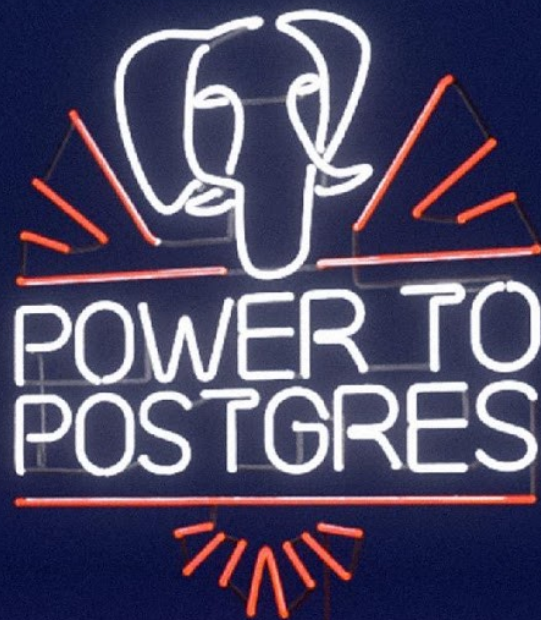# Don't Do This

Jimmy Angelakos
Senior Solutions Architect

PGDay Chicago 2023-04-20

EDB

What is this talk?

- Not all-inclusive

- There is literally nothing you cannot mess up

- Misconceptions

- Confusing things

- Common but impactful mistakes

We'll be looking at

- Bad SQL

- Improper data types

- Improper feature usage

- Performance considerations

- Security considerations

Bad SQL

NOT IN   (i)

Doesn't work the way you expect!

- As in: **SELECT … WHERE … NOT IN (SELECT …)**

- SQL is not Python or Ruby!

  - **SELECT** a **FROM** tab1 **WHERE** a **NOT IN** (1, null); returns NO rows!

  - **SELECT** a **FROM** tab1 **WHERE** a **NOT IN** (**SELECT** b **FROM** tab2);
    same, if any b is **NULL**

- Why is this bad even if no **NULL**s?

  - Query planning / optimization

  - Subplan instead of anti-join

NOT IN

What to do instead?

- Anti-join

- SELECT col
  FROM tab1
  WHERE NOT EXISTS
      (SELECT col
       FROM tab2
       WHERE tab1.col = tab2.col);

NOT IN

Or:

- SELECT col
  FROM tab1
  LEFT JOIN tab2 USING (col)
  WHERE tab2.col IS NULL;

- NOT IN is OK, if you know there are no NULLs

  – e.g. excluding constants: NOT IN (1,3,5,7,11)

BETWEEN

(i)

Especially with TIMESTAMPs

- BETWEEN (1 AND 100) is inclusive (closed interval)

- When is this bad?

```
SELECT sum(amount)
FROM transactions
WHERE transaction_timestamp
BETWEEN ('2023-02-05 00:00' AND '2023-02-06 00:00');
```

BETWEEN

(ii)

Be explicit instead, and use:

```
SELECT sum(amount)
FROM transactions
WHERE transaction_timestamp >= '2023-02-05 00:00'
AND transaction_timestamp < '2023-02-06 00:00';
```

Using upper case in identifiers

For table or column names

- Postgres makes everything lower case unless you double quote it

- CREATE TABLE Plerp (…);
  CREATE TABLE "Quux" (…);

  – Creates a table named plerp and one named Quux

  – TABLE Plerp; works – TABLE "Plerp"; fails

  – TABLE Quux; fails – TABLE "Quux"; works

  – Same with column names

- For pretty column names: SELECT col FROM plerp AS "Pretty Name";

Improper data types

TIMESTAMP (WITHOUT TIME ZONE)

a.k.a. naïve timestamps

- Stores a date and time with no time zone information

    - Arithmetic between timestamps entered at different time zones is meaningless and gives wrong results

