Common DB schema change mistakes

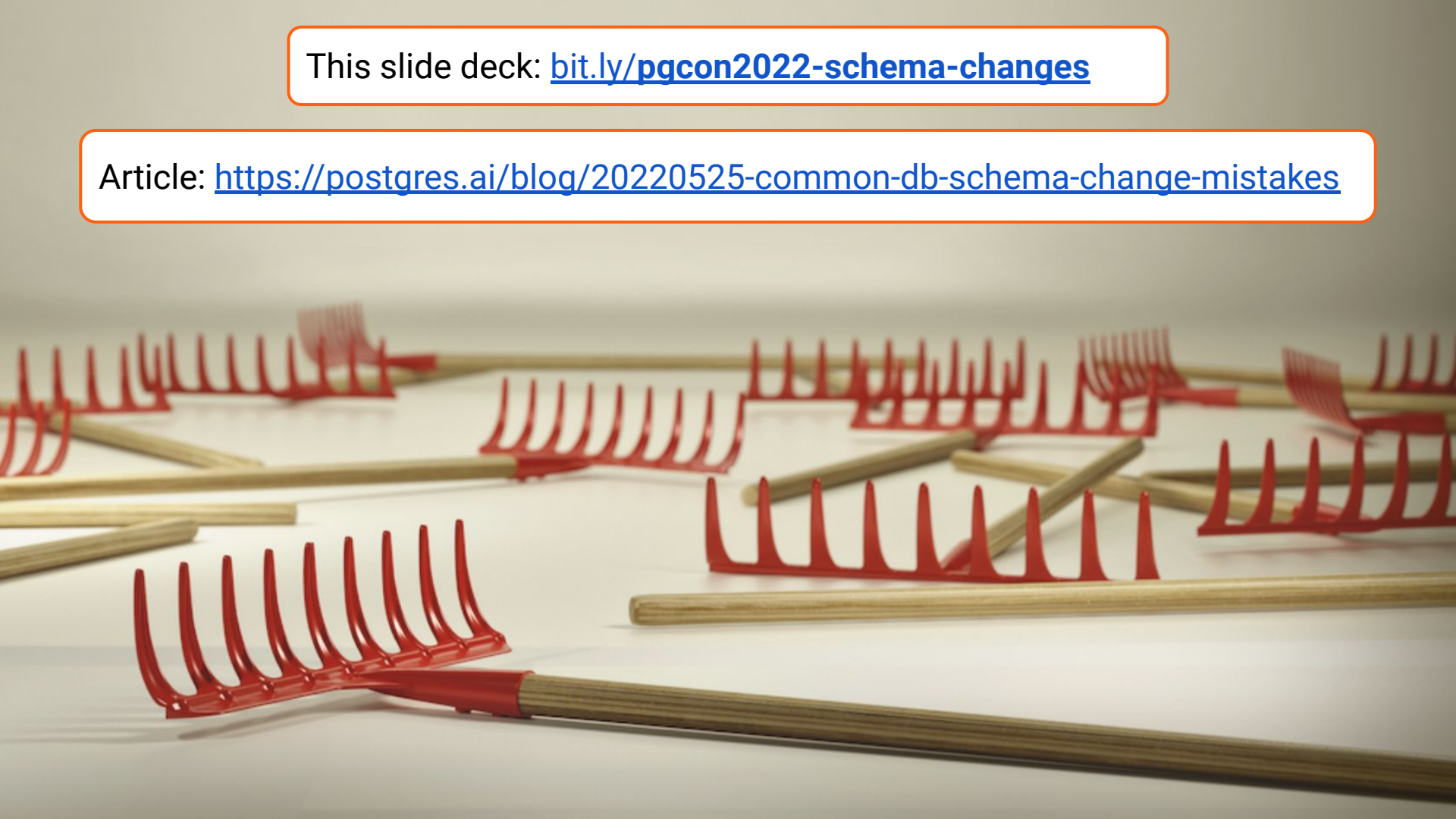
Nikolay Samokhvalov

nik@postgres.ai

Postgres.ai

This slide deck: bit.ly/**pgcon2022-schema-changes**

Article: https://postgres.ai/blog/20220525-common-db-schema-change-mistakes



Preamble…

We found a slow query – looks like a missing index.

We decide to create it.

What can go wrong?


I'M CHECKING NEW INDEXES
FOR OUR POSTGRES DATABASE
NOT ON PRODUCTION, RIGHT?
NOT ON PRODUCTION, RIGHT?
imgflip.com

Nikolay Samokhvalov

PostgreSQL user since 2005

Occasional code contributions (XML, etc.)

Co-founded 3 startups (social media; 2 successful exits)

Helped with Postgres to: GitLab, Chewy, Miro, etc.

Postgres.ai founder

SUBSCRIBE

YouTube **Postgres.tv**

Twitter **@samokhvalov**

@Database_Lab

👉 Created/reviewed more than 1,000 DB changes

Postgres.ai

Help companies with PostgreSQL scalability and performance

Database Lab Engine
– thin clones for Postgres

Database Lab

An abstract example:
- DB size: 10 TiB
- DLE – a single VM with 10 TiB of disk space
- A single DLE provides 30-50 thin clones, each is 10 TiB
  - Engineers work independently
  - CI/CD pipelines run automated tests
- To get a new clone:
  - *~10 seconds and $0 (!)*

Used by: GitLab, QIWI, Allianz, BMW, Burger King, MERCURY

I highly recommend – GitLab's docs and code

GitLab's "Migration Style Guide"
https://docs.gitlab.com/ee/development/migration_style_guide.html

GitLab's migration helpers
https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/database/migration_helpers.rb

Database Lab and Postgres.ai
https://docs.gitlab.com/ee/development/database/database_lab.html

Terminology

DML – database manipulation language
(SELECT / INSERT / UPDATE / DELETE, etc.)

DDL – data definition language

(CREATE …, ALTER …, DROP …)

DB migrations – planned, incremental changes
of DB schema and/or data

DB schema migration & data migration
DB schema evolution, schema versioning
DB change management, and so on

Applying a schema migration to a production database is always a risk

Wikipedia

https://en.wikipedia.org/wiki/Schema_migration

Three big classes of DB migration mistakes

1. Concurrency. Some examples:
   - failed to acquire a lock
   - updating too many rows at once
   - acquired an exclusive lock and left transaction open

2. Correctness. E.g.:
   - unexpected schema deviations
   - schema/app code mismatch
   - unexpected data

3. Miscellaneous. E.g.:
   - reaching `statement_timeout`
   - connection interrupted
   - int4 PKs

[1] Schema mismatch on dev & prod

```
create table t1 (
  id int primary key,
  val text
);


-- dev, test, QA, staging, etc. – OK


-- prod:
ERROR:  relation "t1" already exists
```

[2] Misuse of `IF [NOT] EXISTS`

```
create table if not exists t1 (
  id int primary key,
  val text
);

NOTICE:  relation "t1" already exists, skipping

CREATE TABLE
```

Start using DB schema migration tool

Test changes in CI

- Both DO and UNDO steps are supported (can revert)
- CI: test them all
  - Better: DO, UNDO, and DO again

Test changes in CI

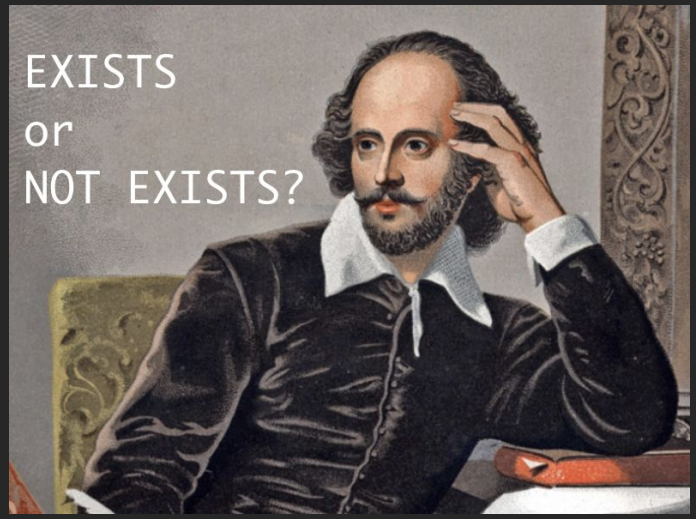
- Both DO and UNDO steps are supported (can revert)
- CI: test them all
  - Better: DO, UNDO, and DO again

Now guess what…

“Thanks” to `IF NOT EXISTS`, we now may leave UNDO empty!

