Don’t do that!

Laurenz Albe

Laurenz Albe

Talking Head

Email

laurenz.albe@cybertec.at

Phone

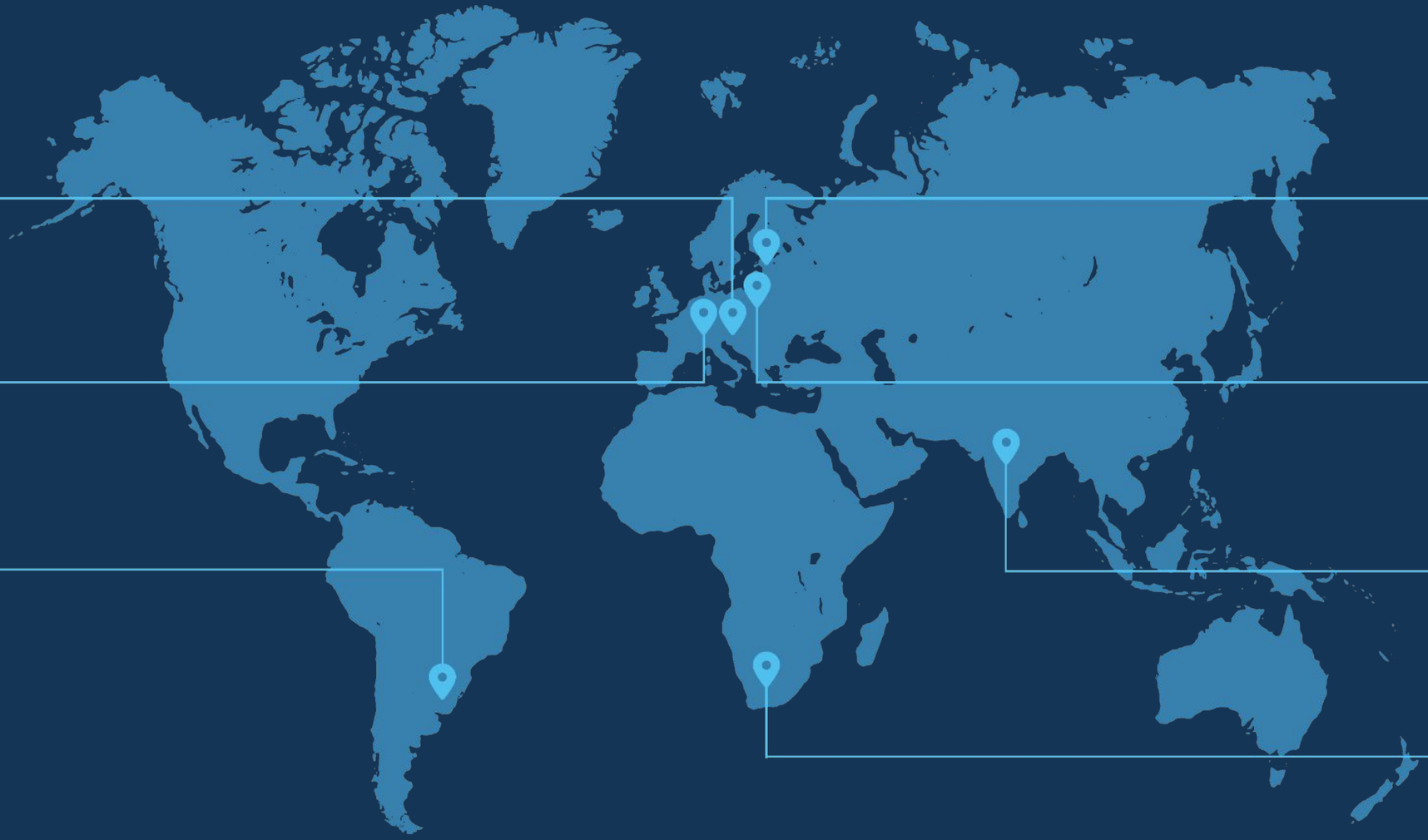
+43 670 605 6265

www.cybertec-postgresql.com

@cybertec-postgresql

www.youtube.com/@cybertecpostgresql

PROUD CONTRIBUTOR TO PostgreSQL
the world's most advanced open source database



AUSTRIA (HQ)

CYBERTEC POSTGRESQL
INTERNATIONAL (HQ)

