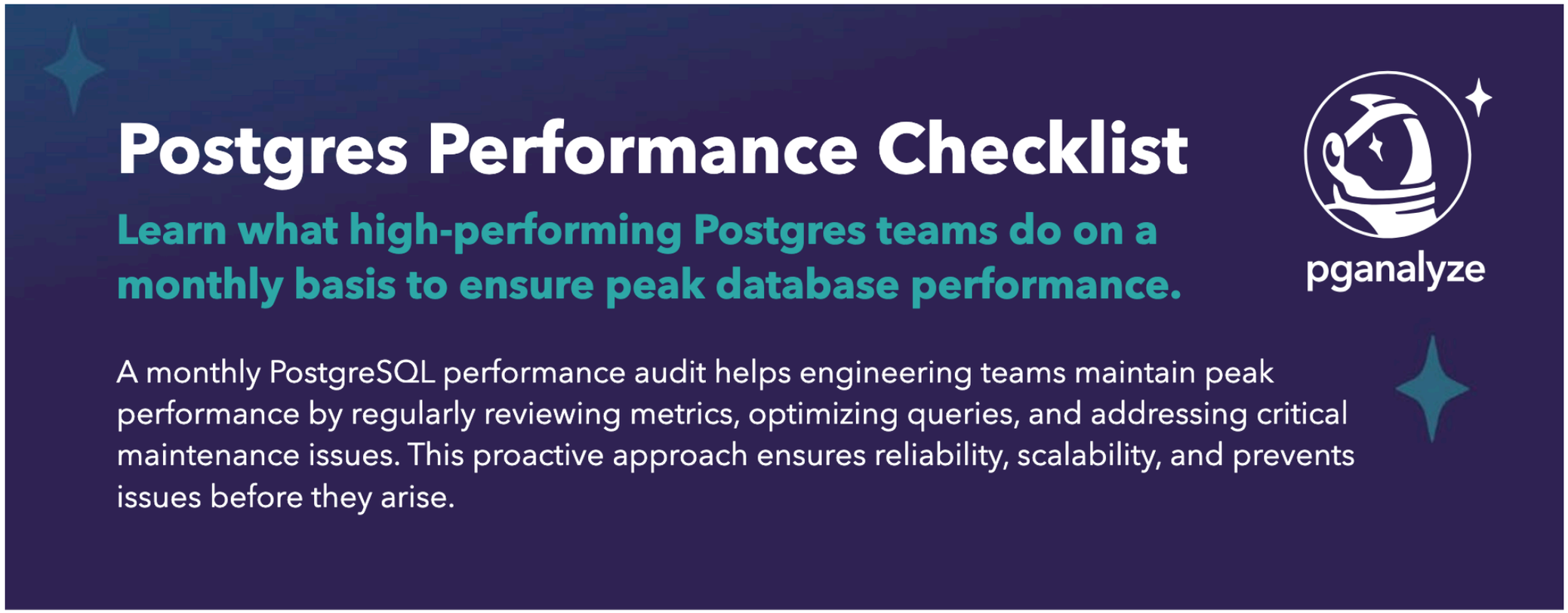
Proactive Postgres Practices to Prevent Performance Bottlenecks

lukas@pganalyze.com

This Webinar is inspired by:

Link



DATABASE OVERVIEW

- ☐ **Review Schema Changes:** Identify and document any significant schema changes made in the last month, ensuring the changes haven't affected performance.

SYSTEM PERFORMANCE

- ☐ **Review System Statistics:** Monitor CPU, I/O, and disk space utilization over the last month to detect performance bottlenecks.
- ☐ **Review Configuration Settings:** Check settings related to Write-Ahead Logging (WAL) and

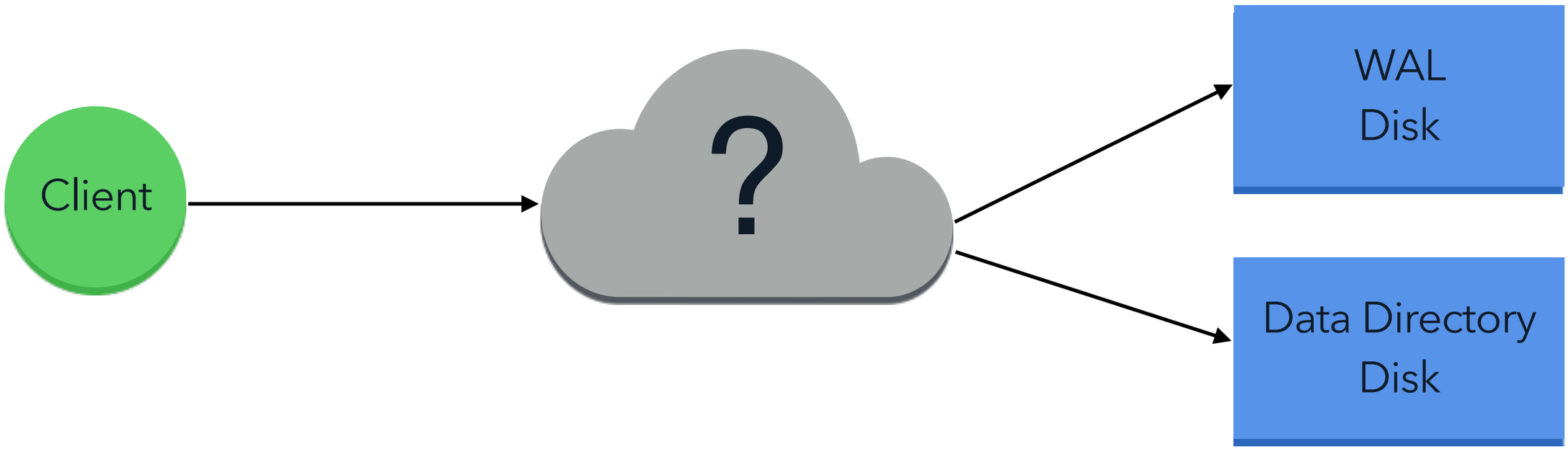
1. A Mental Model for Postgres I/O
2. Tuning Postgres Internal Bottlenecks
3. Table and Index Maintenance
4. Optimizing Queries, Connections & Wait Events

