Practical Guide to Database Locks with Django

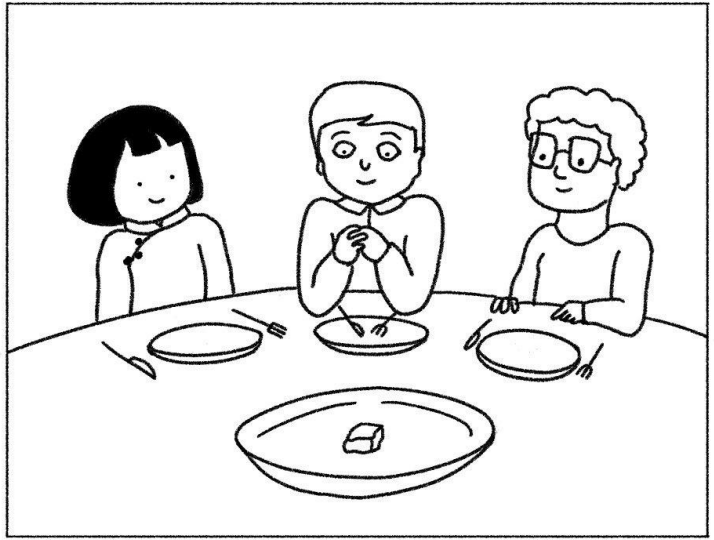
Efe Öge

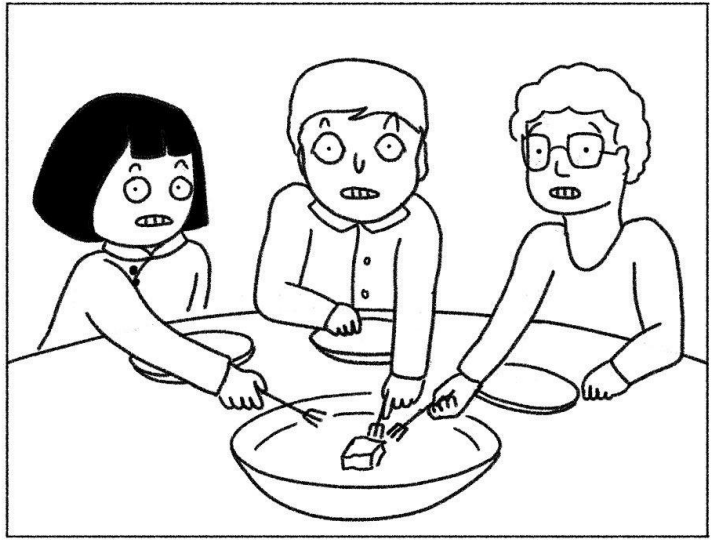
- **Tech Lead & Senior Backend Developer**
- efe.me

Table of contents

- ~30 minutes
- Q&A at the end
- Mainly about PostgreSQL,
  not sure about other databases
- Examples on Django web framework

Why do **we** need lock?


© tinyeyescomics


© tinyeyescomics

It is called
“concurrency”

Event Tickets

Event Tickets

- A platform like Ticketmaster or Songkick for **selling tickets for many events**.
- List and search events, get tickets.
- Imagine **PyCon Portugal 2023**, **Dua Lipa concert** as events.
- Online platform, **multiple users**.

Models

```python
from django.db import models


class Event(models.Model):
    name = models.CharField(max_length=128)
    capacity_left = models.IntegerField()


class Ticket(models.Model):
    user = models.ForeignKey("users.User")
    event = models.ForeignKey("events.Event", related_name="tickets")
```

Creation of Ticket

```python
class Ticket(models.Model):
    # fields..

    @classmethod
    def create(cls, event_id, user_id):
        event = Event.objects.get(id=event_id)

        if event.capacity_left == 0:
            raise ValidationError("There is no space left.")

        ticket = cls.objects.create(event_id=event_id, user_id=user_id)

        event.capacity_left = event.capacity_left - 1
        event.save(update_fields=["capacity_left"])

        ticket.send_invoice()

        return ticket
```

On a sunny day

```
>> pycon_portugal = Event.objects.get(name="PyCon Portugal 2023")
>> print(pycon_portugal.tickets.count())
201
```

- But the capacity was 200!