❌ Don’t:

- `IF [NOT] EXIST`



✅ Do:

- test DO-UNDO-DO in CI
- keep schema up to date in all envs
- don’t ignore or work-around errors

“Three Cases Against IF NOT EXISTS / IF EXISTS in Postgres DDL”

https://postgres.ai/blog/20211103-three-cases-against-if-not-exists-and-if-exists-in-postgresql-ddl

Database Testing Landscape

Schema

Data

Static analysis

Test schema

Test data

Application change management

Test DDL

Test DML

"app"

Dynamic analysis

Single-user DB experiments

EXPLAIN

Multi-user DB experiments

Load and stress testing

"infra"

Infrastructure

Microbenchmarks

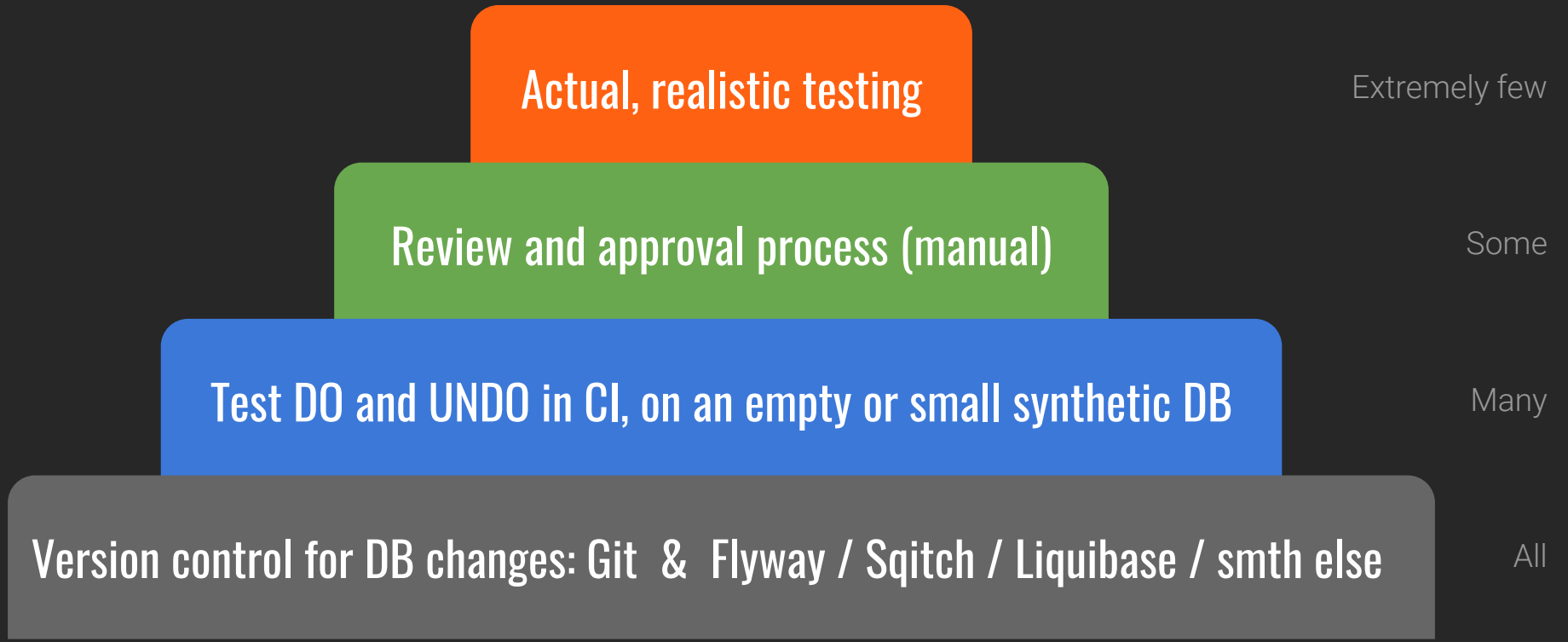
Test backups

Corruption check

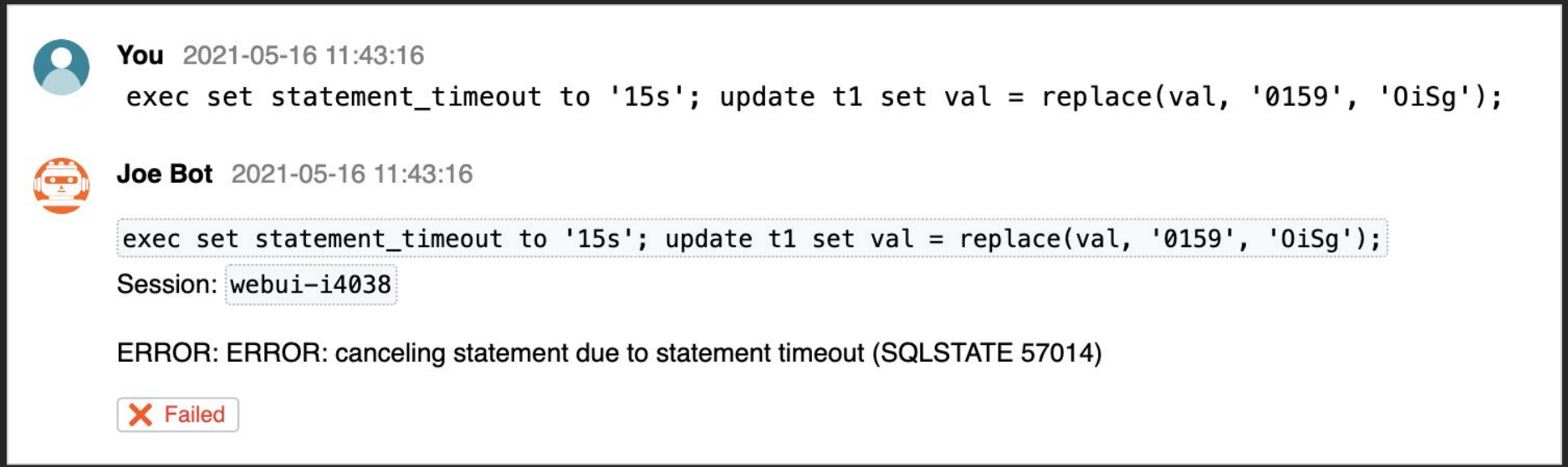
HA tests

...

Reliable database changes – the hierarchy of needs



[3] Failed change due to `statement_timeout`



[4] massive change, unlimited

```
DELETE FROM table1 WHERE … ; -- target 10M rows
```

(or UPDATE)

You 2021-05-16 11:29:58

```
exec create table t1 as
  select id::int, random()::text as val
  from generate_series(1, 10000000) id;

alter table t1 add primary key (id);
```

Joe Bot 2021-05-16 11:29:59

```
exec create table t1 as select id::int, random()::text as val from generate_series(1, 10000000) id; alter table t1
add primary key (id);
```

Session: `webui-i4038`

```
% time      seconds wait_event
------ ------------ ----------------------------
64.82      9.447511 Running
7.92       1.154220 LWLock.WALWriteLock
6.94       1.011216 IO.DataFileExtend
5.69       0.829122 IO.WALWrite
5.27       0.767460 IO.WALSync
2.55       0.370954 IO.DataFileWrite
2.06       0.300581 IO.BufFileWrite
2.04       0.297535 IO.DataFileRead
1.51       0.220348 IO.DataFileImmediateSync
1.21       0.176163 IO.BufFileRead
------ ------------ ----------------------------
100.00    14.575110
```

The query has been executed. Duration: 14.575 s (estimated *for prod: 13.518...116.725 s)*
Estimated timing for production (experimental). How it works

Command

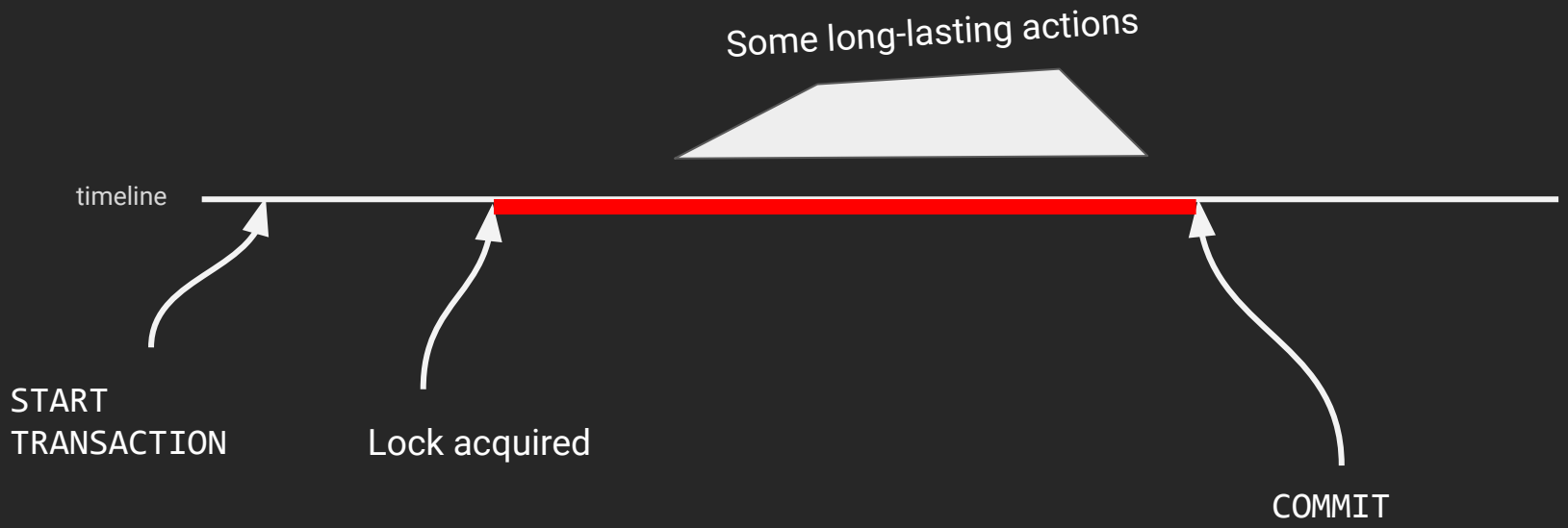
[4] massive change, unlimited

```
test=# explain (buffers, analyze) update t1
       set val = replace(val, '0159', 'OiSg');
```

```
                                                                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------------
 Update on t1  (cost=0.00..189165.00 rows=10000000 width=42) (actual time=76024.507..76024.508 rows=0
loops=1)
   Buffers: shared hit=60154265 read=91606 dirtied=183191 written=198198
   ->  Seq Scan on t1  (cost=0.00..189165.00 rows=10000000 width=42) (actual time=0.367..2227.103
rows=10000000 loops=1)
         Buffers: shared read=64165 written=37703
 Planning:
   Buffers: shared hit=17 read=1 dirtied=1
 Planning Time: 0.497 ms
 Execution Time: 76024.546 ms
(8 rows)

Time: 76030.399 ms (01:16.030)
```

