

Transactions for web developers

Aymeric Augustin - @aymericaugustin

DjangoCon Europe - May 17th, 2013

“Transaction management tools are often made to seem like a **black art.**”

Christophe Pettus (2011)
Life with Object-Relational Mappers

Today's talk

django

Framework

sqlite3

psycopg2

Interface



Database

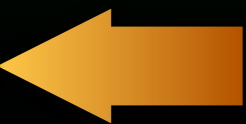


SQL-92

Framework

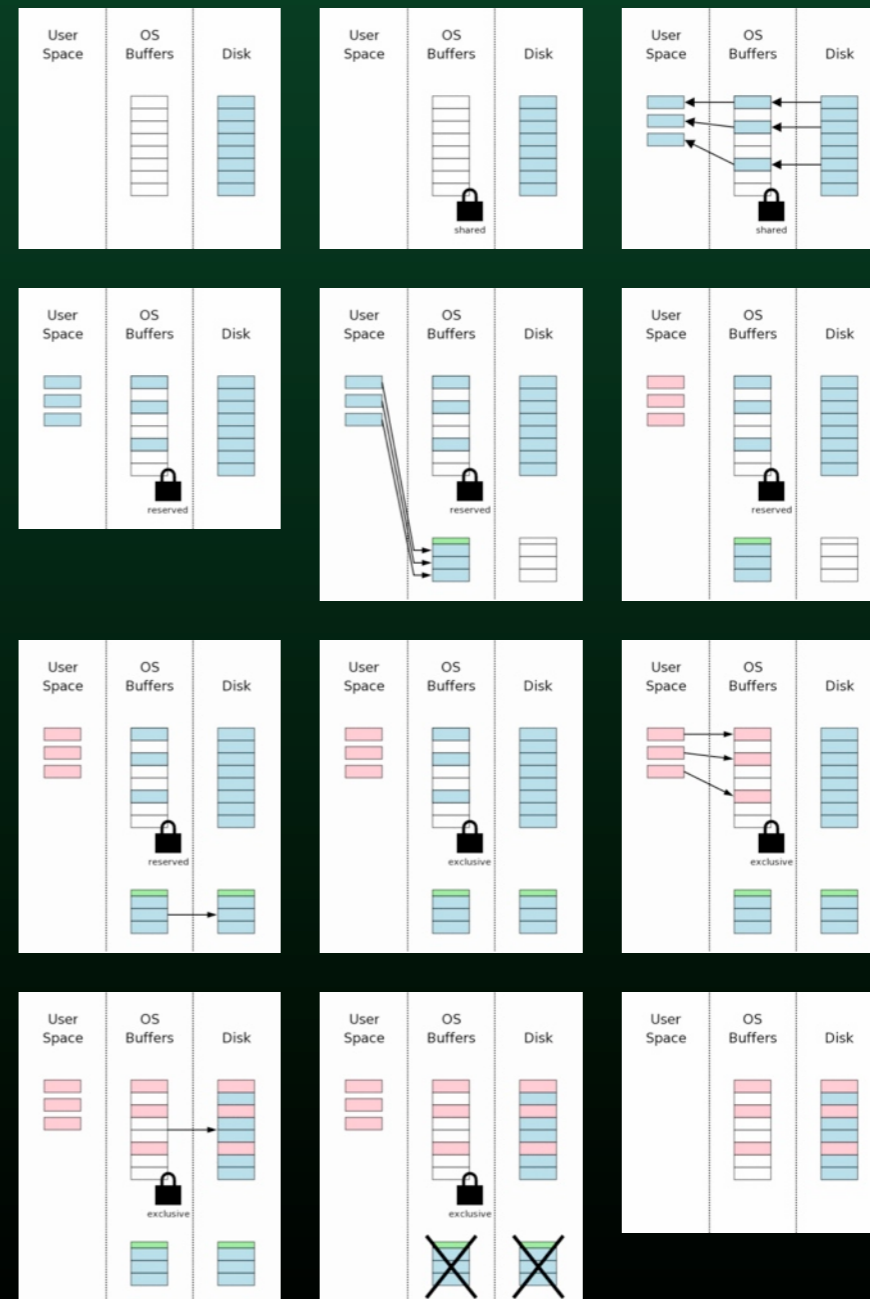
Interface

Database



Definition

“An **SQL-transaction** (sometimes simply called a "transaction") is a sequence of executions of SQL-statements that is **atomic** with respect to recovery.”



Definition

“An **SQL-transaction** (sometimes simply called a "transaction") is a sequence of executions of SQL-statements that is **atomic** with respect to recovery.”