Creation of Ticket

```python
class Ticket(models.Model):
    # fields..

    @classmethod
    def create(cls, event_id, user_id):
        event = Event.objects.get(id=event_id)

        if event.capacity_left == 0:
            raise ValidationError("There is no space left.")

        ticket = cls.objects.create(event_id=event_id, user_id=user_id)

        event.capacity_left = event.capacity_left - 1
        event.save(update_fields=["capacity_left"])

        ticket.send_invoice()

        return ticket
```

Time-of-check to time-of-use

On a sunny day

- **Maria** and **Pedro** would like to buy **PyCon Portugal ticket.**
- There is only **1 seat left.**
- We have multiple processes, app servers…

- Process 1 checks is not capacity_left 0 ✅
- Process 2 checks is not capacity_left 0 ✅
- Process 2 creates the a ticket for **Maria.** ✅
- Process 1 creates the a ticket for **Pedro.** ✅

- Race Condition!
- We created **an extra ticket**! 🤦

Lock the table

Lock the table

```python
class Ticket(models.Model):
    # fields..

    @classmethod
    def create(cls, event_id, user_id):
        with transaction.atomic(), connection.cursor() as cursor:
            cursor.execute('LOCK TABLE tickets_ticket IN EXCLUSIVE MODE;')

            event = Event.objects.get(id=event_id)

            if event.capacity_left == 0:
                raise ValidationError("There is no space left.")

            ticket = cls.objects.create(event_id=event_id, user_id=user_id)

            event.capacity_left = event.capacity_left - 1
            event.save(update_fields=["capacity_left"])

        ticket.send_invoice()

        return ticket
```

slows down!

Lock the row

Lock the row

```python
class Ticket(models.Model):
    # fields..

    @classmethod
    def create(cls, event_id, user_id):
        with transaction.atomic():
            event = Event.objects.select_for_update().get(id=event_id)

            if event.capacity_left == 0:
                raise ValidationError("There is no space left.")

            ticket = cls.objects.create(event_id=event_id, user_id=user_id)

            event.capacity_left = event.capacity_left - 1
            event.save(update_fields=["capacity_left"])

        ticket.send_invoice()

        return ticket
```

You did
Pessimist
Locking

Optimist
and
Pessimist
Lock Strategies

Pessimist Strategy

- “As soon as one user starts to update a record, a lock is placed on it.” —IBM

Optimist Strategy

```python
class Ticket(models.Model):
    # We have new "version" field as extra.
    version = models.PositiveIntegerField(default=0)

    @classmethod
    def create(cls, event_id, user_id):
        event = Event.objects.get(id=event_id)

        if event.capacity_left == 0:
            raise ValidationError("There is no space left.")

        Event.objects.filter(
            id=event_id,
            version=event.version,
        ).update(
            capacity_left=event.capacity_left - 1,
            version=event.version + 1,
        )

        ticket = cls.objects.create(event_id=event_id, user_id=user_id)

        ticket.send_invoice()

        return ticket
```

- ❏ Introduce a “version” field.
- ❏ django-optimistic-lock and django-concurrency packages.

