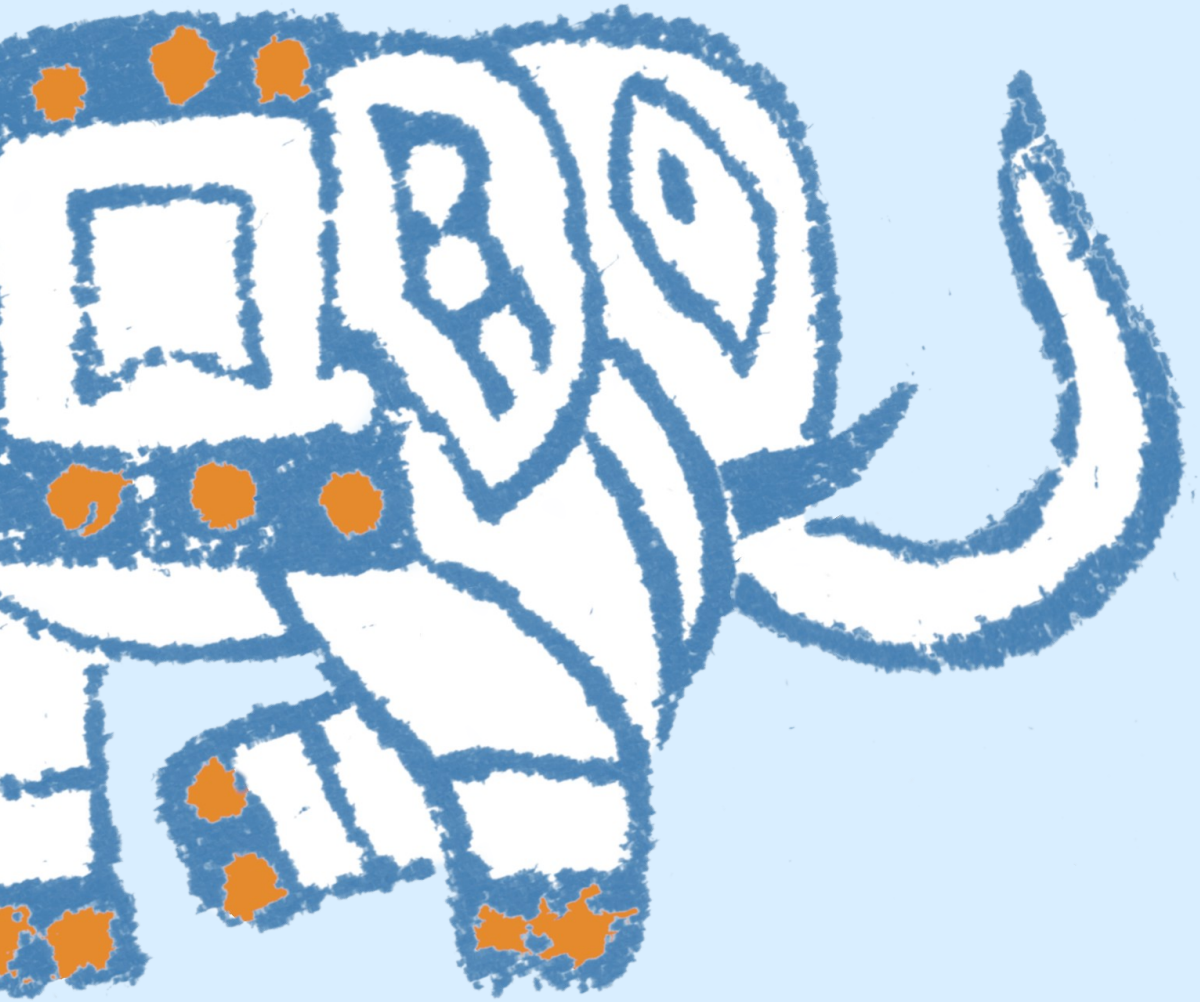

Why can't we continue a transaction after an ERRor?

A journey into subtERRanean world of subtransactions

Egor Rogov, Postgres Professional

In PostgreSQL,
one failed statement aborts the whole transaction.

The transaction could have continued if we had been able to roll the failed statement back…
ACID: A is for Atomicity

```
CREATE TABLE accounts(
  id integer PRIMARY KEY,
  amount numeric NOT NULL CHECK (amount >= 0)
);
INSERT INTO accounts
VALUES (1,1000.00), (2,100.00), (3,0.00);
BEGIN;
UPDATE accounts SET amount = amount - 10.00;
ERROR:  new row for relation "accounts" violates check constraint "accounts_amount_check"
DETAIL:  Failing row contains (3, -10.00).
UPDATE accounts
SET amount = greatest(amount - 10.00, 0.00);
ERROR:  current transaction is aborted, commands ignored until end of transaction block
COMMIT;
ROLLBACK
```

Oracle behaves differently.

