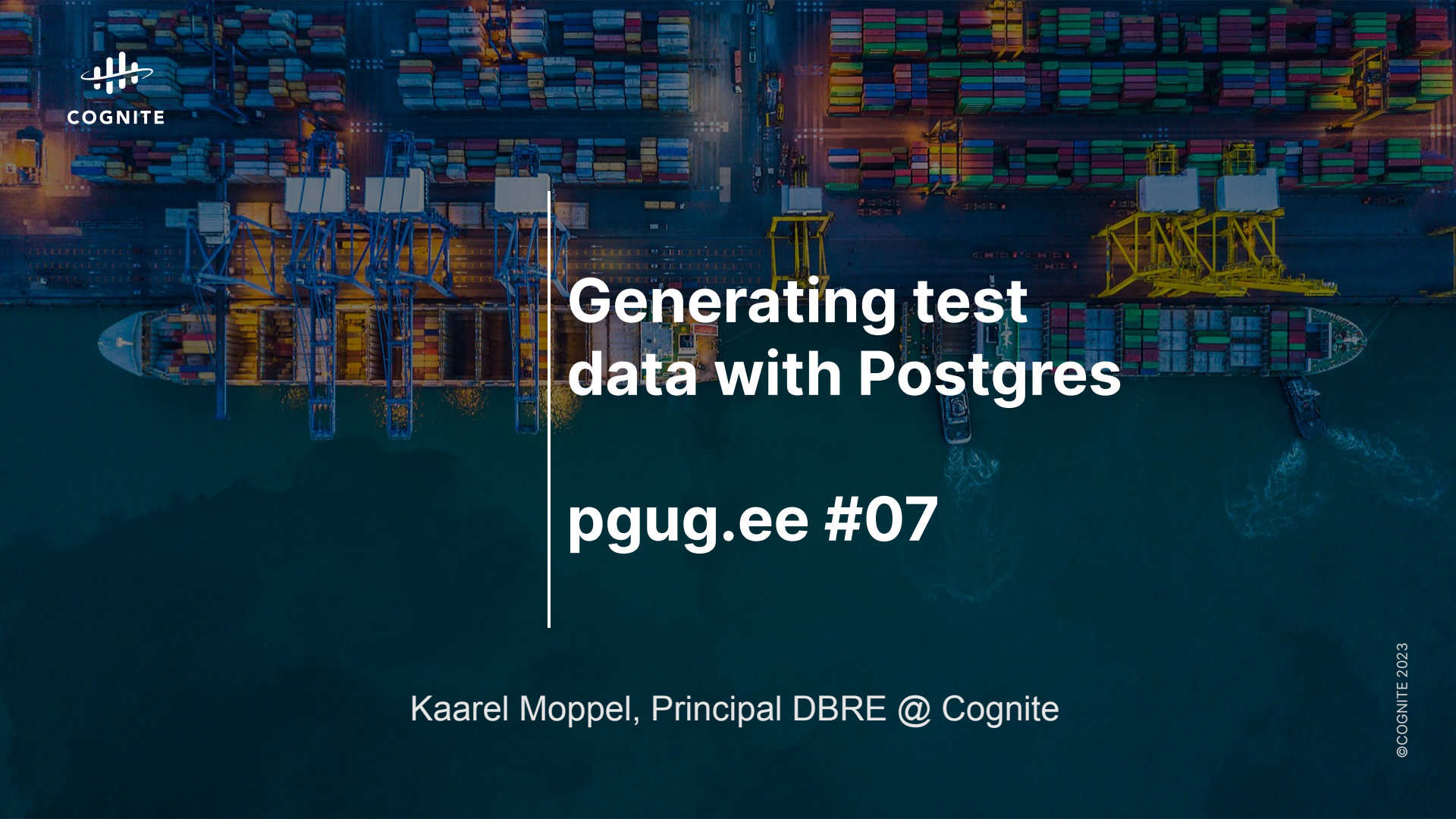


Generating test data with Postgres

pgug.ee #07

Kaarel Moppel, Principal DBRE @ Cognite

COGNITE

Me & Postgres

- **Daily companions since 2011**
  - Schema design
  - Perf troubleshooting
  - Keeping things running / HA on self-managed
  - DB Lifecycle management tooling
  - Consulting / training

Agenda

- Why bother?
- Techniques for test-data generation
- Speeding things up
- Tooling
- Gotchas

Why bother with DB performance testing?

