ODDBIRD

June 14, 2014 by carl in django, postgresql, python, sqlalchemy, transactions.
SQLAlchemy [http://www.sqlalchemy.org/] defaults to implicitly opening a new transaction on your first database query. If you prefer to start your transactions explicitly instead, I've documented here my explorations in getting that to work.

There are several different layers at play, so let's review from the top:

Since version 7.4, PostgreSQL itself always operates in "autocommit" mode. This means that if you haven't started a transaction, and you run a query that changes data, it takes effect immediately; there is no need to commit it, and no option to roll it back.

If you want to group several queries together into an atomic unit of work, such that you can commit them all together or roll them back all together, you first issue a **`BEGIN`** statement, then your queries, and then either a **`COMMIT`** or **`ROLLBACK`** statement.

PEP 249 [http://legacy.python.org/dev/peps/pep-0249/] specifies a common API for all Python database adapter libraries, making it easier to write cross-database-compatible code. Almost all the popular Python database adapters are (or claim or try to be) PEP 249 compliant.

The PEP 249 API specifies a different behavior, which I will call "implicit transactions." There isn't even a way to explicitly start a transaction in the PEP 249 API; no **`begin()`** or similar. Instead, you are always in a transaction; one is automatically started the first time you send a query to the database after opening a connection, and the first time you send a new query after ending the previous transaction with a rollback or commit.

Because Postgres doesn't offer this behavior natively, the Postgres Python adapters (e.g. psycopg2 [http://initd.org/psycopg/docs/]) have to emulate it themselves in order to be PEP 249 compliant. So by default, the first time you send a query to the database, psycopg2 will prefix it with a **`BEGIN`** on your behalf.

In order to get psycopg2 to stop sending these automatic **`BEGIN`** statements and to behave like Postgres natively does, you set the autocommit property [http://initd.org/psycopg/docs/connection.html#connection.autocommit] of your connection object to **`True`**:

```
import psycopg2
conn = psycopg2.connect('dbname=test')
conn.autocommit = True
```

I prefer the 'autocommit' model. It's simple, explicit, and unsurprising. Queries are never grouped together into a transaction unless you ask for one. If you issue a **`COMMIT`** or **`ROLLBACK`**, there is never any doubt about which queries you are committing or rolling back, because you explicitly issued the **`BEGIN`** to start the transaction.

I can see some advantage to the implicit-transaction model for interactive-shell use (you can always roll back your changes if you screw something up, even if you forgot to **`BEGIN`**), but for general use I think it's more error-prone. If you save a change to the database in one place but forget to commit, that change will automatically be wrapped up in the same transaction with later, possibly completely unrelated changes, and may get blindly rolled back along with them. It wraps read-only `SELECT` statements in useless transactions. And for long-running processes, since even a simple `SELECT` implicitly opens a transaction (and you wouldn't intuitively think you'd need to commit or rollback after a `SELECT`), it's very easy to unintentionally end up with connections in the "idle in transaction" state, where they are doing nothing but still may be holding locks and preventing Postgres from compacting tables.

The explicit-transactions (autocommit) model lends itself naturally to Python idioms like decorators or context managers for handling transactions. These clearly wrap a section of code (a function, or the block introduced by the `with` statement) as an atomic unit of work that will be committed or rolled back together. It's more difficult to correctly implement such an idiom in the implicit-transactions model, because there is no explicit start point to a transaction, so prior queries outside the demarcated block might get wrapped up in the same transaction (or you have to check when entering the block whether there are uncommitted queries in an existing transaction, and decide what should be done with them – perhaps having to guess, or raise an error.)

Before we talk about SQLAlchemy [http://www.sqlalchemy.org/] , let's discuss the Django [http://www.djangoproject.com] ORM briefly for comparison purposes.

Up until Django 1.6, Django's transactions API [https://docs.djangoproject.com/en/1.6/topics/db/transactions/#transactions-upgrading-from-1-5] used psycopg2 in its default (PEP 249) mode, but then by default emulated a sort of Python-level "autocommit" mode by automatically issuing commits after ORM-issued queries.

So in other words, if you were using Django pre-1.6 with Postgres, you had psycopg2 emulating implicit-transactions on top of Postgres' native autocommit by automatically issuing **`BEGIN`** statements on your behalf, and then Django emulating autocommit atop psycopg2's emulated non-autocommit atop Postgres' native autocommit, by also automatically issuing **`COMMIT`** statements on your behalf.

Got all that?

Thankfully, in Django 1.6 Aymeric Augustin [https://myks.org/en/] rewrote Django's transaction support (based in part on prior work by Christophe Pettus [https://github.com/Xof/xact]) to use database-level autocommit natively and only open transactions when explicitly requested. The core API (really, the only API) is transactions.atomic [https://docs.djangoproject.com/en/1.6/topics/db/transactions/#django.db.transaction.atomic] , which can work as either a decorator or context manager to define an atomic unit of database work (it can also be nested, using a stack of savepoints [http://www.postgresql.org/docs/9.2/static/sql-savepoint.html] to achieve the effect of nested transactions).