SWITZERLAND

CYBERTEC POSTGRESQL
SWITZERLAND

URUGUAY

CYBERTEC POSTGRESQL
SOUTH AMERICA

ESTONIA

CYBERTEC POSTGRESQL
NORDIC

POLAND

CYBERTEC POSTGRESQL
POLAND

INDIA

CYBERTEC POSTGRESQL
INDIA PRIVATE LIMITED

SOUTH AFRICA

CYBERTEC POSTGRESQL
SOUTH AFRICA

Database Products & Tools

CYBERTEC MIGRATOR

CYPEX

CYBERTEC PGEE

scalefield

PG WATCH

PG SQUEEZE

scalefield secure

CYBERTEC LakeHouse

YAIM

WAL BOUNCER

DATA MASKING

pg_ TIMETABLE

PATRONI ENVIRONMENT SETUP

POSTGRESQL CONFIGURATOR

PG SHOW PLANS

PL/pgSQL_sec CYBERTEC

ISO 27001 CERTIFIED

Introduction

- people keep asking for “best practice”
- I have come to dislike that, because it often means “I don’t want to understand that and I don’t want to think, just tell me what to do”
- it’s much easier to name things that you should avoid
- hence this collection of “worst practices” from my experience as a consultant

Storing timestamps as strings or numbers

Storing timestamps as strings

- it’s a bad idea to store anything as string that isn’t a string
- you’ll end up with dates like `2024-02-30`, `12.4.2024`, `0000-00-00`

  Yes, you could check the values with a check constraint, but using the correct data type checks it automatically.
- `2025-01-23 12:30:00+01` takes 23 bytes as string, but 8 bytes as `timestamp with time zone`
- `'2025-01-23 11:30:00+01' > '2025-01-23 03:30:00-08'` as string

Storing timestamps as offset from the epoch

- don’t store timestamps in seconds since 1070-01-01 00:00:00 UTC
- it’s fundamentally correct, but
  - 1737631800 is harder to read than `'2025-01-23 12:30:00+01'`.
  - date arithmetic becomes more difficult, as you cannot use the timestamp functions and operators directly (and complicated expressions in SQL statements tend to lead to bad performance)
- there are exceptions to this rule, for example if all you ever need to calculate is the difference in seconds
  - but are you sure that the data will never be used for anything else?

About other data types

The same holds for other data types: always use the appropriate database type

- it’s a bad idea to store anything as string that isn’t a string
- store “valid from - valid to” as `tstzrange`
- use PostGIS for geographical coordinates
- use `bytea` for binary data, no encoded string
- use `integer` or `bigint` for integers, not `numeric`
- use `bit varying` for bitmaps
- use `jsonb` for JSON and `xml` for XML

Using 4-byte `integer` for auto-generated primary keys

The problem with auto-generated `integer` keys

- the maximum integer is 2^31 - 1 = 2147483647
- don’t make the same mistake as the people who thought that 2^32 IP addresses would be all mankind could ever need!
- sequence values get “lost” on rollback
- your table might grow bigger than you thought
  (or you might delete and insert a lot)
- The canonical solution for the problem:

  ```
  ALTER TABLE tab ALTER id TYPE bigint;
  ```

  will rewrite the table ⇒ causes a long down time

Good practice: use `bigint`

- play it safe and always use **`bigint`** for auto-generated primary keys
- if the table grows large, you may need it
- if the table is small, wasting four bytes won't matter
- exception: small lookup tables that get referenced in big tables
  ⇒ for those, choose **`integer`** or even **`smallint`**
  - example: a table of the US states

Changing from integer to `bigint` without down time (1)

Add a new column and a trigger that fills it:

```
BEGIN;
ALTER TABLE tab ADD id2 bigint;

CREATE FUNCTION copy_id() RETURNS trigger
   LANGUAGE plpgsql AS
$$BEGIN
   NEW.id2 = NEW.id;
   RETURN NEW;
END;$$;

CREATE TRIGGER copy_id BEFORE INSERT OR UPDATE ON tab
    FOR EACH ROW EXECUTE FUNCTION copy_id();
COMMIT;
```

Changing from integer to `bigint` without down time (2)

Update the existing rows in batches:

```
UPDATE tab SET id2 = id
WHERE id2 IS NULL
  AND id < 1000000;

VACUUM tab;

UPDATE tab SET id2 = id
WHERE id2 IS NULL
  AND id BETWEEN 1000001 AND 2000000;

VACUUM tab;
...
```

Changing from integer to `bigint` without down time (3)

Create a `NOT NULL` constraint and a `UNIQUE` index:

```
ALTER TABLE tab ADD CONSTRAINT tab_id2_notnull CHECK (id2 IS NOT NULL) NOT VALID;

ALTER TABLE tab VALIDATE CONSTRAINT tab_id2_notnull;

ALTER TABLE tab ALTER id2 SET NOT NULL;

ALTER TABLE tab DROP CONSTRAINT tab_id2_notnull;

CREATE UNIQUE INDEX CONCURRENTLY tab_pkey2 ON tab (id2);
```