pganalyze

A Mental Model for Postgres I/O

Write I/O

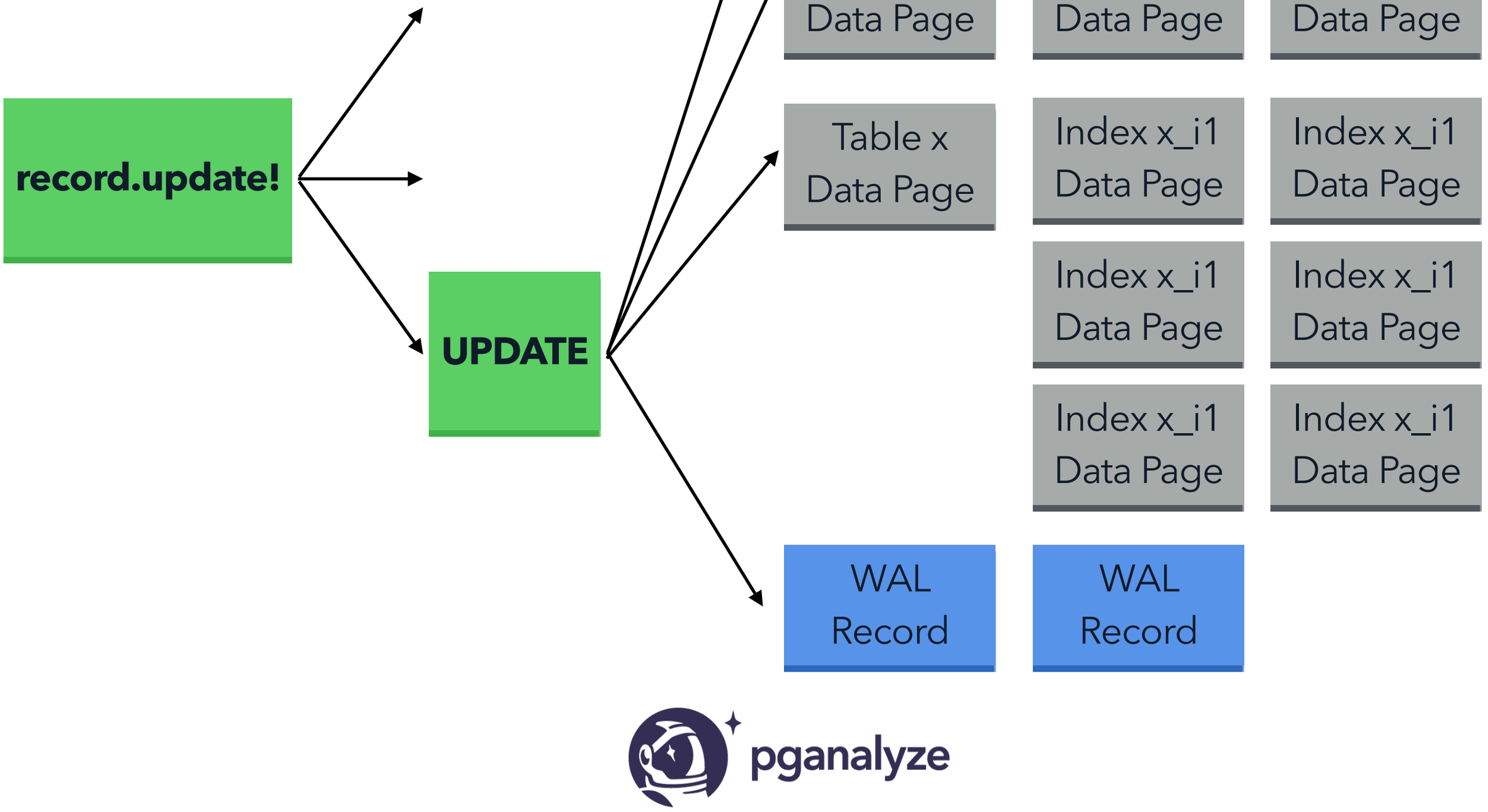
Change that needs to be written to disk



🔧 Change buffer sizes, flush intervals, reduce index writes, etc.

