MACH DAS NICHT!

BY LAURENZ ALBE

TALKING HEAD

CYBERTEC
POSTGRESQL SERVICES & SUPPORT

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


AUSTRIA (HQ)
CYBERTEC POSTGRESQL
INTERNATIONAL (HQ)
ESTONIA
CYBERTEC POSTGRESQL
NORDIC
SWITZERLAND
CYBERTEC POSTGRESQL
SWITZERLAND
POLAND
CYBERTEC POSTGRESQL
POLAND
URUGUAY
CYBERTEC POSTGRESQL
SOUTH AMERICA
INDIA
CYBERTEC POSTGRESQL
INDIA PRIVATE LIMITED
SOUTH AFRICA
CYBERTEC POSTGRESQL
SOUTH AFRICA

EINLEITUNG

- ich werde oft nach „Best Practices” gefragt
- ich mag diese Frage nicht, denn sie bedeutet oft: „Ich will das nicht verstehen und ich will auch nicht nachdenken; sag mir einfach, was ich tun soll!”
- es ist oft einfacher, Fehler aufzuzeigen, als einen Weg zu zeigen, der immer richtig ist
- daher diese Sammlung an „Worst Practices” aus meiner Erfahrung als Berater

TIMESTAMPS ALS STRINGS ODER ZAHLEN SPEICHERN

TIMESTAMPS (DATUM + ZEIT) ALS STRING (ZEICHENKETTE) SPEICHERN

- es ist eine schlechte Idee, etwas als String zu speichern, das in Wirklichkeit kein String ist
- das führt zu Daten wie `2024-02-30, 12.4.2024, 0000-00-00`

  Ja, man kann einen Check Constraint definieren, aber wenn man den richtigen Datentyp nimmt, wird der Wert automatisch überprüft.
- `2025-01-23 12:30:00+01` belegt als String 23 Bytes, aber nur 8 Bytes als `timestamp with time zone`
- `'2025-01-23 11:30:00+01' > '2025-01-23 03:30:00-08'` als String

TIMESTAMPS ALS SEKUNDEN SEIT DER „EPOCHE” SPEICHERN

- Timestamps nicht in Sekunden seit 1970-01-01 00:00:00 UTC speichern
- das ist zwar grundsätzlich korrekt, aber
  - 1737631800 ist schwerer zu lesen als `'2025-01-23 12:30:00+01'`
  - Datumsarithmetik wird komplizierter, weil man die Operatoren und Funktionen für Timestamps nicht direkt verwenden kann (und komplizierte Ausdrücke in SQL-Statements führen oft zu schlechter Performance)
- es gibt Ausnahmen zu dieser Regel, z.B. wenn man nie etwas anderes als die Differenz in Sekunden berechnen will
  - aber wer kann sicher sein, dass die Daten nie anders verwendet werden?

ÜBER ANDERE DATEITYPEN

Dasselbe gilt für andere Arten von Daten: immer den richtigen Datentyp in der Datenbank verwenden

- „gültig von - bis” als **`tstzrange`** speichern
- PostGIS für geographische Koordinaten verwenden
- **`bytea`** für Binärdaten verwenden, keine Verschlüsselung als String
- für ganze Zahlen **`integer`** oder **`bigint`** verwenden, nicht **`numeric`**
- **`jsonb`** für JSON und **`xml`** für XML verwenden

4 BYTE `integer` ALS AUTOMATISCH GENERIERTER PRIMÄRSCHLÜSSEL

DAS PROBLEM AUTOMATISCH GENERIERTER `integer`-SCHLÜSSEL

- der grösste **`integer`** ist 2^31 - 1 = 2147483647
- mach nicht denselben Fehler wie die Leute, die gedacht haben, dass die Welt nie mehr als 2^32 IP-Adressen brauchen wird!
- Sequence-generierte Werte gehen bei **`ROLLBACK`** „verloren”
- die Tabelle könnte grösser als gedacht werden (oder es wird viel eingefügt und wieder gelöscht)
- die kanonische Lösung des Problems

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

  schreibt die Tabelle neu ⇒ führt zu langem Ausfall

WIE MAN ES RICHTIG MACHT: `bigint` VERWENDEN

- bleib auf der sicheren Seite und verwende **`bigint`** für automatisch generierte Schlüssel
- wenn die Tabelle gross wird, ist es vielleicht notwendig
- wenn die Tabelle klein bleibt, schmerzen vier verschwendete Bytes nicht
- Ausnahme: kleine Tabellen, die von grossen Tabellen referenziert werden
  ⇒ hier ist **`integer`** oder **`smallint`** besser
  - Beispiel: Tabelle der Schweizer Kantone