Example

BEGIN;

```
UPDATE cars
  SET status = 'available'
  WHERE id = 1;
```

```
UPDATE rentals
  SET end = CURRENT_TIMESTAMP
  WHERE car_id = 1 AND end IS NULL;
```

COMMIT;

Lifecycle

- A transaction begins with a transaction-initiating statement.
- A transaction ends with a commit, an explicit rollback, or an implicit rollback.

Isolation levels

At this level, these phenomena **cannot** happen:

READ
UNCOMMITTED

-

READ
COMMITTED

dirty read

read a row that is never committed

REPEATABLE
READ

non repeatable read

read different values for the same row

SERIALIZABLE
(default)

phantom read

read different rows for the same query

SERIALIZABLE also guarantees that transactions are serializable.

Statements

- COMMIT
- ROLLBACK

(advanced)

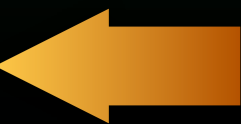
- SET TRANSACTION ...
- SET CONSTRAINTS ...

SQL:1999

Framework

Interface

Database



Savepoints



becomes either

after



SAVEPOINT S3;

or



RELEASE SAVEPOINT S2;

or



ROLLBACK TO SAVEPOINT S2;

Explicit transaction start

```
START TRANSACTION;
```

```
UPDATE cars  
  SET status = 'available'  
 WHERE id = 1;
```

```
UPDATE rentals  
  SET end = CURRENT_TIMESTAMP  
 WHERE car_id = 1 AND end IS NULL;
```

```
COMMIT;
```

Statements

- `SAVEPOINT <svpt>`
- `RELEASE SAVEPOINT <svpt>`
- `ROLLBACK TO SAVEPOINT <svpt>`

(advanced)

- `START TRANSACTION ...`

Key learnings

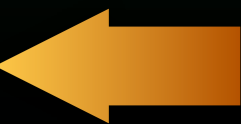
- Statements always run in a transaction.
- Transactions are opened automatically.
- Transactions are advanced technology.

A brief interlude

Framework

Interface

Database



The dreaded PostgreSQL error:

**ERROR: current transaction is aborted,
commands ignored until end of transaction block**

What it actually means:

**a previous statement failed,
the application must perform a rollback**

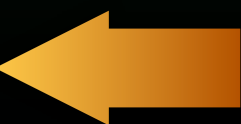
Any “auto-recovery” scheme breaks transactional integrity!