SQLAlchemy [http://www.sqlalchemy.org/] follows the lead of PEP 249 and uses implicit-transactions as its primary mode. Whenever you query the database, a transaction automatically starts (SQLAlchemy doesn't do anything special to make this happen, it's just using the database adapter – **`psycopg2`** in our case – in its default PEP-249-compliant mode), and you can commit that transaction with **`session.commit()`** or roll it back with **`session.rollback()`**.

Although I don't prefer this mode, it's a reasonable choice for SQLAlchemy to rely on PEP 249 consistency across the board rather than implementing custom code for native-autocommit mode in all its many supported databases.

But I'm using Postgres, I know how its native autocommit mode works, and that's the behavior I want with SQLAlchemy. Can I make that work?

I soon found autocommit mode [http://docs.sqlalchemy.org/en/rel_0_9/orm/session.html#autocommit-mode] in SQLAlchemy's documentation, and thought I had my answer – but no such luck. SQLAlchemy's autocommit mode is roughly parallel to the "autocommit" in Django pre-1.6 (albeit smarter): it emulates autocommit over top of non-autocommit database adapters by automatically committing an implicit transaction after you send queries that change the database. It doesn't put the database connections into true autocommit mode, so it still wraps reads in unnecessary transactions.

Happily, setting all of SQLAlchemy's psycopg2 connections into real autocommit became quite easy in SQLAlchemy 0.8.2: SQLAlchemy's psycopg2 "dialect" now exposes an **`AUTOCOMMIT`** transaction isolation level, and selecting it sets **`autocommit=True`** on all the psycopg2 connections.

```
from sqlalchemy import create_engine
engine = create_engine('postgresql://test',
isolation_level="AUTOCOMMIT")
```

We haven't discussed transaction isolation levels yet (and I won't in detail here). They control the visibility of changes between multiple concurrent transactions. The Postgres documentation [http://www.postgresql.org/docs/9.2/static/transaction-iso.html] summarizes the options it provides.

It's a bit odd that SQLAlchemy (and psycopg2 [http://initd.org/psycopg/docs/extensions.html#psycopg2.extensions.ISOLATION_LEVEL_AUTOCOMMIT]) provide **`AUTOCOMMIT`** as a transaction isolation level, since Postgres has no such isolation level, and autocommit-mode isn't really an isolation level at all. (In fact, if you choose the **`AUTOCOMMIT`** "isolation level" in psycopg2, it turns on the connection's **`autocommit`** property but actually resets the real Postgres isolation level to "read committed", the default.)

But regardless, **`isolation_level='AUTOCOMMIT'`** is the incantation needed to get all of our SQLAlchemy psycopg2 connections into true autocommit mode.

If we didn't want to use transactions at all, this would be all we need. SQLAlchemy would happily hum along thinking it has a transaction but actually not having one at all (just like it does for databases that don't support transactions).

But we do want to use transactions, so we need a way to start one. The natural API for this already exists in SQLAlchemy: **`session.begin()`**. Since SQLAlchemy assumes that its database adapter will automatically start transactions, **`session.begin()`** never actually issues a **`BEGIN`** to the database. But we don't actually need to issue **`BEGIN`** ourselves either - we just need to turn off the **`autocommit`** property on our connection, and then **`psycopg2`** will issue the **`BEGIN`** for us.

SQLAlchemy gives us a way to hook into the **`begin()`** call: the **`after_begin`** event, which sends along the relevant database connection. We have to dig through a few layers of connection-wrapping to get down to the actual psycopg2 connection object, but that's not hard:

```
from sqlalchemy import create_engine, event
from sqlalchemy.orm import sessionmaker

engine = create_engine('postgresql://test',
isolation_level="AUTOCOMMIT")
Session = sessionmaker(bind=engine, autocommit=True)
```

```
@event.listens_for(Session, 'after_begin')
def receive_after_begin(session, transaction, connection):
    """When a (non-nested) transaction begins, turn autocommit off."""
    dbapi_connection = connection.connection.connection
    if transaction.nested:
        assert not dbapi_connection.autocommit
        return
    assert dbapi_connection.autocommit
    dbapi_connection.autocommit = False
```

The **`session.begin()`** API can also be used to initiate “nested transactions” using savepoints. In this case autocommit should already have been turned off on the connection by the outer “real” transaction, so we don’t need to do anything. We add in a couple asserts to validate our assumptions about what the autocommit state should be in each case, and in the non-nested case we turn autocommit off.

We also pass **`autocommit=True`** to the **`Session`**; this turns on SQLAlchemy’s autocommit mode (mentioned above). This is necessary to prevent SQLAlchemy from automatically starting a transaction (and thus triggering our **`after_begin`** listener) on the first query.