Concurrency issues – holding an acquired lock for long



Locks are released only in the very end of transaction (COMMIT or ROLLBACK)

[5] Acquire an exclusive lock + wait in transaction

```
begin;

alter table t1 add column c123 int8;

-- do something inside or outside of the database

commit;
```

[6] A Transaction with DDL + massive DML

❌ Don't:

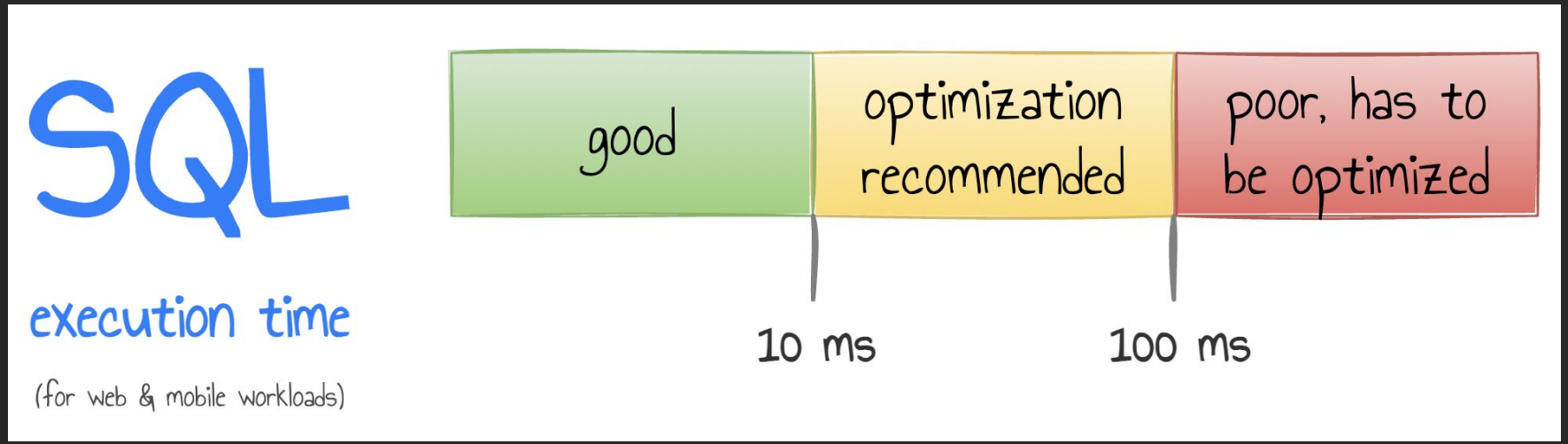
```
begin;

alter table t1 add column c123 int8;

copy ... -- load a lot of data, taking some time

commit;
```

What is “slow” for OLTP?

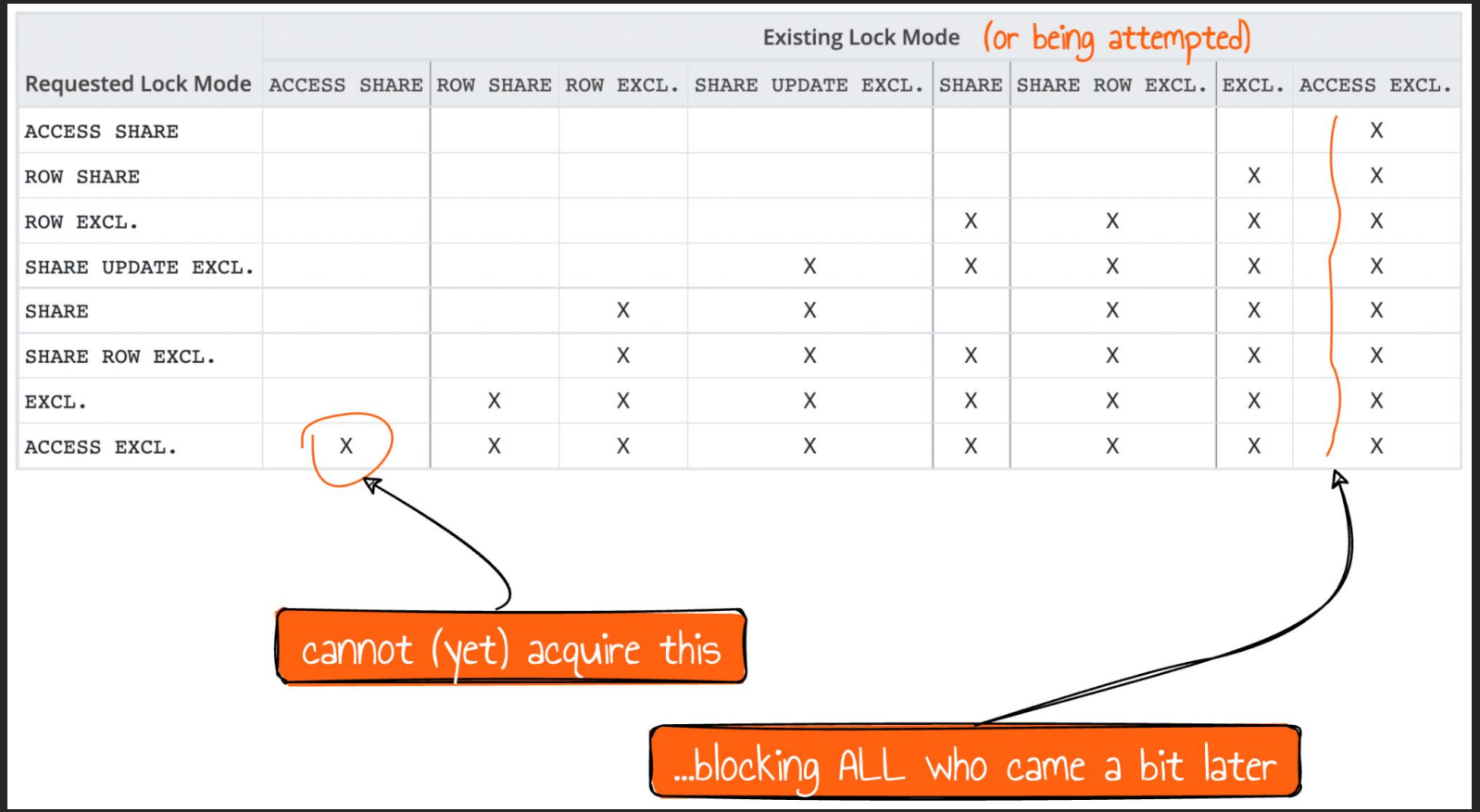


“What is a slow SQL query?” https://postgres.ai/blog/20210909-what-is-a-slow-sql-query

[7] Waiting to acquire an exclusive lock ⇒ blocking others

Can this simple change be dangerous?

```
alter table t1 add column c123;
```



"Zero-downtime Postgres schema migrations need this: lock_timeout and retries"

https://postgres.ai/blog/20210923-zero-downtime-postgres-schema-migrations-lock-timeout-and-retries

The right way – every big/growing project must have it

```
begin;
  set lock_timeout = 50;
  lock table only test in ACCESS EXCLUSIVE MODE;
  set lock_timeout = 0;

  alter table t1 ....;
commit;
```

“Zero-downtime Postgres schema migrations need this: lock_timeout and retries”

https://postgres.ai/blog/20210923-zero-downtime-postgres-schema-migrations-lock-timeout-and-retries

[8] Create an FK

❌ Don't:

```
alter table t1 add constraint ...
  foreign key (...) references t2 (...);
```

✅ Do:

```
-- step 1 (fast)
alter table t1 add constraint fk_t1_123
  foreign key (...) references t2 (...) not valid;

-- step 2 (long) – later, in a separate transaction
alter table t1 validate constraint fk_t1_123;
```

[9] Drop an FK

```
alter table t1 drop constraint ...;
```

– should be deployed with `lock_timeout` and retry logic

[10] Add a CHECK constraint

❌ Don't:

```
alter table t1 add constraint c_t1_123
  check (c123 is not null);
```

✅ Do:

```
-- step 1

alter table t1 add constraint c_t1_123
  check (c123 is not null) not valid;


-- step 2

alter table t1 validate constraint c_t1_123;
```

[11] Add NOT NULL

❌ Don't:

```
alter table t1 alter column c123 set not null;
```

✅ Do:

Trick 1 (PG11+): do with with DEFAULT when creating a new column with some DEFAULT, then backfill, then drop default

Trick 2 (PG12+): add a CHECK with IS NOT NULL (in 2 steps), then SET NOT NULL, then drop the CHECK constraint

[12] Change column’s data type

```
alter table t1 alter column id type int8;
```

Most common approaches:

- Option 1: “new column”
- Option 2: “new table”

None of them are easy.

