Postgres test data generation 101

Kaarel Moppel, Freelance PostgreSQL Consultant
pgDay Nordic 2025, Copenhagen

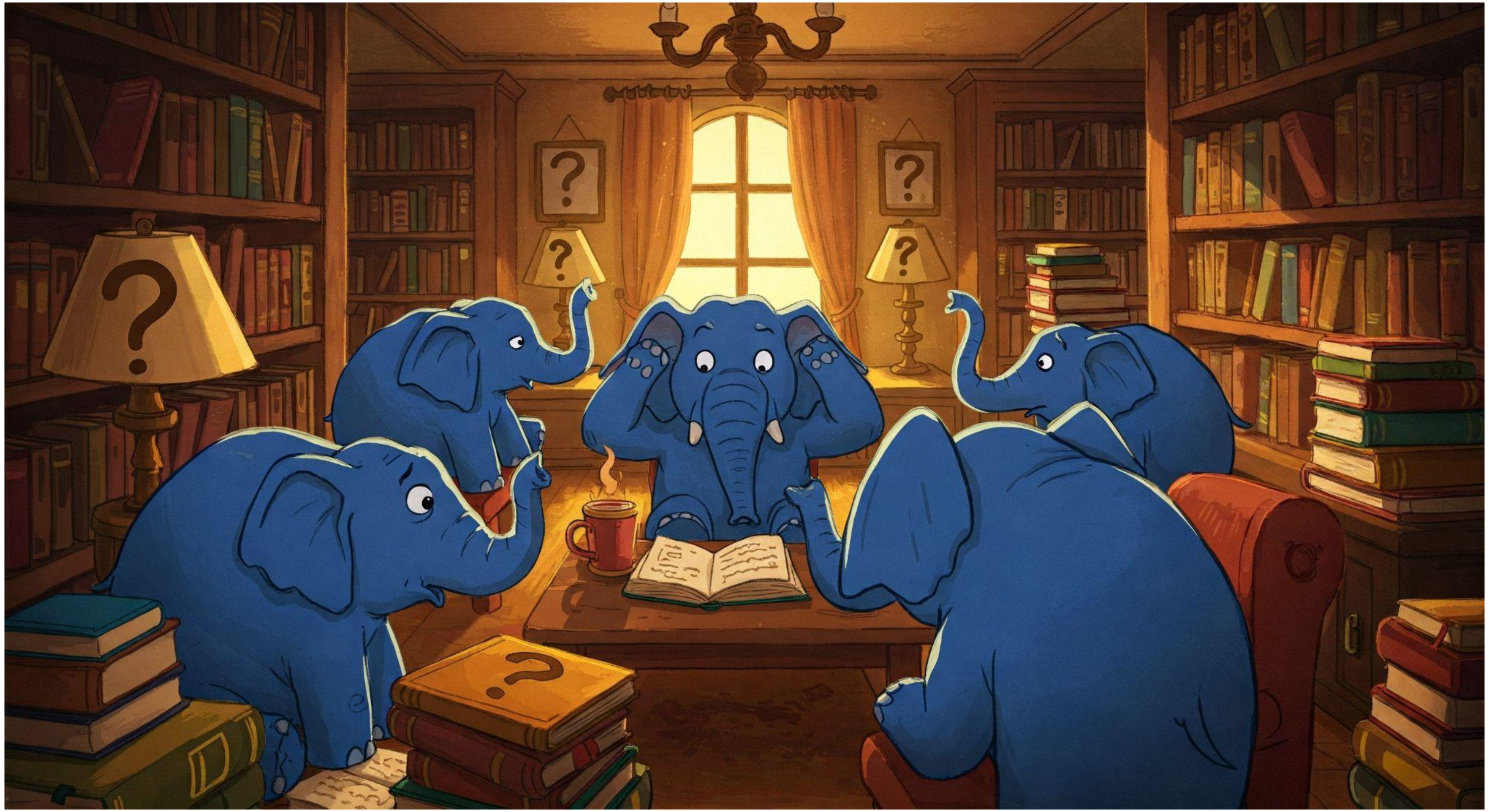
$ whoami

- Full-time “wrestling” with databases since 2007
- 20K+ hours in the Postgres ecosystem
    - Many hats along the way
    - Have developed somekind of a gut feeling on “Postgres”-y things if anything
- Up for Postgres related consulting
    - https://kmoppel.github.io/ (Blog & Contact)

Agenda

- The “why”
- Common techniques
- Advanced techniques
- Tooling
- AI-assisted tooling
- Speeding things up
- Gotchas

The “why”



Why bother with DB testing / validation ?

Nobody is asking that question for app code, right ?

As a consultant I’m seeing over and over again that the initial DB layout (or even the single-node approach in general) was completely not suitable for upcoming *known* data growth / request counts ...

Such that in a few years $someone will have to deal with:

- Jumpy or allout bad query performance
- Manual DB maintenance routines
- Unplanned work / incidents / downtime
- Massive HW upscaling to cover some peaks and sleep peacefully

Could have been avoided with some pretty basic* DB-side validation!

Benefits of test data generation / performance testing

- Setting up a benchmark forces one to think a bit more about the DB design
- Takes away some FUD around DB internals
  - Might be “forced” to learn about a Postgres setting or two
  - Makes future experimenting more cheap / accessible!
- Should bring out some obvious performance and concurrency bottlenecks
  - Assures that the design can handle the projected workload
- Validates approx query performance / TPS per $$
- Validates hardware / cloud provider degradation and settings
  - Not all clouds are created equal

Benefits of test data generation / performance testing

- If have a lot of data (+ incoming streams) can gauge how doable / time-consuming / expensive database migrations might be in the future
    - The “oh, we’ll then just migrate” myth
- How much space / $$ would backups and snapshots allocate for huge DBs after compression / deduplication steps?
- How much WAL will we be generating, can LR even keep up?
- How much time would it take to run a “pg_dump” or PITR restore clone from a future life-sized DB?
    - Just in case - “pg_dump” is not a good backup strategy

The “elephant in the room”

Gaps in DB side knowledge and lack of awareness on importance

- Often considered “someone else’s” territory - meaning just overlooked or testing limited to unit / functional / integration tests across the DB boundary
- The app frameworks / deployment systems often get in the way
    - Testdata tasks are pretty long-running
- Hard to fix the knowledge gap in a short time obviously (even with AI)...but there are a few basic techniques that should give the 80% result with “little effort”™

