Das 1×1 der Testdatengenerierung mit Postgres

Kaarel Moppel, Freelance PostgreSQL Consultant
pgDay.ch 2025, Rapperswil

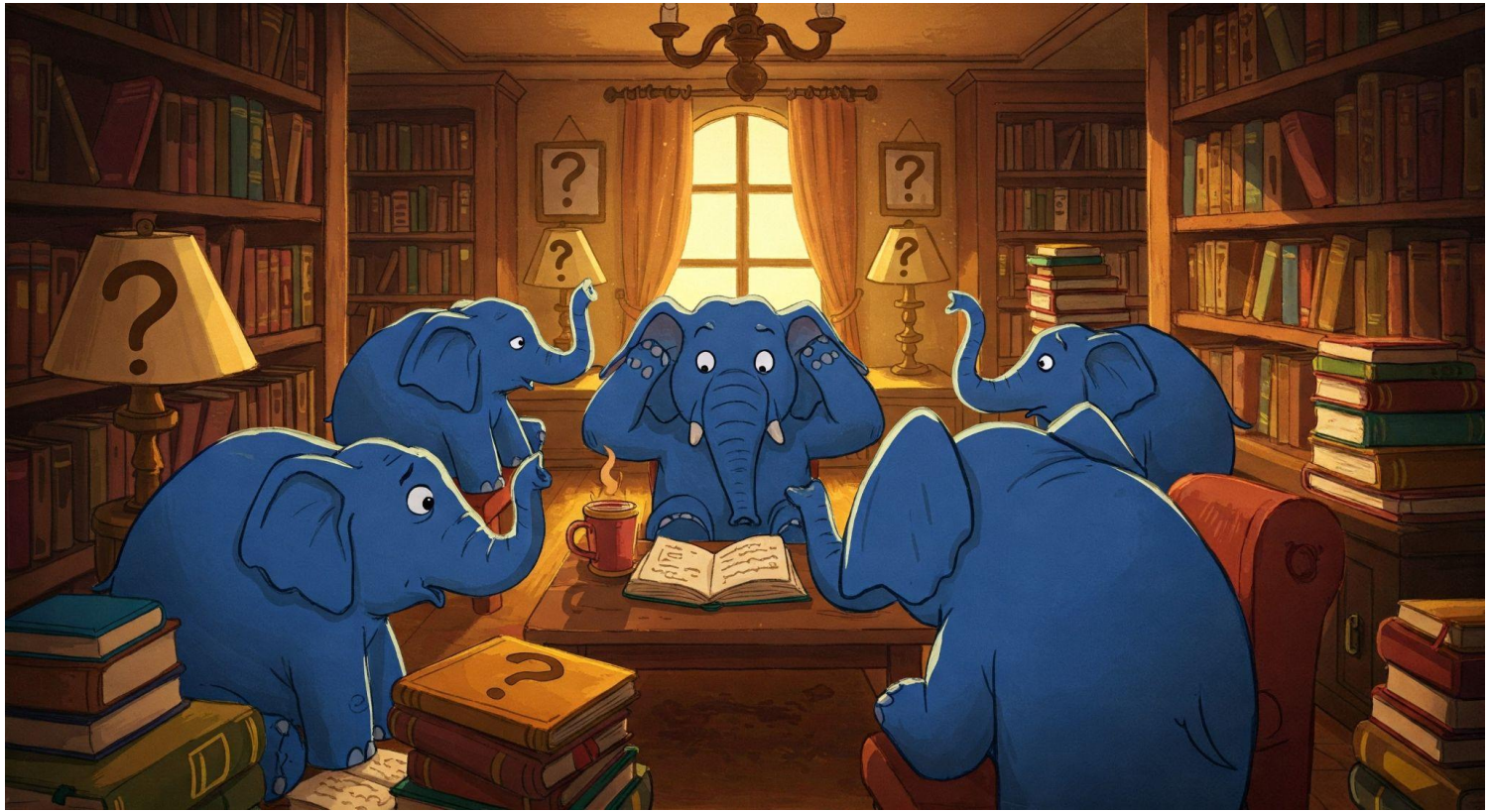
$ whoami

- "Kämpfe" mit Datenbanken seit 2007
- 20K+ Stunden im Postgres-Ökosystem
  - Viele verschiedene Rollen im Laufe der Zeit
  - Wenn überhaupt etwas - habe ein gewisses Bauchgefühl für alle Postgres-nahe Dinge entwickelt
- Als Freelancer unterstütze Leute bei allen möglichen Themen rund um Postgres
- https://kmoppel.github.io/
  - Mein Postgres Blog & Kontakt

Agenda

- Das „Warum?“
- Gängige Techniken
- Fortgeschrittene Techniken
- Werkzeuge
- Beschleunigungstricks
- Stolperfallen

Das „Warum?“



Warum sich überhaupt die Mühe machen?

Niemand stellt diese Frage beim App-Code, oder?

Als Berater sehe ich immer wieder, dass das anfängliche DB-Layout (oder der Single-Node-Ansatz generell) völlig ungeeignet für das absehbare* Datenwachstum / Anfragevolumen war...

Das führt in ein paar Jahren zu:

- Sprunghafter oder sehr schlechter Abfrageperformance
- Schweißtreibende Ad-hoc DB-Wartungsaktionen / Migrationen
- Ungeplante Arbeit / Ausfälle und panische Meetings
- Massiven Hardware “Overprovisioning” um ruhig schlafen zu können

Vorteile der Testdatengenerierung / Benchmarking

- Ein Testframework / Benchmark zwingt einen dazu, über das Design nachzudenken
- Reduziert FUD bzgl. der DB-Interna
- Offenbart offensichtliche Performance- und Parallelitätsprobleme
- Validiert wie viel TPS (Transaktionen pro Sekunde) pro Dollar ungefähr erzielt werden
- Wie Hardware / Cloud mit extremer Belastung umgeht
    - Besonders wichtig bei Managed Postgres!