OHNE SERVICEAUSFALL VON `integer` AUF `bigint` UMSTELLEN (1)

Neue Spalte und einen Trigger, der sie befüllt, hinzufügen:

```
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;
```

OHNE SERVICEAUSFALL VON `INTEGER` AUF `BIGINT` UMSTELLEN (2)

Die bestehenden Zeilen in Tranchen aktualisieren:

```
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;
...
```

OHNE SERVICEAUSFALL VON `INTEGER` AUF `BIGINT` UMSTELLEN (3)

`NOT NULL` und **`UNIQUE`**-Constraint anlegen:

```
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);
```

OHNE SERVICEAUSFALL VON `INTEGER` AUF `BIGINT` UMSTELLEN (4)

Alte Spalte löschen, neue umbenennen, Trigger weg, neuer Primärschlüssel:

```
BEGIN;

-- wirft Fehler, wenn von anderen Tabellen referenziert
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;
```

SPALTE kommentar ALS varchar(255) ANLEGEN

SPALTE `kommentar` ALS `VARCHAR(255)` ANLEGEN

- irgendjemand wird 300 Zeichen einfügen wollen
- das **`ALTER TABLE`** ist billig, aber unnötig
- wenn die Applikation keine Längenbeschränkung erfordert, ist der Datentyp **`text`** der beste
- **`text`** und **`varchar`** haben dieselbe Implementation
- keine Performancekosten für **`text`**
  im Gegenteil – man spart sich den Längentest

NULL-WERTE IN ALLEN SPALTEN ZULASSEN

NULL-WERTE IN ALLEN SPALTEN ZULASSEN

- passiert leicht, weil das (leider) in SQL die Standardeinstellung ist
- die Erfahrung lehrt: irgendwie kommt immer ein NULL-Wert in die Daten
  ⇒ schlecht für die Datenqualität
- NULL macht Abfragen komplizierter (schwerer für den Optimierer)

```
WHERE col <> 42 OR col IS NULL -- oder besser
WHERE col IS DISTINCT FROM 42

a JOIN b ON acol IS NOT DISTINCT FROM bcol -- nicht indexierbar
```

- es ist leicht, `NOT NULL` zu deaktivieren, aber das Gegenteil ist schwierig (siehe Code-Beispiel oben)
- im Zweifelsfall Spalten anfangs als `NOT NULL` definieren

„LARGE OBJECTS” VERWENDEN

WAS SIND LARGE OBJECTS?

- besondere, nicht standardkonforme Schnittstelle: **`lo_create`, `lo_open`, `lowrite`, `loread`, `lo_close`, `lo_unlink`**, . . .
- Daten stehen im Katalog **`pg_largeobject`**
- jedes Large Object hat eine **`oid`**, die man als Referenz in einer Tabelle speichert
- die Dokumentation sagt:
  PostgreSQL also supports a storage system called “TOAST” [. . .]
  This makes the large object facility partially obsolete.

PROBLEME MIT LARGE OBJECTS

- keine referenzielle Integrität zwischen dem Large Object und der Tabellenzeile, die es referenziert
  - braucht Trigger oder regelmässiges **`vacuumlo`** für Integrität
- vor PostgreSQL v17 werden bei **`pg_upgrade`** alle Large Objects in einer einzigen Transaktion gelesen und geschrieben
  - bei vielen Large Objects kann das ein Upgrade unmöglich machen
    ⇒ v17 hat das verbessert, aber es ist immer noch langsam

EMPFEHLUNGEN FÜR LARGE OBJECTS

- kein Vorteil bei Large Objects, mit den folgenden Ausnahmen:
  - man braucht Objekte über 1GB (braucht man aber nicht)
  - man möchte das Schreiben in Stücken „streamen”
- Finger weg von Large Objects – lieber **`bytea`** verwenden
- Large Objects werden leider noch oft verwendet, weil der JDBC-Treiber von PostgreSQL sie für die Standardmethoden **`getBLOB()`** und **`setBLOB()`** verwendet
- wenn man in Hibernate Spalten als **`@Lob`** oder **`@Lob String`** deklariert, bekommt man Large Objects

ENUM-DATENTYPEN FÜR VERÄNDERLICHE LISTEN VERWENDEN

ÜBER ENUM-DATENTYPEN

- Erzeugen mit

  ```
  CREATE TYPE kanton AS ENUM ('Freiburg', 'Zug', 'Ticino', ...);
  ```

- man kann einen neuen Kanton hinzufügen:

  ```
  ALTER TYPE kanton ADD VALUE 'Vorarlberg';
  ```

  (aber vor v17 kann man den neuen Wert nicht in derselben Transaktion verwenden)