Autocommit

Framework

Interface

Database

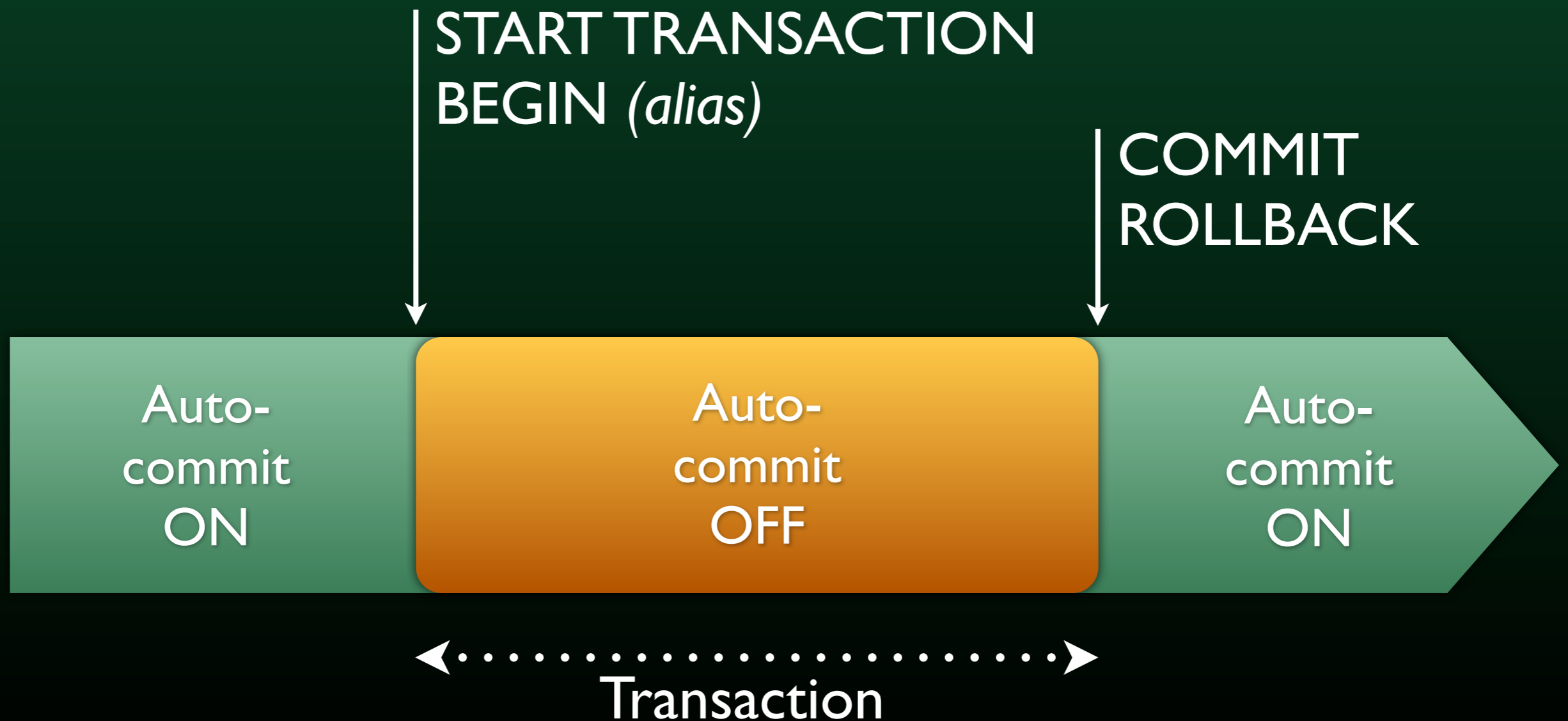


What does it mean?

- Commit implicitly after each statement.
- Wrap each statement in its own transaction.
- Just execute my query KTHXBYE!

- Most databases default to autocommit.

Transactions in autocommit



Autocommit in PostgreSQL

- The server always uses autocommit.
- Clients can emulate standard behavior by inserting implicit BEGIN statements.
- In psql: `\set autocommit off`

Autocommit in SQLite

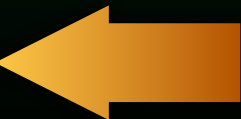
- Transaction semantics are tightly related to the implementation of atomic commit.
- SQLite automatically starts a transaction before all statements except `SELECT`.
- It automatically commits such transactions as soon as all statements finish executing.

Python client libraries

Framework

Interface

Database



DB API 2.0 - PEP 249

- Connection
 - Performs commits and rollbacks
- Cursor
 - Executes queries
 - Fetches results

Transactions in PEP 249

- “If the database supports an auto-commit feature, this must be initially off.”
- “An interface method may be provided to turn it back on.”
- “Closing a connection without committing the changes first will cause an implicit rollback to be performed.”

Example

```
>>> from psycopg2 import connect
>>> cnx = connect(...); cur = cnx.cursor()

>>> cur.execute("INSERT INTO cars "
...             "VALUES (1, 'available');")

>>> cnx = connect(...); cur = cnx.cursor()

>>> cur.execute("SELECT * FROM cars;")
>>> cur.fetchall()
[]
```

Example

```
>>> from psycopg2 import connect
>>> cnx = connect(...); cur = cnx.cursor()

>>> cur.execute("INSERT INTO cars "
...             "VALUES (1, 'available');")
>>> cnx.commit()

>>> cnx = connect(...); cur = cnx.cursor()

>>> cur.execute("SELECT * FROM cars;")
>>> cur.fetchall()
[(1, 'available')]
```

Transactions in psycopg2

- psycopg2 tracks the transaction state.
- It inserts a BEGIN before each statement, unless there's already a transaction in progress.
 - Even before SELECT statements.
 - “Idle in transaction”.
- `cnx.autocommit = True` disables this behavior.

Transactions in sqlite3

- sqlite3 tracks the transaction state.
- It parses statements to insert BEGIN or COMMIT:
 - SELECT: —
 - INSERT, UPDATE, DELETE, REPLACE: BEGIN
 - Any other statement: COMMIT
- `cnx.isolation_level = None` disables this behavior.
 - This is totally unrelated to the isolation level!

Transactions in sqlite3

- sqlite3 tracks the transaction state.
- It parses statements and inserts before:
 - SELECT: —
 - INSERT, UPDATE, DELETE, REPLACE: BEGIN
 - Any other statement: COMMIT
- **This is broken by design.**
 - “SAVEPOINT foo” triggers a commit!

Key learnings

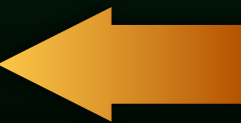
- The DB API requires the same transactional behavior as the SQL standard.
- Client libraries for databases that always autocommit have to emulate this behavior.
- But you can turn it off and use autocommit.