Vorteile der Testdatengenerierung / Perf-Tests

Besonders wichtig wenn es um “Big Data” geht:

- Wie viel Speicherplatz/$$ würden Backups und Snapshots beanspruchen?
- Wie lange ein Dump oder PITR dauert?
- Wie viel WAL erzeugen wir?
  - Kann der LR / Archivierung mithalten?
- Migrationendauer abschätzen
  - Der Mythos - „Ach, dann migrieren wir einfach“

PS Nur für den Fall – „Dumps“ sind keine echte Backup-Strategie :)

🐘 Der Elefant im Raum

Lücken im DB-Wissen und mangelndes Problembewusstsein!

- Wird oft als „nicht mein Zuständigkeitsbereich“ betrachtet
  - Einfach ignoriert oder Tests enden an der DB-Grenze
  - Die Cloud kümmert sich um alles? Oder?
- App-Frameworks oder CI/CD-Systeme stehen oft im Weg
  - Performance-Testing Aufgaben sind meistens langlaufend ...
- Wissen lässt sich nicht natürlich schnell aufbauen
  - Selbst nicht mit KI :)
  - Aber es gibt schon einige einfache Techniken, die mit wenig* Aufwand die Pareto-Wirkung erzielen ...

Ein Fenster im SDLC-Prozess finden

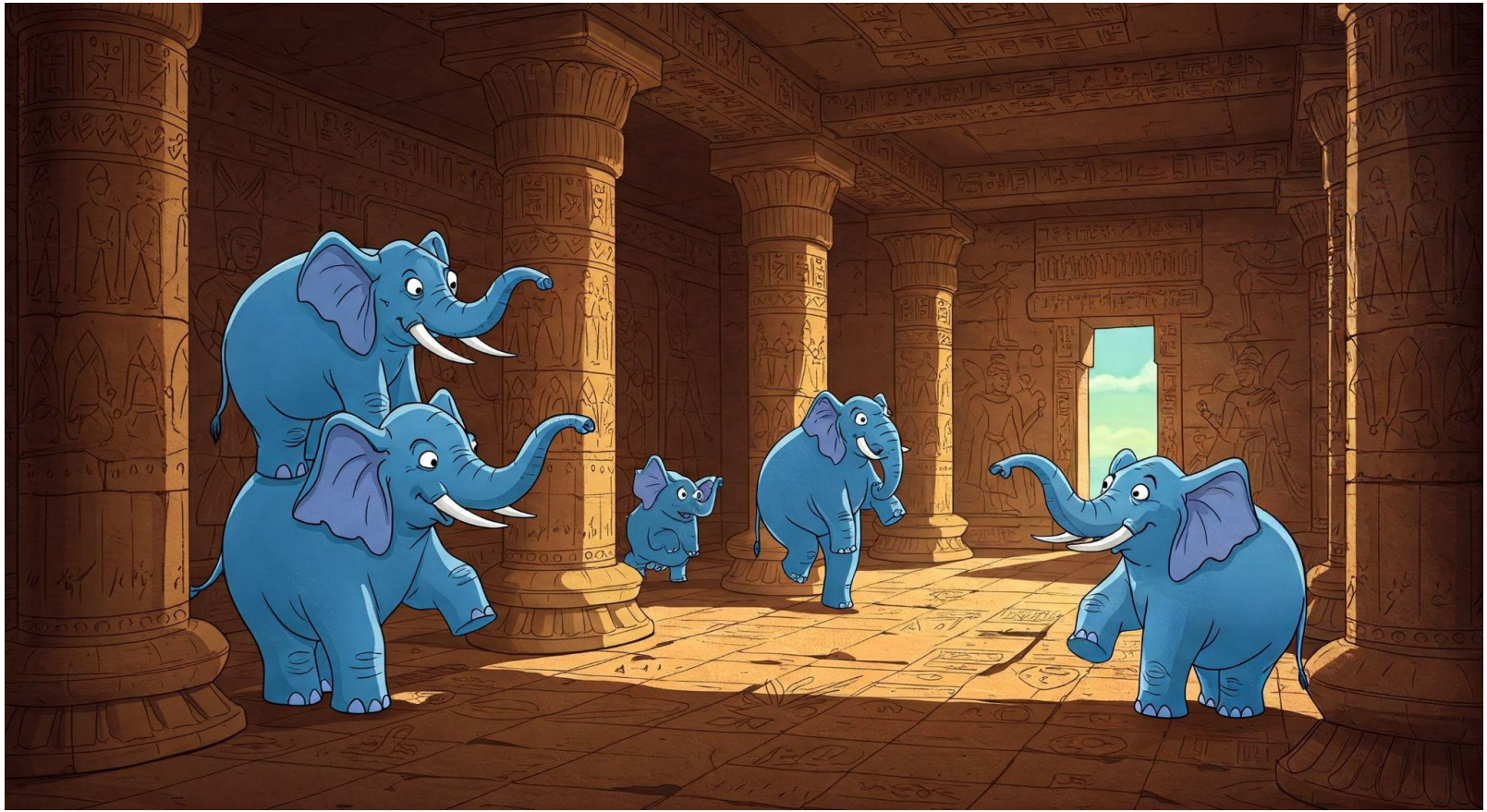
Das Einbinden von „Datenbank-Dingen“ ist oft ein Problem!

Datenbanken sind oft weder „Fleisch“ noch „Fisch“... Auch schlimm - viele Startups, sogar Scaleups, wollen keine Postgres DBAs mehr (oder slicht finden keine?)...