Rollback:

- is based on undo log
- transaction-level
- statement-level
- block-level for MVCC

So what's the difference with PostgreSQL?

Recap

ACID: I is for Isolation

4 levels in the standard,
Snapshot Isolation to rul'em all

ACID: C is for Consistency

Multi-Version Concurrency Control

```
BEGIN ISOLATION LEVEL REPEATABLE READ;
SELECT * FROM accounts;
 id | amount
----+---------
  1 | 1000.00
  2 |  100.00
  3 |    0.00
(3 rows)
    UPDATE accounts SET amount = amount + 1.00;
    SELECT * FROM accounts;
     id | amount
    ----+---------
      1 | 1001.00
      2 |  101.00
      3 |    1.00
    (3 rows)
SELECT * FROM accounts;
 id | amount
----+---------
  1 | 1000.00
  2 |  100.00
  3 |    0.00
(3 rows)
COMMIT;
```

MVCC in PostgreSQL

pageinspect

Terminology:

- heap ≈ table
- row
- row version ≈ tuple
- outdated row version = dead tuple

```
CREATE EXTENSION pageinspect;
TRUNCATE accounts;
INSERT INTO accounts VALUES (1,100.00);
SELECT * FROM accounts;
 id | amount
----+--------
  1 | 100.00
SELECT lp
FROM heap_page_items(get_raw_page('accounts',0));
 lp
----
  1
BEGIN;
UPDATE accounts SET amount = amount + 10.00;
SELECT * FROM accounts;
 id | amount
----+--------
  1 | 110.00
SELECT lp
FROM heap_page_items(get_raw_page('accounts',0));
 lp
----
  1
  2
```

xmin & xmax,

x = xact = transaction

Transaction ID (vs SCN in Oracle)

Transaction status: CLOG

Rollback, but not undo

- rollback is fast
- no need for block-level rollback
- bloat and vacuum

```
SELECT lp, t_xmin, t_xmax
FROM heap_page_items(get_raw_page('accounts',0));
 lp | t_xmin | t_xmax
----+--------+--------
  1 |    758 |    759
  2 |    759 |      0
ROLLBACK;
SELECT * FROM accounts;
 id | amount
----+--------
  1 | 100.00
SELECT lp, t_xmin, t_xmax
FROM heap_page_items(get_raw_page('accounts',0));
 lp | t_xmin | t_xmax
----+--------+--------
  1 |    758 |    759
  2 |    759 |      0
SELECT pg_xact_status('758'::xid8);
 pg_xact_status
----------------
 committed
SELECT pg_xact_status('759'::xid8);
 pg_xact_status
----------------
 aborted
```

Rollback = status change,
that's why PostgreSQL is not able
to rollback a statement,
only a transaction as a whole

But what about savepoints?

```
TRUNCATE accounts;
INSERT INTO accounts VALUES (1,100.00);
BEGIN;
UPDATE accounts SET amount = amount + 100.00;
SAVEPOINT sp;
UPDATE accounts SET amount = amount + 1.00;
ROLLBACK TO sp;
COMMIT;
SELECT * FROM accounts;
 id | amount
----+--------
  1 | 200.00
SELECT lp, t_xmin, t_xmax
FROM heap_page_items(get_raw_page('accounts',0));
 lp | t_xmin | t_xmax
----+--------+--------
  1 |    761 |    762
  2 |    762 |    763
  3 |    763 |      0
SELECT pg_xact_status('762'::xid8);
 pg_xact_status
----------------
 committed
SELECT pg_xact_status('763'::xid8);
 pg_xact_status
----------------
 aborted
```

Subtransactions

Main transaction

- subtransaction 1
- ✗ subtransaction 2 (aborted)
- subtransaction 3

Subtransactions

- ✗ Main transaction (aborted)
  - ✗ subtransaction 1
  - ✗ subtransaction 2 (aborted)
  - ✗ subtransaction 3

Subtransactions

- ✓ Main transaction (committed)
  - ✓ subtransaction 1
  - ✗ subtransaction 2 (aborted)
  - ✓ subtransaction 3

...and autonomous transactions

PL/pgSQL exceptions

How do we trap an error?

```
DO $$
BEGIN
  RAISE NOTICE '%', 1/0;
END;
$$;
ERROR:  division by zero
CONTEXT:  SQL expression "1/0"
PL/pgSQL function inline_code_block line 3 at RAISE

DO $$
BEGIN
  RAISE NOTICE '%', 1/0;
EXCEPTION
  WHEN OTHERS THEN
    RAISE NOTICE 'exception';
END;
$$;
NOTICE:  exception
DO
```

PL/pgSQL exceptions

How do we trap an error?
Subtransactions!