Changing from integer to `bigint` without down time (4)

Drop the old column and rename the new one, drop the trigger and create a new primary key constraint:

```
BEGIN;

-- works only if not referenced by a foreign key
ALTER TABLE tab DROP id;

DROP TRIGGER copy_id ON tab;

DROP FUNCTION copy_id();

ALTER TABLE tab RENAME id2 TO id;

ALTER TABLE tab ADD PRIMARY KEY USING INDEX tab_pkey2;

COMMIT;
```

Define a `comment` column as `varchar(255)`

Define a `comment` column as `varchar(255)`

- someone will want to insert 300 characters
- the `ALTER TABLE` is cheap, but unnecessary
- if the application does not enforce a length limit, define the column as `text`
- `text` and `varchar` have the same implementation
- no performance penalty for `text`
  on the contrary – you avoid the length check

Define all columns nullable

Define all columns nullable

- easy to do, because nullable is the default in SQL (more's the pity!)
- experience tells: most nullable columns will eventually hold a NULL
  ⇒ bad for data quality
- NULL makes queries complicated (harder for the optimizer)

  ```
  WHERE col <> 42 OR col IS NULL -- or better:
  WHERE col IS DISTINCT FROM 42
  ```

  a `JOIN b ON acol IS NOT DISTINCT FROM bcol` -- cannot be indexed
- it is easy to change from `NOT NULL` to nullable, but not the other way around
  (see the sample code from before!)
- in case of doubt, initially define columns as `NOT NULL`

Use large objects

What are large objects?

- special, non-standard API: `lo_create`, `lo_open`, `lowrite`, `loread`, `lo_close`, `lo_unlink`, . . .
- data stored in the catalog table `pg_largeobject`
- each large object has an oid, which you can store in a table to refer to it
- the documentation says:

  PostgreSQL also supports a storage system called “TOAST” [. . .] This makes the large object facility partially obsolete.

Problems with large objects

- no referential integrity between the large object and the table row that uses it
  - needs a trigger or regular `vacuumlo` run to maintain integrity
- before PostgreSQL v17, large objects get dumped and restored in a single transaction during `pg_upgrade`
  - if you have many large objects, upgrade may become impossible
    ⇒ v17 improved that, but the upgrade is still slow

Recommendations for large objects

- there is really no benefit in using large objects, with these exceptions:
  - you need objects exceeding 1GB (but you don’t)
  - you need to stream writes to the object
- don’t touch large objects, except with a long pole – use `bytea` instead
- large objects are unfortunately still used a lot, because the PostgreSQL JDBC driver has the standard methods `getBLOB()` and `setBLOB()` operate on large objects
- if you define a column as `@Lob` or `@Lob String` in Hibernate, you’ll end up with large objects

Use an ENUM type for lists that can change

Using ENUM types

- Created with

```
CREATE TYPE state AS ENUM ('Ohio', 'California', 'Alabama', ...);
```

- you can add a new state:

```
ALTER TYPE state ADD VALUE 'Ontario' AFTER 'Ohio';
```

  (but before v17, the new value cannot be used in the same transaction)

- you can rename a state:

```
ALTER TYPE state RENAME VALUE 'California' TO 'Hot Oven';
```

ENUM: Problems and recommendation

- you cannot delete a state:

```
ALTER TYPE state DROP VALUE 'Alabama';
ERROR:  dropping an enum value is not implemented
```

- when in doubt, use a lookup table

```
CREATE TABLE state (
  id smallint PRIMARY KEY,
  name text UNIQUE NOT NULL
);
```

- use ENUM types only for lists that can never lose entries

Define a check constraint that can become `FALSE`

The problem with check constraints that become `FALSE` over time

- a bad example:

```
ALTER TABLE tab
ADD CHECK (col > current_timestamp);
```

- but the condition that is initially `TRUE` becomes `FALSE` over time (is not “retroactively deterministic” in SQL standard terms)
- once the condition has become `FALSE`, any later update of the row will lead to an error – even if a different column is updated
- make sure your check constraints are retroactively deterministic (and remember that word to impress others!)