Mein Vorschlag:

- Ad-hoc-Validierung durch “irgendjemanden” mit etwas DB Know-how
  - Einfache SQL- oder Python-Skripte die Nummern liefern sind unendlich besser als nichts!
  - DB-Engines sind meist stabil / robust – das erste Vorab-Testing ist das wichtigste!
- Bei Bedarf ein bisschen weiteren Automatisierung aufsetzen
  - z.B. ein kleines Framework mit rot-grün Feedback für Nicht-Wizards
  - Sodass wenn Risiken erkannt werden man “günstig” etwas tiefer reinschauen kann

Gängige Techniken



Techniken - generate_series()

generate_series() in der “Hosentasche” ist Pflicht!

- Generatorfunktion
  - ähnlich zu “*range*” in Python
- Unterstützt Zahlen, Daten, Zeitstempel
- Unterstützt Step / Stride / Schritt

```
select generate_series(1, 10, 5);
 generate_series
-----------------
               1
               6
(2 rows)
```

Techniken - generate_series()

select d::date, i from
  generate_series(current_date-6, current_date, '1d'::interval) with ordinality x(d, i);

```
postgres=# select d::date, i from
  generate_series(current_date-6, current_date, '1d'::interval) with ordinality x(d, i);
     d      | i
------------+---
 2025-06-20 | 1
 2025-06-21 | 2
 2025-06-22 | 3
 2025-06-23 | 4
 2025-06-24 | 5
 2025-06-25 | 6
 2025-06-26 | 7
(7 rows)
```

Techniken - generate_series()

Q: Wie groß wird unsere Event-Tabelle in 3 Jahren bei 50 Inserts/Sekunde?

```
CREATE UNLOGGED TABLE measurement (
    id int8 GENERATED ALWAYS AS IDENTITY,
    created_on timestamptz,
    value1 float,
    value2 float
);

INSERT INTO measurement (created_on, value1, value2)
SELECT gs,   0,   0
FROM
    generate_series(CURRENT_DATE - '3 years'::interval, now(), '20ms') gs;
...
```

Techniken - Zufallsdaten erzeugen

```
SELECT random(); -- Wert zw. 0.0 und < 1.0

SELECT string_agg(
    substr('ABCDEFGHJKLMNPQRSTUVWXYZ23456789',
    (random() * 31 + 1)::int, 1), ''
) FROM generate_series(1, 8);

SELECT random_normal(100, 10) FROM generate_series(1, 10); -- v16+

-- Aus der "tablefunc" Erweiterung
SELECT * FROM normal_rand(1000, 5, 3); -- Mittelwert 5, Standardabweichung 3

SELECT setseed(0.666);  -- Wiederholbare “Zufallsdaten”
```

PS Für ernsthaftere Zufallsdaten: “pgcrypto” oder Teile aus *gen_random_uuid()*

Techniken - CASE WHEN random() Verkettung

Ein Klassiker, um Zufälligkeit zu erhöhen bzw. etwas Jitter hinzuzufügen, indem man einige random()-s verkettet

```
SELECT
   CASE WHEN random() < 0.5 THEN
      true
   ELSE
      false
   END AS x;

SELECT
   CASE WHEN random() < 0.02 THEN
      random() * 100
   WHEN random() < 0.1 THEN
      random() * 10
   ELSE
      random()
   END AS x;
```

Techniken - PL/pgSQL für komplexere Logik

Idealerweise sollte man im bei reine SQLs / SQL-Funktionen bleiben*, aber wenn die Logik schon zu unlesbar wird, ist PL/pgSQL immer noch eine gute Wahl.

```
SELECT (array_shuffle(string_to_array('abcd', NULL)))[1];
 vs
SELECT random_choice(array['a', 'b', 'c', 'd']);

CREATE OR REPLACE FUNCTION random_choice (items anyarray)
    RETURNS anyelement
    LANGUAGE plpgsql  AS $$
DECLARE
    len int; idx int;
BEGIN
    len := array_length(items, 1);
    idx := 1 + floor(random() * len)::int;
    RETURN items[idx];
END; $$;
```

Techniken - TABLESAMPLE

Um größere gemischte / randomisierte Datenmengen schneller selektieren, unterstützt PostgreSQL die SQL:2003-Standardklausel TABLESAMPLE:

```
-- Nimm ~30% von Daten
SELECT * FROM pgbech_accounts TABLESAMPLE SYSTEM (30) ;
```

SYSTEM - ganze Blöcke zufällig
BERNOULLI - Zeilen zufällig
SYSTEM_ROWS - genaue Zeilenanzahl über zufällige Blöcke
(Erweiterung “*tsm_system_rows*”)

Tehnicken - LATERAL

Lateral ermöglicht „Generatoren“ auf JOIN-Ebene - d.h., für jede Eingabezeile (die “Linke” Tabelle) möchten wir *dynamisch*, anhand bestimmter Kriterien, Faktenzeilen auswählen. Eine **unverzichtbare** Technik für Daten-/Datenbankingenieure!

```
SELECT a.* FROM pgbench_branches b
    JOIN LATERAL (SELECT bid, aid, abalance FROM pgbench_accounts
        WHERE bid = b.bid ORDER BY abalance DESC LIMIT 2) a ON TRUE;
```

bid	aid	abalance
1	76562	26593
1	3634	22217
2	198007	21171
2	126249	20959
3	288385	26151
3	202054	24357

(6 rows)

Tehnicken - LATERAL

PS Auch variable Zeilenanzahl pro Gruppe möglich! Praktisch für ML-Trainingsdaten z. B.

```
-- Vorausgesetzt, es ist eine „treibende“ Tabellenspalte verfügbar

SELECT  a.* FROM  pgbench_branches b
    JOIN LATERAL (
        SELECT  * FROM  pgbench_accounts
        WHERE  bid = b.bid LIMIT b.rowlimit /* Or directly: (random()*6)::int  */
     ) a ON TRUE;
```

Fortgeschrittene Techniken

FT - pgbench

Pgbench ist ein leichtgewichtiges und Einfach zu benutzendes Benchmarking-Tool, das mit Postgres kommt.