[13] Create an index

❌ Don't:

```
create index i_t1_c123
  on t1 using btree (c123);
```

✅ Do:

```
create index concurrently i_t1_c123
  on t1 using btree (c123);
```

[14] Drop an index

❌ Don't:

```
drop index i_t1_c123;
```

✅ Do:

```
drop index concurrently i_t1_c123;
```

[15] Renaming objects

```
alter table t1 rename to ..;

alter table t1 rename column .. to …;
```

(and so on)

Application code may not expect it:

- app code not yet deployed
- app code deployed before commit

[16] Add a column with DEFAULT

```
alter table t1 add column c1 int8 default -1;
```

No worries unless you’re on Postgres 10 or older
(otherwise– some headache, if the table has a lot of rows)

Good article: https://brandur.org/postgres-default

[17] Failed `CREATE INDEX CONCURRENTLY`

CREATE INDEX CONCURRENTLY may fail

If it does, an invalid index is present – must be cleaned up first:

```
test=# select indexrelid, indexrelid::regclass as indexname, indisvalid
from pg_index
where not indisvalid and indexrelid::regclass::text = 'mytable_title_idx';

 indexrelid |     indexname     | indisvalid
------------+-------------------+------------
      26401 | mytable_title_idx | f
(1 row)
```

[18] Create a table with int4 PK

❌ Don't:

```
create table t1 (
  id int4 primary key,
  ts timestamptz
);
```

✅ Do:

```
create table t1 (
  id int8 primary key,
  ts timestamptz
);
```

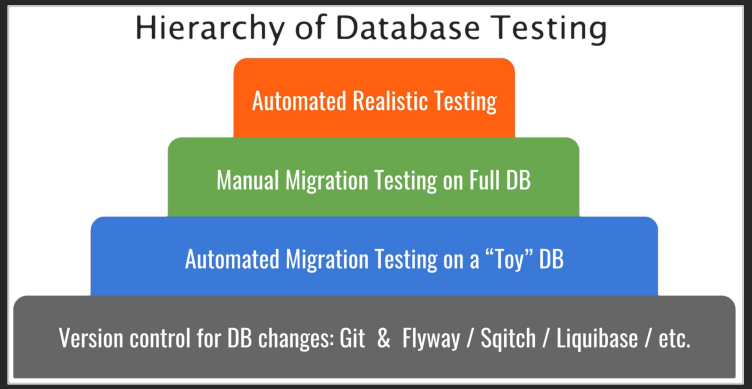
[19] Limiting text too much

[20] REFRESH MATERIALIZED VIEW (w/o CONCURRENTLY)

[21] Ignoring vacuum and bloat issues

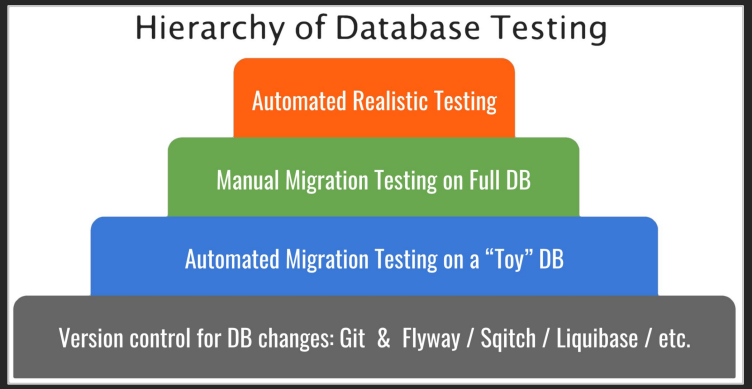
Database Migration Testing with Database Lab

- Realistic migration testing is hard
- No testing = unexpected problems

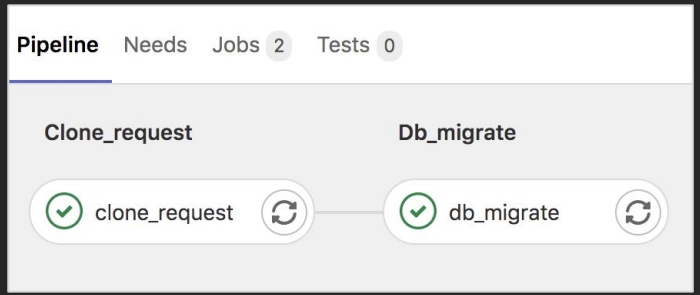


Database Migration Testing with Database Lab

- Realistic migration testing is hard
- No testing = unexpected problems



- Database Lab makes realistic testing *easy*



Summary

1. **Test DB changes in CI (*with data*)**

   Consider:  with DB Migration Checker

2. **Create automation for *each* case**

   A good example: GitLab's `migration_helpers.rb`

3. **Share your ideas with us!**

   👉 Slack.Postgres.ai @Database_Lab

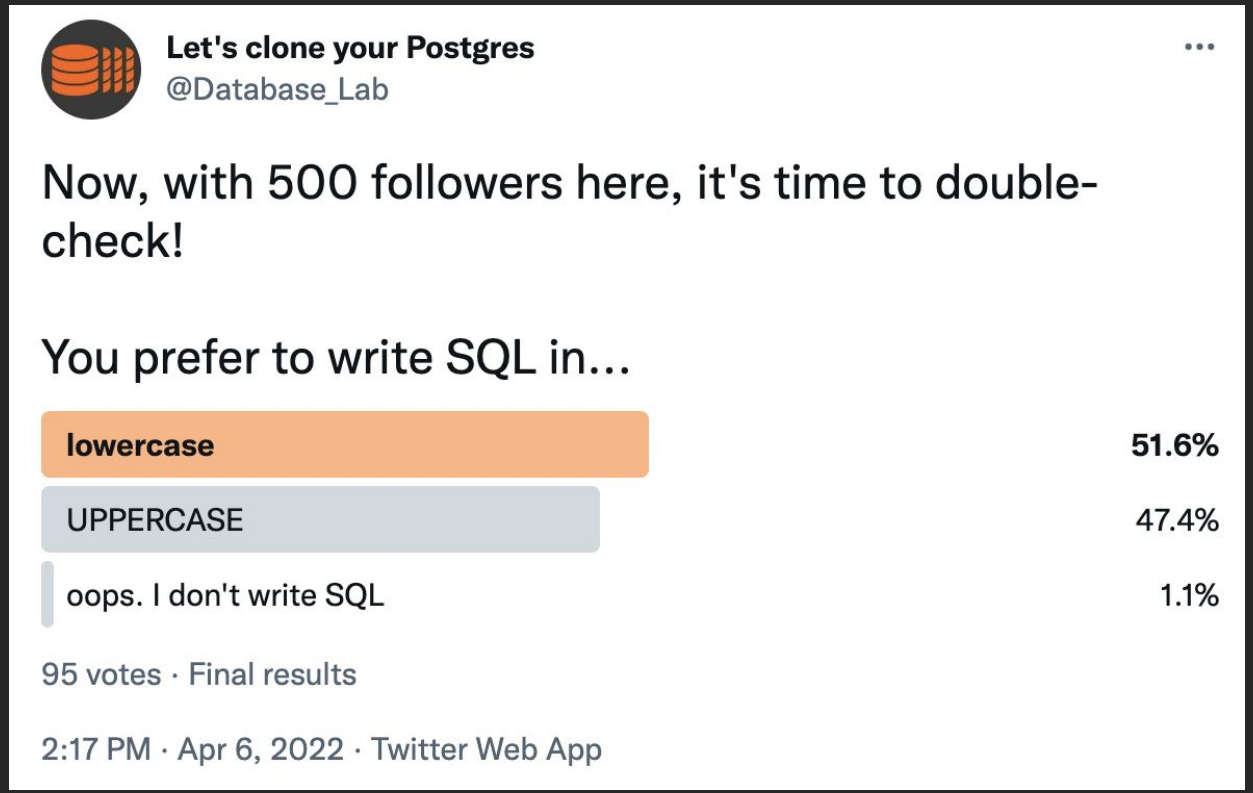
Thank you!

Slack (EN): **Slack.Postgres.ai**

Twitter: **@Database_Lab**


TO BE
CONTINUED...

lowercase for sql queries is ok



Let's clone your Postgres
@Database_Lab

Now, with 500 followers here, it's time to double-check!

You prefer to write SQL in...

lowercase	**51.6%**
UPPERCASE	47.4%
oops. I don't write SQL	1.1%

95 votes · Final results

2:17 PM · Apr 6, 2022 · Twitter Web App

Some examples of failures due to lack of testing

- Incompatible changes – production has different DB schema than dev & test
- Cannot deploy – hitting `statement_timeout` – too heavy operations

- During deployment, we've got a failover
- Deployment lasted 10 minutes, the app was very slow (or even down)

- Two weeks after deployment, we realize that the high bloat growth we have now has been introduced by that deployment
- Deployment succeeded, but then we have started to see errors