Nobody is asking that question for app code and algorithms, right ?

For quite a few backends I have seen that the initial DB layout and the single-node approach was completely not suitable for actual 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 / costly migrations

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

Benefits of DB performance testing

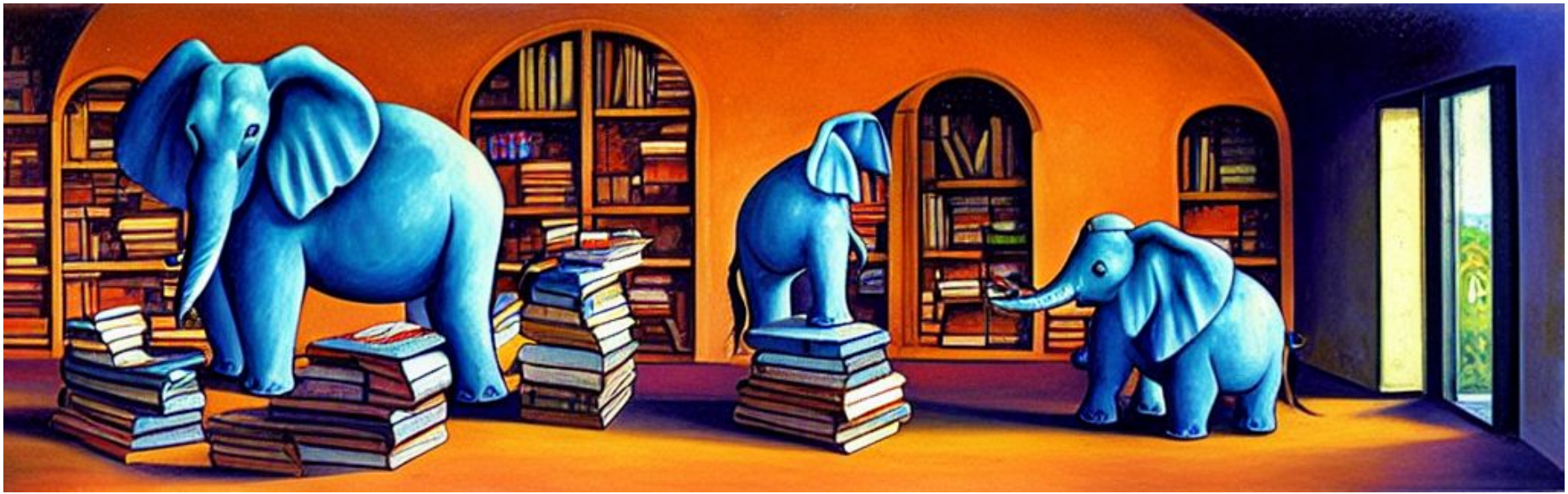
- Setting up test cases forces one to think more about the DB design
- Takes away some “fear” / FUD around DB internals
  - Makes future experimenting more cheap / accessible
- Should bring out some obvious concurrency bottlenecks
- Indicates approx query performance / TPS per $$
- Validates hardware / cloud provider degradation and settings
  - Not all clouds are created equal
  - E.g. was hugely visible for Azure’s 1st gen “Single Server” offering
- Can gauge how time-consuming / expensive database migrations might be in the future

Assures that the chosen design can handle the expected workload (plus some)!