Problems in Django \leq 1.5

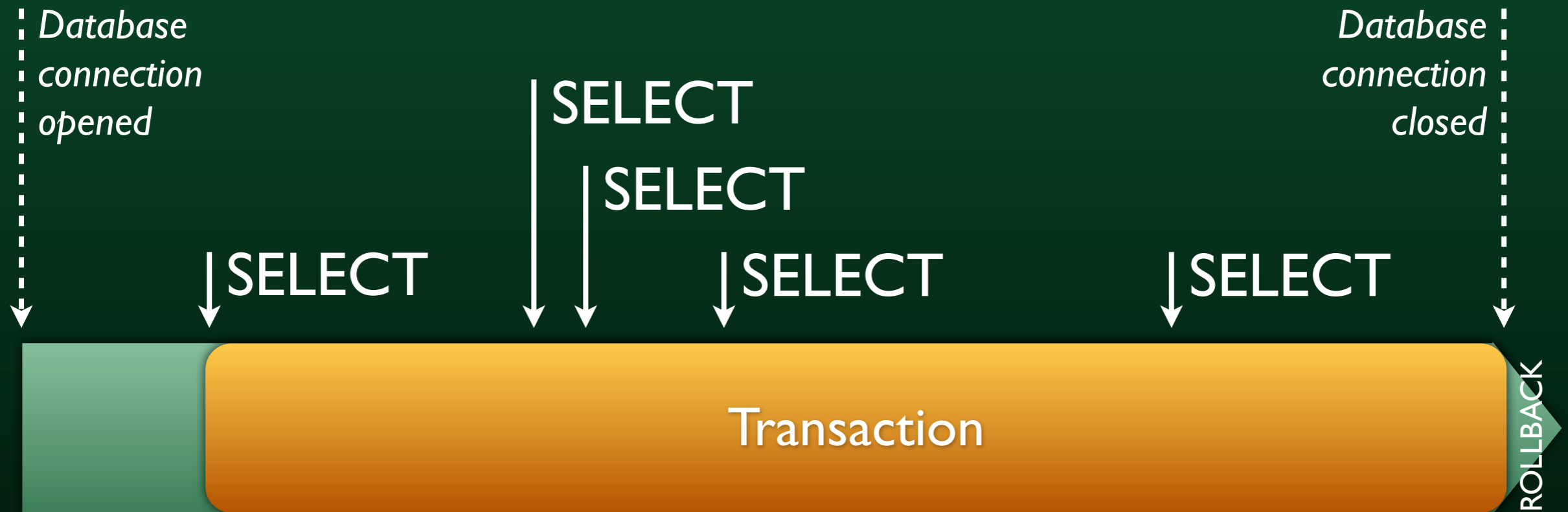
Framework

Interface

Database

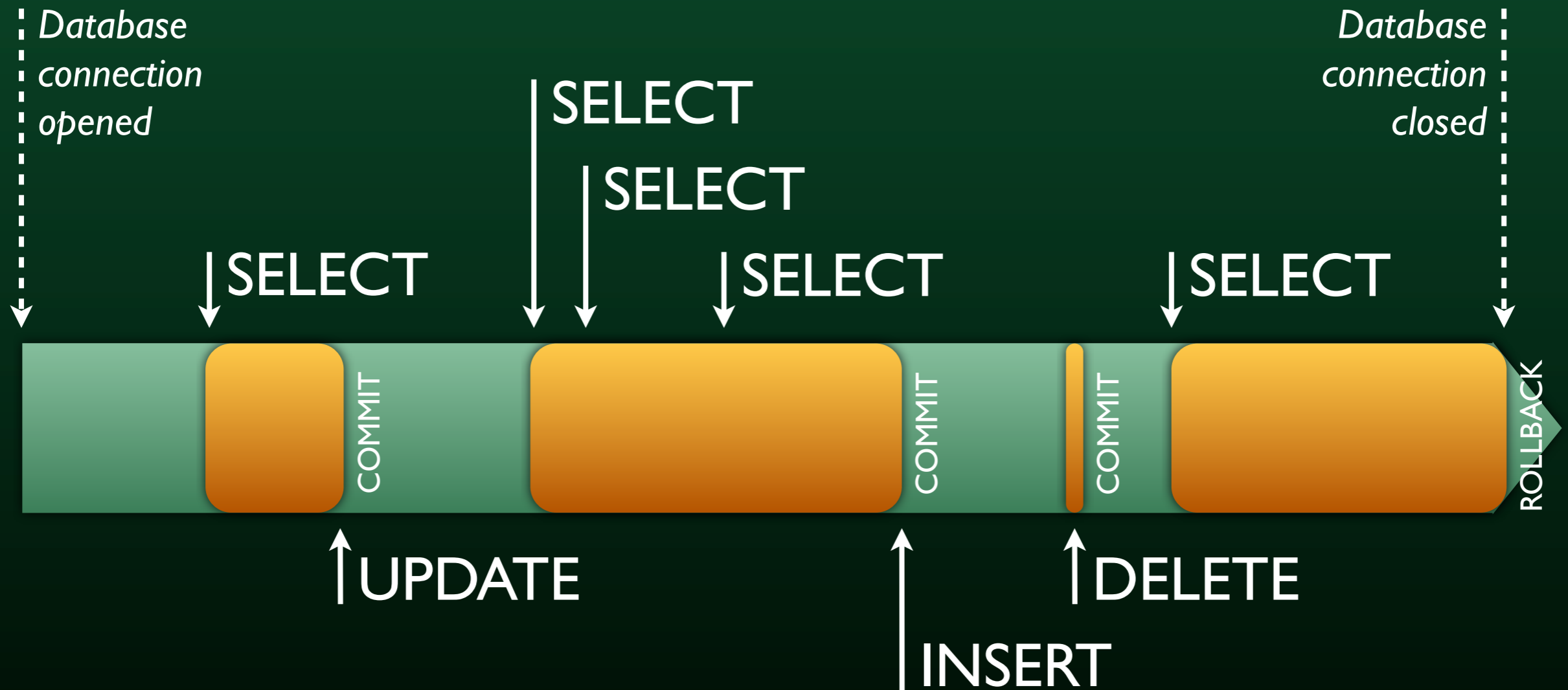