This is the piece that I’m least happy with, as it means we have to worry about what is meant by the vague warnings [http://docs.sqlalchemy.org/en/rel_0_9/orm/session.html#autocommit-mode] in the documentation that Session autocommit is a “legacy mode of usage” that “can in some cases lead to concurrent connection checkouts” and that we should turn off the Session’s autoflush and autoexpire features. So far I haven’t done the latter; waiting to see what (if any) problems ensue in practice.

One thing remains lacking from our implementation. When a transaction ends, we need to restore that connection to autocommit mode again.

This isn't entirely straightforward. SQLAlchemy gives us three events corresponding to the end of a transaction: **`after_rollback`**, **`after_commit`**, and **`after_transaction_end`**. But these are all fired after the connection has been "closed" (that is, returned to the connection pool), and **`after_transaction_end`** is only fired once per SQLAlchemy **`SessionTransaction`** object, which can involve multiple connections. For both of these reasons, none of those events provide us with a connection object.

In order to get around this, I maintain a mapping of **`SessionTransaction`** objects to the connection(s) that have had **`autocommit`** turned off due to that transaction. Then I listen to **`after_transaction_end`** and restore autocommit on all the appropriate connections:

```
from sqlalchemy import create_engine, event
from sqlalchemy.orm import sessionmaker

engine = create_engine('postgresql://test',
isolation_level="AUTOCOMMIT")
Session = sessionmaker(bind=engine, autocommit=True)

dconns_by_trans = {}

@event.listens_for(Session, 'after_begin')
def receive_after_begin(session, transaction, connection):
    """When a transaction begins, turn autocommit off."""
    dbapi_connection = connection.connection.connection
    if transaction.nested:
        assert not dbapi_connection.autocommit
        return
    assert dbapi_connection.autocommit
    dbapi_connection.autocommit = False
    dconns_by_trans.setdefault(transaction, set()).add(
        dbapi_connection)


@event.listens_for(Session, 'after_transaction_end')
def receive_after_transaction_end(session, transaction):
    """Restore autocommit where this transaction turned it off."""
    if transaction in dconns_by_trans:
        for dbapi_connection in dconns_by_trans[transaction]:
            assert not dbapi_connection.autocommit
            dbapi_connection.autocommit = True
        del dconns_by_trans[transaction]
```

Now that we have autocommit mode working, here's an example of a rough equivalent to transaction.atomic [https://docs.djangoproject.com/en/1.6/topics/db/transactions/#django.db.transaction.atomic] for SQLAlchemy (unlike transaction.atomic [https://docs.djangoproject.com/en/1.6/topics/db/transactions/#django.db.transaction.atomic] this doesn't work as a decorator, but adding that is just a matter of some boilerplate):

```
from contextlib import contextmanager
from sqlalchemy.orm import Session as BaseSession



class Session(BaseSession):
    def __init__(self, *a, **kw):
        super(Session, self).__init__(*a, **kw)
        self._in_atomic = False

    @contextmanager
    def atomic(self):
        """Transaction context manager.
        Will commit the transaction on successful completion
        of the block, or roll it back on error.

        Supports nested usage (via savepoints).

        """
        nested = self._in_atomic
        self.begin(nested=nested)
        self._in_atomic = True

        try:
            yield
        except:
            self.rollback()
            raise
        else:
            self.commit()
        finally:
            if not nested:
                self._in_atomic = False
```

It would be possible to implement this same context manager in SQLAlchemy's default implicit-transactions mode: you just leave out the call to **`session.begin()`** in the non-nested case (because an implicit transaction will already have been created). But that implementation then suffers from the bug where database queries from prior to the context-managed block could be included within its transaction. With some further cleverness you might be able to figure out on entering the context manager whether the existing transaction is already "dirty" (though this is not trivial to determine), and then either raise an error or implicitly commit the existing transaction — but neither of these solutions are appealing compared to the conceptual simplicity of autocommit & explicit transactions.

Is this all worth it? Perhaps not; it's possible to work around the problems with implicit transactions by being careful. And I'm not yet clear on the costs of this approach – just how bad are the problems SQLAlchemy's docs warn about with its autocommit mode?

In any case, while I understand why SQLAlchemy is well-advised to generally follow PEP 249 for its default behavior, I would love if it had (scary-warning-free) support for an "autocommit and explicit transactions" mode on those databases/adapters with good support for it.

The code from this post is pulled together in a gist [https://gist.github.com/carljm/57bfb8616f11bceaf865] . I also have tests for it, but they are currently integrated with the project where I'm using this. If there's enough interest (and it works well on this project) I might be convinced to package it up and release it properly.

Thanks to Mike Bayer [https://twitter.com/zzzeek] for writing SQLAlchemy(!) and for pointing me towards the **`AUTOCOMMIT`** "isolation level" setting. Thanks to Christophe Pettus [http://thebuild.com/blog/] for my initial education in Postgres' transaction behavior, and Aymeric Augustin [https://myks.org/en/] for the excellent implementation in Django 1.6+.

Have feedback on this post? Talk to us on Twitter [https://twitter.com/oddbird] , or email birds@oddbird.net [mailto:birds@oddbird.net] .