The problem - DB testing rarely performed

Gaps in DB side knowledge and lack of awareness on importance*

- Often considered “someone else’s” territory - meaning DB just overlooked or limited to functional / integration testing
- The app frameworks / deployment systems often get in the way or the DB-related app layers change too fast
  - OK to have them just as standalone SQL or Python scripts for local / ad-hoc testing - better than nothing!
  - Don’t need to run constantly - DB engines / clouds are pretty stable, the initial pre-rollout verification is the most critical
- Hard to fix the knowledge gap in a short time obviously...but there are some basic techniques with Postgres that should give the 80% result with “little effort”™



[blue elephants in a study, lots of books, cartoon style]

Techniques - generate_series()

The **generate_series()** function is a must have tool in a Postgres devs toolbox!

- A “virtual table” to draw sequences / rows from
- Similar to Python’s “range”
- Supports numeric and date / timestamp data types and variable steps

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

Techniques - generate_series()

select * from generate_series(current_date, current_date + '11mons'::interval, '1month') with ordinality x(n, i);

n	i
2023-10-25 00:00:00	1
2023-11-25 00:00:00	2
2023-12-25 00:00:00	3
2024-01-25 00:00:00	4
2024-02-25 00:00:00	5
2024-03-25 00:00:00	6
2024-04-25 00:00:00	7
2024-05-25 00:00:00	8
2024-06-25 00:00:00	9
2024-07-25 00:00:00	10
2024-08-25 00:00:00	11
2024-09-25 00:00:00	12

(12 rows)

Techniques - randomizing

```
SELECT random(); -- float /  double precision between 0.0 <= x < 1.0

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


SELECT setseed(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

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

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

SELECT
    CASE WHEN random() < 0.05 THEN
        least (ceil(random() * 10)::int, 10)
    WHEN random() < 0.2 THEN
        1
    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

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

SELECT random_choice(array['a', 'b', 'c', 'd']);
```

Techniques - LATERAL Joins

Lateral enables “generators”- i.e. for each input “parent” row, we want to dynamically generate “child” or “fact” rows with slightly varying column data. Again a MUST HAVE technique in a data 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

A bit more tougher case is with variable rowcounts multiplier per group - Postgres doesn't allow variables / randomization directly in the LIMIT clause, but there's a secret workaround to add a temporary columns to the driving table.

```
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
    ) a ON TRUE;

ALTER TABLE  pgbench_branches DROP COLUMN rowlimit ;
```

Advanced techniques - pgbench

Pgbench is a lightweight and easy to use benchmarking tool / framework that comes with Postgres.

- Revolves around a simplistic OLTP banking transactions schema (by default)
- 3 built-in tests (tpcb-like*, simple-update, select-only)
- Can be easily scripted, parallelized, rate-limited, ...

pgbench --initialize --scale=1 # 1 scale unit = 100k bank accounts ~ 13MB of main table data
pgbench --client=2 --time=10 # Do default transactions (3 UPD, 1 SEL, 1 INS) 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)
```

Advanced techniques - pgbench

Beware - output KPIs are measured from client side - can **differ a lot** from DB side metrics!

$ pgbench -c 2 -T 10 --report-per-command

```
...
latency average = 0.892 ms
tps = 2242.164129 (without initial connection time)
statement latencies in milliseconds and failures:
        0.024           0  BEGIN;
        0.144           0  UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
        0.053           0  SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
        0.057           0  UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
        0.055           0  UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
        0.043           0  INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :delta,
CURRENT_TIMESTAMP);
        0.515           0  END;
```

VS

```
krl@postgres=# select mean_exec_time from pg_stat_statements where query ~ '^UPDATE pgbench_accounts' ;
   mean_exec_time
______________________
 0.09260896579176205
```

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 - BUT can use custom SQL files or “pgbench” scripts to play with variables, different types of randomness, fetch some setup data from DB / shell etc

$ 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, id as table id, 1 as pgbench_helper FROM public.table_info
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 =
'my_rows') x ORDER BY random() LIMIT 1 \gset

\set shard_id random(0, 9)

select shard_id, key, data, last_changed_time
from my rows
where shard id = :shard id
and project id = :project_id
and table id = :table id
and last changed time > ':tz'::timestamptz
and last changed time <= ':tz'::timestamptz + '6h'::interval
order by shard_id asc, last_changed_time desc, key asc limit 1001;
```