pganalyze

Write I/O



pganalyze

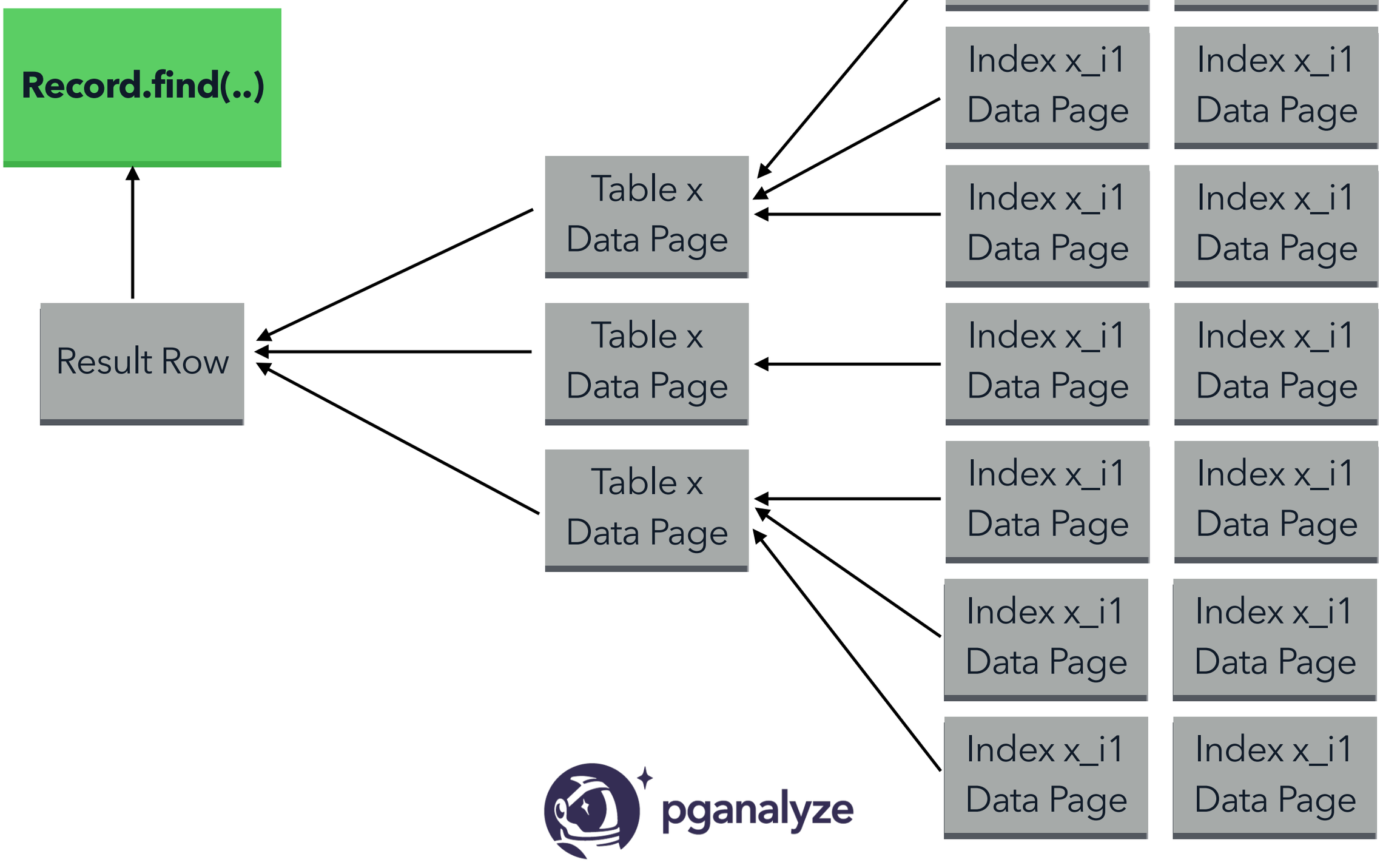
Read I/O

Provide a certain query result to the client

🔧 Improve parallelism, improve indexes, improve caching

pganalyze

Read I/O

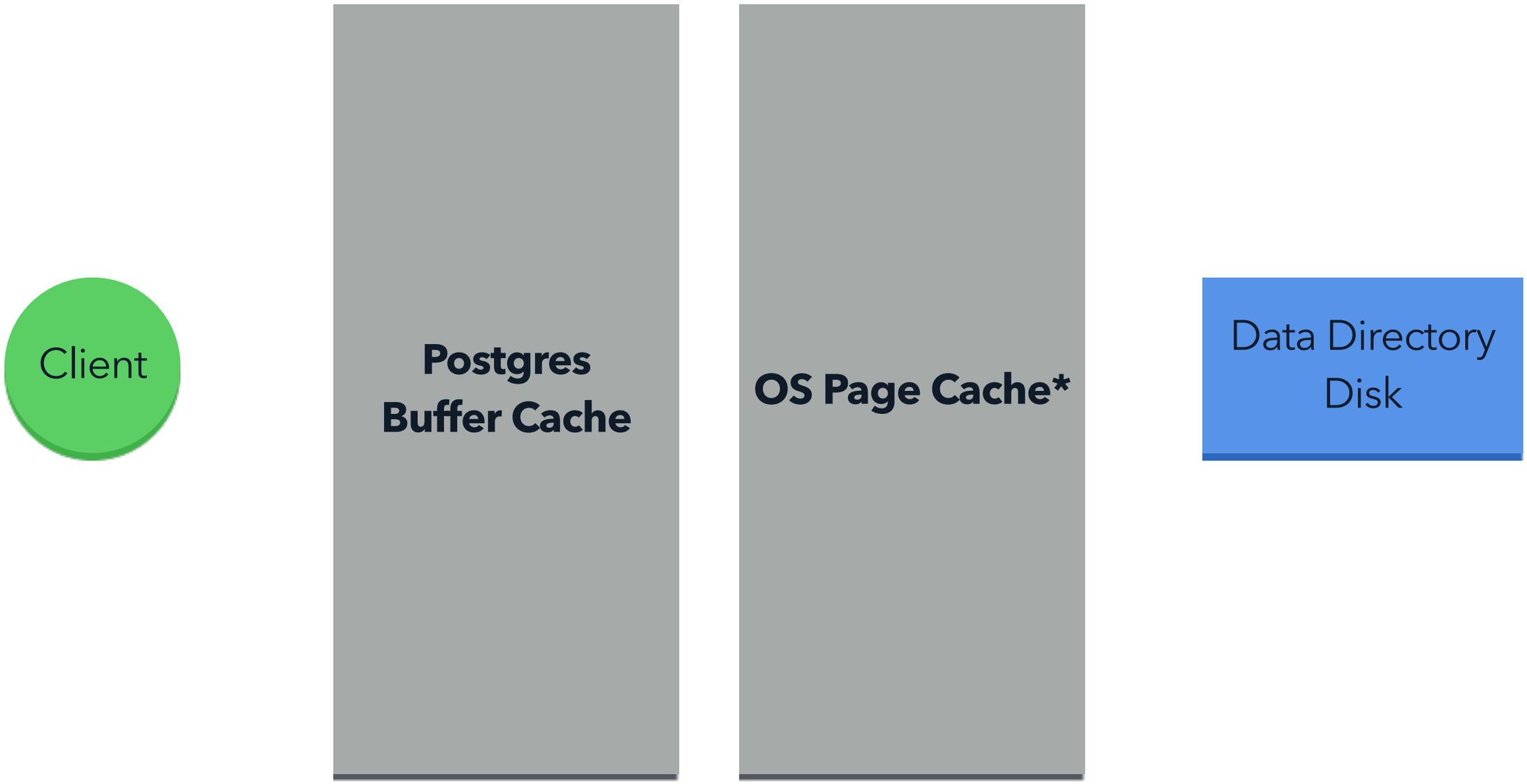

Record.find(..)
Result Row
Table x
Data Page
Table x
Data Page
Table x
Data Page
Index x_i1
Data Page
Index x_i1
Data Page
Index x_i1
Data Page
Index x_i1
Data Page
Index x_i1
Data Page
Index x_i1
Data Page
Index x_i1
Data Page
Index x_i1
Data Page
Index x_i1
Data Page
Index x_i1
Data Page
Index x_i1
