Database Connection Pooling in Django

1. Introduction

Database connection pooling is a technique to reuse existing database connections instead of creating new ones for every request. This helps reduce connection overhead and improves performance.

Django supports connection pooling via the `CONN_MAX_AGE` setting, which controls how long a database connection remains open before being closed.

---

2. How Django Handles Connection Pooling

By default, Django creates a **new database connection** for each request and closes it after the request is completed. This can lead to performance issues when handling multiple concurrent requests.

To enable connection pooling, we use:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'mydb',
        'USER': 'myuser',
        'PASSWORD': 'mypassword',
        'HOST': 'localhost',
        'PORT': '5432',
        'CONN_MAX_AGE': 600,  # Keep connections open for 10 minutes
    }
}
```

Database Pooling

Behavior of `CONN_MAX_AGE`

1. If a connection is **available and within the timeout**, Django **reuses it**.
2. If a connection is **idle for more than 10 minutes**, Django **closes it**.
3. If a new request comes after 10 minutes, Django **creates a new connection**.

---

3. Does Pooling Require Query Modifications?

No! Django automatically manages connection reuse when `CONN_MAX_AGE` is set. Example:

```
user = User.objects.get(id=1)  # This query will reuse an open connection if available
```

You don't need to manually handle pooling in your queries.

---

4. Database Connection Limits

Every database has a limit on the maximum number of concurrent connections:

- **PostgreSQL** (`max_connections`): Default **100** (can be increased via `ALTER SYSTEM SET max_connections = 500;`)
- **MySQL** (`max_connections`): Default **151** (can be changed in `my.cnf`)

If too many connections are created, the database may reject new connections, leading to errors.

---

5. Optimizing Pooling with PgBouncer

Database Pooling

If your application has **high concurrent traffic**, using PgBouncer can further optimize pooling. PgBouncer allows many users to share a small number of actual database connections.

PgBouncer Configuration (`pgbouncer.ini`)

```
[pgbouncer]
max_client_conn = 1000
default_pool_size = 50
pool_mode = transaction  # Best for Django
```

Update Django Settings for PgBouncer

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'mydb',
        'USER': 'myuser',
        'PASSWORD': 'mypassword',
        'HOST': '127.0.0.1',  # PgBouncer host
        'PORT': '6432',  # PgBouncer port
        'CONN_MAX_AGE': 600,
    }
}
```

PgBouncer Benefits

- ✅ Reduces database connection overhead
- ✅ Supports **1000+ users** with only **50 real DB connections**
- ✅ Prevents hitting `max_connections` limit

---

6. Best Practices

Database Pooling

✔ **Use a single database user** for Django {

```
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'mydb',
        'USER': 'myuser',
        'PASSWORD': 'mypassword',
        'HOST': '127.0.0.1',  # PgBouncer host
        'PORT': '6432',  # PgBouncer port
        'CONN_MAX_AGE': 600,
    }
}
```

(`myuser`) instead of separate users per request. ✔ **Increase `max_connections`** if your database supports high traffic. ✔ **Enable PgBouncer** for production environments. ✔ **Monitor active connections** with:

```
SELECT * FROM pg_stat_activity;
```

---

7. Conclusion

- **Django's `CONN_MAX_AGE`** allows automatic connection pooling.
- **PgBouncer** improves scalability by reducing the number of actual DB connections.
- **No need to modify queries**, as Django manages connections automatically.
- For **high concurrency**, use **PgBouncer** + **optimized DB settings**.