- **TIMESTAMPTZ (TIMESTAMP WITH TIME ZONE)** stores a moment in time

    - Arithmetic works correctly

    - Displays in your time zone, but can display it **AT TIME ZONE**

- Don't use **TIMESTAMP** to store UTC because the DB doesn't know it's UTC

TIMETZ

Or TIME WITH TIME ZONE has questionable usefulness

- Only there for SQL compliance

  - Time zones in the real world have little meaning without dates

  - Offset can vary with Daylight Savings

  - Not possible to do arithmetic across DST boundaries

- Use TIMESTAMPTZ instead

CURRENT_TIME

Is TIMETZ. Instead use:

- CURRENT_TIMESTAMP or now() for a TIMESTAMPTZ

- LOCALTIMESTAMP for a TIMESTAMP

- CURRENT_DATE for a DATE

- LOCALTIME for a TIME

CHAR(n) / VARCHAR(n)

Padded with whitespace up to length n

- Padding spaces are ignored when comparing

    - But not for pattern matching with LIKE & regular expressions!

- Actually not stored as fixed-width field!

    - Can waste space storing irrelevant spaces

    - Performance-wise, spend extra time stripping spaces

    - Index created for CHAR(n) may not work with a TEXT parameter

- company_name VARCHAR(50) → Peterson's and Sons and Friends Bits & Parts Limited

- To restrict length, just enforce CHECK constraint

- Bottom line: just use TEXT (VARCHAR)

MONEY

Get away

- Fixed-point

  - Doesn't handle fractions of a cent, etc. – rounding may be off!

- Doesn't store currency type, assumes server LC_MONETARY

- Accepts garbage input:

  ```
  # SELECT ',123,456,,7,8.1,0,9'::MONEY;
      money
  ----------------
   £12,345,678.11
  (1 row)
  ```

- Just use NUMERIC and store currency in another column

SERIAL

Used to be useful shorthand but now more trouble than it's worth

- Non SQL Standard

- Permissions for sequence created by SERIAL need to be managed separately from the table

- CREATE TABLE ... LIKE will use the same sequence!

- Use identity columns instead:

  CREATE TABLE tab (id BIGINT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY, content TEXT);

- With an identity column, you don't need to know the name of the sequence:
  ALTER TABLE tab ALTER COLUMN id RESTART WITH 1000;

- BUT: if application depends on a serial sequence with no gaps (e.g. for receipt numbers), generate that in the application

Improper feature usage

SQL_ASCII
Is not a database encoding

- No encoding conversion or validation!

    - Byte values **0-127** interpreted as ASCII
    - Byte values **128-255** uninterpreted

- Setting behaves differently from other character sets

- Can end up storing a mixture of encodings

    - And no way to recover original strings

CREATE RULE

RULEs are not the same as TRIGGERs

- Rules don't simply apply conditional logic

    - They rewrite queries to modify or add extra queries
    - All non-trivial rules will probably have unintended side-effects
    - Non SQL Standard

- If you are not creating writable VIEWs, use TRIGGERs instead

- Look for Depesz's exhaustive blog post on rules:

    https://www.depesz.com/2010/06/15/to-rule-or-not-to-rule-that-is-the-question

CREATE TABLE (...) INHERITS ...   (i)

Table inheritance

- Seemed like a good idea before ORMs...

- e.g. **CREATE TABLE** events (id **BIGINT**, ... many columns ...);
        CREATE TABLE meetings (scheduled_time **TIMESTAMPTZ**)
        INHERITS events;

- Was used to implement partitioning (< PG 10)

- Incompatible with declarative partitioning (>= PG 10):

  - One cannot inherit from a partitioned table

  - One cannot add inheritance to a partitioned table

CREATE TABLE (…) INHERITS … (ii)

How to undo table inheritance

- You can replace table inheritance with foreign key relations

- Create a new table to hold the data, and add the FK column:

  ```
  CREATE TABLE new_meetings LIKE meetings;
  ALTER TABLE new_meetings ADD item_id BIGINT;
  ```

- Copy data from old table into new one (may take a long time):

  ```
  INSERT INTO new_meetings
  SELECT *, id FROM meetings;
  ```