Default behavior (≤ 1.5)



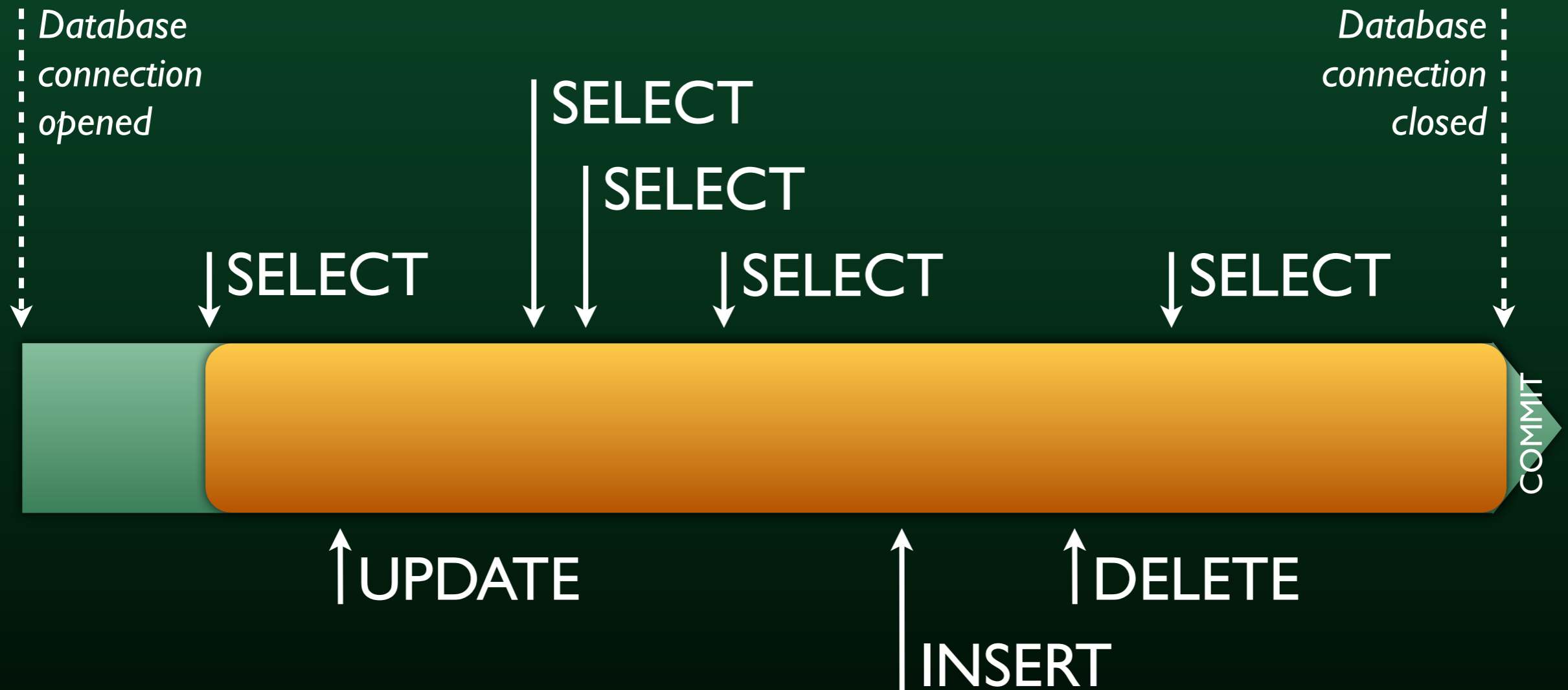
Django runs with an open transaction.

