s'"
```

Don't make either of these mistakes.

As discussed in SQL injection protection, using the `params` argument and leaving the placeholders unquoted protects you from SQL injection attacks, a common exploit where attackers inject arbitrary SQL into your database. If you use string interpolation or quote the placeholder, you're at risk for SQL injection.

Executing custom SQL directly

Sometimes even `Manager.raw()` isn't quite enough: you might need to perform queries that don't map cleanly to models, or directly execute `UPDATE`, `INSERT`, or `DELETE` queries.

In these cases, you can always access the database directly, routing around the model layer entirely.

The object `django.db.connection` represents the default database connection. To use the database connection, call `connection.cursor()` to get a cursor object. Then, call `cursor.execute(sql, [params])` to execute the SQL and `cursor.fetchone()` or `cursor.fetchall()` to return the resulting rows.

For example:

```
from django.db import connection


def my_custom_sql(self):
    with connection.cursor() as cursor:
        cursor.execute("UPDATE bar SET foo = 1 WHERE baz = %s", [self.baz])
        cursor.execute("SELECT foo FROM bar WHERE baz = %s", [self.baz])
        row = cursor.fetchone()

    return row
```

To protect against SQL injection, you must not include quotes around the `%s` placeholders in the SQL string.

Note that if you want to include literal percent signs in the query, you have to double them in the case you are passing parameters:

```
cursor.execute("SELECT foo FROM bar WHERE baz = '30%'")
cursor.execute("SELECT foo FROM bar WHERE baz = '30%%' AND id = %s", [self.id])
```

If you are using more than one database, you can use `django.db.connections` to obtain the connection (and cursor) for a specific database. `django.db.connections` is a dictionary-like object that allows you to retrieve a specific connection using its alias:

```
from django.db import connections

with connections["my_db_alias"].cursor() as cursor:
    # Your code here
    ...
```

By default, the Python DB API will return results without their field names, which means you end up with a `list` of values, rather than a `dict`. At a small performance and memory cost, you can return results as a `dict` by using something like this:

```
def dictfetchall(cursor):
    """
    Return all rows from a cursor as a dict.
    Assume the column names are unique.
    """
    columns = [col[0] for col in cursor.description]
    return [dict(zip(columns, row)) for row in cursor.fetchall()]
```

Another option is to use `collections.namedtuple()` from the Python standard library. A `namedtuple` is a tuple-like object that has fields accessible by attribute lookup; it's also indexable and iterable. Results are immutable and accessible by field names or indices, which might be useful:

```
from collections import namedtuple


def namedtuplefetchall(cursor):
    """
    Return all rows from a cursor as a namedtuple.
    Assume the column names are unique.
    """
    desc = cursor.description
    nt_result = namedtuple("Result", [col[0] for col in desc])
    return [nt_result(*row) for row in cursor.fetchall()]
```

The `dictfetchall()` and `namedtuplefetchall()` examples assume unique column names, since a cursor cannot distinguish columns from different tables.

Here is an example of the difference between the three:

```
>>> cursor.execute("SELECT id, parent_id FROM test LIMIT 2")
>>> cursor.fetchall()
((54360982, None), (54360880, None))

>>> cursor.execute("SELECT id, parent_id FROM test LIMIT 2")
>>> dictfetchall(cursor)
[{'parent_id': None, 'id': 54360982}, {'parent_id': None, 'id': 54360880}]

>>> cursor.execute("SELECT id, parent_id FROM test LIMIT 2")
>>> results = namedtuplefetchall(cursor)
>>> results
[Result(id=54360982, parent_id=None), Result(id=54360880, parent_id=None)]
>>> results[0].id
54360982
>>> results[0][0]
54360982
```

Connections and cursors

`connection` and `cursor` mostly implement the standard Python DB-API described in **PEP 249** — except when it comes to transaction handling.

If you're not familiar with the Python DB-API, note that the SQL statement in `cursor.execute()` uses placeholders, `"%s"`, rather than adding parameters directly within the SQL. If you use this technique, the underlying database library will automatically escape your parameters as necessary.

Also note that Django expects the `"%s"` placeholder, *not* the `"?"` placeholder, which is used by the SQLite Python bindings. This is for the sake of consistency and sanity.

Using a cursor as a context manager:

```
with connection.cursor() as c:
    c.execute(...)
```

is equivalent to:

```
c = connection.cursor()
try:
    c.execute(...)
finally:
    c.close()
```

Calling stored procedures

```
CursorWrapper.callproc(procname, params=None, kparams=None)
```

Calls a database stored procedure with the given name. A sequence (`params`) or dictionary (`kparams`) of input parameters may be provided. Most databases don't support `kparams` . Of Django's built-in backends, only Oracle supports it.

For example, given this stored procedure in an Oracle database:

```
CREATE PROCEDURE "TEST_PROCEDURE"(v_i INTEGER, v_text NVARCHAR2(10)) AS
    p_i INTEGER;
    p_text NVARCHAR2(10);
BEGIN
    p_i := v_i;
    p_text := v_text;
    ...
END;
```

This will call it:

```
with connection.cursor() as cursor:
    cursor.callproc("test_procedure", [1, "test"])
```