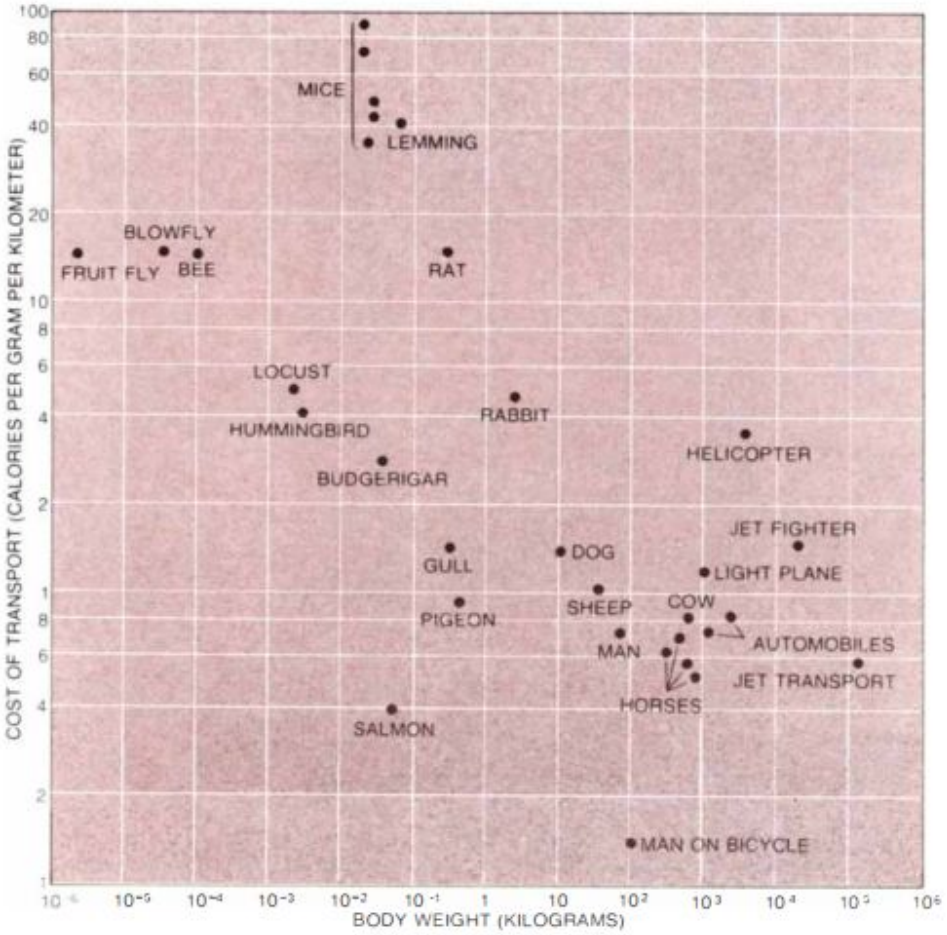
We need better tools

SCIENTIFIC AMERICAN

BICYCLE TECHNOLOGY

ONE DOLLAR

March 1973



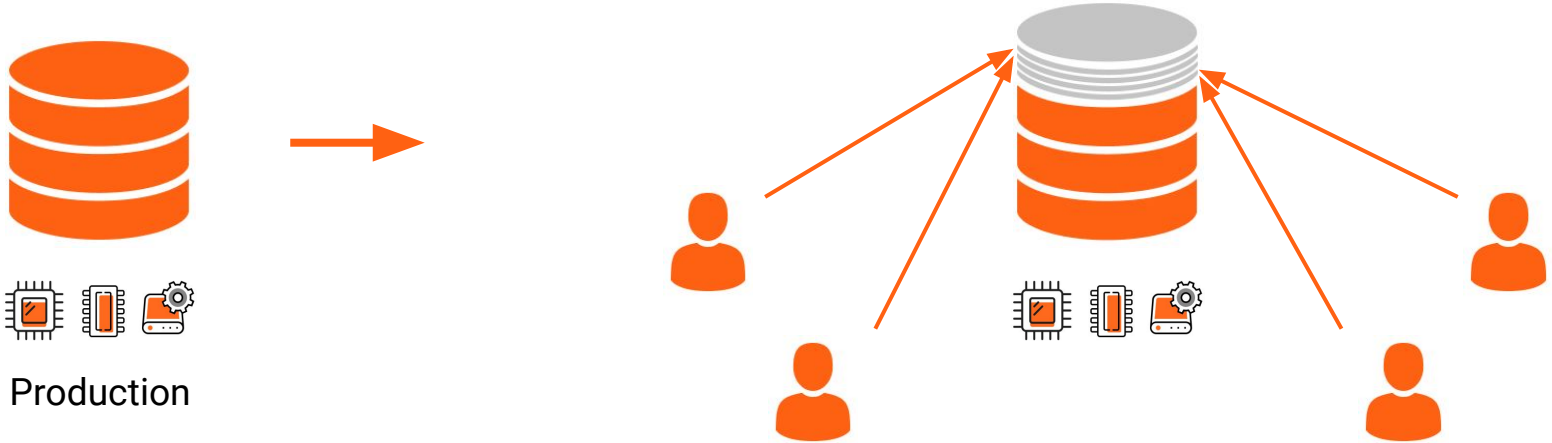
Steve Jobs (1980)

1) We, humans, are great tool-makers.
   We amplify human abilities.

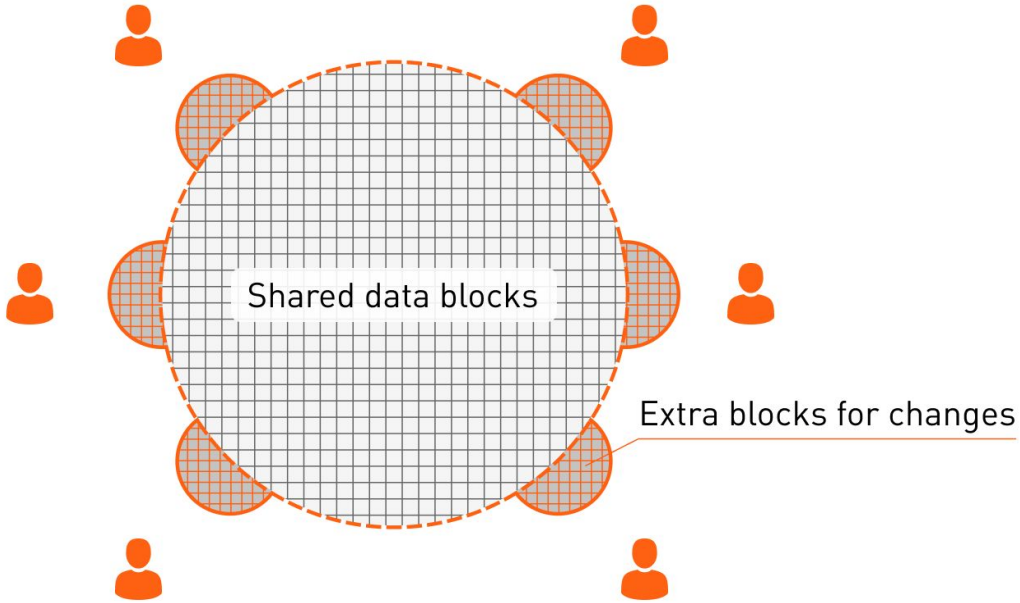
2) Something special happens
   when you have 1 computer and 1 person.

   It's very different that having 1 computer and 10 persons.

“1 database copy – 10 persons”



“1 database copy – 1 person”



Thick copy of production (any size)

Thin clone (size starts from 1 MB, depends on changes)

Database Lab – Open-core model

The Database Lab Engine (DLE)

Open-source (AGPLv3)

- Thin cloning – API & CLI
- Automated provisioning and data refresh
- Data transformation, anonymization
- Supports managed Postgres (AWS RDS, etc.)

https://gitlab.com/postgres-ai/database-lab

The Platform (SaaS)

Proprietary (freemium)

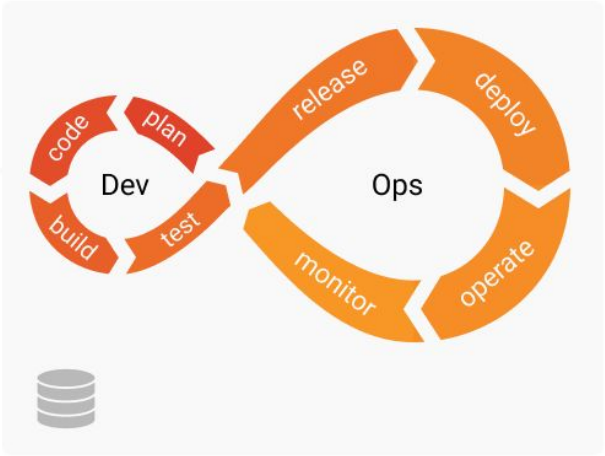
- Web console – GUI
- Access control, audit
- History, visualization
- Support

https://postgres.ai/

^^ use these links to start using it for your databases ^^

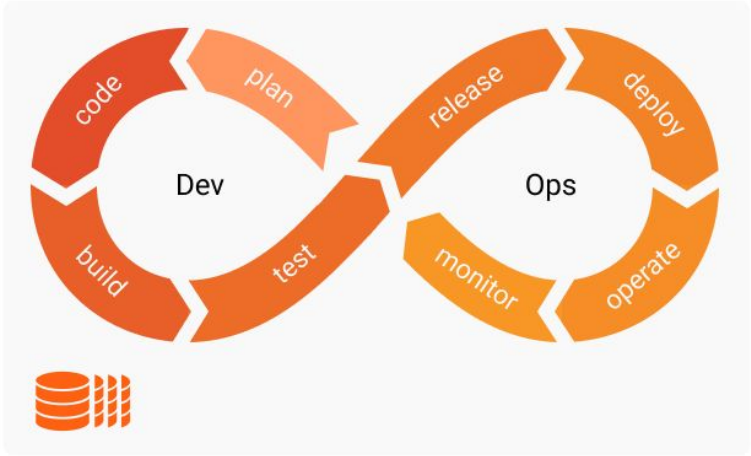
Database Lab unlocks "Shift-left testing"

Development bottlenecks
(with standard staging DB)



- ✕ Bugs: difficult to reproduce, easy to miss
- ✕ Not 100% of changes are well-verified
- ✕ SQL optimization is hard
- ✕ Each non-prod big DB costs a lot
- ✕ Non-prod DB refresh takes hours, days, weeks

Frictionless development
(with Database Lab)



- ✓ Bugs: easy to reproduce, and fix early
- ✓ 100% of changes are well-verified
- ✓ SQL optimization can be done by anyone
- ✓ Non-prod DB refresh takes seconds
- ✓ Extra non-prod DBs doesn't cost a penny

Database experiments on thin clones – yes and no

Yes

- Check execution plan – Joe bot 

  - EXPLAIN w/o execution
  - EXPLAIN (ANALYZE, BUFFERS)
    - (timing is different; structure and buffer numbers – the same)
- Check DDL
  - index ideas (Joe bot)
  - auto-check DB migrations (CI Observer)
- Heavy, long queries: analytics, dump/restore
  - No penalties!
    (think hot_standby_feedback, locks, CPU)

No

- Load testing
- Regular HA/DR goals
  - backups
    - (but useful to check WAL stream, recover records by mistake)
  - hot standby
    - (but useful to offload very long-running SELECTs)

DB migration testing – “stateful tests in CI”

What we want from testing of DB changes:

- Ensure the change is valid
- It will be executed in appropriate time
- It won’t put the system down

…and:

- What to expect? (New objects, size change, duration, etc.)

