Database Connection Pool

1. psycopg (database adapter):

The psycopg connection pool changes, but only slightly, the business of obtaining a connection, compared to not using the pool. We still need to return connections to the pool, otherwise (if you call getconn more times than putconn) we will empty it.

Regarding the implementation of pooling in databases. I approached the maintainer of the psycopg. And he said that a good 90% of the pool code is independent of psycopg and from postgres. As such we can start from the psycopg pool implementations and move most of the implementation to a base class, while leaving subclasses to deal with the different databases.

But there is a concern as well with psycopg, it's a database adapter and Django implementation would be an ORM one and it makes little difference on the API design level. Django connections wrap psycopg connections, then the ORM builds on top of them. I think that there is only a minimal place where to create a psycopg connection in a different way (from the pool rather than from scratch) and the rest should remain pretty unchanged.

Although maybe, if we create a pool of Django connections, then we might not even need to have different subclasses per different backend. If the DB API connection and the Django connection wrapping it are tightly paired. A way to implement it would still be to take the psycopg pool code and adapt it where necessary.

2. Peewee ORM:

Peewee ORM has a pool implementation, but I have not seen any async version of it. The pool implementation had some psycopg2 imports which were mostly to check the possible status of a transaction.

```
try:
    from psycopg2.extensions import TRANSACTION_STATUS_IDLE
    from psycopg2.extensions import TRANSACTION_STATUS_INERROR
    from psycopg2.extensions import TRANSACTION_STATUS_UNKNOWN
except ImportError:
    TRANSACTION_STATUS_IDLE = \
            TRANSACTION_STATUS_INERROR = \
            TRANSACTION_STATUS_UNKNOWN = None
```

They had another import as well:

```
import heapq
```

I had talked with the maintainer of peewee to ask if it is possible to wrap Django around the peewee pool implementation. The maintainer said there's no chance of integrating the two, especially because **asgi and async don't provide a notion of a context local**, which was doubtful. We can use the contextvars module to address that issue. (this is a kind of race condition issue) And I guess there are other ways to make the threads safe. And it might work if we can emulate what peewee does with thread locals. Peewee ORM implementation has an advantage and a disadvantage. The advantage is that it has a common component with Django, it has thread locals for its connection pools, and Django's built-in connection pooler relies on thread-local storage. The disadvantage is that thread-local storage does not work well with asynchronous code.

Peewee can still be worked around. But I still think that we should use the contextvars module over thread local storage because **the main difference is the variables are isolated per *context* rather than per thread, where the context is either per thread or per asyncio Task.**

Peewee ORM has different classes for each database implementation. We can implement that on the API design level and then we need to create an `asynchronous pool class`, which could be achieved with the help of psycopg's AsyncConnectionPool
The `asynchronous pool class` can be assigned as a main class and then we can mix individual database class with the `asynchronous pool class`:

> class AsyncPooledPostgresqlDatabase = AsyncPooledDatabase + PostgresqlDatabase

3. Pgbouncer(Pooling Middleware):

pgbouncer is a lightweight connection pooler for PostgreSQL that can be used to improve the performance and scalability of PostgreSQL database connections. When using Django with ASGI, persistent database connections can be a challenge since ASGI is an asynchronous framework and Django's built-in connection pooler, which relies on thread-local storage, does not work well with asynchronous code. To use pgbouncer with Django's persistent database connections under ASGI, we need to configure Django to use a custom database backend that supports pgbouncer. One such backend is the django-db-pool package, which provides a custom backend that can be used with pgbouncer.

i) We need to install the package:
```
pip install django-db-pool

```
ii) Next, we need to update our Django settings to use the `dbpool.postgrespool` backend instead of the default `django.db.backends.postgresql` backend:
```
DATABASES = {
    'default': {
```

```
        'ENGINE': 'dbpool.postgrespool',
        'NAME': 'your_database_name',
        'USER': 'your_database_user',
        'PASSWORD': 'your_database_password',
        'HOST': 'your_database_host',
        'PORT': 'your_database_port',
        'OPTIONS': {
            'connection_pool_class': 'pgbouncer.pool.PooledConnection',
            'max_overflow': 10,
            'pool_size': 5,
            'pool_timeout': 30,
        },
    }
}

```
In the above code, the `OPTIONS` dictionary contains the configuration options for `django-db-pool`. The `connection_pool_class` option specifies that we want to use `PooledConnection` from the `pgbouncer` package as the connection pool implementation. The `max_overflow` option specifies the maximum number of connections that can be created beyond the pool size, and the `pool_size` option specifies the number of connections that should be kept in the pool. The `pool_timeout` option specifies the number of seconds a connection can be idle before it is closed and removed from the pool.