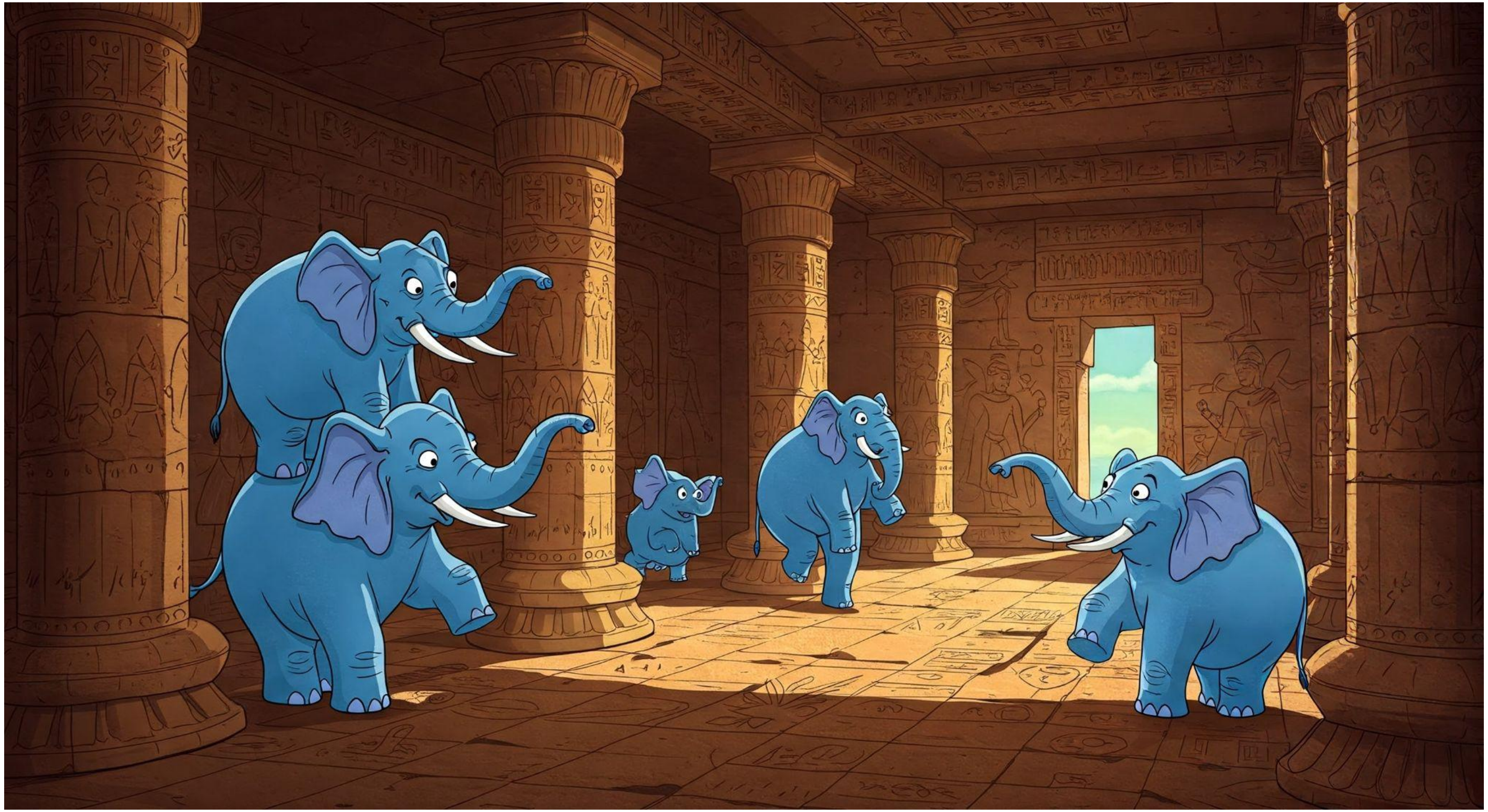
Finding a window in the SDLC process

Finding where to plug in “database stuff” is a real problem actually - even in today’s world of abstractions - a database is often not “meat”, not “fish”. And worse - even scale-ups, not to mention startups, don’t want* or can’t find a DBA ...

I’d advocate for a two-pronged approach:

- Just task someone “DB-able” with ad-hoc one-time validation
  - Ignoring team’s common CI/CD flow of doing things if needed. Standalone SQL or Python scripts and some result numbers are infinitely better than nothing!
  - DB engines are pretty stable and try to use robustly scaling algorithms - the initial pre-rollout verification is the most critical!
- Setting up some automation to be able to run through a more extensive / slow DB pipeline also “per need”, when some danger / long term possible side-effects identified
  - A mini-version of the former so to say, with some visible feedback so that “non-wizards” could be alarmed

Must have techniques



Techniques - generate_series()

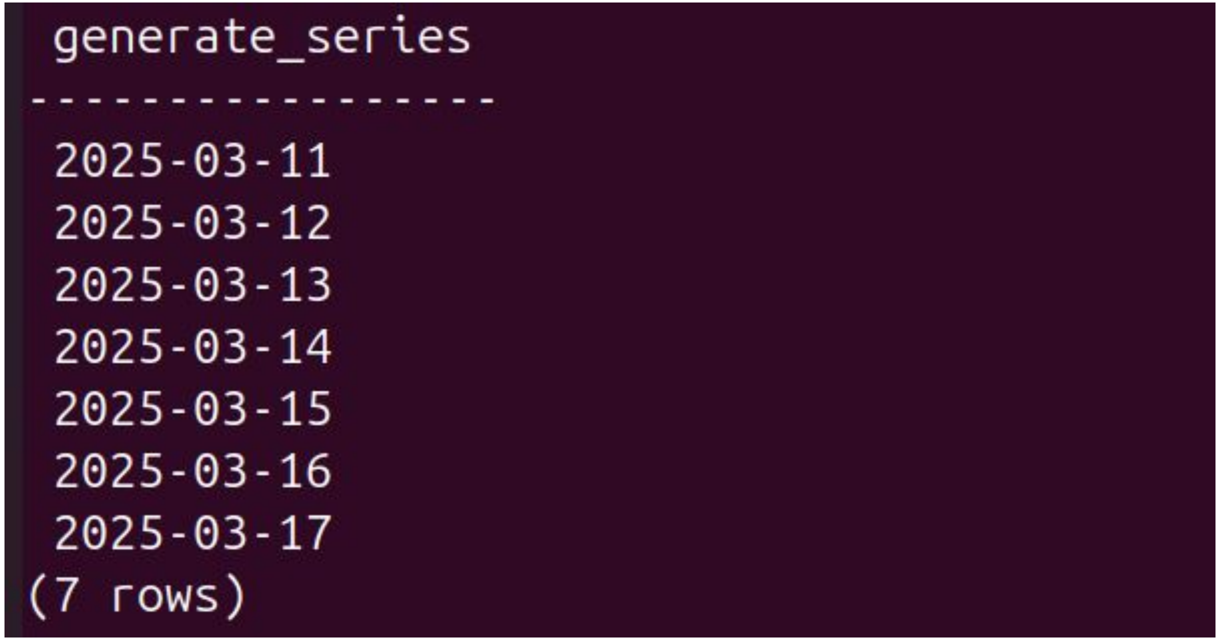
The **generate_series()** function is a must have tool in a Postgres dev's toolbox!

- A generator function to "draw" sequences / rows from
- Similar to Python's "range"
- Supports numerals and dates / timestamps
- Step / stride

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

Techniques - generate_series()

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



```
 generate_series
-----------------
 2025-03-11
 2025-03-12
 2025-03-13
 2025-03-14
 2025-03-15
 2025-03-16
 2025-03-17
(7 rows)
```

Techniques - generate_series()

Q: How big would our event table will look like after 3years, assuming we have 50 INSERT’s per second?

```
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;
...
```

Techniques - randomizing

```
SELECT random(); -- float /  double precision between 0.0 <= x < 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+

-- From the “tablefunc” extension
SELECT * FROM normal_rand(1000, 5, 3); -- 1k values with a mean of 5 and stddev 3

SELECT setseed(0.666); -- to have repeatable “random” data
```

PS For more serious / expensive randomization try the “pgcrypto” extension or extract some parts from gen_random_uuid()

Techniques - CASE WHEN random() chaining

A classic to randomize between a few choices or increase randomness / add some jitter by chaining together a few *random()*-s

```
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;
```

Techniques - PL/pgSQL

Ideally one should remain in pure SQL or SQL functions “territory” (faster*), but if logic gets too unreadable PL/pgSQL is a good choice still for “in-DB” generation

```
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 $$
DECLARE
    len int; idx int;
BEGIN
    len := array_length(items, 1);
    idx := 1 + floor(random() * len)::int;
    RETURN items[idx];
END; $$;
```

Techniques - TABLESAMPLE

To choose larger chunks of shuffled / randomized data PostgreSQL supports the SQL:2003 standard TABLESAMPLE clause:

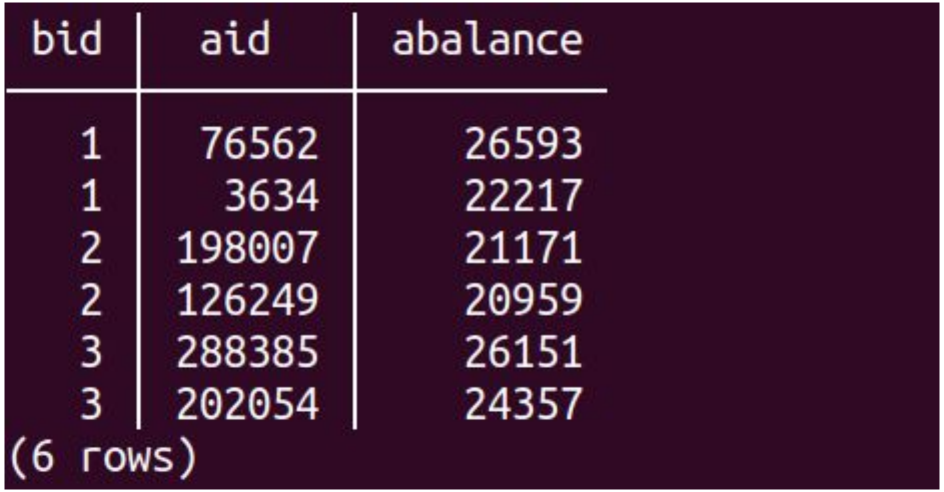
```
-- Take 30% of data
SELECT * FROM pgbech_accounts TABLESAMPLE SYSTEM (30) ;
```

SYSTEM - takes some random blocks, takes all records from those block
BERNOULLI - scans the whole table and randomizes individual rows
SYSTEM_ROWS - exact rows. Need to create the "tsm_system_rows" extension

Techniques - LATERAL Joins

Lateral enables JOIN-level "generators"- i.e. for each input "parent" / "left side" row, we want to dynamically choose "child" or "fact" rows based on some criteria. A **MUST HAVE technique** in a data / database engineers toolbox!

```
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)
```

Techniques - LATERAL Joins

PS Also variable rowcounts per group is possible! Especially in ML model training can be crucial that all groups are represented, albeit a little.

```
-- Assuming have some “driving” table column available
-- ALTER TABLE  pgbench_branches ADD rowlimit int DEFAULT (6*random())::int ;

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;
```

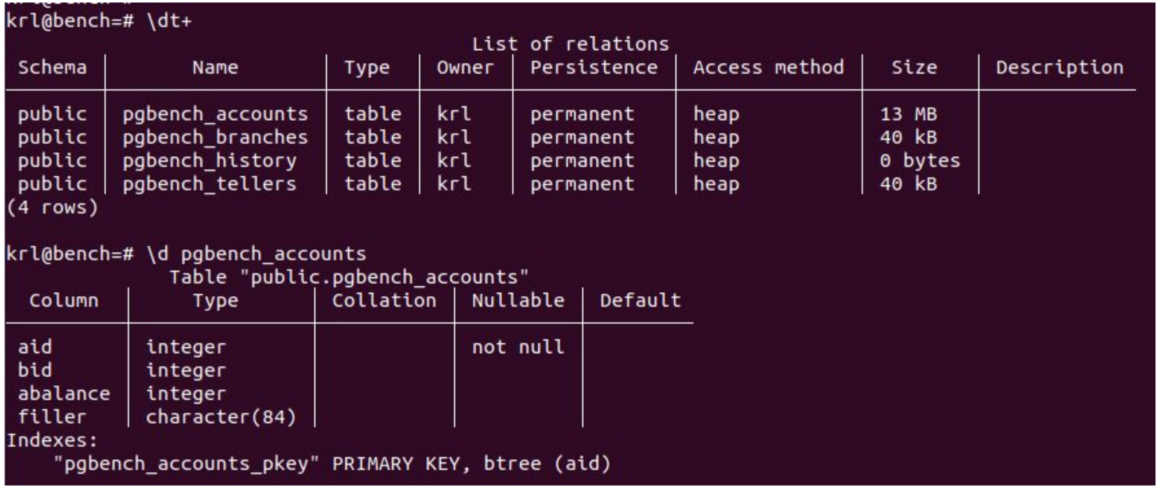
Advanced techniques

Advanced techniques - pgbench

Pgbench is a lightweight and easy to use benchmarking tool / framework tfrom the Postgres project (might not be bundled with “client” packages though)

- Revolves around a simplistic OLTP-style banking schema (by default, TCP-B like)
- Can be scripted and parallelized

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)
```

FYI - to “grok” the magic scale units I’m usually using a little helper up on JSFiddle . More here on how the formula was derived.

Advanced techniques - custom pgbench scripts

The default schema / test scripts are rarely useful outside of stress testing or getting an approximate latency feel for indexed key operations.
Can use **custom SQL** files or “pgbench” scripts to play with custom schemas, variables, different types of randomness, fetch some setup data from DB / shell, if / else.

```
$ pgbench  --show-script simple-update

-- simple-update: <builtin: simple update>
\set aid random(1, 100000 * :scale)
\set bid random(1, 1 * :scale)
\set tid random(1, 10 * :scale)
\set delta random(-5000, 5000)
BEGIN;
UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :delta,
CURRENT_TIMESTAMP);
END;
```

Advanced techniques - custom pgbench scripts

```
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;
```

Advanced techniques - custom pgbench scripts

```
# A short version of a some actual test I ran to choose a partitioning strategy

# Set up the schema / import data distributions from production
...

# Reset internal Postgres stats counters
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

# Analyze the metrics ...
```

Advanced techniques - using real table stats

Allows to easily generate “near to real life” distributions. In case the real values are not a secret, would needs some custom handling / hashing otherwise

```
SELECT
  schemaname,
  tablename,
  attname,
  null_frac,
  avg_width,
  n_distinct,
  most_common_freqs,
  correlation,
  most_common_vals::text::text[],  -- assuming no secrecy issues
  histogram_bounds::text::text[]   -- has real values in it
FROM pg_stats
WHERE tablename IN ('pgbench_accounts','...');
```

Advanced techniques - increasing stats precision

If want to “test clone” (*) a larger existing DB distribution, one should know that the Postgres stats are by default very lossy - ANALYZE scans max 30k pages (~234 MB). If your data changes rapidly or is skewy then defaults are not enough!

A workaround is to increase the “stats target” temporarily, update stats, export, roll back.