- Basiert auf einem vereinfachten OLTP Banking-Schema (TCP-B ähnlich).
- Kann leicht parallelisiert und geskriptet werden.

pgbench --initialize --scale=1 # 1 scale unit = 100k bank accounts ~13MB of main table data
pgbench -n --select-only --client=2 --time=10 # Do key reads for 10s from 2 sessions

```
krl@bench=# \dt+
                                    List of relations
 Schema |       Name       | Type  | Owner | Persistence | Access method |  Size   | Description
--------+------------------+-------+-------+-------------+---------------+---------+-------------
 public | pgbench_accounts | table | krl   | permanent   | heap          | 13 MB   |
 public | pgbench_branches | table | krl   | permanent   | heap          | 40 kB   |
 public | pgbench_history  | table | krl   | permanent   | heap          | 0 bytes |
 public | pgbench_tellers  | table | krl   | permanent   | heap          | 40 kB   |
(4 rows)

krl@bench=# \d pgbench_accounts
            Table "public.pgbench_accounts"
  Column  |     Type      | Collation | Nullable | Default
----------+---------------+-----------+----------+---------
 aid      | integer       |           | not null |
 bid      | integer       |           |          |
 abalance | integer       |           |          |
 filler   | character(84) |           |          |
Indexes:
    "pgbench_accounts_pkey" PRIMARY KEY, btree (aid)
```

PS - Um die magischen *“Scale”-Einheiten* zu verstehen, gibt’s einen JSFiddle Skript. Mehr zur Herleitung der Formel finden Sie hier

FT - pgbench-Skripte

Die Standardschemata ist außerhalb von Stresstests, oder um ein ungefähres Gefühl für die Latenzen zu bekommen, selten nützlich.

Aber - man kann **eigene Schemas** und pgbench-Skripte nutzen!

- Um Verteilungen ähnlich zu deine Workloads definieren
- Werte aus DB oder Shell zu ziehen
- Bedingungen wie (if/else) einzubauen
- ...

$ pgbench --show-script select-only

```
-- select-only: <builtin: select only>
\set aid random(1, 100000 * :scale)
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
```

FT - Benutzerdefinierte pgbench-Skripte

```
SELECT project_id, table_id, 1 as pgbench_helper
FROM public.table_metadata ORDER BY random() LIMIT 1 \aset

SELECT tz - (random()*1000)::int * '1ms'::interval as tz, 1 as
pgbench_helper FROM (select unnest(histogram_bounds::text::timestamptz[])
tz from pg_stats where attname = 'last_changed_time' and schemaname =
'public' and tablename = 'datatable') x ORDER BY random() LIMIT 1 \gset

\set shard_id random(0, 9)

select row_key, data, last_changed_time
from datatable
where shard_id = :shard_id
and project_id = :project_id and table_id = :table_id
and last_changed_time > ':tz'::timestamptz
order by row_key limit 1001;
```

FT - Benutzerdefinierte pgbench-Skripte

Ein Test den ich zur Auswahl einer Partitionierungsstrategie durchgeführt habe

Einrichten des Schemas / Importieren von Datenverteilungen aus der Produktion
...

Interne Postgres-Statistikzähler zurücksetzen
psql -c “SELECT pg_stat_statements_reset()” -c “SELECT pg_stat_reset()”

The scales are from analyzing prod pg_stat_statements calls data
pgbench -n -f ins_upd.sql**@1** -f sel_1.sql**@30** \
  -f sel_2.sql**@20** -f sel_3.sql**@10** -f sel_4.sql**@5** \
  -f sel_5.sql**@5** -f del_gc.sql**@1** \
  --client=32 --jobs 2 **-T 86400** -P 1800 &> run.log

Analysieren Sie die DB- und App-Metriken ...

FT - Nutzung realer Tabellen-Statistiken

Ermöglicht die einfache Generierung realitätsnahe Verteilungen! Falls die reale Werte keie Geheimnisse beinhalten ...

```
SELECT
  schemaname,
  tablename,
  attname,
  null_frac,
  avg_width,
  n_distinct,
  most_common_freqs,
  correlation,
  most_common_vals::text::text[],  -- vorausgesetzt keine Secrets
  histogram_bounds::text::text[]   -- hat echte Werte drin
FROM pg_stats
WHERE tablename IN ('pgbench_accounts','...');
```

FT - Präzision der Statistiken erhöhen

Vorsicht: Standardmäßig sind Postgres Statistiken sehr reduziert! ANALYZE scannt max. 30k Blöcke (~234 MB) - wenn sich die Daten relativ schnell ändern, oder verzerrt sind - reichen Standardwerte (*default_statistics_target*) nicht aus!

Eine Problemumgehung besteht darin, das „Statistikziel“ vorübergehend zu erhöhen, die Statistiken zu aktualisieren, zu exportieren und ein Rollback durchzuführen*.

```
begin;
set default_statistics_target to 400 ; -- ~1GB scannen
analyze pgbench_accounts ; -- PS wird Autovacuum blokieren!
\copy ... -- pg_stats exportieren
rollback; -- NB! Commit könnte einige Pläne umkrempeln …
```

Ein Beispiel für den Export und Import von Statistiken - hier.