Default behavior (≤ 1.5)



Django “auto-commits” in `save()`, `delete()`, `update()` and `bulk_create()`.

Transaction middleware



One HTTP request \approx one transaction. Commit on success, roll back on exception.

Only for the default database!

May or may not apply to middleware!

High-level APIs

```
from django.db import transaction

with transaction.autocommit():
    # Like Django's default behavior.

with transaction.commit_on_success():
    # Like the transaction middleware.

with transaction.commit_manually():
    # Don't use it.
```

Low-level APIs

```
from django.db import transaction
```

```
transaction.commit()  
transaction.rollback()
```

```
sid = transaction.savepoint()
```

```
transaction.savepoint_commit(sid)  
transaction.savepoint_rollback(sid)
```

How do they interact with Django's transaction management?

Behind the scenes

- Django maintains a stack of transaction management states:
 - “Auto” (False): the ORM commits every change.
 - “Managed” (True): Django doesn’t commit.
- Django maintains a “dirty” flag:
 - Set automatically by the ORM after writes.
 - Must be set manually after raw SQL writes.

Nesting doesn't work well

```
from django.db import transaction
from music.models import Guitar

guitars = (Guitar.objects
           .filter(brand="Fender"))

with transaction.commit_on_success():
    guitars.update(brand="Gibson")
    with transaction.autocommit():
        guitars.compute_ratings()
```

compute_ratings() raises an exception. Is the brand change saved?

Nesting doesn't work well

```
from django.db import transaction
from music.models import Guitar

guitars = (Guitar.objects
            .filter(brand="Fender"))

with transaction.commit_on_success():
    guitars.update(brand="Gibson")
    with transaction.commit_manually():
        guitars.compute_ratings()
```

compute_ratings() raises an exception. Is the brand change saved?

Key learnings

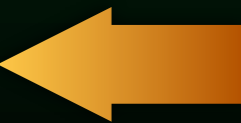
- If you never understood transaction management in Django \leq 1.5, that's fine.
- The transaction middleware is a reasonable idea.
- The decorators/context managers don't work well, especially when they're nested.