Perfect Lab for database experiments

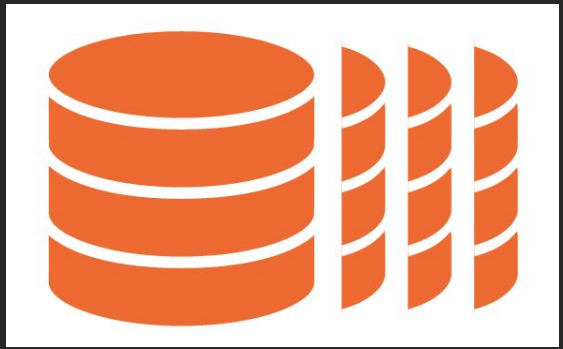
- Realistic conditions – as similar to production as possible
  - The same schema, data, environment as on production
  - Very similar background workload
- Full automation
- “Memory” (store, share details)
- Low iteration overhead (time & money)
- Everyone can test independently

allowed to fail → allowed to learn

Database experiments with Database Lab today (2021)

- Realistic conditions – as similar to production as possible
  - The same schema, data, environment as on production
  - ~~Very similar background workload~~
- Fine automation
- “Memory” (store, share details)
- Low iteration overhead (time & money)
- Everyone can test independently

  able to fail → able to learn



Why Database Lab was created

- Containers, OverlayFS (file-level CoW)

  CI: `docker pull ... && docker run ...`

    – OK only for tiny (< a few GiB) databases

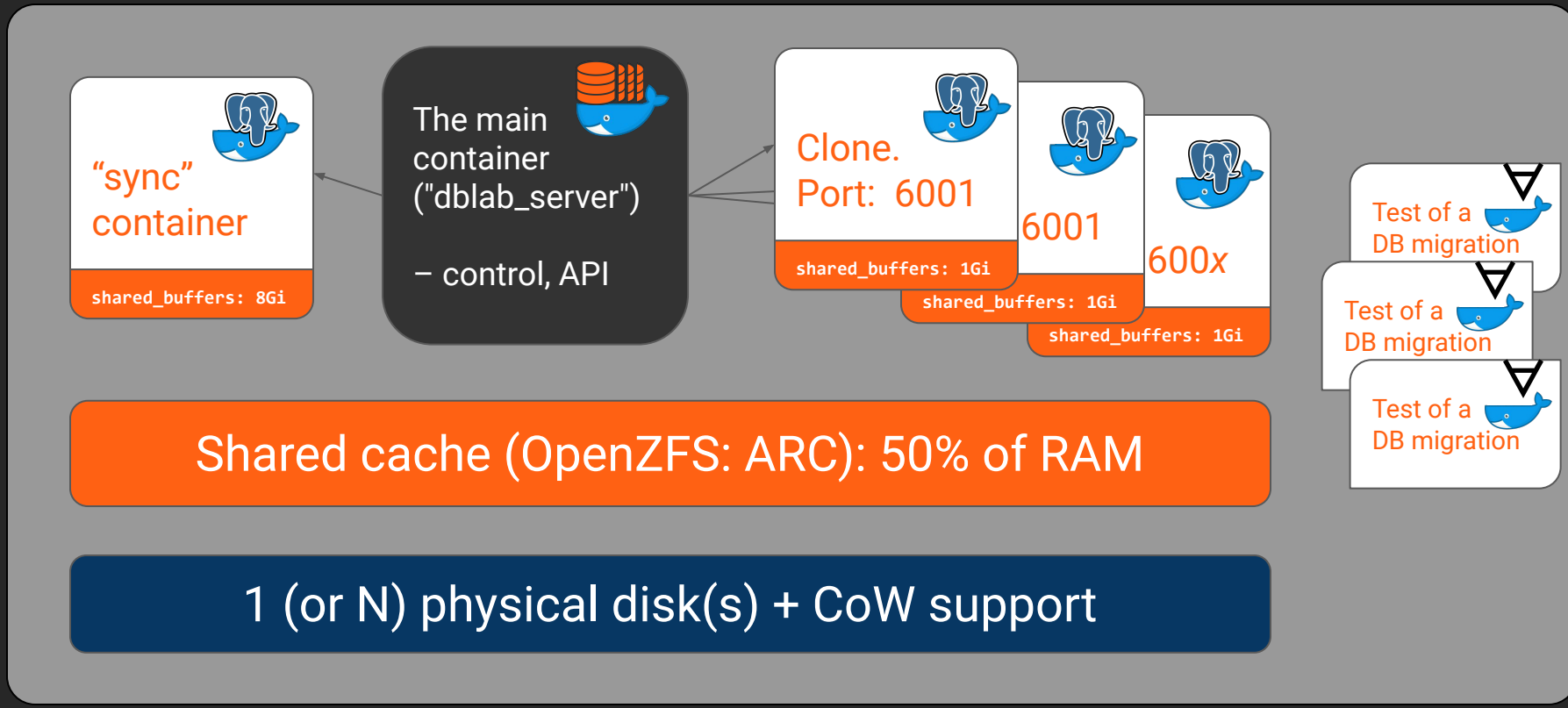
- Existing solutions: Oracle Snap Clones, Delphix, Actifio, etc.
  $$$$, not open

    – OK only for very large enterprises

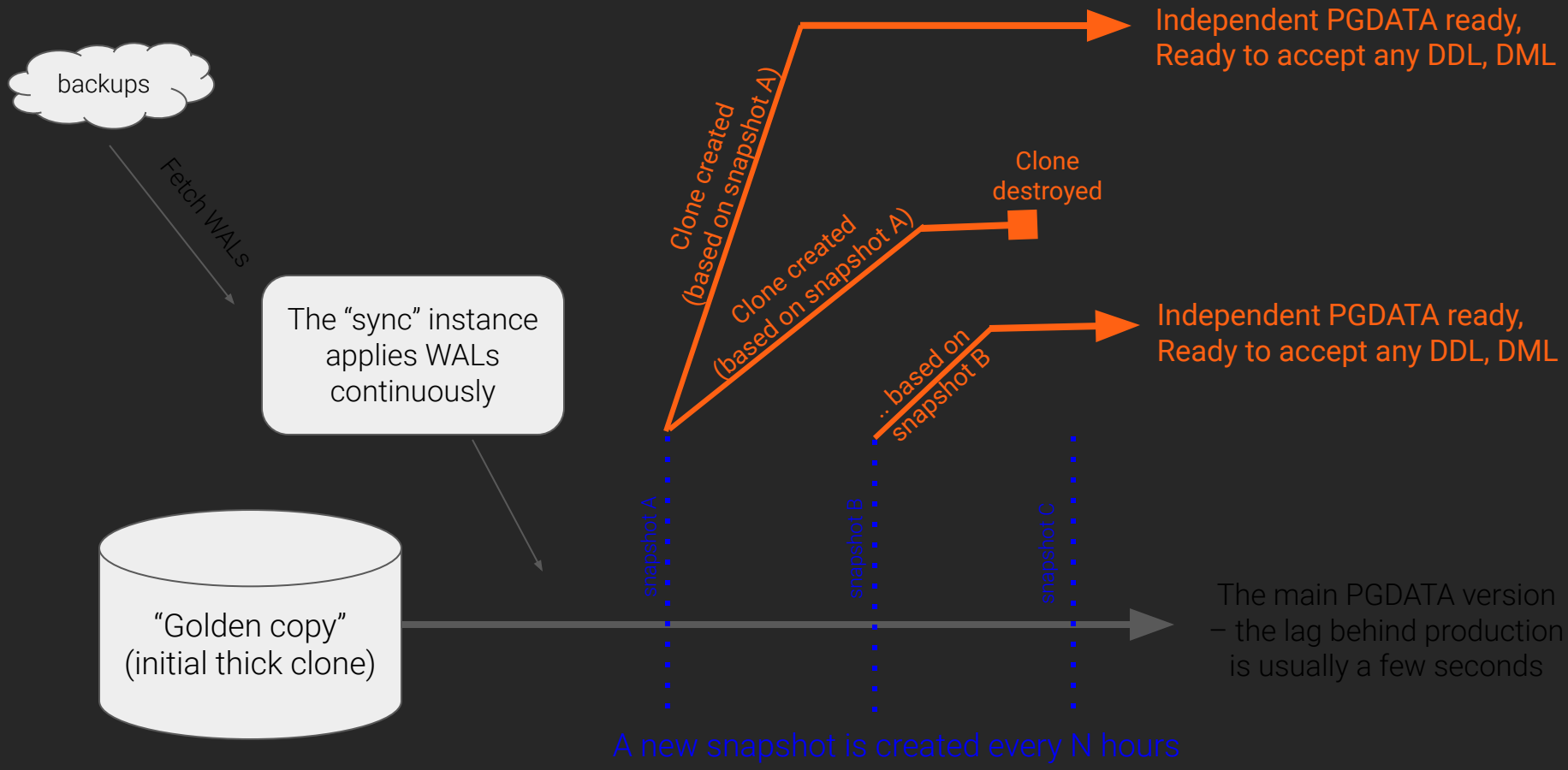
Companies that do need it today

- 10+ engineers
- Multiple backend teams (or plans to split soon)
- Microservices (or plans to move to them)
- 100+ GiB databases
- Frequent releases