- Create required constraints, indexes, triggers etc. for new_meetings

CREATE TABLE (...) INHERITS ... (iii)

How to undo table inheritance (continued)

- **Very dirty hack** (if your table is huge) - create the FK but do not validate it now to avoid the full table scan:

    ```
    ALTER TABLE new_meetings
    CONSTRAINT event_id_fk
    FOREIGN KEY (event_id)
    REFERENCES events (id)
    NOT VALID;
    ```

- If doing this on a live system, create a trigger to replicate changes coming into **meetings** also into **new_meetings**

- Normally one should not touch pg_catalog directly, but we can
  UPDATE pg_constraint **SET** convalidated = true **WHERE** conname = 'event_id_fk';
  as we are confident that data in FK column is valid (as exact copy of the original table)

CREATE TABLE (...) INHERITS ... (iv)

How to undo table inheritance (continued)

- Inside a transaction, perform all the DDL at once:

```
DO $$
BEGIN
    ALTER TABLE meetings RENAME TO old_meetings;
    ALTER TABLE new_meetings RENAME TO meetings;
    DROP TABLE old_meetings;
    -- IMPORTANT: Create trigger to INSERT/UPDATE/DELETE items in
    -- events as they get changed in meetings - it's easy as now
    -- we have the FK.
    COMMIT;
END $$ LANGUAGE plpgsql;
```

Partitioning by multiple keys   (i)

Is not partitioning on multiple levels

- Be careful!

- CREATE TABLE transactions (… , location_code **TEXT**, tstamp **TIMESTAMPTZ**)
  PARTITION BY RANGE (tstamp, location_code);

- CREATE TABLE transactions_2023_02_a
  PARTITION OF transactions
  FOR VALUES FROM ('2023-02-01', 'AAA') TO ('2023-03-01', 'BAA');

- CREATE TABLE transactions_2023_02_b
  PARTITION OF transactions
  FOR VALUES FROM ('2023-02-01', 'BAA') TO ('2023-03-01', 'BZZ');

  ERROR: partition "transactions_2023_02_b" would overlap partition
  "transactions_2023_02_a"

Partitioning by multiple keys   (ii)

Subpartitioning is what you actually need

- CREATE TABLE transactions (… , location_code **TEXT**, tstamp **TIMESTAMPTZ**)
  PARTITION BY RANGE (tstamp);

- CREATE TABLE transactions_2023_02
  PARTITION OF transactions
  FOR VALUES FROM ('2023-02-01') **TO** ('2023-03-01')
  PARTITION BY HASH (location_code);

- CREATE TABLE transactions_2023_02_p1
  PARTITION OF transactions_2023_02
  FOR VALUES WITH (**MODULUS** 4, **REMAINDER** 0);

Performance considerations

Number of connections (i)

Don't overload your server for no reason

- max_connections = 5000

- Every client connection spawns a separate backend process

    - IPC via semaphores & shared memory

    - Risk: CPU context switching

- Accessing the same objects from multiple connections may incur many Lightweight Locks (LWLocks or "latches")

    - Lock becomes heavily contended, lots of lockers slow each other down

    - You may be making your data hotter for no reason

    - No queuing, more or less random

Number of connections (ii)

- Pre-PG 13: **Snapshot contention**

    - Each transaction has an MVCC snapshot – even if idle!

- Contention often caused by too much concurrency

    - Insert a connection pooler (e.g. **PgBouncer**) between application and DB
    - Allow fewer connections into the DB, make the rest queue for their turn
    - "Throttle" or introduce latency on the application side, to save your server performance

        - Sounds counter-intuitive!
        - Doesn't necessarily slow anything down – queries may execute faster!

High transaction rate (i)

Just because you can, doesn't mean you should

- Postgres assigns an identifier to each transaction

    - Unsigned 32-bit int (4.2B values)

    - Circular space, with a visibility horizon

- **XID wraparound:** you try to read a very old tuple that is > 2.1B XIDs in the past

- Very heavy OLTP workloads can go through 2.1B transactions in a short time

    - For you, that's the future! (invisible)

    - **Freezing:** Flag tuple as "frozen" which is known to always be in the past

- Need to make sure FREEZE happens before XID wraparound

High transaction rate (ii)

What can you do?