Solution in Django \geq 1.6

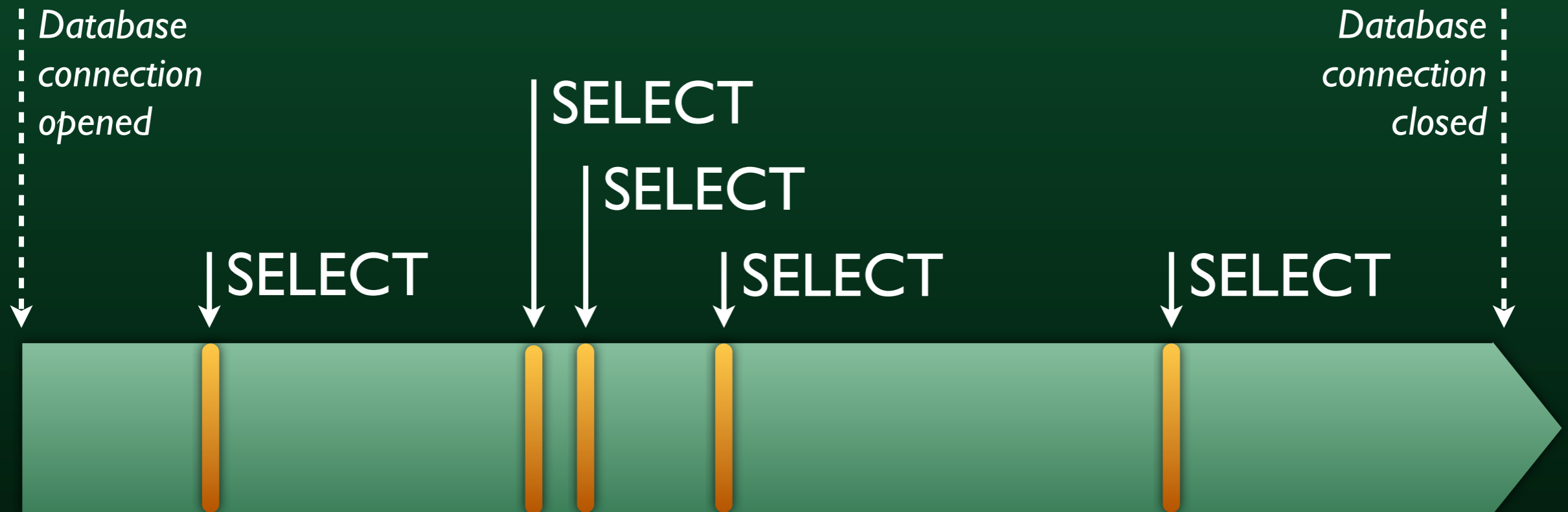
Framework

Interface

Database

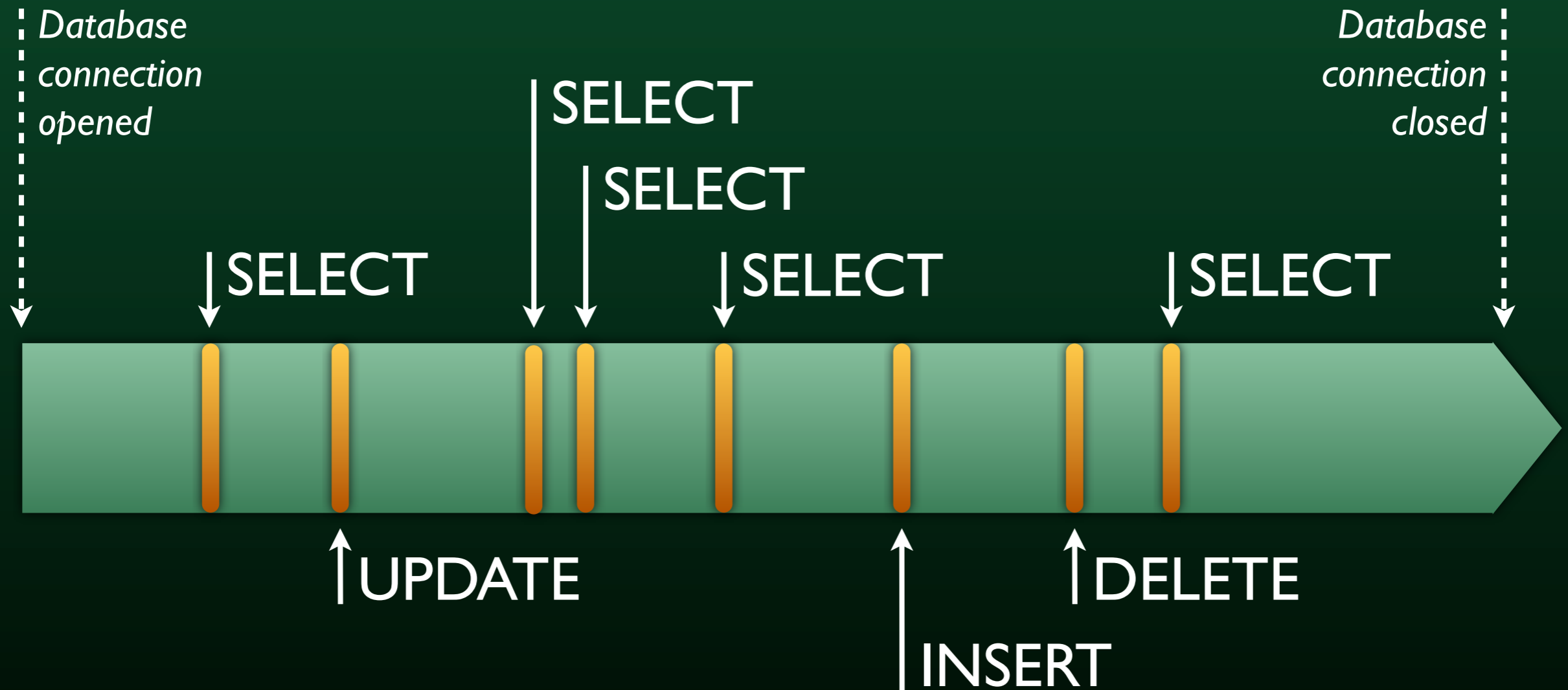


Default behavior (≥ 1.6)