Inside the Database Lab Engine 2.x



DLE – the data flow (physical mode)



How snapshots are created (ZFS version)

- Create a “pre” ZFS snapshot (R/O)
- Create a “pre” ZFS clone (R/W)
- DLE launches a temporary “promote” container
  - If needed, performs “preprocessing” steps (bash)
  - Uses “pre” clone to run Postgres and promote it to primary state
  - If needed, performs “preprocessing” SQL queries
  - Performs a clean shutdown of Postgres
- Create a final ZFS snapshot that will be used for cloning

Major topics of automated (CI) testing on thin clones

- Security

  https://postgres.ai/docs/platform/security

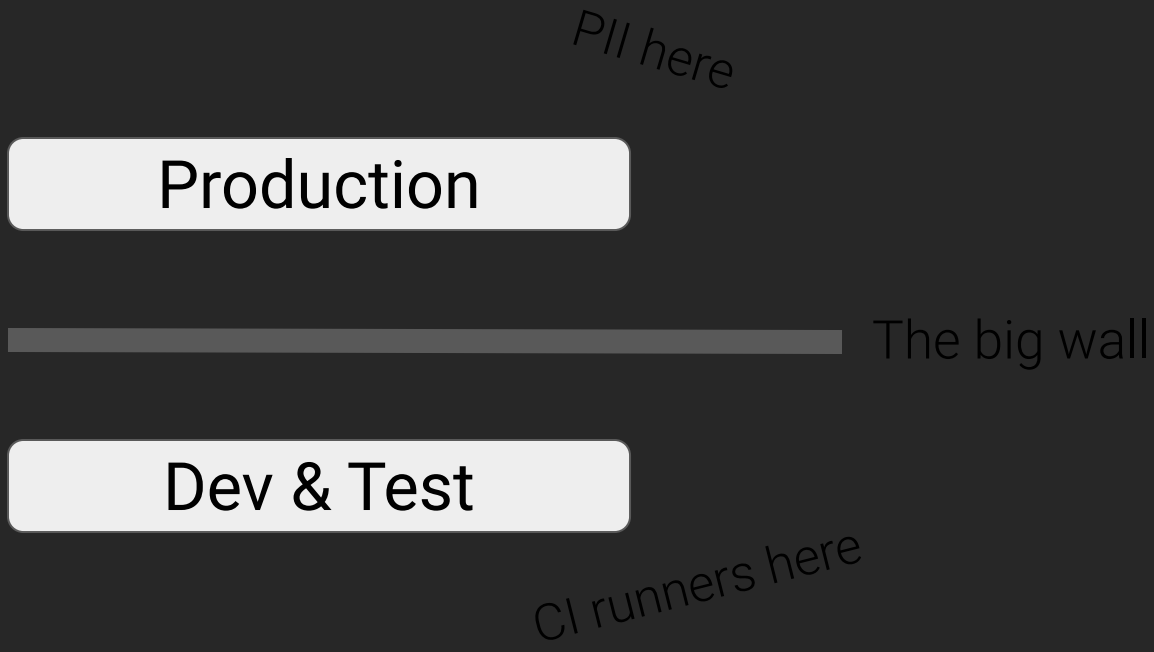
- Capturing dangerous locks

  CI Observer:
https://postgres.ai/docs/database-lab/cli-reference#subcommand-start-observation

- Forecast production timing

  Timing estimator: https://postgres.ai/docs/database-lab/timing-estimator

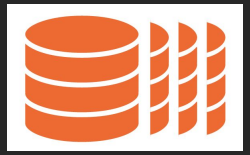
Making the process secure: where to place the DLE?