```
begin;
set default_statistics_target to 400 ; -- ~1GB
analyze pgbench_accounts ; -- PS will block Autovacuum!
\copy ... -- export pg_stats
rollback; -- NB! Commit could flip some plans
```

One stats exporting-importing example here:

Advanced techniques - jumping over FK hurdles

When populating some “real” app schema, it can be tedious to insert test data - as all Foreign Keys need to be satisfied ...

Or with performance testing we might only care about a few critical / fast-growing tables, not the correctness of the whole “spiderweb”.

Workarounds:

- A schema dump with minimal table definitions only:
  pg_dump --section=pre-data mydb
- A custom schema dump with a few tables only:
  pg_dump -t customers -T '*bigtable*' mydb
- A Postgres session level hack to disable background FK triggers:
  SET session_replication_role TO replica ; ⚠

Tooling

Tooling - no shortage

The “problem space” is not new actually - quite some tools out there! They assist mostly with test data generation, but also with anonymization and load testing / actual benchmarking.

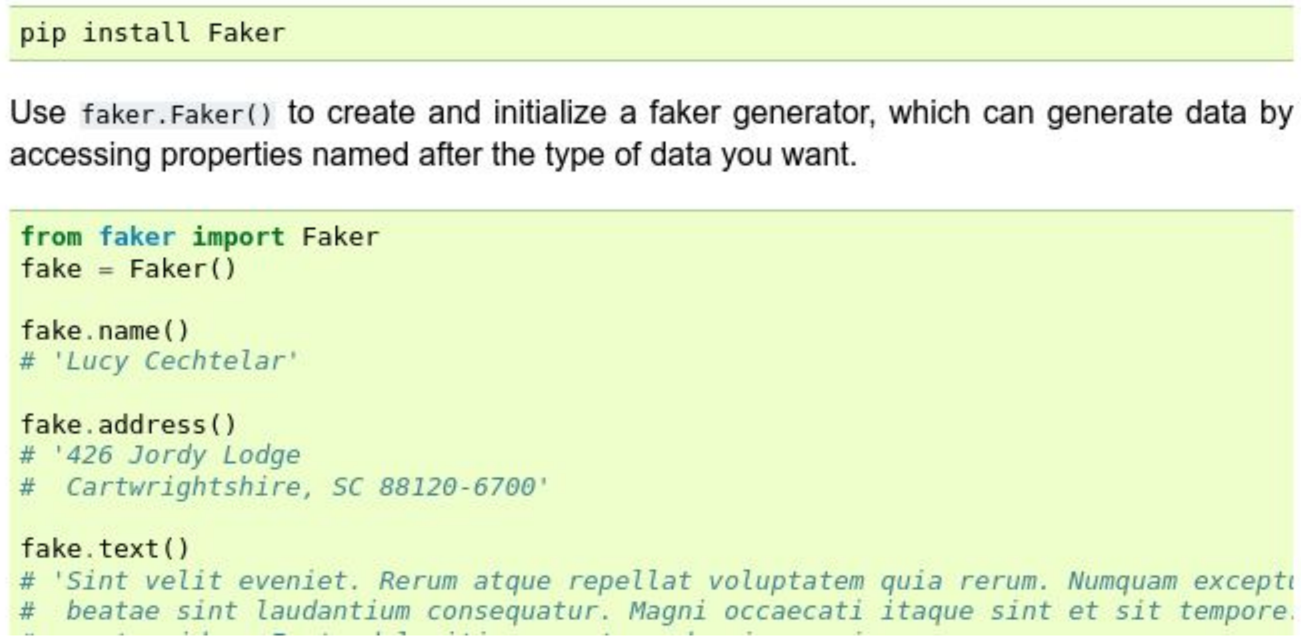
Benefits of generating synthetic test data:

- Privacy / security - no real data can leak
- Faster development - don’t have to wait for any green lights / gate-keeping
- Flexibility - can often speed up things if to disconnect from real world constraints
    - Some duplicate / cheap “filler” data OK in most cases
    - Rarely actually need all columns for example to be authentic to develop “a feature or two”, just similar size / volume and index cardinality
- With “AI” can get pretty close to “authentic” nowadays
    - Could run into time / latency / cost issues though ...

Tooling - generation

The “faker” Python library / CLI - “mother” on many other tools / wrappers

- Built-in providers and Community providers / extra domains



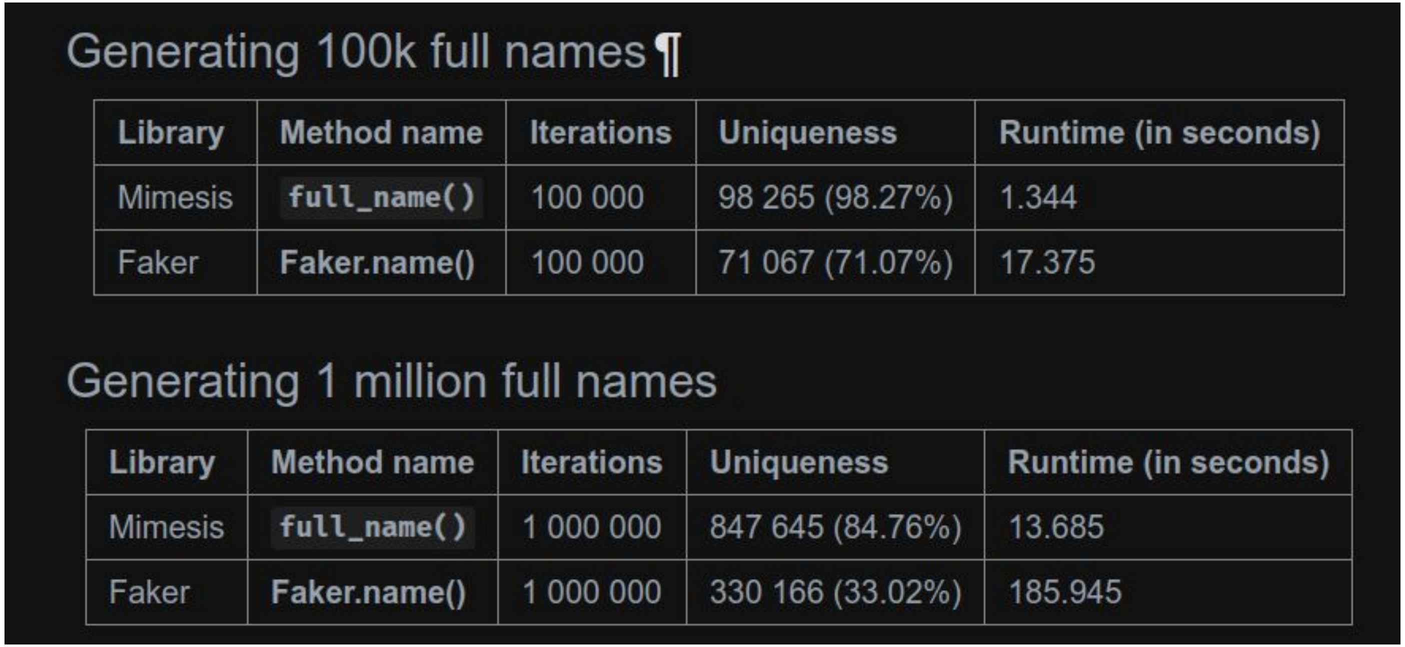
```
pip install Faker
```

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.
```

Tooling - generation

mimesis - or a sort of faker++ in Python - faster and with a bit more humanly 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