FT - Fremdschlüssel-Constraints umgehen

Für Performance-Tests brauchen wir evtl. nicht das ganze Datenmodell korrekt.

Wenn der Fokus nur auf einigen kritischen Tabellen liegt und nicht auf der Richtigkeit des gesamten “Spinnennetzes”, kann man ein paar Workarounds anwenden:

- Schema mit nur Tabellendefinitionen
  pg_dump --section=pre-data mydb
- Ein benutzerdefinierter Schema-Dump mit nur wenigen Tabellen:
  pg_dump -t customers -T '*bigtable*' mydb
- Ein Postgres Hack um Hintergrund FK-Triggern zu Deaktivieren:
  SET session_replication_role TO replica ; ⚠

Werkzeuge

Werkzeuge - kein Mangel!

Die Testdatenerzeugung ist kein neues Problem – es gibt viele Tools! Sie unterstützen hauptsächlich bei der Generierung von Testdaten, aber auch bei der Anonymisierung und beim Lasttest bzw. beim eigentlichen Benchmarking.

Vorteile synthetischer Daten:

- Kein Risiko für Datenschutz / DSGVO*
- Schnelleres Entwickeln - muss nicht unnötig auf grüne Ampeln warten
- Muss nicht perfekt sein an alle Ecken - doppelte oder Dummy-Daten meist OK
- Mit „KI“ kann man heutzutage schon ziemlich nah an „authentisch“ herankommen
  - Es könnte jedoch zu Zeit-, Latenz- und Kostenproblemen kommen ...

Werkzeuge - Testdatengenerierung

- faker (Python): Mutter vieler Wrapper, CLI, viele Domains
  - Integrierte Domains und zusätzliche Community-Domains

```
pip install Faker
```

Use `faker.Faker()` to create and initialize a faker generator, which can generate data by accessing properties named after the type of data you want.

```
from faker import Faker
fake = Faker()

fake.name()
# 'Lucy Cechtelar'

fake.address()
# '426 Jordy Lodge
#  Cartwrightshire, SC 88120-6700'

fake.text()
# 'Sint velit eveniet. Rerum atque repellat voluptatem quia rerum. Numquam exceptu
#  beatae sint laudantium consequatur. Magni occaecati itaque sint et sit tempore.
```

Werkzeuge - Testdatengenerierung

mimesis - eine Art Faker++ in Python – schneller und mit etwas mehr menschlichem “Touch”

Generating 100k full names ¶

Library	Method name	Iterations	Uniqueness	Runtime (in seconds)
Mimesis	`full_name()`	100 000	98 265 (98.27%)	1.344
Faker	**Faker.name()**	100 000	71 067 (71.07%)	17.375

Generating 1 million full names

Library	Method name	Iterations	Uniqueness	Runtime (in seconds)
Mimesis	`full_name()`	1 000 000	847 645 (84.76%)	13.685
Faker	**Faker.name()**	1 000 000	330 166 (33.02%)	185.945

Werkzeuge - Testdatengenerierung

PostgreSQL Anonymizer - Dalibos postgresql_faker-Funktionalitäten wurden verschoben

- Einige integrierte Seed-Datensätze + Funktionen

```
SELECT anon.dummy_last_name();
 dummy_last_name
-------------------------
 Tillman

SELECT anon.dummy_last_name_locale('fr_FR');
 dummy_last_name_locale
-------------------------
 Granier

SELECT anon.dummy_last_name_locale('pt_BR');
 dummy_last_name_locale
-------------------------
 Barreto
```

Currently 7 locales are available: ar_SA, en_US(default), fr_FR, ja_JP, pt_BR, zh_CN, zh_TW.

Werkzeuge - Testdatengenerierung

pgsynthdata - Generiert Datenbank mit der gleichen Struktur wie das Original, aber mit synthetischen Daten.

- Verwendet “pg_stats”, ermöglicht spezialisierte Generatoren über Kommentare

pgsynthdata mydb generate mydb_gen

pgsynthdata -t "table_a:table_b:table_c" mydb generate mydb_gen --drop

pgsynthdata -u testuser -p 1234 --verbose mydb generate mydb_gen -d -m 2.0 --owner testuser2

pgsynthdata mydb show

pgsynthdata mydb analyze mydb_gen

Werkzeuge - Testdatengenerierung

Synth - ein ziemlich vielversprechender deklarativer Generator (leider “war” ?)

- Kann Schema und erforderliche Generatoren aus Live-DBs ableiten

You can use the `synth import` command to automatically generate Synth schema files from your Postgres, MySQL or MongoDB database:

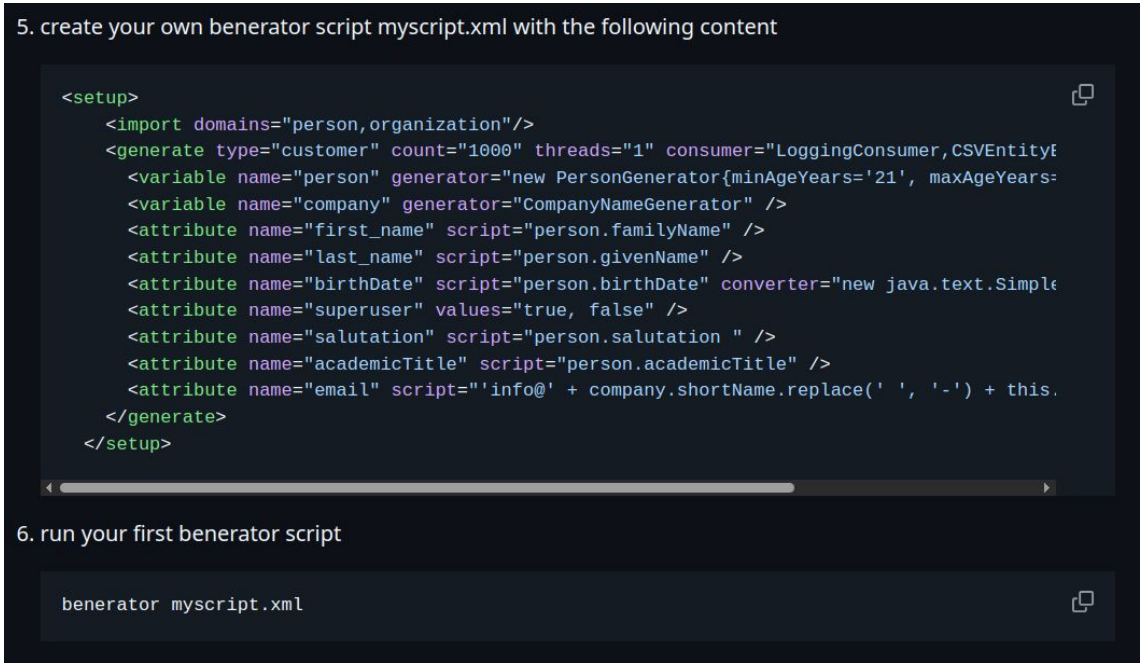
```
$ synth import tpch --from postgres://user:pass@localhost:5432/tpch
Building customer collection...
Building primary keys...
Building foreign keys...
Ingesting data for table customer...  10 rows done.
```

Finally, generate data into another instance of Postgres:

```
$ synth generate tpch --to postgres://user:pass@localhost:5433/tpch
```

Werkzeuge - Testdatengenerierung

Benerator - XML Schema-Beschreibungen zu Dateien / RDBMS umwandeln.
DATAMIMIC - Eine modernere „KI“-gestützte Version von Benerator



Werkzeuge - Testdatengenerierung

Es ist nicht wirklich schwer, für ein begrenztes Business-Domain etwas zusammenhacken!

Ich z.B. verfüge über einen kleinen Satz reiner SQL-Funktionen für Hauptdatentypen - um zufällige oder “range”-bezogene, pseudozufällige Daten zu generieren.

```
INSERT INTO table_x
SELECT
    tdgen.random_bigint() as id,
    tdgen.random_choice('{1,2,3,4}'::int8[]) as project_id,
    tdgen.random_text(10, 20) as description,
    tdgen.random_choice('{1,1,1,0,0,2}'::int8[]) as depth,
    tdgen.random_jsonb(0,2) as metadata,
    tdgen.random_epoch_micros('2021-11-01') / 1000 as last_updated_time,
    tdgen.random_bigint() % 1e6 as dataset_id
FROM
    generate_series(1, 1e6);
```

Werkzeuge - AD 2025

Werkzeuge - KI-gestützt

Here's a SQL script to generate 1,000,000 test rows in `pgbench_accounts` with varied `abalance` and `filler` values:

sql Copy Edit

```sql
-- Ensure at least some branches exist
INSERT INTO public.pgbench_branches (bid, bbalance, filler)
SELECT i, (random() * 100000)::int, md5(i::text)::char(88)
FROM generate_series(1, 100) i;

-- Insert 1 million rows into pgbench_accounts with varied abalance and filler
INSERT INTO public.pgbench_accounts (aid, bid, abalance, filler)
SELECT
    i,
    (random() * 100)::int + 1,  -- Assign to a random branch (1-100)
    (random() * 20000 - 10000)::int,  -- Random balance between -10,000 and 10,000
    md5(i::text || random()::text)::char(84)  -- Random filler value
FROM generate_series(1, 1000000) i;
```

Zumindest ein guter Anfang!

Verwendeter ChatGPT prompt link

Werkzeuge - KI-gestützt (lokal)

```
In [5]: from ollama import chat
   ...: from ollama import ChatResponse
   ...:
   ...: response: ChatResponse = chat(model='llama3.2', messages=[
   ...:   {
   ...:     'role': 'user',
   ...:     'content': '''Please generate an SQL INSERT with dummy data for
   ...:     the following PostgreSQL table structure:
   ...:     CREATE TABLE t_user(id int8, first_name text, last_name text,
   ...:     date_of_birth date, email text);'''
   ...:   },
   ...: ])
   ...: print(response['message']['content'])
   ...:
Here's an example SQL insert statement using PostgreSQL:

```sql
INSERT INTO t_user (id, first_name, last_name, date_of_birth, email)
VALUES
(1, 'John', 'Doe', '1990-05-12', 'johndoe@example.com'),
(2, 'Jane', 'Smith', '1985-03-20', 'janesmith@example.com'),
(3, 'Michael', 'Brown', '1970-01-15', 'michaelbrown@example.com');
```
```

Einfacher als mit Ollama geht es eigentlich nicht

Werkzeuge - KI-gestützt (lokal)

```python
from sdv.datasets.demo import download_demo

real_data, metadata = download_demo(
    modality='single_table',
    dataset_name='fake_hotel_guests')
```

guest_email	has_ rewards	room_ type	amenities_ fee	checkin_ date	checkout_ date	room_ rate	billing_address	credit_card_ number
michaelsanders @shaw.net	False	BASIC	37.89	27 Dec 2020	29 Dec 2020	131.23	49380 Rivers Street Spencerville, AK 68265	4075084747483975747
randy49 @brown.biz	False	BASIC	24.37	30 Dec 2020	02 Jan 2021	114.43	88934 Boyle Meadows Conleyberg, TN 22063	180072822063468
webermelissa @neal.com	True	DELUXE	0.00	17 Sep 2020	18 Sep 2020	368.33	0323 Lisa Station Apt. 208 Port Thomas, LA 82585	38983476971380
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

https://github.com/sdv-dev/SDV

Test Datasets, Modellen-Ableitung von echten Daten mit verschiedene Algorithmen.