- Can batching help?

    - Does application really need to commit everything atomically?

    - Batch size 1000 will have 1/1000th the burn rate

- Increase effectiveness of autovacuum

    - More efficient FREEZE

Turning off autovacuum (i)

a.k.a. the MVCC maintenance operation. Yeah, don't.

- Removes dead tuples, freezes tuples (among other things)

- Has overhead

  - Scans tables & indexes
  - Needs, obtains, and waits for locks
  - Has limited capacity by default

- People are concerned about overhead

  - Alternative is worse! You can't avoid VACUUM in Postgres (yet).
  - You can outrun it (and then you'll need VACUUM FULL)

Turning off autovacuum (ii)

For most production workloads, defaults are too low

- Make it work harder to avoid problems

- Increase potency via:

    - maintenance_work_mem (1GB is good)
    - autovacuum_max_workers
    - autovacuum_vacuum_cost_delay / autovacuum_vacuum_cost_limit

Explicit locking

(i)

a.k.a. heavyweight locks

- Table-level (e.g. SHARE) or row-level (e.g. FOR UPDATE)

- Conflict with other lock modes (e.g. ACCESS EXCLUSIVE with ROW EXCLUSIVE)

- Block read/write access totally leading to waits

- Disastrous for performance

  – Unless your application is exquisitely crafted

  – Hint: it isn't

Explicit locking (ii)

Lock contention: waiting for explicit locks

- Avoid explicit locking!

- Use SSI (Serializable Snapshot Isolation, SERIALIZABLE isolation level)

- Make application tolerant

    - Allow it to fail and retry

- Slightly reduced concurrency, but:

    - No blocking, no explicit locks needed (SIReadLocks, rw-conflicts)

    - Best performance choice for some application types

Security considerations

psql --W or --password
Request password before attempting connection

- It will ask for a password even if the server doesn't require one

- Unnecessary: **psql** will always ask for a password if required by server

- Insecure: You may think you're logging in with a password
  - But the server may be in **trust** mode and letting you in anyhow
  - Also, you may be entering the wrong password and still getting in
  - From a different client, you may get a surprise!

listen_addresses = "*"

Listening for connections from clients

- There's a **reason** the default is 'localhost' (only TCP/IP loopback)

- Make sure you only enable the interfaces and networks which you actually want to have access to the database server

- e.g. Internet connection on one network & private network on another interface

- Don't advertise your presence: **3,600,000** MySQL/MariaDB servers (port 3306) found exposed on the Internet in May 20

Come In WE'RE OPEN

pg_hba.conf → trust

Host-Based Authentication

- Called that for a reason, i.e. configuring with host ... like:

  host mydb myuser 10.10.10.10/32 md5

- trust with host(ssl) is a **Very Bad Idea**

  - Even for local e.g. improper user can connect to the DB

  - Postgres might be fine, but other software on the same server could be compromised

- Default to giving access only where strictly necessary (better safe...)

Database owned by superuser

Do you really need to?

- Use superuser only for management of global objects

  - Such as users

  - Good security practice

- Superuser bypasses a lot of checks

- (Bad) code that's normally harmless could be exploited in harmful way with superuser access

- Try to restrict database ownership to standard users

Thank you!

Find me on Mastodon: @vyruss@fosstodon.org

Photo: "The Devil's Beef Tub", Scotland

Thank you!

PG Day Chicago 2023 Sponsors

PLATINUM SPONSOR

DRW

GOLD SPONSORS

pgEdge · datAvail · Microsoft · aws · Quest Where Next Meets Now.

Silver Sponsors

DBeaver · PGX POSTGRESQL EXPERTS, INC. · FUJITSU · instaclustr Now part of Spot by NetApp

Community Partner

Chicago PostgreSQL User Group