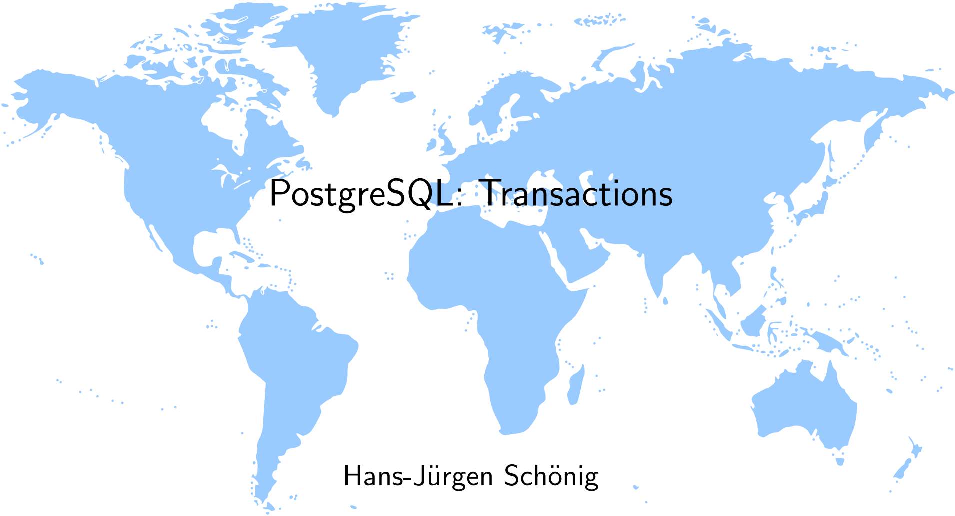

PostgreSQL: Transactions

Hans-Jürgen Schönig

November 9, 2015

Introduction

PostgreSQL transactions

- PostgreSQL uses the ACID transaction model:
  - Atomic
  - Consistent
  - Isolated
  - Durable
- All transactions are ACID compliant

The transaction model

- A transaction has to complete or it will fail entirely.
- The basic principle: Everything or nothing
  - Other databases such as Oracle use different models
  - Oracle can commit partially correct transactions
  - This requires more error handling

An example (1)

```
test=# BEGIN;
test=# SELECT 1;
...
test=# SELECT 1 / 0;
ERROR:  division by zero
test=# SELECT 1 / 0;
ERROR:  current transaction is aborted, commands
    ignored until end of transaction block
test=# SELECT 1 / 0;
ERROR:  current transaction is aborted, commands
    ignored until end of transaction block
test=# COMMIT;
ROLLBACK
```

An example (2)

- After the error the transaction can only ROLLBACK
- All commands will be ignored
- A transaction has to be correct

Using savepoints

- Savepoints can help to avoid errors.
- Savepoints live inside a transaction
- You can return to any savepoint inside a transaction

```
test=# \h SAVEPOINT
Command:     SAVEPOINT
Description: define a new savepoint within the
    current transaction
Syntax:
SAVEPOINT savepoint_name
```

ROLLBACK TO SAVEPOINT

```
test=# \h ROLLBACK TO SAVEPOINT
Command:     ROLLBACK TO SAVEPOINT
Description: roll back to a savepoint
Syntax:
ROLLBACK [ WORK | TRANSACTION ] TO
    [ SAVEPOINT ] savepoint_name
```

Transaction time

- Time inside a transaction is “frozen”
- now() will return the same time inside a transaction
- This is important to know:

```
DELETE FROM data WHERE field < now();
```

- If now() was not constant, the result would be more or less random

Locking and concurrency

Locking

- In case of concurrent transactions locking is essential
- Otherwise the behavior of the database would be unpredictable
- PostgreSQL tries to minimize locking as much as possible
- When possible no locks or fine grained row locks are used

Rules

- A transaction is only seen by others once it has been committed.
- A transaction can see its own changes.
- Modified rows are locked and can only be changed by one person at a time.
- Data is persisted by COMMIT
- ROLLBACK is virtually free
- SELECTs and data changes can coexist

Rules

- If two transactions try to modify the same row, one has to wait:

```
UPDATE data SET id = id + 1
```

- If 1000 people do this concurrently, the ID will be incremented by EXACTLY 1000.

Safety (1)

- Is it safe?

```
BEGIN;
SELECT   seat
FROM     t_flight
WHERE    flight = 'AB4711'
    AND  booked = false;

....
UPDATE   t_flight
    SET  booked = true
    WHERE    seat = ...;
```

Safety (2)

- If two people do the same thing at the same time they would overwrite each other
- Concurrency is the hidden danger here
- No locking is in place to protect users
- SELECT FOR UPDATE comes to the rescue

Safety (3)

- SELECT FOR UPDATE locks a row as if we would modify it instantly.
- Concurrent writes have to wait until we commit or rollback.
- TIP: Use NOWAIT to make sure a transaction does not wait forever. This is especially important on the web.