Data Page
Index x_i1
Data Page
pganalyze

The best I/O operation is

the one that didn't happen.

(e.g. because we avoided table bloat, thus not even having a particular data page)

The Two Caches That Matter

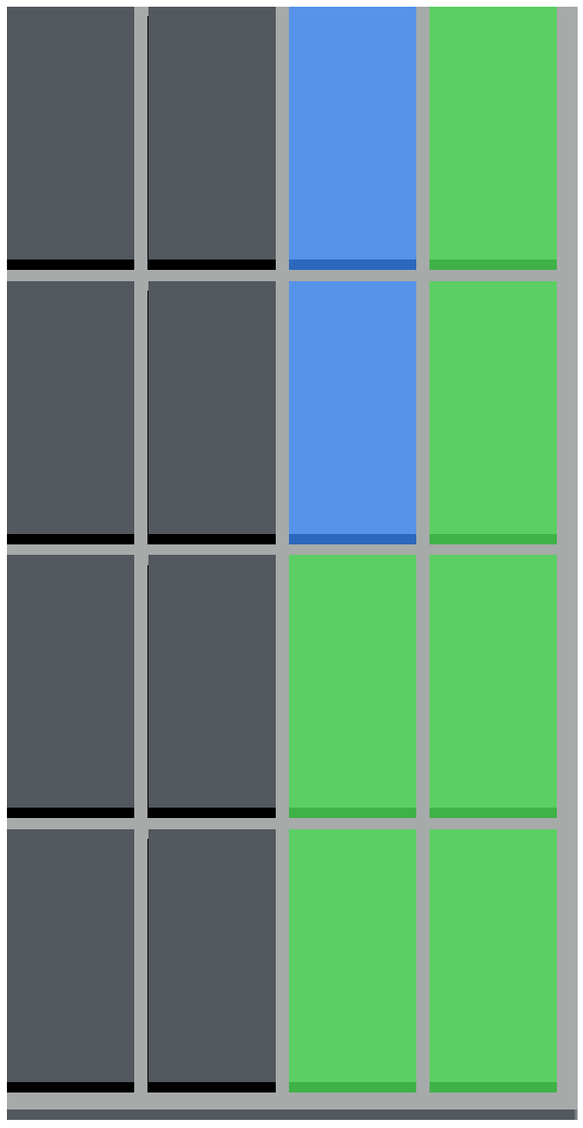


pganalyze

* except on Aurora

Postgres Shared Buffers

8kB Page



e.g. 128kB shared_buffers

Pinned Buffer
Actively being read/used by a backend

Dirty Buffer
Previously written to, not yet persisted to disk

Unpinned Buffer
Unused buffer, may contain cached data

pganalyze

Postgres Shared Buffers

What?