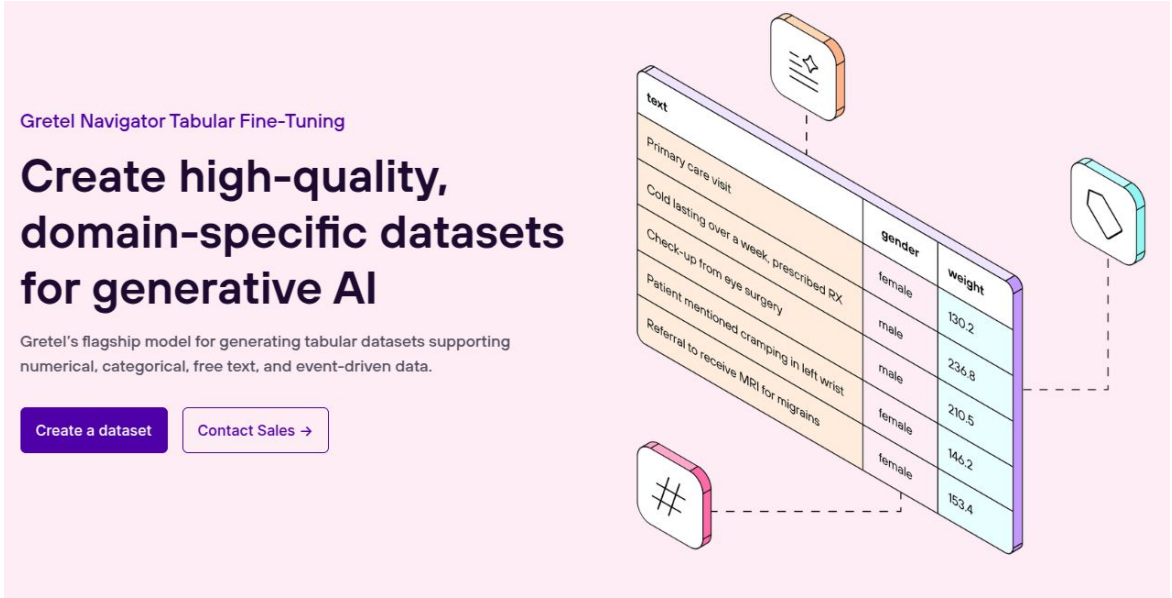
Werkzeuge - KI-gestützt (lokal, komplex)

```python
import pandas as pd
from mostlyai.sdk import MostlyAI

# load original data
repo_url = 'https://github.com/mostly-ai/public-demo-data'
df_original = pd.read_csv(f'{repo_url}/raw/dev/census/census.csv.gz')

# instantiate SDK
mostly = MostlyAI()

# train a generator
g = mostly.train(config={
        'name': 'US Census Income',          # name of the generator
        'tables': [{                         # provide list of table(s)
            'name': 'census',                # name of the table
            'data': df_original,             # the original data as pd.DataFrame
            'tabular_model_configuration': { # tabular model configuration (optional)
                'max_training_time': 2,      # cap runtime for demo; set None for max accuracy
                # model, max_epochs,,..      # further model configurations (optional)
                'differential_privacy': {    # differential privacy configuration (optional)
                    'max_epsilon': 5.0,      # - max epsilon value, used as stopping criterion
                    'delta': 1e-5,           # - delta value
                }
            },
            # columns, keys, compute,..      # further table configurations (optional)
        }]
    },
    start=True,                              # start training immediately (default: True)
    wait=True,                               # wait for completion (default: True)
)
```

https://github.com/mostly-ai/mostlyai

Python-Toolkit für hochpräzise, datenschutzsichere synthetische Daten. Lokaler und Cloud-Modus, Export trainierter Modelle

Werkzeuge - KI-gestützt (lokal, komplex)



https://github.com/gretelai/gretel-synthetics

Benötigt quasi einen KI/ML-Hintergrund …

Werkzeuge - Benchmarking

Die bekanntesten dürften sein:

- pgbench - ein “Grundnahrungsmittel” für jeden, der mit Postgres mehr zu tun hat
- sysbench - Skriptfähiger Datenbank- und Systemleistungsbenchmark (Lua)
- HammerDB - TPC-C und TPC-H Belastungstesten (Oracle, MS SQL Server, IBM Db2, PostgreSQL, MySQL, MariaDB), TCL Skripte
- Benchbase - Multi-DB, Multi-Model (früher OLTPBench), das beste Benchmark-Modellunterstützung
- Time Series Benchmark Suite - zwei Tests (Systemüberwachung, IoT), alle Haupt-DBs aus dem TSDB-Bereich unterstützt
- pgbent (ein Klon von pgbench-tools): versch. Klient / Scale / PG Konfigurations-Kombinationen testen und visualisieren

Werkzeuge - Subsetting & Anonymisierung

- PostgreSQL Anonymizer - eine Erweiterung zum Maskieren oder Ersetzen personenbezogener Daten (PII) oder vertraulicher Geschäftsdaten
- greenmask - Postgres-Subsetting, Anonymisierung und synthetische Generierung
- pg_sample - Teilmenge mit Wahrung der referenziellen Integrität
- Jailer - ein GUI-fokussierter Subsetter
- pg_anonymize - eine Erweiterung zur Steuerung der Anonymisierung durch Postgres-Sicherheitslabels
- pg_dump | sed ... | psql