Advanced techniques - custom pgbench scripts

A short version of an actual test I ran to verify partitioning effects

```
# Set up the schema / import data distributions
...

# 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 with different values

```
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
NOT schemaname IN ('pg_catalog', 'information_schema')
AND 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
```

Advanced techniques - jumping over FK hurdles

When rolling out the default app schema, it can be tedious to insert test data as all Foreign Keys need to be satisfied :/

But with performance testing we often only care about a few critical tables, not the correctness of the whole “spiderweb”.

Workarounds:

- A custom schema dump with minimal table definitions only:
  pg_dump --section=pre-data -t tblx -t '*bigdata*' mydb
- A Postgres session level hack to disable background FK triggers*:
  SET session_replication_role TO replica ;

Advanced techniques - AI

Seems one can also already put robots to work in this area 🤖 ...

Python ML libs, Pandas dataframes:

https://www.youtube.com/watch?v=iYngoCRnM1Q


MUST...GO...FASTER
imgflip.com

Speeding things up - fast disk filling

Generating randomized data is pretty CPU intensive.

If the goal is just to fill the disk, to see how the DB instance behaves with huge volumes in general, what latencies we're gonna get when caching is minimal - one should 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 CPU expensive.

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

Also not much point to modify columns that are not used by the testcase (especially if not indexed) - just match the byte-size of rows.

```
INSERT INTO big_data(...)
 SELECT important_col+random(), unimportant1, unimportant2
 FROM big_data LIMIT 1e6 ;
```

Speeding things up - using similar “open” data

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

One can load those up and selectively insert some similar columns into your own schema, or stir up a bit.

https://wiki.postgresql.org/wiki/Sample_Databases
https://www.kaggle.com/datasets
https://datasetsearch.research.google.com/
https://datahub.io/collections

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

# One should prefer INSERT .. ON CONFLICT DO NOTHING but sometimes need to deal
# with duplicates - most common SQL version on how to do that is here
psql -c “$delete duplicates if any ...”

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

Tooling - lot of 3rd party choice

Besides these standalone techniques there are of course lots of tools out there, that assist with test data generation, load testing / actual benchmarking.

I've also tried quite a few of those...but mostly concluded that they create as many problems as they solve :) It's mostly a one-off task per design, so you want something super simple to get some 80% certainty with 20% time.

Some things to semi-recommend though:

- 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 though)
- postgresql_faker - A Postgres extension around the popular Python Faker Library
- Synth - Introspect an existing DB, generate dummy data into another DB (*)
- mimesis - A Python lib with a more human-like touch compared to Faker

Gotchas

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

- Generating only random data is not “real life”
- 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 real life expectations. And 10GB is not big data!
  - Also the test runtime should be as long as tolerable
- Ideally testing should happen on a few SKUs, cloud* has variance
- Testing on very low-end cloud instances doesn’t make much sense - they’re throttled on many parameters
- Avoid huge transactions with 10M+ rows - loop in chunks for better visibility / resumability
  - For DB side looping scripts avoid standard stored functions / anonymous DO $$ blocks, prefer CREATE PROCEDURE / CALL syntax + COMMIT after loop iterations

Additional links / tools

- https://www.postgresql.org/docs/16/pgbench.html
- https://www.postgresql.org/docs/16/tablefunc.html
- https://www.postgresql.org/docs/16/functions-string.html
- https://www.postgresql.org/docs/16/functions-math.html
- https://kmoppel.github.io/2022-12-23-generating-lots-of-test-data-with-postgres-fast-and-faster/
- https://github.com/timescale/benchmark-postgres
- https://techcommunity.microsoft.com/t5/azure-database-for-postgresql/how-to-benchmark-performance-of-citus-and-postgres-with-hammerdb/ba-p/3254918
- https://github.com/Wisser/Jailer

COGNITE

THANK YOU!

QUESTIONS?