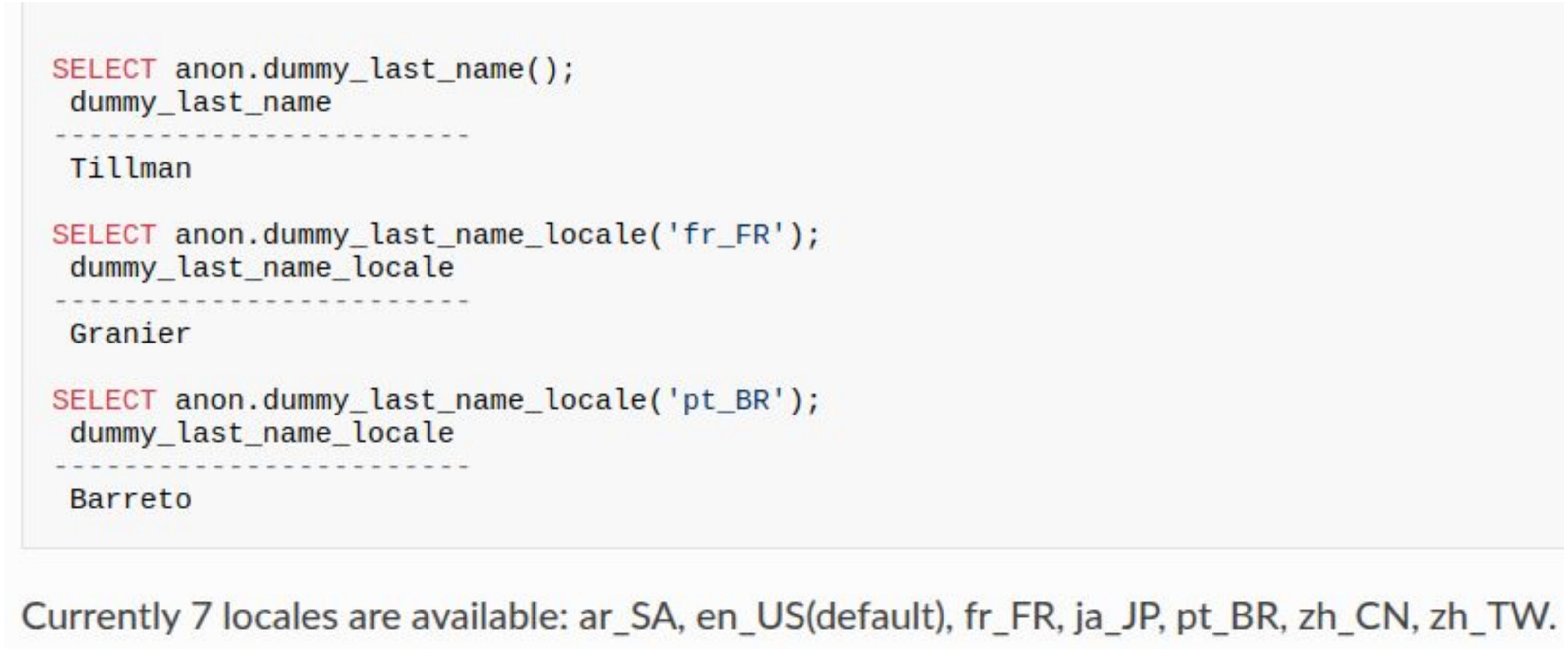
Generating 1 million full names

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

Tooling - generation

PostgreSQL Anonymizer - Dalibo's postgresql_faker functionalities have moved

- Some built-in seed datasets + custom functions



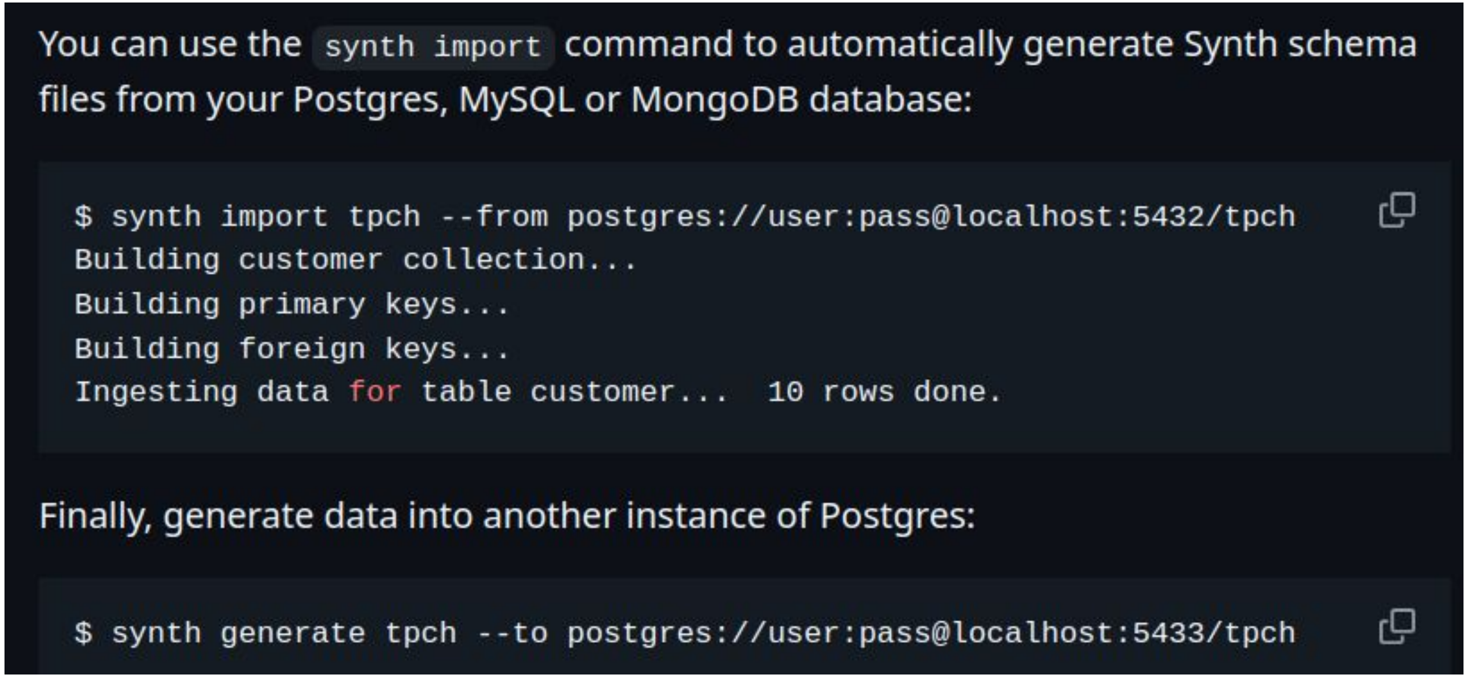
```
SELECT anon.dummy_last_name();
 dummy_last_name
------------------------
 Tillman

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.

Tooling - generation

Synth - was a pretty promising declarative generator

- With option to infer data type / nature from live DB-s



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

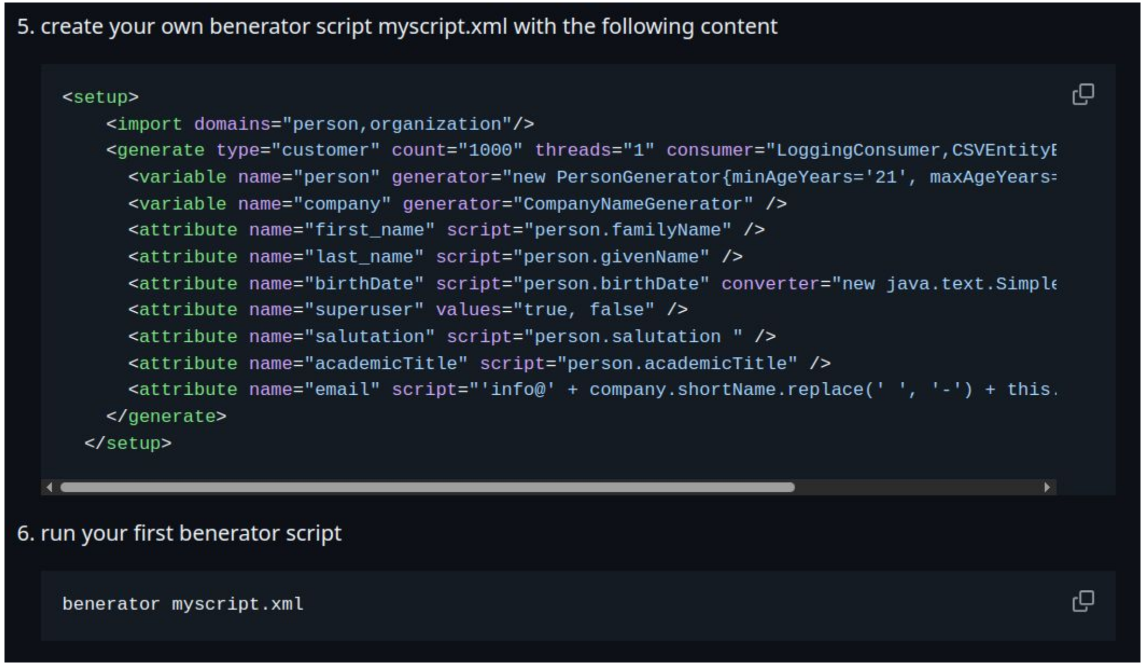
```
$ synth import tpch --from postgres://user:pass@localhost:5432/tpch
Building customer collection...
Building primary keys...
Building foreign keys...
Ingesting data for table customer...  10 rows done.
```

Finally, generate data into another instance of Postgres:

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

Tooling - generation

Benerator - “old style” XML description to files / RDBMs
DATAMIMIC - a more modern “AI”-aided version of Benerator



5. create your own benerator script myscript.xml with the following content