One more thing, The forgotten locks…

A quiz?

```sql
SELECT *
FROM events_event
LIMIT 5
```

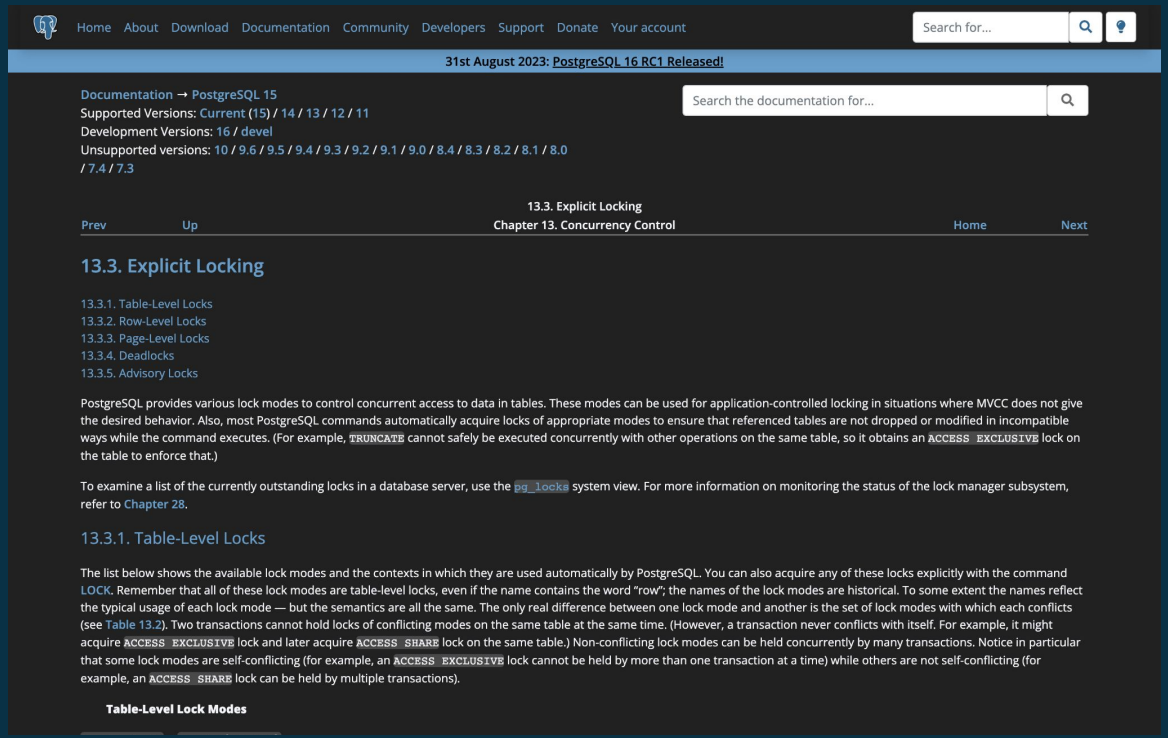
- ❏ AccessShareLock (table)

Another quiz?

```
1 INSERT INTO events_event (name, capacity_left)
2 VALUES ('PyCon Turkey', 100);
```

- ❏ RowExclusiveLock (table)

13.3. Explicit Locking Documentation



Home About Download Documentation Community Developers Support Donate Your account

Search for...

31st August 2023: PostgreSQL 16 RC1 Released!

Documentation → PostgreSQL 15
Supported Versions: Current (15) / 14 / 13 / 12 / 11
Development Versions: 16 / devel
Unsupported versions: 10 / 9.6 / 9.5 / 9.4 / 9.3 / 9.2 / 9.1 / 9.0 / 8.4 / 8.3 / 8.2 / 8.1 / 8.0 / 7.4 / 7.3

Search the documentation for...

13.3. Explicit Locking
Prev Up Chapter 13. Concurrency Control Home Next

13.3. Explicit Locking

13.3.1. Table-Level Locks
13.3.2. Row-Level Locks
13.3.3. Page-Level Locks
13.3.4. Deadlocks
13.3.5. Advisory Locks

PostgreSQL provides various lock modes to control concurrent access to data in tables. These modes can be used for application-controlled locking in situations where MVCC does not give the desired behavior. Also, most PostgreSQL commands automatically acquire locks of appropriate modes to ensure that referenced tables are not dropped or modified in incompatible ways while the command executes. (For example, `TRUNCATE` cannot safely be executed concurrently with other operations on the same table, so it obtains an `ACCESS EXCLUSIVE` lock on the table to enforce that.)

To examine a list of the currently outstanding locks in a database server, use the `pg_locks` system view. For more information on monitoring the status of the lock manager subsystem, refer to Chapter 28.

13.3.1. Table-Level Locks

The list below shows the available lock modes and the contexts in which they are used automatically by PostgreSQL. You can also acquire any of these locks explicitly with the command LOCK. Remember that all of these lock modes are table-level locks, even if the name contains the word "row"; the names of the lock modes are historical. To some extent the names reflect the typical usage of each lock mode — but the semantics are all the same. The only real difference between one lock mode and another is the set of lock modes with which each conflicts (see Table 13.2). Two transactions cannot hold locks of conflicting modes on the same table at the same time. (However, a transaction never conflicts with itself. For example, it might acquire `ACCESS EXCLUSIVE` lock and later acquire `ACCESS SHARE` lock on the same table.) Non-conflicting lock modes can be held concurrently by many transactions. Notice in particular that some lock modes are self-conflicting (for example, an `ACCESS EXCLUSIVE` lock cannot be held by more than one transaction at a time) while others are not self-conflicting (for example, an `ACCESS SHARE` lock can be held by multiple transactions).