Safety (4)

- SELECT FOR UPDATE locks out concurrent writes.
- Sometimes not everybody wants the same row.
- Sometimes you are happy with any rows as long as you are the only one touching it.

```
SELECT ...
FROM     t_flight
WHERE    flight = 'AB4711'
LIMIT    1
FOR UPDATE SKIP LOCKED;
```

Safety (5)

- SKIP LOCKED allows many people to book a flight at the same time.
- Everybody will get his own row.
- SKIP LOCKED has been introduced in PostgreSQL 9.5

FOR UPDATE vs. FOR SHARE

- Sometimes FOR UPDATE is too harsh.
- A simple SELECT might be too weak.
- Consider: While you are looking at a row you want to make sure that it is not removed. However, it is perfectly fine if two people need the row at the same time.

A practical use cases

- Imagine you got two tables: Currency + Account.
- If you modify an account you want to make sure that the currency is not removed while you change the account.
- The foreign key ensures that (internally a FOR SHARE equivalent is used)

Dangerous locking (1)

```
SELECT  ...
FROM    currency AS c, account AS a
WHERE   a.currency_id = c.id
    AND a.id = '4711'
FOR UPDATE;
```

- Which rows in which tables are locked?

Dangerous locking (2)

- In this case only one person at a time can draw money
- The entire currency is locked because of a single person.
- PostgreSQL cannot know what you are planning to change
- Therefore both sides of the join need locking

Dangerous locking (3)

- More fine grained locking:

```
SELECT   ...
FROM     currency AS c, account AS a
WHERE    a.currency_id = c.id
    AND a.id = '4711'
FOR UPDATE OF a;
```

- Only accounts will be changed.
- Currency needs no locking

More options

- SELECT is ways more powerful:

```
[ FOR { UPDATE | NO KEY UPDATE | SHARE | KEY SHARE }
    [ OF table_name [, ...] ] [ NOWAIT | SKIP LOCKED ]
      [...] ]
```

Locking entire tables

- Entire tables can be locked
- PostgreSQL knows 8 types of locks
- The most important ones:
  - ACCESS EXCLUSIVE: No reads, no writes
  - EXCLUSIVE: Reads are ok, writes are blocked
  - SHARE: Used by CREATE INDEX, writes are blocked
  - ACCESS SHARE: Conflicts with DROP TABLE and alike
- Try to avoid table locks when possible

Transaction isolation

Transaction isolation

- In SQL transactions are isolated from each other
- Depending on the level of isolation the way data is seen changes.
- The ANSI SQL standard defined 4 transaction isolation levels:
  - READ UNCOMMITTED + READ COMMITED are the same in PostgreSQL
  - REPEATABLE READ (snapshot isolation, good for reporting)
  - SERIALIZABLE (SSI transactions for even higher isolation)

READ COMMITTED (1)

- READ COMMITTED is the default isolation level
- A transaction can see the effect of committed transactions.

```
BEGIN;
SELECT sum(field) FROM tab;
    returns 43
            <-- concurrent write
SELECT sum(field) FROM tab;
    returns 57
COMMIT;
```

READ COMMITTED (2)

- READ COMMITTED is bad for reporting
- Queries inside the transaction do not operate on the same "snapshot".
- Higher isolation is needed for reporting

REPEATABLE READ (1)

- The entire transaction will use the same snapshot.
- Visibility does not change inside the transaction.

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT ...
SELECT ...
COMMIT;
```

REPEATABLE READ (2)

- There is no performance penalty for REPEATABLE READ
- The only difference is that different rows are visible
- NOTE: This does not apply to SERIALIZABLE (it is significantly slower)

SERIALIZABLE

- Transactions are separated even more.
- Consider the following example:
  - We want to store prison guards
  - Each prison should only have ONE guard at a time
- Of course this can be solved with LOCK TABLE
- However, LOCK TABLE does not scale beyond one CPU core

How SERIALIZABLE works

- PostgreSQL keeps track of data touched using predicate locking
- Locking rows is not enough
- We also have to protect ourselves against future rows
- The main idea is to create the illusion that a user is alone
- However, behind the scenes things are as parallel as possible
- Conflicts can happen and transactions can be aborted

How to use SERIALIZABLE

- FOR UPDATE, LOCK TABLE, etc. are not needed anymore
- Users can write code without worrying about concurrency
- PostgreSQL will resolve conflicts automatically
- "Pivot transactions" are aborted

Additional considerations

Managing lock_timeout

- lock_timeout can be set to abort a statement if a lock is held for too long.
- lock_timeout is in milliseconds.

Deadlocks

- Transactions can fail due to deadlocks
- In case deadlocks happen, PostgreSQL will resolve them automatically.
- The deadlock detection will kick in after deadlock_timeout has been reached.
- Deadlocks can happen regardless of the isolation level.

Tracking locks: pg_locks

- pg_locks can be used to check for pending locks
- Administrators can check, which transaction is waiting on which transaction
- Rows level conflicts can even be observed at the row level (page + tuple)
- In many cases it is easier to check the "waiting" flag in pg_stat_activity

Sequences and transactions (1)

- Sequences cannot be rollbacked

```
test=# CREATE SEQUENCE seq_a;
test=# BEGIN;
test=# SELECT nextval('seq_a');
 nextval
---------
       1
test=# ROLLBACK;
test=# SELECT nextval('seq_a');
 nextval
---------
       2
```

Sequences and transactions (2)

- Never use sequences to assign numbers to business transactions
- Gaps in your invoice numbers are not allowed
- Different means are needed