```
<setup>
    <import domains="person,organization"/>
    <generate type="customer" count="1000" threads="1" consumer="LoggingConsumer,CSVEntityE
      <variable name="person" generator="new PersonGenerator{minAgeYears='21', maxAgeYears=
      <variable name="company" generator="CompanyNameGenerator" />
      <attribute name="first_name" script="person.familyName" />
      <attribute name="last_name" script="person.givenName" />
      <attribute name="birthDate" script="person.birthDate" converter="new java.text.Simple
      <attribute name="superuser" values="true, false" />
      <attribute name="salutation" script="person.salutation " />
      <attribute name="academicTitle" script="person.academicTitle" />
      <attribute name="email" script="'info@' + company.shortName.replace(' ', '-') + this.
    </generate>
  </setup>
```

6. run your first benerator script

```
benerator myscript.xml
```

Tooling - generation

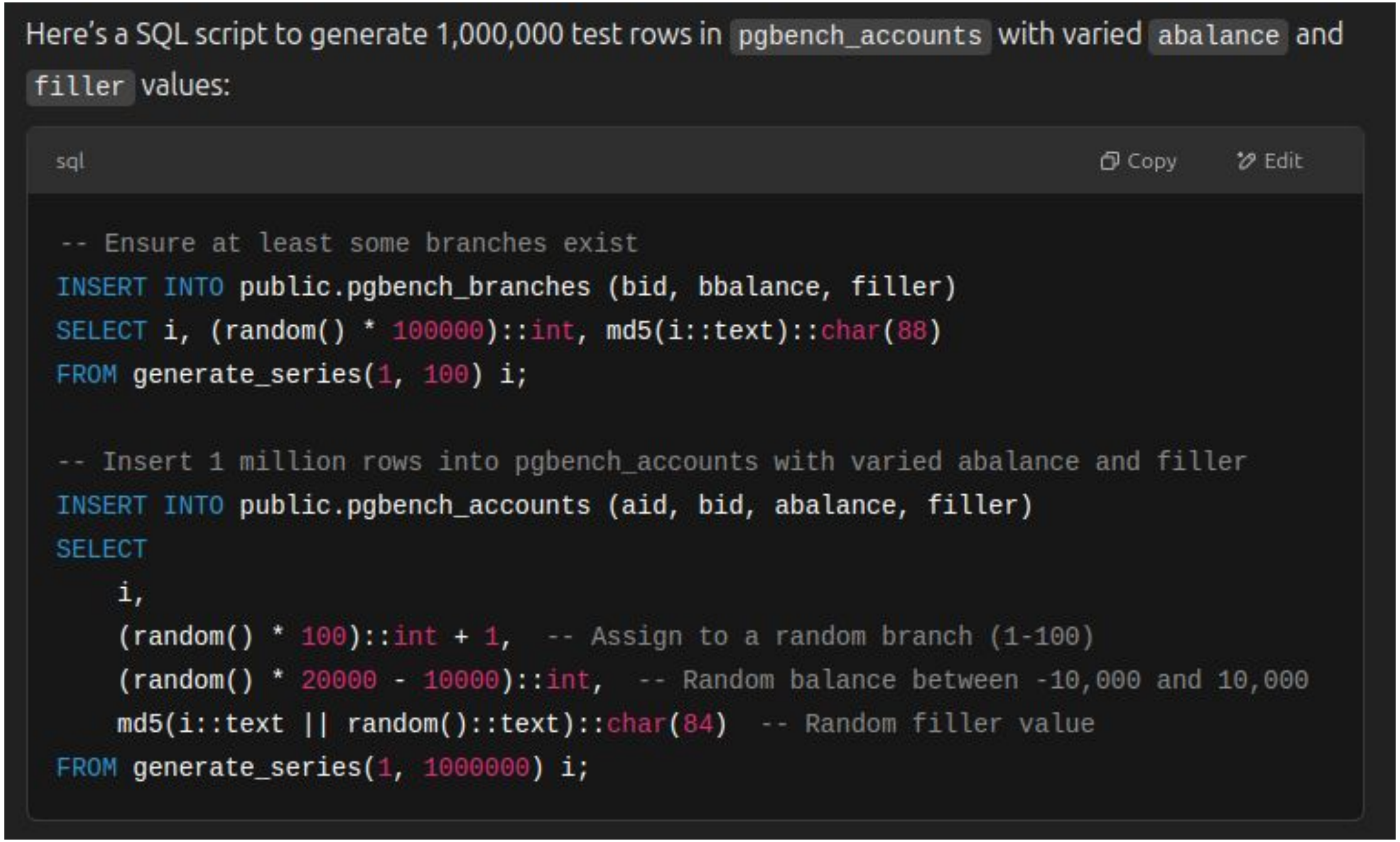
Not really that hard to throw together “something” for a limited business domain project. For example I’ve also got a small set of pure-SQL functions for myself for main data types / object classes to generate random, or choiced / ranged, pseudo-random data.

The benefit of pure-SQL is that it runs on all flavors of Postgres and doesn’t need any complex installation or privileges and can fire up “pgbench” quickly.

```
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);
```

Tooling - AD 2025

Tooling - generation, AI-assisted (cloud)



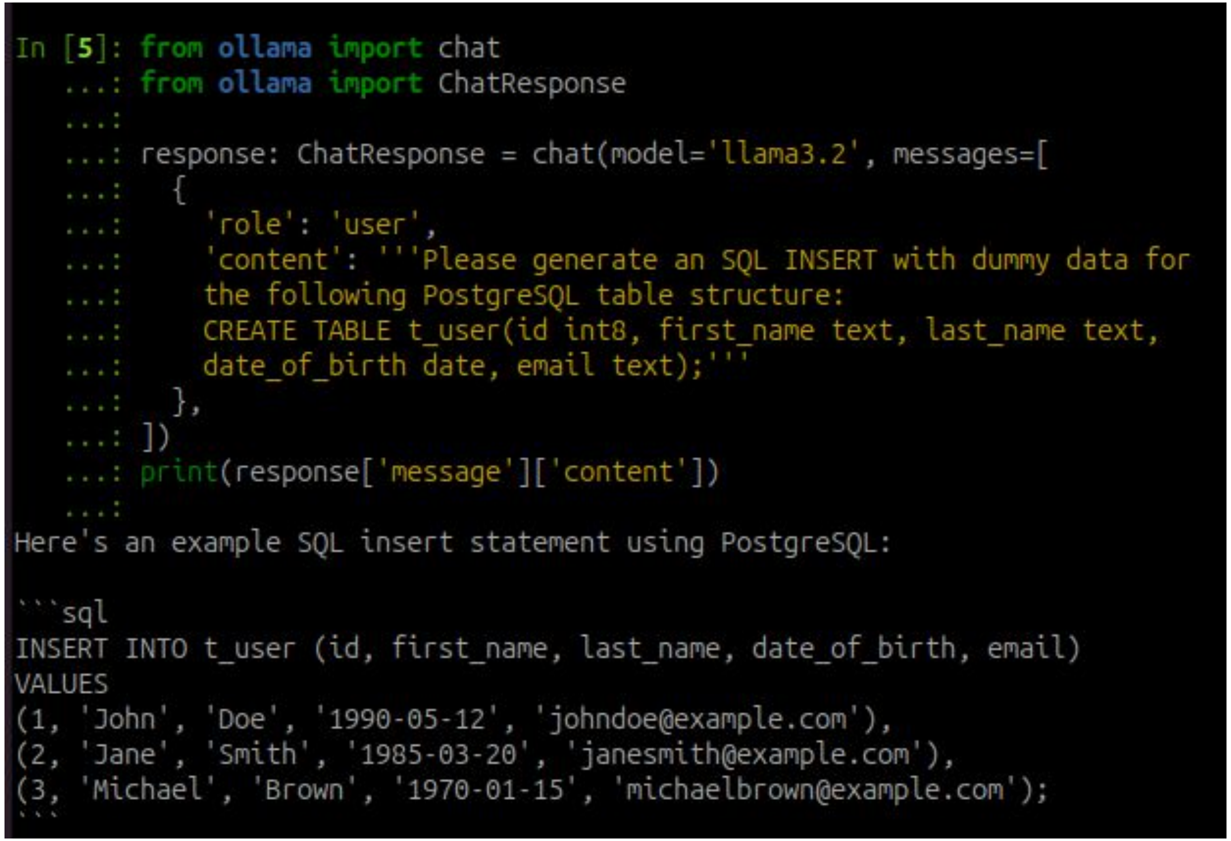
Here's a SQL script to generate 1,000,000 test rows in `pgbench_accounts` with varied `abalance` and `filler` values:

```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;
```

A promising start at least!

Used ChatGPT prompt link

Tooling - generation, AI-assisted (local)

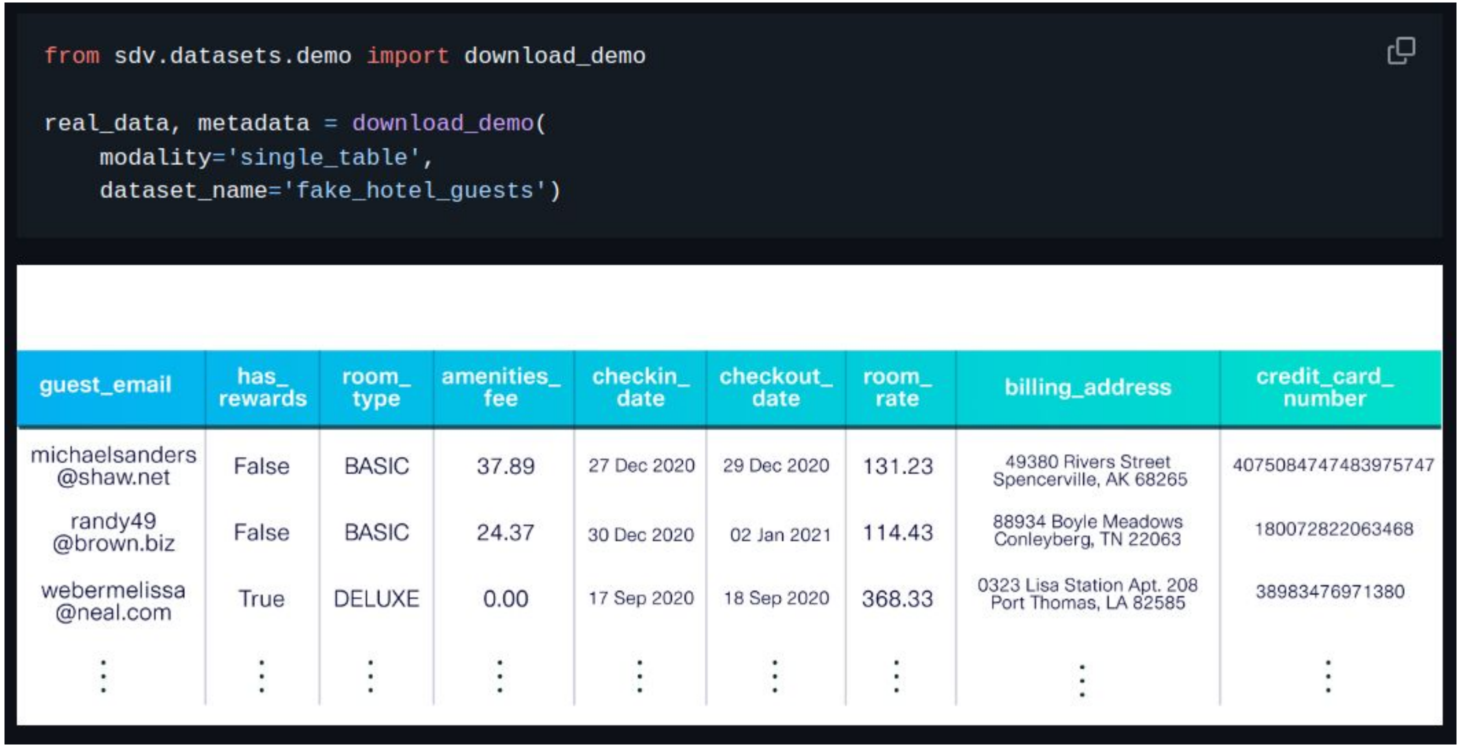


~~~
In [5]: from ollama import chat
   ...: from ollama import ChatResponse
   ...:
   ...: response: ChatResponse = chat(model='llama3.2', messages=[
   ...:   {
   ...:     'role': 'user',
   ...:     'content': '''Please generate an SQL INSERT with dummy data for
   ...:     the following PostgreSQL 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');
```
~~~

Couldn't get much simpler really with Ollama:

1. Install Ollama
2. Start Ollama (ollama serve)
3. pip install ollama
4. Run the Python code (for better DB injection in dict or JSON format + use streaming mode)

Tooling - generation, AI-assisted (local)

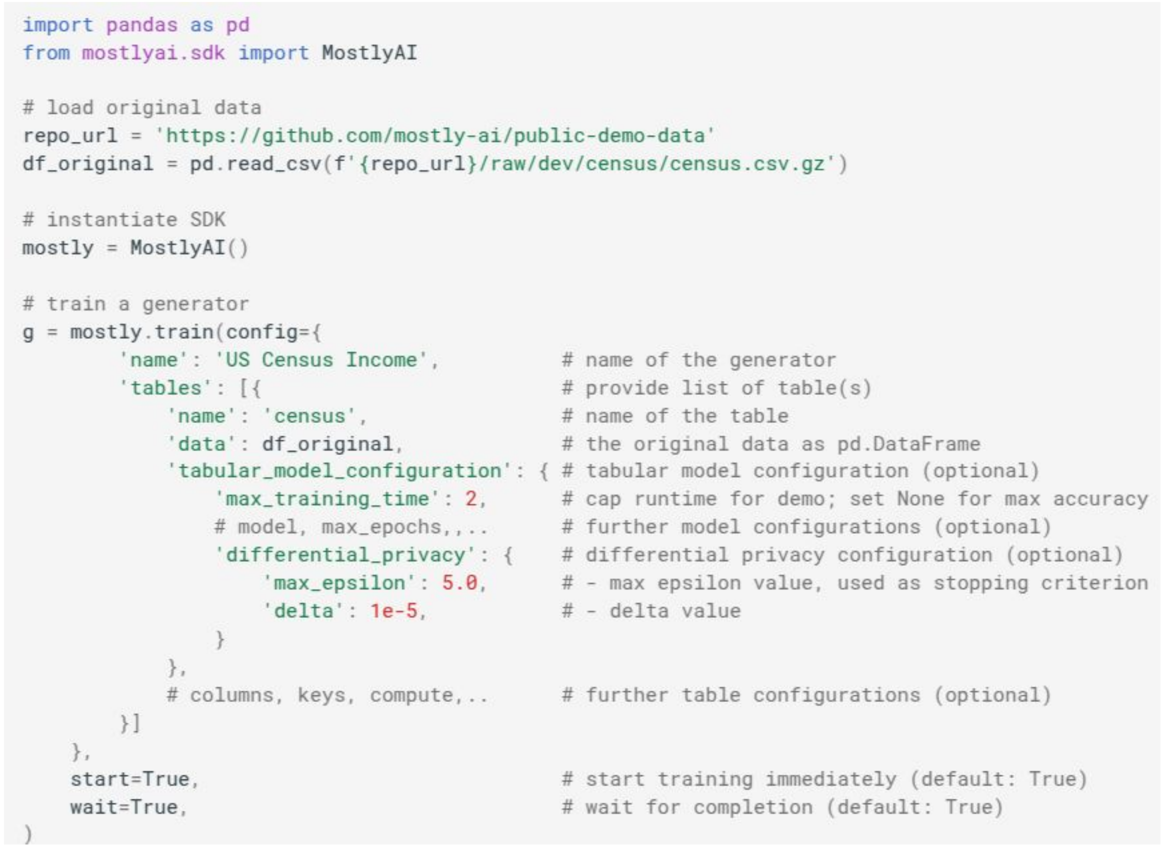


```python
from sdv.datasets.demo import download_demo

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

Tooling - generation, AI-assisted (more knobs)

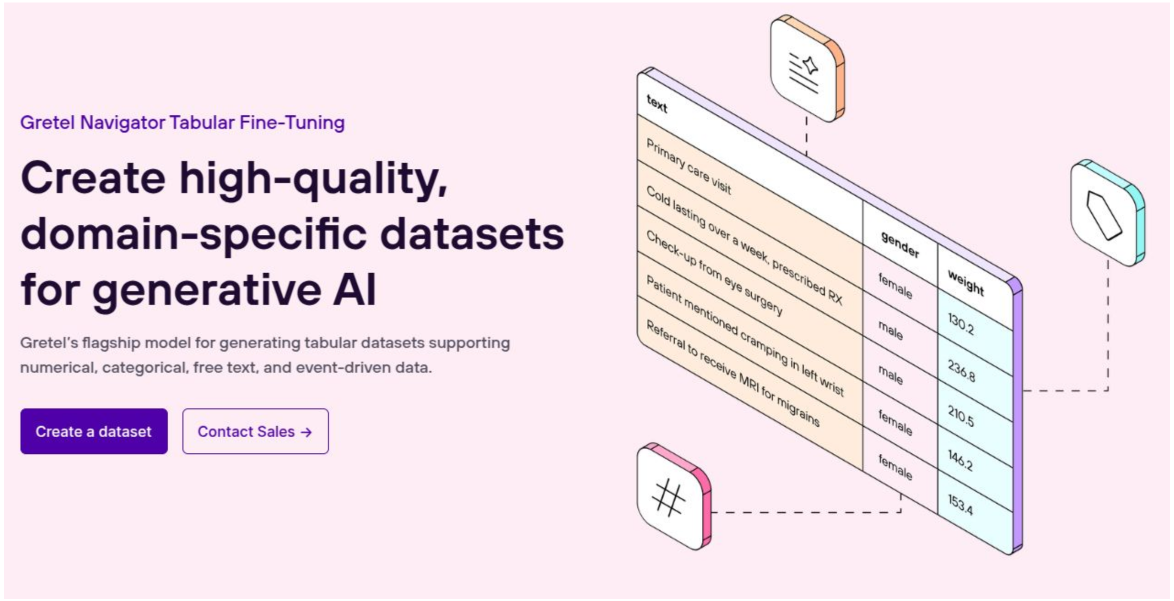