Table-Level Lock Modes

Databases
also
need lock
for themself

Table Level Locks

- ACCESS SHARE
- ROW SHARE
- ROW EXCLUSIVE
- SHARE UPDATE EXCLUSIVE
- SHARE
- SHARE ROW EXCLUSIVE
- EXCLUSIVE
- ACCESS EXCLUSIVE

Conflicts of Table Level Locks

Lock Mode	ACCESS SHARE	ROW SHARE	ROW EXCLUSIVE	SHARE UPDATE EXCLUSIVE	SHARE	SHARE ROW EXCLUSIVE	EXCLUSIVE	ACCESS EXCLUSIVE
ACCESS SHARE								X
ROW SHARE							X	X
ROW EXCLUSIVE					X	X	X	X
SHARE UPDATE EXCLUSIVE				X	X	X	X	X
SHARE			X	X		X	X	X
SHARE ROW EXCLUSIVE			X	X	X	X	X	X
EXCLUSIVE		X	X	X	X	X	X	X
ACCESS EXCLUSIVE	X	X	X	X	X	X	X	X

Conflicts of Table Level Locks

Lock Mode	ACCESS SHARE	ROW SHARE	ROW EXCLUSIVE	SHARE UPDATE EXCLUSIVE	SHARE	SHARE ROW EXCLUSIVE	EXCLUSIVE	ACCESS EXCLUSIVE
ACCESS SHARE								X
ROW SHARE							X	X
ROW EXCLUSIVE					X	X	X	X
SHARE UPDATE EXCLUSIVE				X	X	X	X	X
SHARE			X	X		X	X	X
SHARE ROW EXCLUSIVE			X	X	X	X	X	X
EXCLUSIVE		X	X	X	X	X	X	X
ACCESS EXCLUSIVE	X	X	X	X	X	X	X	X

Row Level Locks

- FOR UPDATE
- FOR NO KEY UPDATE
- FOR SHARE
- FOR KEY SHARE

Row Level Locks

Lock Mode	FOR KEY SHARE	FOR SHARE	FOR NO KEY UPDATE	FOR UPDATE
FOR KEY SHARE				X
FOR SHARE			X	X
FOR NO KEY UPDATE		X	X	X
FOR UPDATE	X	X	X	X

pglocks.org

PostgreSQL Lock Conflicts

Database engineering course | *@hnasr* |

PostgreSQL Lock Conflicts

This tool shows all commands and locks in postgres. If you select a command, it lists the locks that it acquires, commands that conflicts with it and commands that are allowed to run concurrently with it (with no conflict or blocking). If you select a lock, it lists commands that acquire the lock and what are the other conflicting locks.

Locks

1. AccessShareLock (table)
2. RowShareLock (table)
3. RowExclusiveLock (table)
4. ShareUpdateExclusiveLock (table)
5. ShareLock (table)
6. ShareRowExclusiveLock (table)
7. ExclusiveLock (table)
8. AccessExclusiveLock (table)
9. FORKEYSHARE (row)
10. FORSHARE (row)
11. FORNOKEYUPDATE (row)
12. FORUPDATE (row)

- ❏ I don’t care about lock conflicts.
- ❏ I care about **conflicts of SQL Commands**.
- ❏ Thanks to **Hussein Nasser**.

Downtime on deployments with migration?

Commands conflicting with SELECT

- **VACUUM FULL**
- TRUNCATE
- **REINDEX**
- DROP TABLE
- ALTER TABLE SET/DROP DEFAULT
- ALTER TABLE RENAME
- ALTER TABLE DROP CONSTRAINT
- ALTER TABLE DROP COLUMN
- ALTER TABLE ALTER CONSTRAINT
- ALTER TABLE ADD COLUMN
- ALTER TABLE ADD CONSTRAINT
- …

Commands not conflicting with SELECT

- VACUUM
- REINDEX CONCURRENTLY
- CREATE INDEX CONCURRENTLY
- …

Downtime on deployments with migration?

Concurrent Migration Operations

- Django supports **AddIndexConcurrently** and **RemoveIndexConcurrently**.
- Change **AddIndex** to **AddIndexConcurrently** in the migration file.
- DDL (Data Definition Language) txns -CREATE, DROP, ALTER, TRUNCATE- don’t support atomic. Then, Set atomic = False in the migration file.
- Add index first concurrently, Then have another deployment.

Auchtung!

- Don't create NOT NULL column in a deployment for large tables.
- Don't create column with default value in large table.

Obrigado!

Slides are available
at efe.me

Any Question?