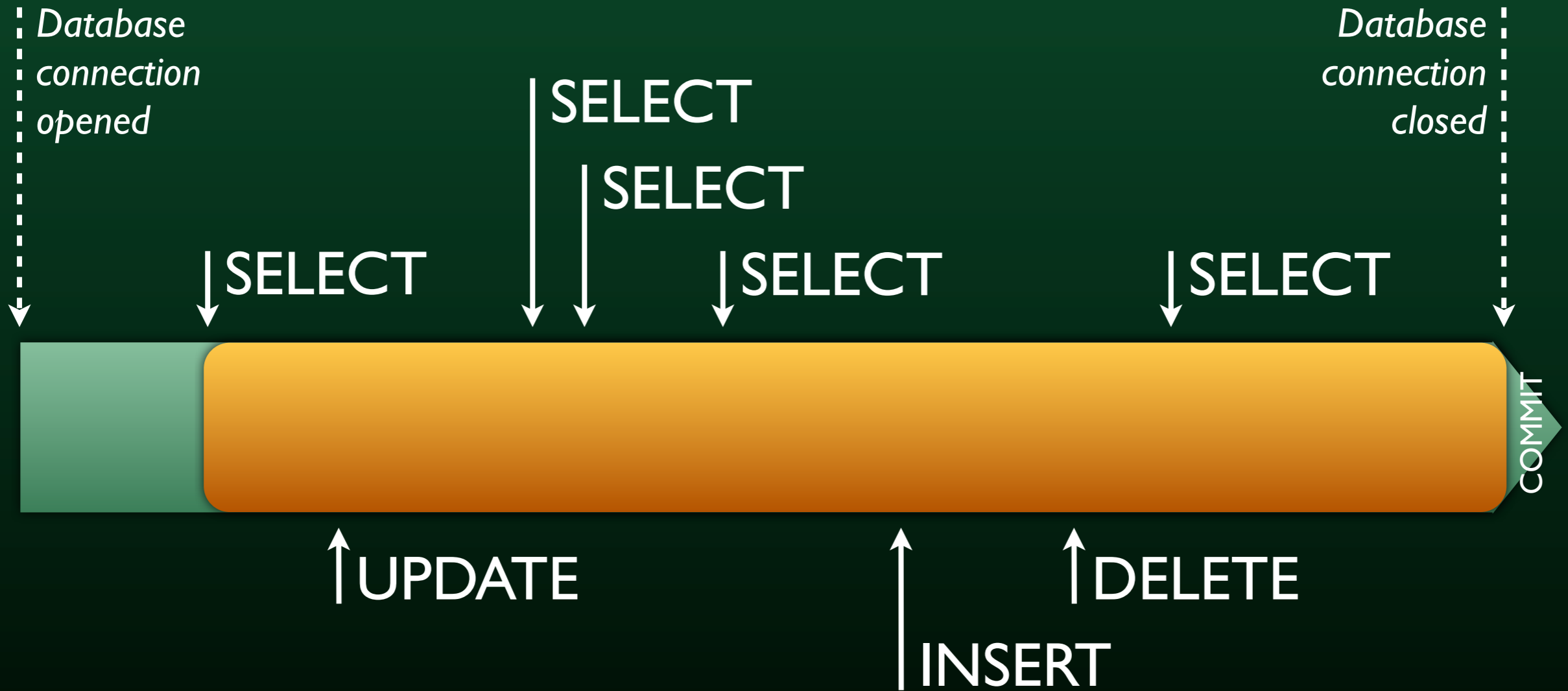
Django ≥ 1.6 uses database-level autocommit.

Default behavior (≥ 1.6)



Django ≥ 1.6 uses database-level autocommit.

ATOMIC_REQUESTS



One HTTP request = one transaction. Commit on success, roll back on exception.
Per database. Only for the view function.

High-level API: **atomic**

- Usable as a decorator or as a context manager.
- Commits on success, rolls back on exceptions.
- Can be nested without any restrictions.
 - Concept stolen from **xact** by Christophe Pettus.
- Guarantees atomicity.
 - “*Don’t Give Users Guns Aimed At Feet*” principle.

High-level API: **atomic**

```
from django.db import IntegrityError
from django.db import transaction

@transaction.atomic
def viewfunc(request):
    do_stuff()

    try:
        with transaction.atomic():
            do_stuff_that_could_fail()
    except IntegrityError:
        handle_exception()

    do_more_stuff()
```

Low-level APIs

- In case you want to implement your own transaction management.
- There aren't many sane ways to combine them.

Key learnings

- If you don't understand transaction management in Django \geq 1.6, get to work!
- `ATOMIC_REQUESTS` is still a reasonable idea.
- Use `atomic` whenever you need atomicity.

Thank you!

Questions?

<https://docs.djangoproject.com/en/dev/topics/db/transactions/>