Reference count (pin count) is increased to avoid contents being replaced by others,
additional buffer locks may be taken (e.g. to replace content),
buffer is used (read and/or written to) and pin count decreased once work is done.

Who can do it?

1. Individual backends trying to read data / use cache data
2. Background processes (for similar reasons)

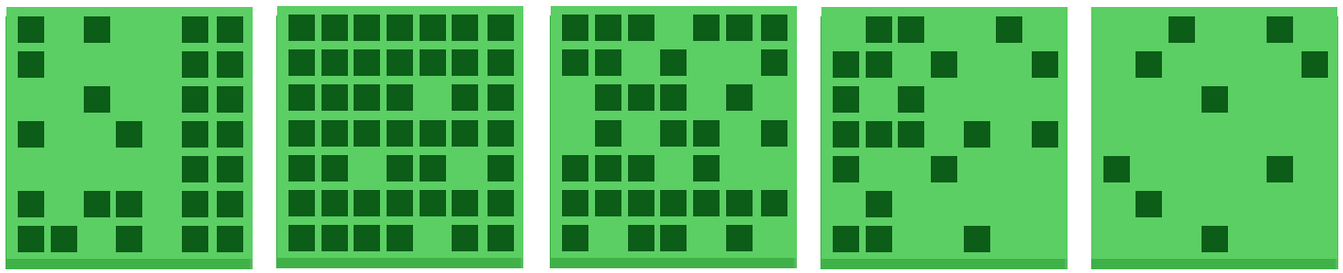
pganalyze

pganalyze