- man kann einen Kanton umbenennen:

  ```
  ALTER TYPE kanton RENAME VALUE 'Freiburg' TO 'Fribourg';
  ```

ENUM: PROBLEME UND KORREKTE VERWENDUNG

- man kann keinen Kanton entfernen:

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

- im Zweifelsfall eine Referenztabelle verwenden:

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

- ENUM-Typen nur für Listen verwenden, bei denen *nie* ein Wert entfernt wird

CHECK-CONSTRAINT DEFINIEREN, DER SPÄTER FALSE WIRD

DAS PROBLEM BEI CHECK-CONSTRAINTS, DIE SPÄTER `FALSE` WERDEN

- ein warnendes Beispiel:

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

- die ursprünglich wahre Bedingung wird im Lauf der Zeit falsch (ist nicht „retroaktiv deterministisch” im Sinne des SQL-Standards)
- sobald die Bedingung **`FALSE`** wird, scheitert jede Änderung der Zeile – sogar wenn eine ganz andere Spalte geändert wird
- Check-Constraints immer retroaktiv deterministisch definieren (den Begriff kann man sich merken, um Eindruck zu schinden)

CHECK-CONSTRAINTS, DIE ANDERE TABELLEN REFERENZIEREN

- wollen sicherstellen, dass ein gewisser Name in einer anderen Tabelle steht
- eine Unterabfrage im Check-Constraint geht nicht, aber wir können mit einer Funktion schummeln:

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

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

- jedoch überprüft das die Bedingung nach Löschen einer Zeile in **`other`** nicht. . .
- bei Dump/Restore könnte **`tab`** zuerst eingespielt werden, was zu einem Fehler führt
  ⇒ kann Backup nicht einspielen

ÜBER DIE STABILITÄT VON FUNKTIONEN LÜGEN

ÜBER DIE STABILITÄT VON FUNKTIONEN LÜGEN

- `CREATE FUNCTION name( ... ) RETURNS ...`
  `{ IMMUTABLE | STABLE | VOLATILE }`
  `LANGUAGE ... AS ...`
- “**`IMMUTABLE`**” ist das Versprechen, dass eine Funktion für die selben Argumente immer dasselbe Ergebnis liefern wird
- PostgreSQL überprüft manche Aspekte (z.B. wird ein **`UPDATE`** einen Fehler auslösen), aber im Wesentlichen glaubt es die Behauptung
- über **`IMMUTABLE`** lügen kann folgendes bewirken:
  - korrupte Indexe
  - Zeilen, die in der falschen Partition landen
  - falsche Werte in **`GENERATED`** Spalten
  - allgemein gesagt, falsche Abfrageergebnisse und Datenkorruption

BEISPIEL: KORRUPTER INDEX DURCH SCHLECHTE IMMUTABLE FUNKTION

```
-- hängt von "timezone" ab, nicht wirklich 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
```

EIN ENTITY-ATTRIBUTE-VALUE DESIGN VERWENDEN

MOTIVATION FÜR EIN ENTITY-ATTRIBUTE-VALUE DESIGN

Wenn man Klassen von Objekten dynamisch erzeugen möchte, könnte dieses Design reizvoll sein:

```
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));
```

ABFRAGEN UND DML MIT EINEM ENTITY-ATTRIBUTE-VALUE DESIGN

- ein Objekt mit object `N` Attributen lesen muss `N+1` Zeilen lesen
- ein Objekt mit `N` Attributen einfügen erfordert `N+1 INSERT`s
- ein Objekt mit `N` Attributen löschen erfordert `N+1 DELETE`s
  ⇒ diese Operationen werden viel langsamer sein
- ein Attribut ändern erfordert ein einzelnes `UPDATE`
  - das kann tatsächlich etwas schneller sein
  - aber mehrere Spalten ändern bedeutet mehrere `UPDATE`s
- weiters bedeuten die 24 Bytes Header pro Tabellenzeile eine beträchtliche Speicherplatzverschwendung

EIN „EINFACHER JOIN” MIT EINEM 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';
```

⇒ Kein Kommentar!

EMPFEHLUNGEN FÜR ALTERNATIVEN

- einfach kein Entity-Attribute-Value Design verwenden
- echt nicht
- tatsächlich ist es viel besser, die Applikation **`CREATE TABLE`** ausführen zu lassen
- wenn man keine Objekte erzeugen möchte, kann man **`jsonb`** verwenden
  - allgemeine Attribute (**`objectid`**) werden normale Tabellenspalten
  - benutzerdefinierte Attribute werden JSON-Attribute

FRAGEN?

LAURENZ ALBE

SENIOR CONSULTANT

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