Where to place the DLE? Current approach

PII here

Production



High-level API calls

The big wall

Dev & Test

CI runners here

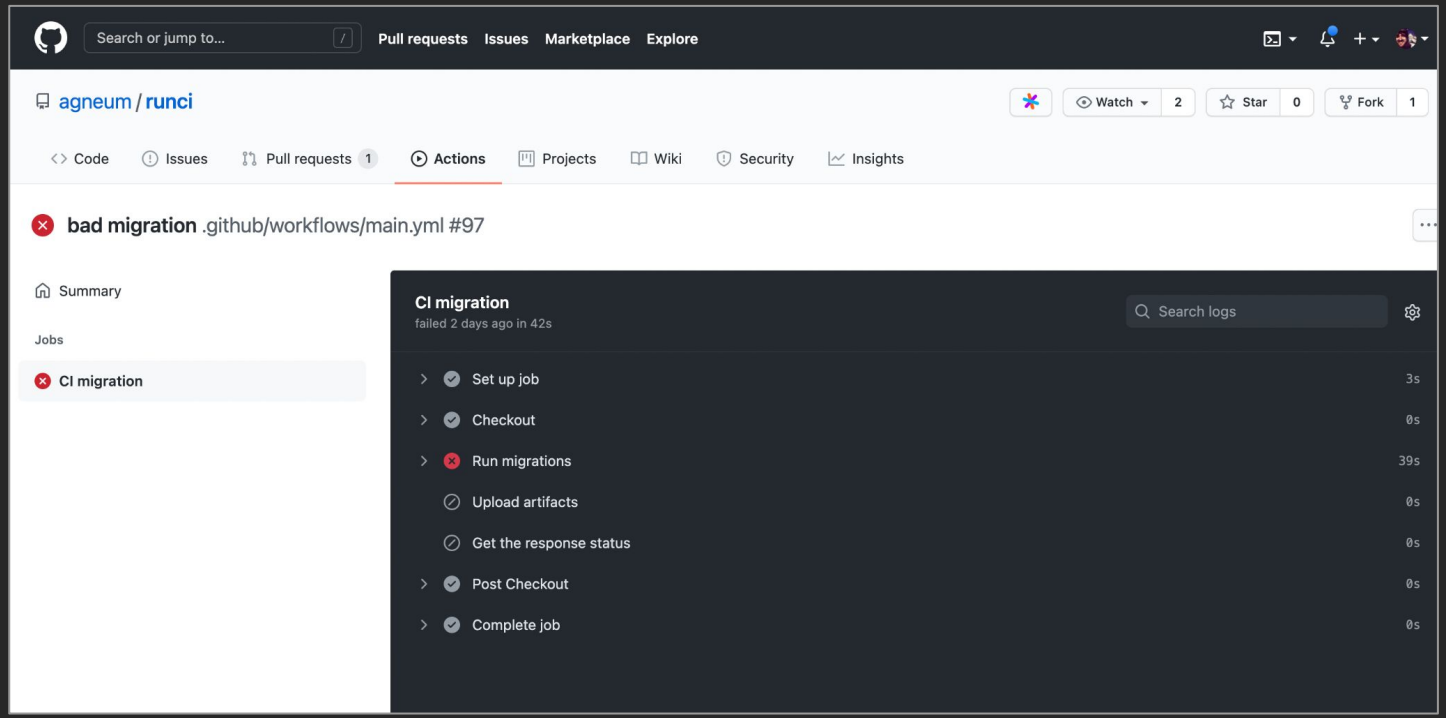
GitHub

GitLab

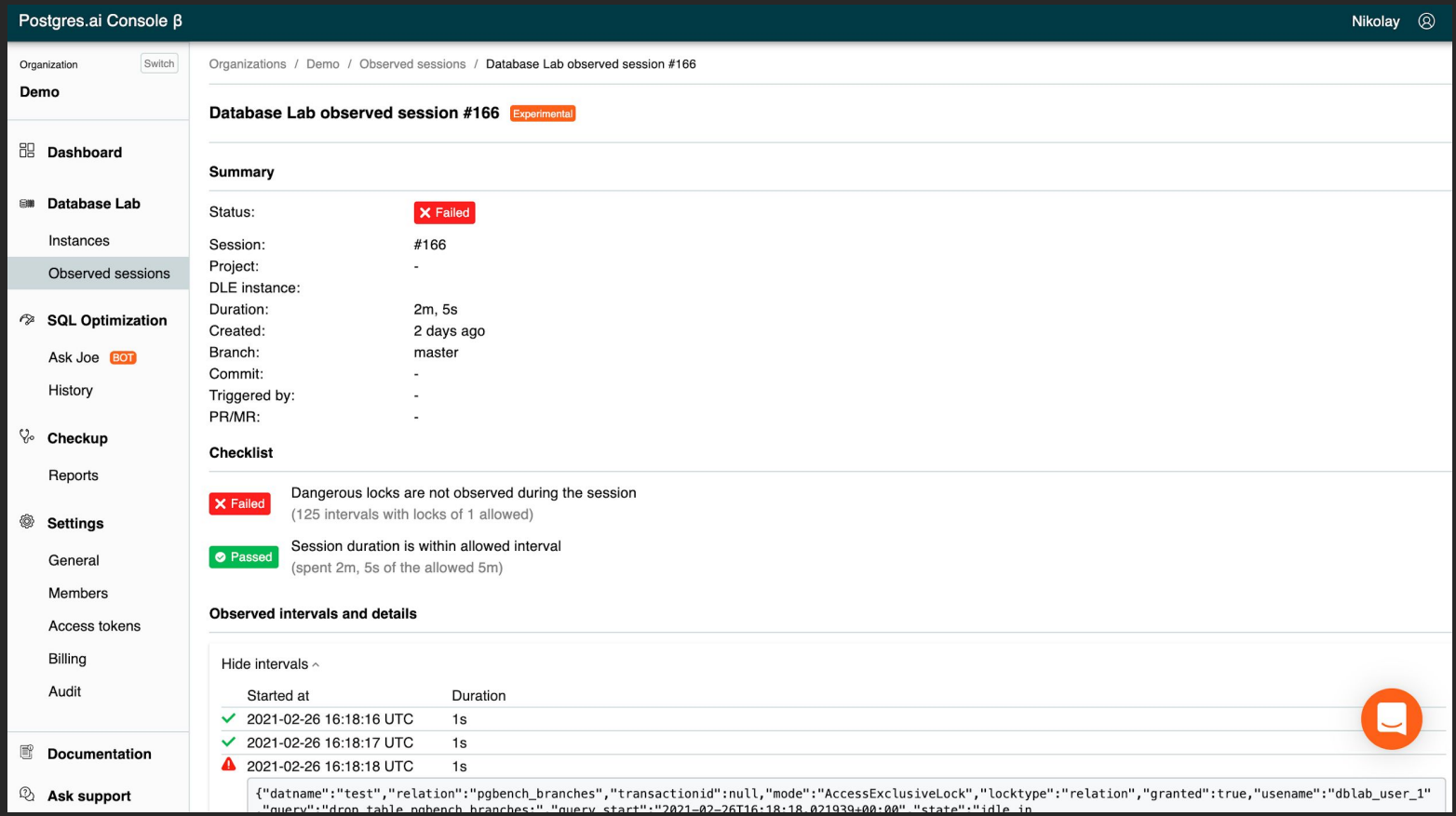
Jenkins

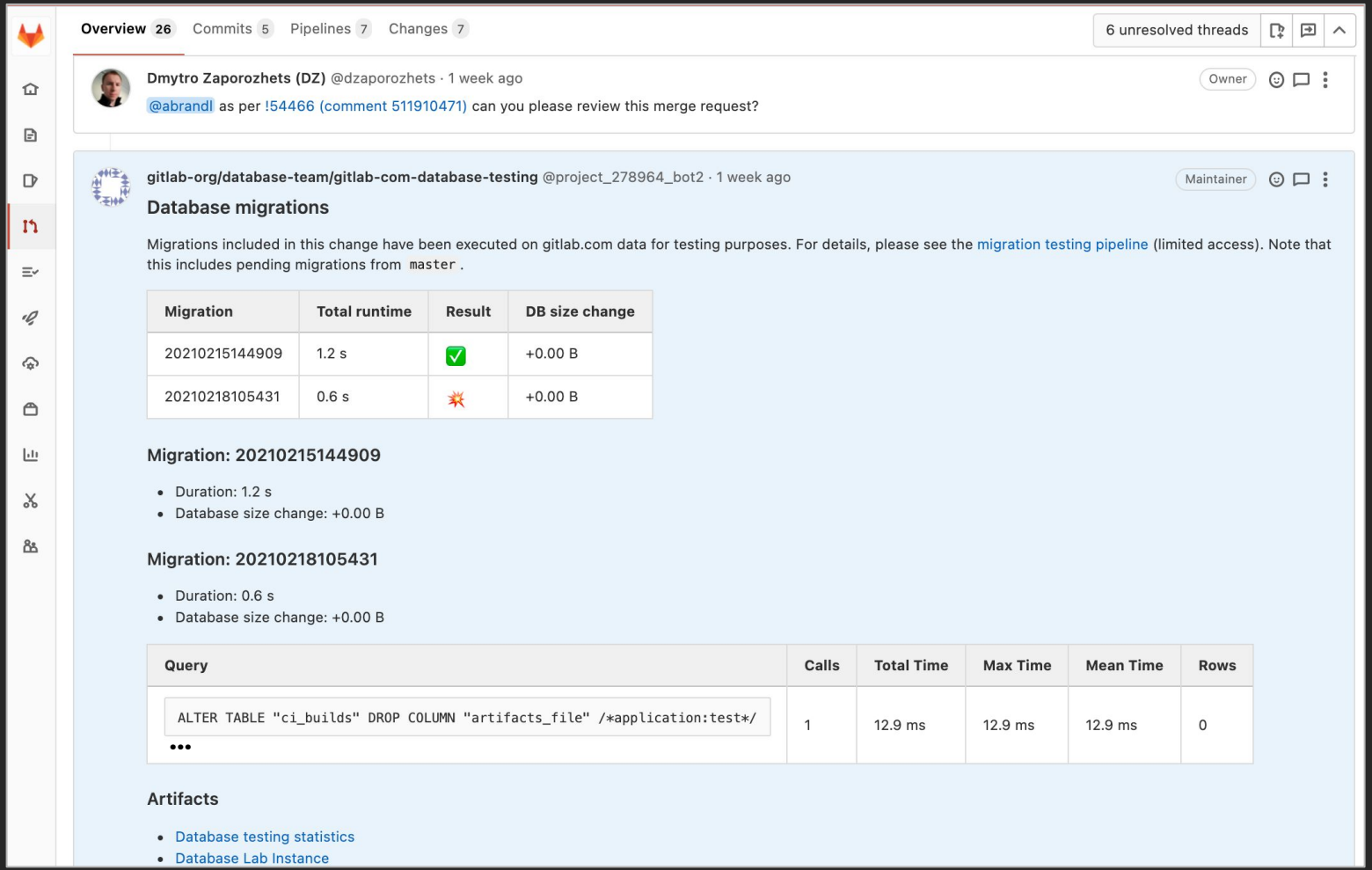
How it looks like: CI part

Example: GitHub Actions:
https://github.com/agneum/runci/runs/2519607920?check_suite_focus=true



More about dangerous lock detection





Overview 26 Commits 5 Pipelines 7 Changes 7

6 unresolved threads

Dmytro Zaporozhets (DZ) @dzaporozhets · 1 week ago

Owner

@abrandl as per !54466 (comment 511910471) can you please review this merge request?

gitlab-org/database-team/gitlab-com-database-testing @project_278964_bot2 · 1 week ago

Maintainer

Database migrations

Migrations included in this change have been executed on gitlab.com data for testing purposes. For details, please see the migration testing pipeline (limited access). Note that this includes pending migrations from `master`.

Migration	Total runtime	Result	DB size change
20210215144909	1.2 s	✅	+0.00 B
20210218105431	0.6 s	💥	+0.00 B

Migration: 20210215144909

- Duration: 1.2 s
- Database size change: +0.00 B

Migration: 20210218105431

- Duration: 0.6 s
- Database size change: +0.00 B

Query	Calls	Total Time	Max Time	Mean Time	Rows
`ALTER TABLE "ci_builds" DROP COLUMN "artifacts_file" /*application:test*/` •••	1	12.9 ms	12.9 ms	12.9 ms	0

Artifacts

- Database testing statistics
- Database Lab Instance

February 19, 2021 – https://gitlab.com/gitlab-org/gitlab/-/merge_requests/54564#note_512678910

Example: GitLab.com, testing database changes using Database Lab

- Full automation
- GitLab CI/CD pipelines securely work with Database Lab
- Database Lab clones ~10 TiB database in ~10 seconds

Read their blueprint:

https://docs.gitlab.com/ee/architecture/blueprints/database_testing/

More about production timing estimation

Experimental, WIP: https://postgres.ai/docs/database-lab/timing-estimator

```
estimator:
    readRatio: 1
    writeRatio: 1
    profilingInterval: 20ms
    sampleThreshold: 100
```

```
LOG: Profiling process 63 with 10ms sampling
% time      seconds wait_event
------ ------------ ------------------------------
57.30     17.715111 IO.DataFileRead
25.53      7.893916 Running
3.55       1.097738 IO.DataFileExtend
2.55       0.787341 LWLock.WALWriteLock
2.25       0.696663 IO.BufFileRead
2.14       0.662457 IO.BufFileWrite
2.12       0.654081 IO.WALInitWrite
1.62       0.499461 IO.WALInitSync
1.09       0.335660 IO.WALWrite
0.98       0.301637 IO.DataFileImmediateSync
0.81       0.250249 IO.WALSync
0.07       0.020805 LWLock.WALBufMappingLock
------ ------------ ------------------------------
100.00    30.915119
```

Summary:

```
Time: 3.148 s
  - planning: 0.168 ms
  - execution: 3.147 s (estimated* for prod: 2.465...2.693 s)
    - I/O read: 627.267 ms
    - I/O write: 3.644 ms

Shared buffers:
  - hits: 1016393 (~7.80 GiB) from the buffer pool
  - reads: 16395 (~128.10 MiB) from the OS file cache, including disk I/O
  - dirtied: 16395 (~128.10 MiB)
  - writes: 280 (~2.20 MiB)
```

Summary – available in PR/MR and visible to whole team

- When, who, status
- Duration (in the Lab + estimated for production)
- Size changes, new objects
- Dangerous locks
- Error stats
- Transaction stats
- Query analysis summary
- Tuple stats
- WAL generated, checkpointer/bgwriter stats
- Temp files stats

Example (WIP): https://gitlab.com/postgres-ai/database-lab/-/snippets/2083427

More artifacts, details – restricted access

- System monitoring (resources utilization)
- pg_stat_*
- pg_stat_statements, pg_stat_kcache
- logerrors
- Postgres log
- pgBadger (html, json)
- wait event sampling
- perf tracing, flamegraphs; or eBPF
- Estimated production timing

https://gitlab.com/postgres-ai/database-lab/-/issues/226

Database Lab Roadmap

https://postgres.ai/docs/roadmap

- Lower the entry bar
  - Simplify installation
  - Simplify the use
  - Easy to integrate
  - *** **** * *******

Where to start

Postgres.ai/docs/

Slack.Postgres.ai