Tuning Postgres Internal Bottlenecks

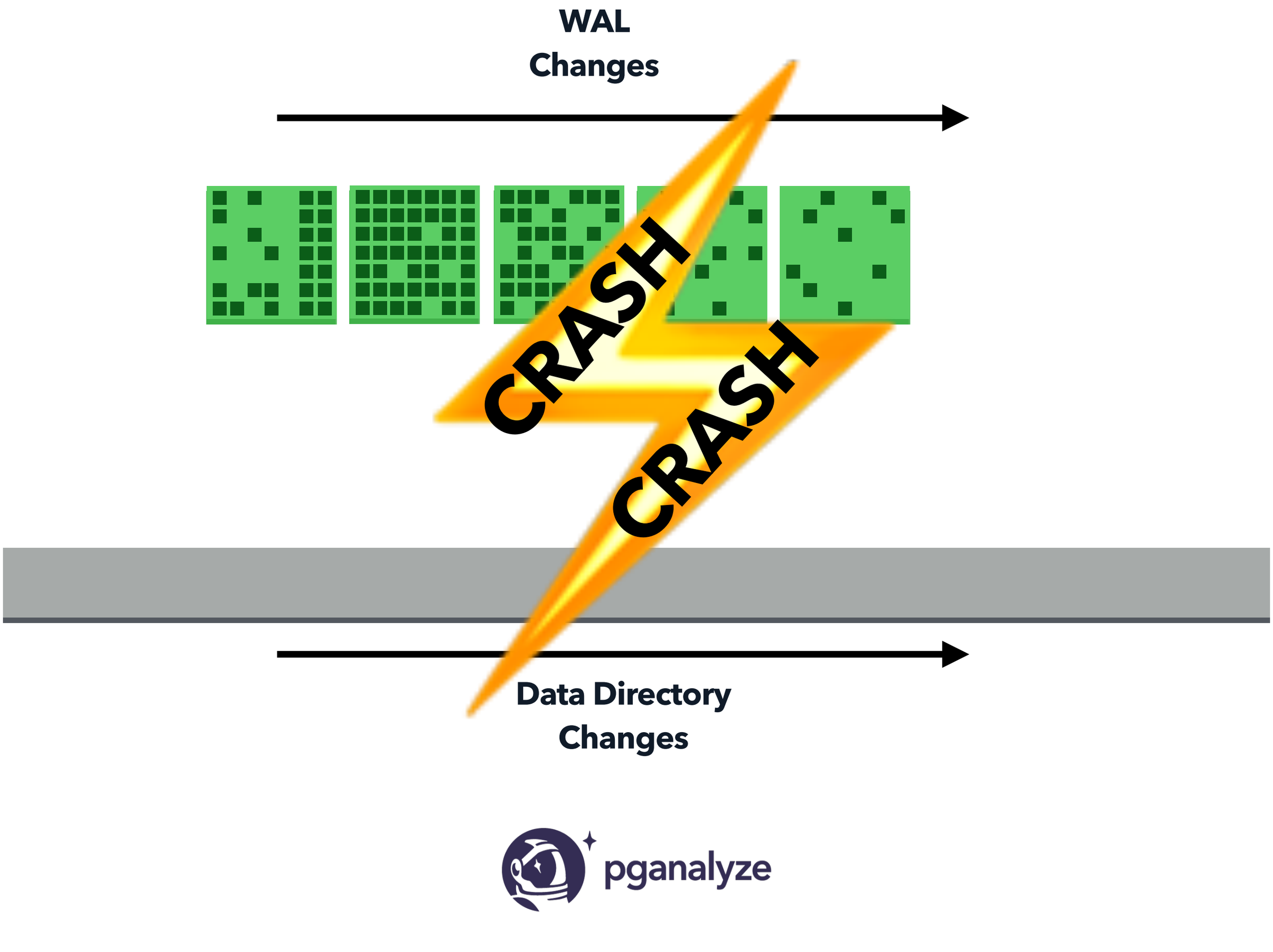
Checkpoints & WAL Buffers

WAL
Changes



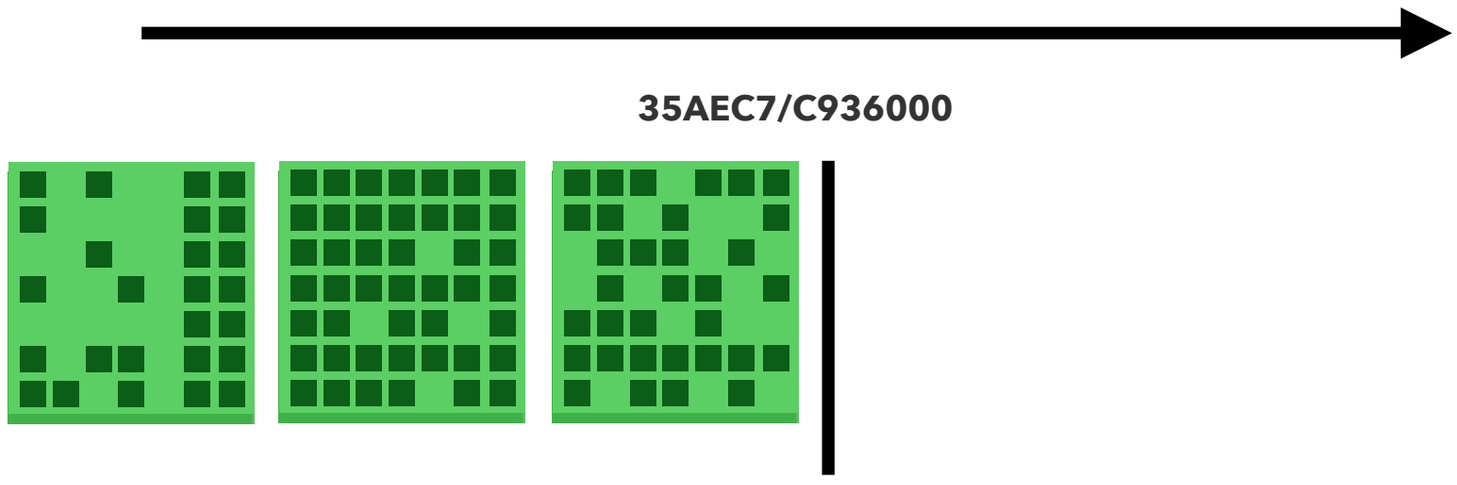
Data Directory
Changes

pganalyze



pganalyze

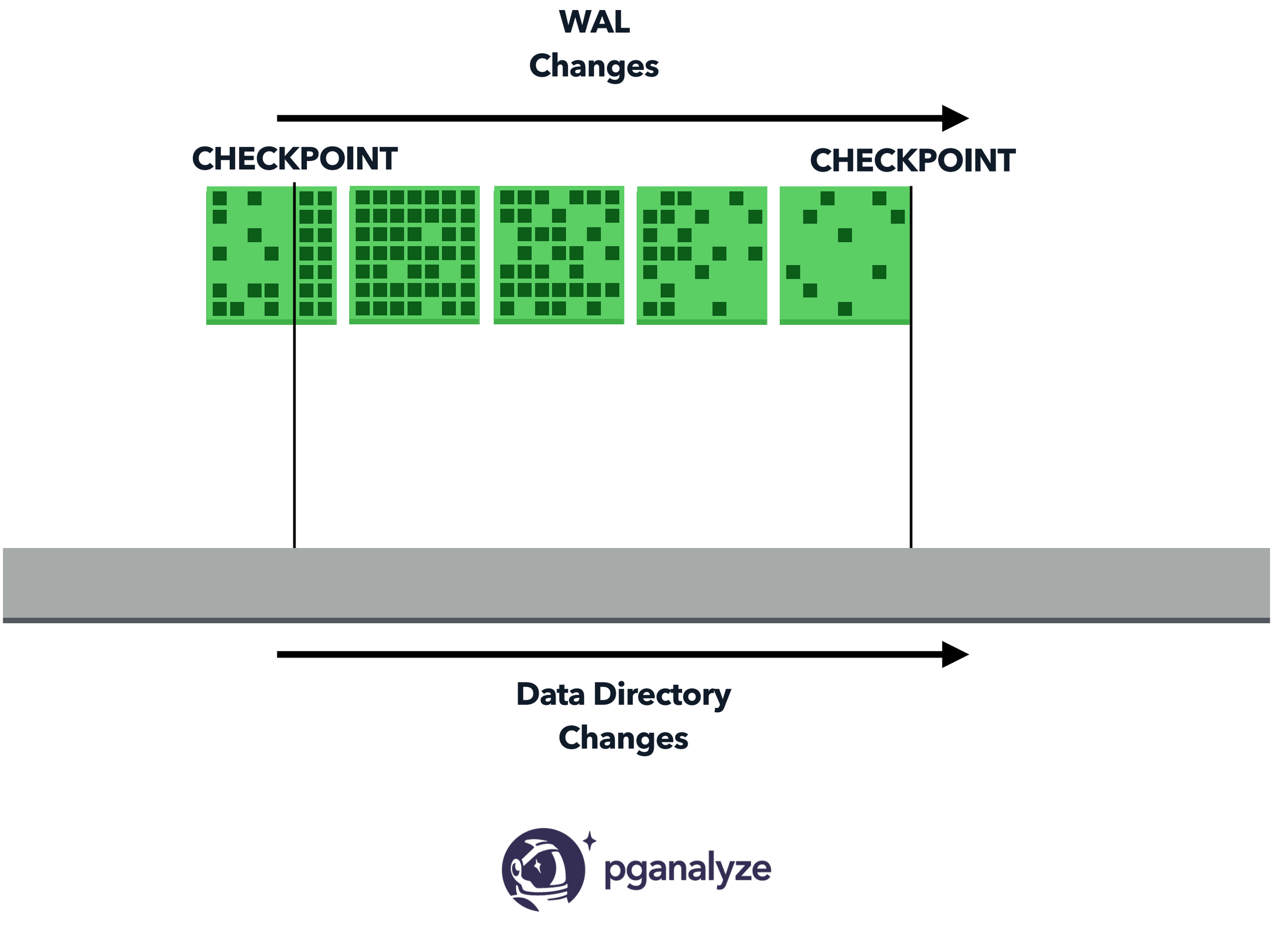
WAL
Changes



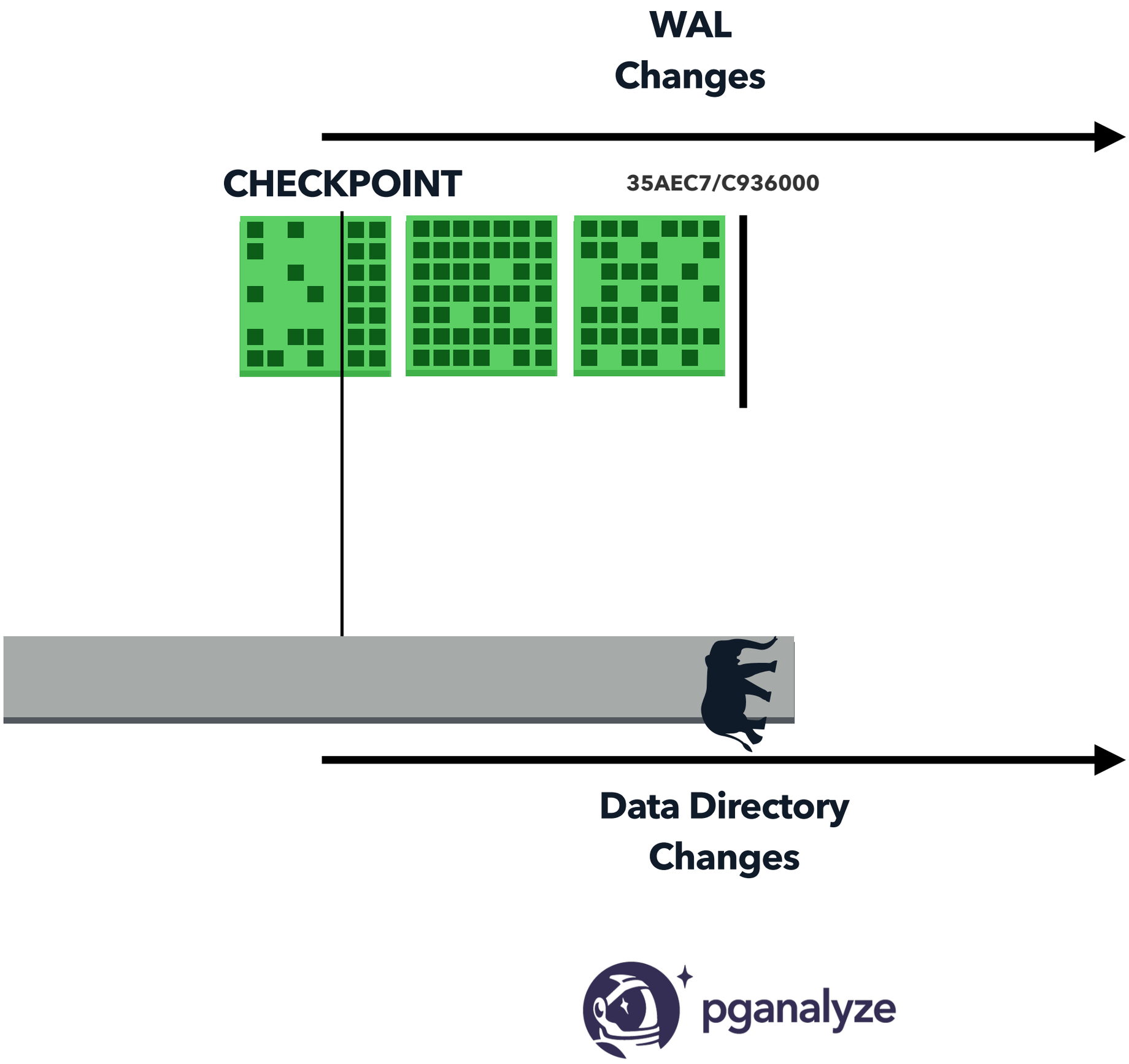
Data Directory
Changes

What is the state of
the data directory?!
(How far did it get synchronized?)