Failed block is rolled back to implicit savepoint (unlike in Oracle)

```
DO $$
BEGIN
  INSERT INTO accounts VALUES (2, 100.00);
  RAISE NOTICE '%', 1/0;
EXCEPTION
  WHEN OTHERS THEN
     RAISE NOTICE 'exception';
END;
$$;
NOTICE:  exception
DO
SELECT * FROM accounts;
 id | amount
----+--------
  1 | 200.00
SELECT lp, t_xmin, t_xmax
FROM heap_page_items(get_raw_page('accounts',0));
 lp | t_xmin | t_xmax
----+--------+--------
  1 |    761 |    762
  2 |    762 |    763
  3 |    763 |      0
  4 |    765 |      0
SELECT pg_xact_status('765'::xid8);
 pg_xact_status
----------------
 aborted
```

Why not setting a savepoint
before each statement?

Perfectly possible,
but not a good idea

```
\timing on
DO $$ BEGIN
  FOR i IN 1 .. 10_000_000 LOOP
    BEGIN
    END;
  END LOOP;
END; $$;
DO
Time: 223.854 ms
...
    BEGIN
    EXCEPTION
      WHEN others THEN NULL;
    END;
...
DO
Time: 3711.709 ms (00:03.712)
...
    BEGIN
      RAISE NOTICE '%', 1/0;
    EXCEPTION
      WHEN others THEN NULL;
    END;
...
DO
Time: 38429.202 ms (00:38.429)
```

Long and sad tale of SQL/JSON

- 2016 — introduced in SQL:2016
- 2017 — first patch
- 2019 — JSON/Path get into PG12
- 2022 — other stuff almost get into PG15, but was reverted

Why?

Long and sad tale of SQL/JSON

- 2016 — introduced in SQL:2016
- 2017 — first patch
- 2019 — JSON/Path get into PG12
- 2022 — other stuff almost get into PG15, but was reverted

Why? Subtransactions!

- 2024 — finally get into PG 17

```
SELECT JSON_QUERY('{"foo": [1,2]}'::jsonb, '$.foo'
RETURNING int[] ERROR ON ERROR);
 json_query
------------
 {1,2}
(1 row)

SELECT JSON_QUERY('{"foo": "bar"}'::jsonb, '$.foo'
RETURNING int[] ERROR ON ERROR);
ERROR:  expected JSON array

SELECT JSON_QUERY('{"foo": "bar"}'::jsonb, '$.foo'
RETURNING int[] NULL ON ERROR);
 json_query
------------

(1 row)
```

It’s okay to use exception when you need them

But quite often you can do without exceptions

```
CREATE OR REPLACE PROCEDURE upd_or_ins(
  acc_id integer, acc_amount numeric
) AS $$
BEGIN
  INSERT INTO accounts VALUES (acc_id, acc_amount);
EXCEPTION
  WHEN unique_violation THEN
     -- PERFORM pg_sleep(10); -- anything can happen
     UPDATE accounts SET amount = amount + acc_amount
WHERE id = acc_id;
END;
$$ LANGUAGE plpgsql;

CALL update_or_insert(2, 50.00);

    DELETE FROM accounts WHERE id = 2;

SELECT * FROM accounts;
 id | amount
----+--------
  1 | 200.00
(1 row)

INSERT INTO accounts VALUES (2, 50.00)
ON CONFLICT(id) DO UPDATE
SET amount = accounts.amount + excluded.amount;
```

When overhead is not a concern: ON_ERROR_ROLLBACK

```
\set ON_ERROR_ROLLBACK on
BEGIN;
UPDATE accounts SET amount = amount - 1000.00;
ERROR:  new row for relation "accounts" violates check constraint "accounts_amount_check"
DETAIL:  Failing row contains (1, -890.00).
UPDATE accounts
SET amount = greatest(amount - 1000.00, 0.00);
UPDATE 1
COMMIT;
COMMIT
```

Takeaways

Snapshot Isolation and MVCC

Rollback ≠ undo

Subtransactions to implement:

- rollback to savepoint
- exception handling
- on-error-rollback mode

Mind the overhead!

Thanks!

Any questions?

edu@postgrespro.ru