Werkzeuge - Fazit

Ich habe persönlich ziemlich viele dieser Tools ausprobiert ... bin aber überwiegend zu folgendem Schluss gekommen:

- Jedes Werkzeug bringt wie üblich einige Einschränkungen mit sich
  - Diese treten meist erst nach einigen ersten, einfacheren Meilensteinen zutage
- Leider stolpert man häufig auf Fehler und Leistungsbarrieren
- Da größere “Front-up” DB Testprojekte eher selten sind (bei mir mindestens), vergisst man oft die Tool-Details - und die Tools ändern sich natürlich auch ständig

Daher, empfehle ich bei einmaligen Projekten mit einfachere Anforderungen - vielleicht ist es OK **mit etwas ganz Einfachem zu beginnen?**

- SQL
- PL/pgSQL
- Anonymisierung von Dumps oder Snapshots
- Geprüfte „Seed“-Datensätze gemischt und multipliziert

Beschleunigungstricks

Beschleunigungstricks - schnelles Füllen der Platte

Die Generierung gut randomisierter Daten ist sehr CPU-intensiv.

Wenn das Ziel lediglich darin besteht, die Festplatte zu füllen, um zu sehen, wie sich die Datenbank bei großen Datenmengen generell verhält, welche Latenzen wir bei minimalem Caching bekommen, oder welche Fehlermeldungen auftreten - kann man Folgendes einsetzen:

- **UNLOGGED Tabellen** - überspringt WAL/Transaktionsprotokoll, daher deutlich weniger Schreiblast...aber ☠️-Gefahr beim Crashen
- **FILLFACTOR reduzieren** - Der Fillfactor ist ein Attribut auf Tabellenebene, das angibt, wie dicht die Zeilen “gepackt” werden
  - Niedrigerer Fillfactor → Wir belasten die Festplatten stärker

Beschleunigungstricks - „Seed“-Daten

Wieder - Die Generierung gute, realistische und insbesondere längere Datensets ist sehr CPU aufwendig ...

Wenn möglich, verwenden Sie vorhandene kleine Datensätze wieder, egal ob generiert oder aus der Produktion - mischen Sie sie ein wenig auf und fügen Sie sie erneut ein – fertig!

```
INSERT INTO x(data)
  SELECT data || random()::text FROM x LIMIT 1e4;
```

PS Habe über ähnliche Tricks irgendwann mal auch geschrieben: https://kmoppel.github.io/2022-12-23-generating-lots-of-test-data-with-postgres-fast-and-faster/

Beschleunigungstricks - verwendung von “Open” Data

Handelt es sich um eine relativ standardisierte Anwendung (CRM, Webshop/Verkaufsdaten, Lagerbestand), gibt es bereits zahlreiche vorhandene Datensätze.

Man kann diese reinladen und dann selektiv relevante Datensätze / Spalten in das eigene Test-Schema einfügen ...

https://wiki.postgresql.org/wiki/Sample_Databases
https://www.kaggle.com/datasets
https://huggingface.co/datasets
https://datasetsearch.research.google.com/
https://registry.opendata.aws/
https://datahub.io/collections
https://github.com/kmoppel/pg-open-datasets

Beschleunigungstricks - Indizes als letztes!

Kann einen riesengroßen Unterschied machen!

Was ich oft mache:

pg_dump --section=pre-data $prod | psql $dev

pgbench -n -f gen_testdata.sql -t 1000000 -c 16

psql -c “delete duplicates if any ...” # Eine Implementierung hier

pg_dump --section=post-data $prod | psql $dev

Die Strategie der „Entschleunigung“

Bei linearem Wachstum, oder gut Korrelierte Aktivitäten (TSP→ Hardware Belastung), ist es manchmal* tatsächlich eine gute Idee, nicht zu versuchen, Produktionsumgebungen akribisch zu replizieren, sondern sich bewusst für schwache Hardware zu entscheiden!

- Und stattdessen etwas Mathe machen … ⚠
- Kann einiges an Zeit und Geld sparen

Sehr relevant auch für den modernen “Serverless“ Ansatz – Autoscaling kann offensichtliche Probleme durch transparente Skalierung verbergen!

- CU-s limitieren!

Stolperfallen

Stolperfallen

Einige Dinge, die Sie beim Testen mit synthetischen Daten beachten sollten:

- Testen Sie nicht mit Spielzeug-Datensätzen!
- Postgres kann nach einer längeren Periode normaler Aktivität aufgrund von „Aufblähung“ ziemlich verlangsamt werden
  - Beim Testen sollte man es einrechnen!
- Leistungstests sollte man idealerweise nicht auf einem einzelnen Hardwareknoten durchgeführt werden, sondern auf mehreren verschiedenen, um die Auswirkungen zu sehen

Stolperfallen

- Bei Cloud-Instanzen der unteren Preisklasse werden die IOPS und auch die Disk/Netzwerkbandbreite schwer gedrosselt (plus “Bursting)
- Vermeiden Sie übergroße Transaktionen mit über 100M+ Zeilen
  - Gehen Sie besser mit Schleifen voran, um eine bessere Sichtbarkeit / Wiederaufnehmbarkeit zu gewährleisten
- Erwarten Sie nicht, dass synthetische Tests die Produktion exakt widerspiegeln können!
  - User Aktionen > Pseudozufälligkeit
  - Caching spielt eine entscheidende Rolle bei Big-Data und ist schwer komplett richtig einzustimmen

Zum Mitnehmen

Sogar einfache, schnelle Tests, mit produktionsähnlichen Daten - die konkrete Zahlen liefern - sind deutlich verlässlicher als bloßes Hoffen oder Bauchgefühl

VIELEN DANK!

FRAGEN?

FOLIEN