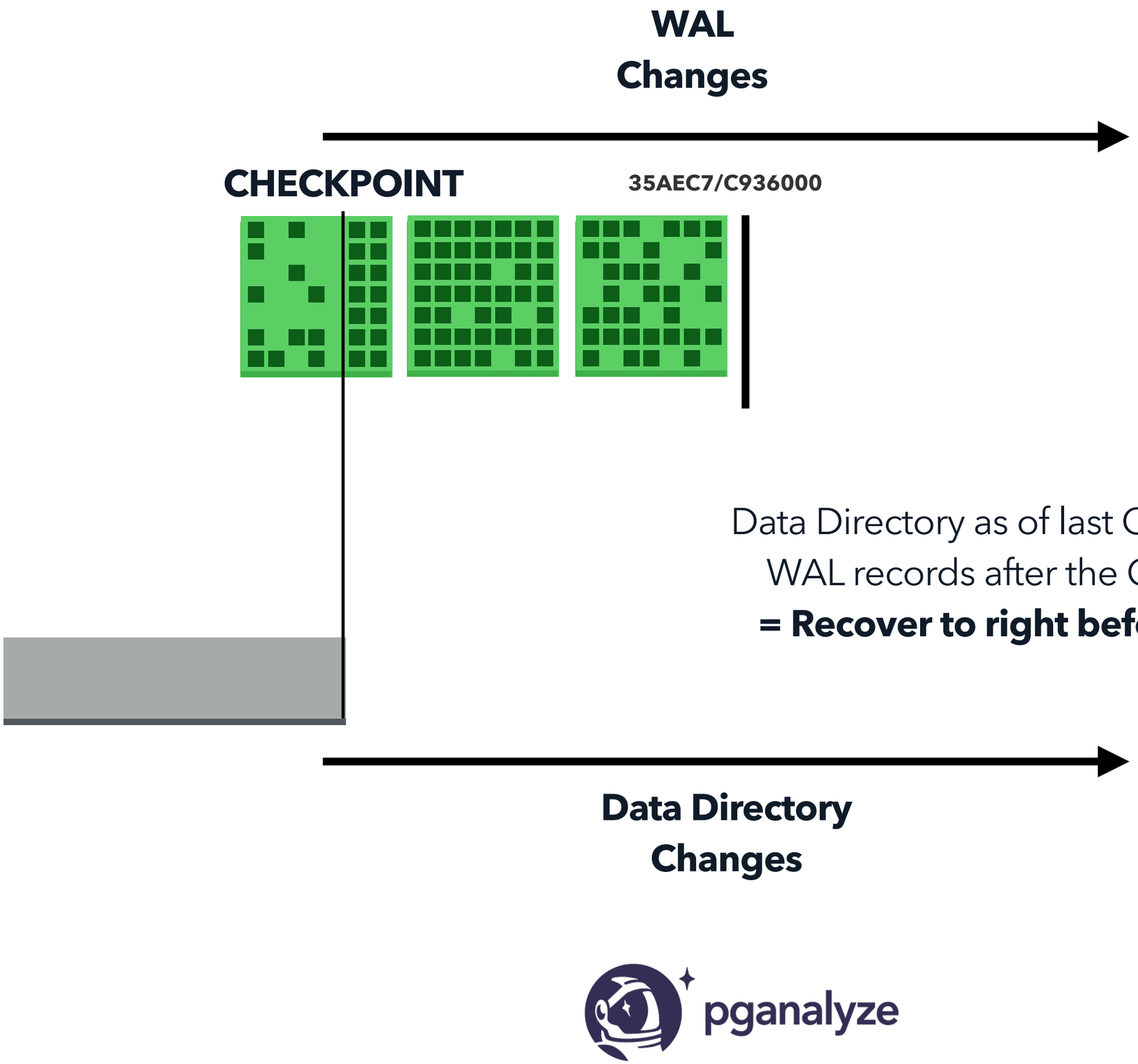
pganalyze


WAL
Changes
CHECKPOINT
CHECKPOINT
Data Directory
Changes

pganalyze

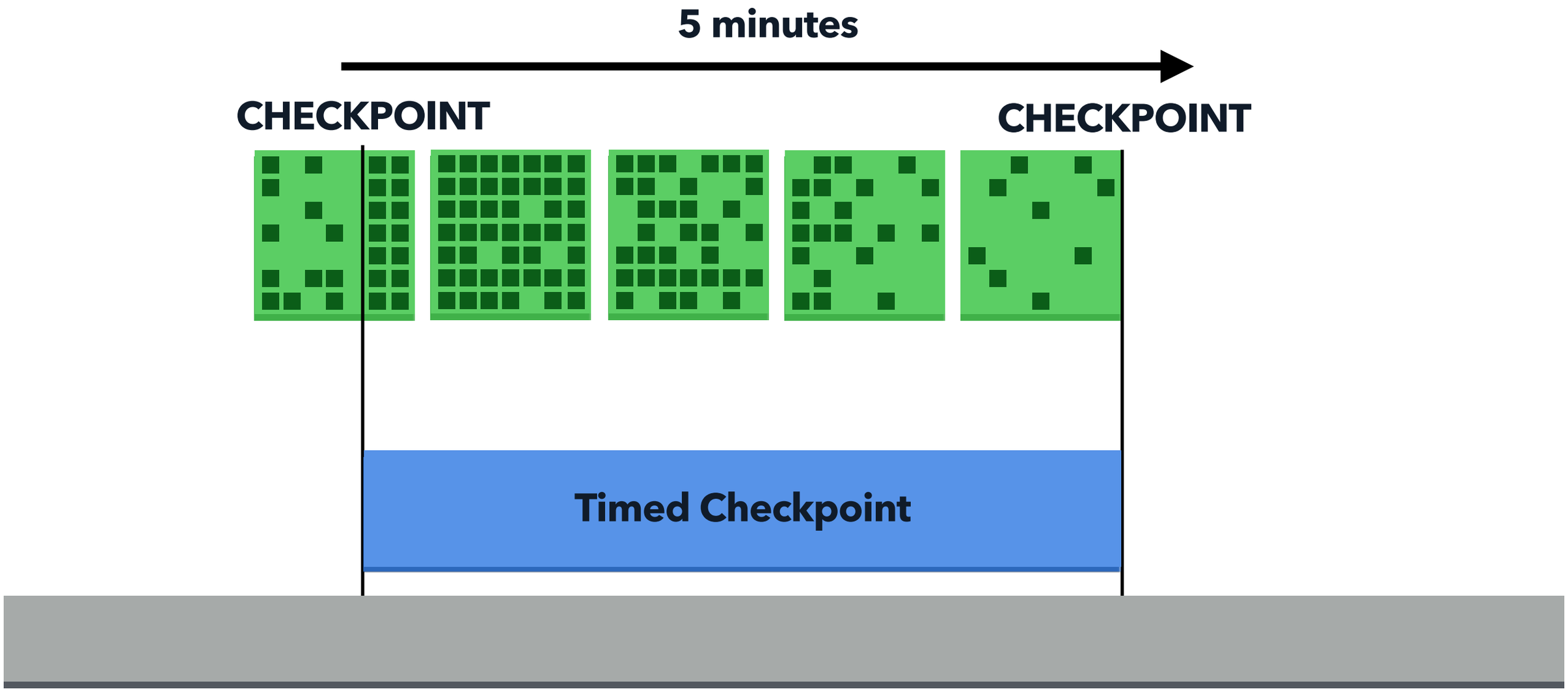

WAL
Changes
CHECKPOINT
35AEC7/C936000
Data Directory
Changes

pganalyze


WAL
Changes
CHECKPOINT
35AEC7/C936000
Data Directory as of last CHECKPOINT +
WAL records after the CHECKPOINT
= Recover to right before the crash
Data Directory
Changes