Bad check constraints that reference other tables

- we want to make sure that a certain name exists in another table
- writing a subquery into a check constraint will fail, but we can cheat with a function:

```
CREATE FUNCTION f(text) RETURNS boolean
RETURN EXISTS (SELECT FROM other WHERE upper(name) = upper($1));

ALTER TABLE tab
ADD CHECK (f(name));
```

- this won't check the constraint if a row is deleted from `other`...
- if you dump and restore the database, the data for `tab` could be restored first, which would lead to an error ⇒ cannot restore the backup

Lie about a function’s immutability

Lying about a function's immutability

- `CREATE FUNCTION name( ... ) RETURNS ...`
  `{ IMMUTABLE | STABLE | VOLATILE }`
  `LANGUAGE ... AS ...`
- `IMMUTABLE` promises that a function always will always return the same result for the same arguments
- PostgreSQL performs some sanity checks, but in general, it believes your claim
- lying about `IMMUTABLE` can result in:
  - corrupted indexes
  - rows ending up in the wrong table partition
  - incorrect values in generated columns
  - in general, bad query results and data corruption

Example: corrupted index caused by bad `IMMUTABLE` function

```
-- depends on "timezone", not really IMMUTABLE
CREATE FUNCTION get_hour(timestamp with time zone) RETURNS integer
   IMMUTABLE RETURN CAST (extract(hour FROM $1) AS integer);

CREATE TABLE ts (t timestamp with time zone);
CREATE UNIQUE INDEX ON ts (get_hour(t));

SET timezone = 'UTC';

INSERT INTO ts VALUES ('2024-11-19 22:00:00 Europe/Vienna');
INSERT 0 1

SET timezone = 'Asia/Kolkata';

INSERT INTO ts VALUES ('2024-11-19 22:00:00 Europe/Vienna');
INSERT 0 1
```

Use an Entity-Attribute-Value design

Motivation for an Entity-Attribute-Value design

If you want to create entities on the fly, you might find the following design attractive:

```
CREATE TABLE objects (
   objectid bigint PRIMARY KEY);

CREATE TABLE attstring (
   objectid bigint REFERENCES objects ON DELETE CASCADE NOT NULL,
   attname text NOT NULL,
   attval text,
   PRIMARY KEY (objectid, attname));

CREATE TABLE attint (
   objectid bigint REFERENCES objects ON DELETE CASCADE NOT NULL,
   attname text NOT NULL,
   attval integer,
   PRIMARY KEY (objectid, attname));
```

Queries and DML with an Entity-Attribute-Value design

- fetching an object with `N` attributes has to fetch `N+1 rows`
- inserting one object with `N` attributes leads to `N+1 INSERTS`
- deleting one object with `N` attributes leads to `N+1 DELETEs`
  ⇒ these operations will be much slower
- updating one attribute will be a single `UPDATE`
  - that might actually be a bit faster
  - but update of several columns will become several `UPDATEs`
- on top of all that, the 24 bytes header for each table row waste considerable storage space

A “simple” join with an Entity-Attribute-Value design

```
SELECT e1a1.attval AS person_name,
       e1a2.attval AS person_id,
       e2a1.attval AS address_street,
       e2a2.attval AS address_city
FROM attint AS e1a2
   JOIN attstring AS e1a1
      ON e1a2.objectid = e1a1.objectid
   LEFT JOIN attint AS e2a0
      ON e1a2.attval = e2a0.attval
   LEFT JOIN attstring AS e2a1
      ON e2a0.objectid = e2a1.objectid
   LEFT JOIN attstring AS e2a2
      ON e2a0.objectid = e2a2.objectid
WHERE e1a1.attname = 'name'
  AND e1a2.attname = 'persnr'
  AND e2a0.attname = 'persnr'
  AND e2a1.attname = 'street'
  AND e2a2.attname = 'city';
```

No comment!

Recommendations for alternatives

- don’t use an Entity-Attribute-Value design
- just don’t
- it is actually much better to have the application run `CREATE TABLE`
- if you want to avoid creating objects, use `jsonb`
  - common attributes (`objectid`) are normal table columns
  - user-defined attributes become JSON attributes

Questions?

Link for feedback

Affiliations & Recognitions

ISO 27001 CERTIFIED

PG4B

PROUD CONTRIBUTOR TO PostgreSQL
the world's most advanced open source database

OSB Open Source Business ALLIANCE
Bundesverband für digitale Souveränität e.V.

Open Alliance
For PostgreSQL Education

Our partners at PGConf.EU

readyset

redgate

OpenSource^DB
Your Trusted Data Partner

Microsoft

PROVENTA

Open Alliance

For PostgreSQL Education

Your Pathway to Verified PostgreSQL Skills

Scan for Updates

oapg-edu.org

PGDay Austria returns in 2026

Scan for updates

pgday.at