```python
import pandas as pd
from mostlyai.sdk import MostlyAI

# 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 for high-fidelity, privacy-safe Synthetic Data. Local and client modes, trained model exporting / re-use

Tooling - generation, AI-assisted (more knobs)



Gretel Navigator Tabular Fine-Tuning

Create high-quality, domain-specific datasets for generative AI

Gretel's flagship model for generating tabular datasets supporting numerical, categorical, free text, and event-driven data.

text	gender	weight
Primary care visit	female	130.2
Cold lasting over a week, prescribed RX	male	236.8
Check-up from eye surgery	male	210.5
Patient mentioned cramping in left wrist	female	146.2
Referral to receive MRI for migrains	female	153.4

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

Quite advanced already, need some AI/ML background

Tooling - benchmarking

Most known ones should be:

- pgbench - a staple really for anyone orbiting Postgres
- JMeter - Not exactly DB focused, but battle-tested and can do scripting, parallelism, query param randomization / fetching from DB
- sysbench - Scriptable database and system performance benchmark (DB scripts in Lua)
- HammerDB - TPC-C and TPC-H load testing / benchmarking (Oracle, MS SQL Server, IBM Db2, PostgreSQL, MySQL, MariaDB, scripting in TCL)
- Benchbase - (formerly OLTPBench), Multi-DBMS, Multi-model, best benchmark support
- Time Series Benchmark Suite - two tests (system monitoring, IoT), main DBs from the TSDB space
- pgbench-tools - run combinations of database sizes, concurrent client count, Postgres configuration sets

Tooling - subsetting / anonymization

- PostgreSQL Anonymizer - an extension to mask or replace personally identifiable information (PII) or commercially sensitive data from a Postgres database
- greenmask - PostgreSQL subsetting, anonymization and synthetic generation
- pg_sample - subsetting with RI kept
- Jailer - a GUI focused subsetter
- pg_anonymize - an extension to control anonymization by Postgres security labels
- pg_dump | sed ...

Tooling - summary

I’ve personally tried quite a few of those tools...but mostly concluded:

- Not that they create as many problems as they solve, but every tool brings some limitations as usual
  - Which tend to surface only when past some first simpler milestones
  - Consider if you really need all those features
- Sadly common pretty to hit bugs or performance / parallelism walls
- As bigger front-up DB testing projects are rare-ish, one tends to forget about the tool details and the tools change

Thus for one-off kickstarts or schema suitability evaluation tasks, where also requirements differ a lot project-by-project and re-use might be harder - you probably want to start with something super-simple, like good’ol:

  - SQL
  - PL/pgSQL
  - Anonymization dumps or snapshots
  - Safe “seed” datasets mashed and mixed

Speeding things up

Speeding things up - fast disk filling

Generating well randomized data is pretty CPU intensive ...

If the goal is just to fill the disk, to see how the DB behaves with huge volumes in general, what latencies we're gonna get when caching is minimal, or if alerting kicks in, what errors we get - one could employ:

- **Unlogged tables** - skips WAL / transaction log, much less writing overhead / locking
  - Data will not survive a server crash though!
- **Lowering the "fillfactor"** - fillfactor is a table-level attribute, saying how densely we pack the rows into data pages
  - Lower FF → we pressure the disks more heavily

Speeding things up - use of “seed” data

Again - generating good, real life looking, and especially longer column data is expensive.

If possible - re-use some existing small dataset, be it generated or from production, mix it up a bit and re-inject - voila!

```
INSERT INTO x(data)
  SELECT data || random()::text FROM x LIMIT 1e2;
```

(I’ve blogged about some other similar tricks also in the past)
https://kmoppel.github.io/2022-12-23-generating-lots-of-test-data-with-postgres-fast-and-faster/

Speeding things up - using similar “open” data

If the app is a relatively standard one (CRM, Webshop / Sales facts, Inventory) there are quite some existing datasets out there.

One can load those up and selectively insert some similar columns into your own schema (with some randomization)

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

Speeding things up - applying indexes in the last step

Makes a huuuge difference!

What I commonly do:

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

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

psql -c “delete duplicates if any ...” # 1 version on how to do here

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

The “slowing things down” strategy

A counter-intuitive trick can come handy time-to-time also ...

With linear growth plus a simple schema (or just a verified correlation to more data / activity) it’s actually a good idea to not try to meticulously replicate production environments, but going for weak hardware on purpose!

- And do some math instead ... ⚠
- Can save quite some time / $$

Very relevant also for the modern “serverless” approach - which could hide obvious problems via transparent scaling!

- Limit CU-s!

Gotchas

Gotchas

Some things to be aware of in regards to testing with synthetic data:

- Postgres can slow down quite considerably after a longer period of normal activity due to “bloat” - to account for that the datasets should always be larger than expected in real life or some “fillfactor” should be set (~80%).
    - Also the test runtime should be as long as tolerable
- Ideally perf testing should not happen on a single hardware node, but on a few different ones to see the effects
- Very low-end cloud instances are usually throttled also on IOPS and network bandwith (bursting kicking in and out can make things even more fun)
- Avoid huge transactions with 100M+ rows, loop in chunks for better visibility / resumability
    - For DB side looping prefer CREATE PROCEDURE / CALL syntax + batch COMMITs

Don’t expect synthetic testing to cover all production problems …

THANK YOU!

QUESTIONS?

SLIDES