Once we have configured Django to use django-db-pool, we can use Django's persistent database connections as usual. The connection pooler will manage the connections to PostgreSQL and ensure that they are properly reused and released, improving the performance and scalability of your application. But there are some difficulties around this implementation. The author of `django-db-pool` has specifically mentioned that **code has not been rigorously tested in high-volume production systems.**

The package is dependent on a few things, connection pooling is implemented by thinly wrapping a psycopg2 connection object with a pool-aware class. The actual pool implementation is psycop2g's built-in ThreadedConnectionPool, which handles thread safety for the pool instance.
But I think to use pgbouncer with Django's persistent database connections under ASGI, we need to configure Django to use a custom backend that supports pgbouncer.

4. SQLAlchemy (Database toolkit for python):

SQLAlchemy has a more advanced database connection pooling implementation. Django ORM and SQLAlchemy are both Object Relational Mappers (ORMs) that provide abstractions for working with relational databases. However, they have different design goals and implementation details, so they cannot be simply wrapped around each other.

That being said, Django 3.1 introduced support for ASGI (Asynchronous Server Gateway Interface) through the ASGI application instance, which is responsible for asynchronously

handling HTTP requests and responses. To use ASGI with Django, we need to use an ASGI server such as Daphne, Uvicorn or Hypercorn, and an ASGI middleware that can handle Django's database connections asynchronously.

Another similar approach could be to use an asynchronous database library like SQLAlchemy Core. SQLAlchemy Core provides an async API that allows for non-blocking database access, which is better suited for ASGI.
To use SQLAlchemy with Django, we can create a custom database backend that uses SQLAlchemy as the underlying database driver.

1. We need to install some requirements:
```
pip install sqlalchemy databases

```
2. Create a custom database backend in Django that uses SQLAlchemy:
```
from django.db.backends.base.base import BaseDatabaseWrapper
from databases import DatabaseURL
from sqlalchemy import create_engine

class AsyncSQLAlchemyDatabaseWrapper(BaseDatabaseWrapper):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.database_url = DatabaseURL(self.settings_dict['NAME'])
        self.engine = create_engine(str(self.database_url))

    def get_new_connection(self, conn_params):
        return self.engine.connect()

    def init_connection_state(self):
        pass

    def create_cursor(self, name=None):
        return self.connection

    def _close(self):
        self.connection.close()

```
3. Configure Django to use the custom database backend by updating the `DATABASES` setting in your `settings.py` file:
```
DATABASES = {
    'default': {
        'ENGINE': 'path.to.AsyncSQLAlchemyDatabaseWrapper',
        'NAME': 'your_database_name',
    }
```

}

```
There are some packages that I have seen that use SQLAlchemy to implement the same,
1. django-db-connection-pool
2. django-dbpool-backend
3. django-postgrespool
```

What is the issue with the persistent database connections under ASGI?

The issue could be traced down to the **AsyncToSync** function, when a thread already has a running async loop, **AsyncToSync** can't run things on that loop if it's blocked on synchronous code that is always above you in the call stack. According to the documentation of ASGI, the only real solution is to have a variant of **ThreadPoolExecutor** that executes any *thread_sensitive* code on the **outermost synchronous** thread - either the main thread or a single spawned subthread. And if the outermost layer of the program is synchronous, then all async code run through AsyncToSync will run in a per-call event loop in arbitrary subthreads, while all thread_sensitive code will run in the main thread. Now my conclusion would be that there is a per-call event loop, there are many connections to the database through async code and it is not possible for async code to maintain persistent connections.

Summary

Django's persistent database connections rely on the underlying database driver to manage the connection pool. When using ASGI, the database driver needs to support asynchronous I/O, and Django needs to use an ASGI-compliant database backend.

Thanks,

Kaushik