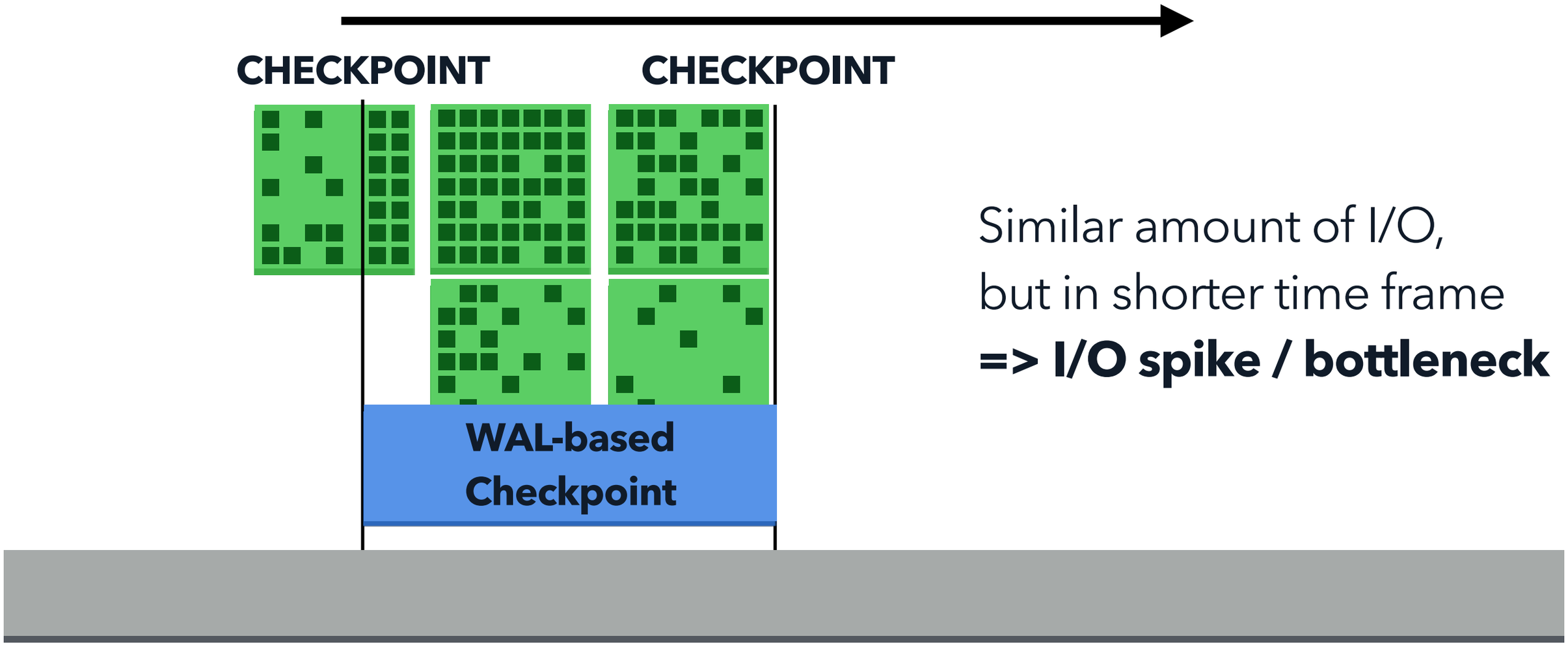
pganalyze

Timed vs WAL-based Checkpoints



checkpoint_timeout = 300s

pganalyze



Similar amount of I/O,
but in shorter time frame
=> I/O spike / bottleneck

max_wal_size = 1GB

pganalyze

Write-Ahead Log / Checkpoints

Setting	Current Value	Default Value	Source
checkpoint_completion_target	0.9	0.9	configuration file
checkpoint_flush_after	256 KB	256 KB	default
checkpoint_timeout	300 s	300 s	default
checkpoint_warning	30 s	30 s	default
max_wal_size	50 GB	1 GB	configuration file
min_wal_size	192 MB	80 MB	configuration file

Log Insights - W40: Checkpoint starting

Checkpoint to synchronize the data directory has started

Learn more

Postgres Server Log

Show all log events

Reason for Checkpoint Starting:

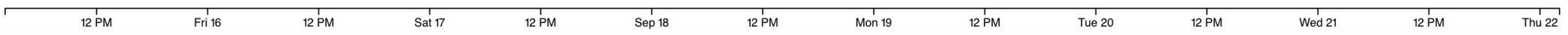
1826 × time

1 × wal

Related Config Parameters:

```
checkpoint_completion_target = 0.9
checkpoint_timeout = 300 s
max_wal_size = 50 GB
```

Jump to time:



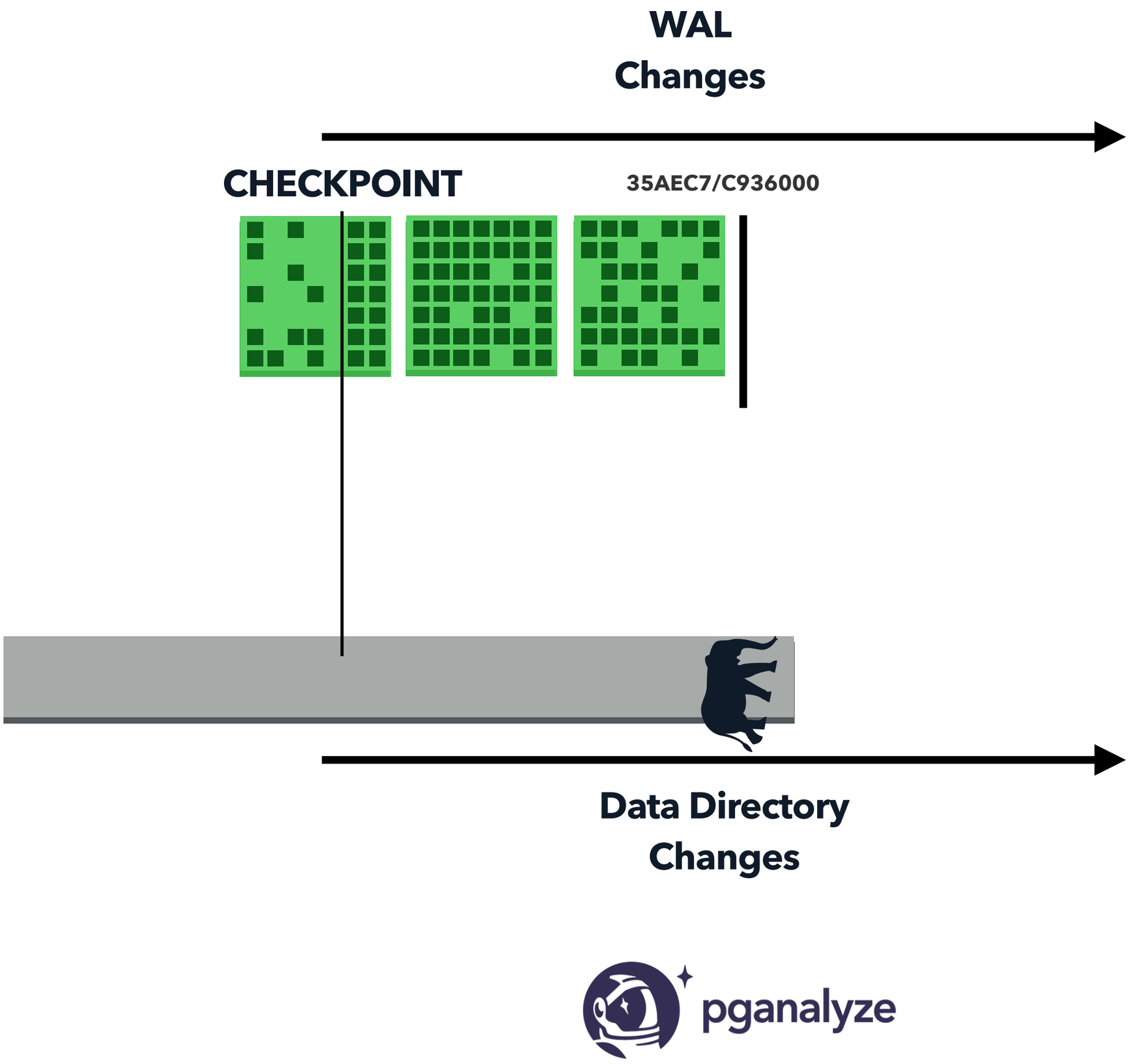

```
Sep 22 02:04:27am PDT W40 4498 LOG: checkpoint starting: time
Sep 22 01:59:27am PDT W40 4498 LOG: checkpoint starting: time
Sep 22 01:54:27am PDT W40 4498 LOG: checkpoint starting: time
Sep 22 01:49:27am PDT W40 4498 LOG: checkpoint starting: time
Sep 22 01:44:27am PDT W40 4498 LOG: checkpoint starting: time
Sep 22 01:39:27am PDT W40 4498 LOG: checkpoint starting: time
Sep 22 01:34:27am PDT W40 4498 LOG: checkpoint starting: time
Sep 22 01:29:27am PDT W40 4498 LOG: checkpoint starting: time
Sep 22 01:24:27am PDT W40 4498 LOG: checkpoint starting: time
Sep 22 01:19:27am PDT W40 4498 LOG: checkpoint starting: time
Sep 22 01:14:27am PDT W40 4498 LOG: checkpoint starting: time
Sep 22 01:09:27am PDT W40 4498 LOG: checkpoint starting: time
Sep 22 01:04:27am PDT W40 4498 LOG: checkpoint starting: time
```

pganalyze

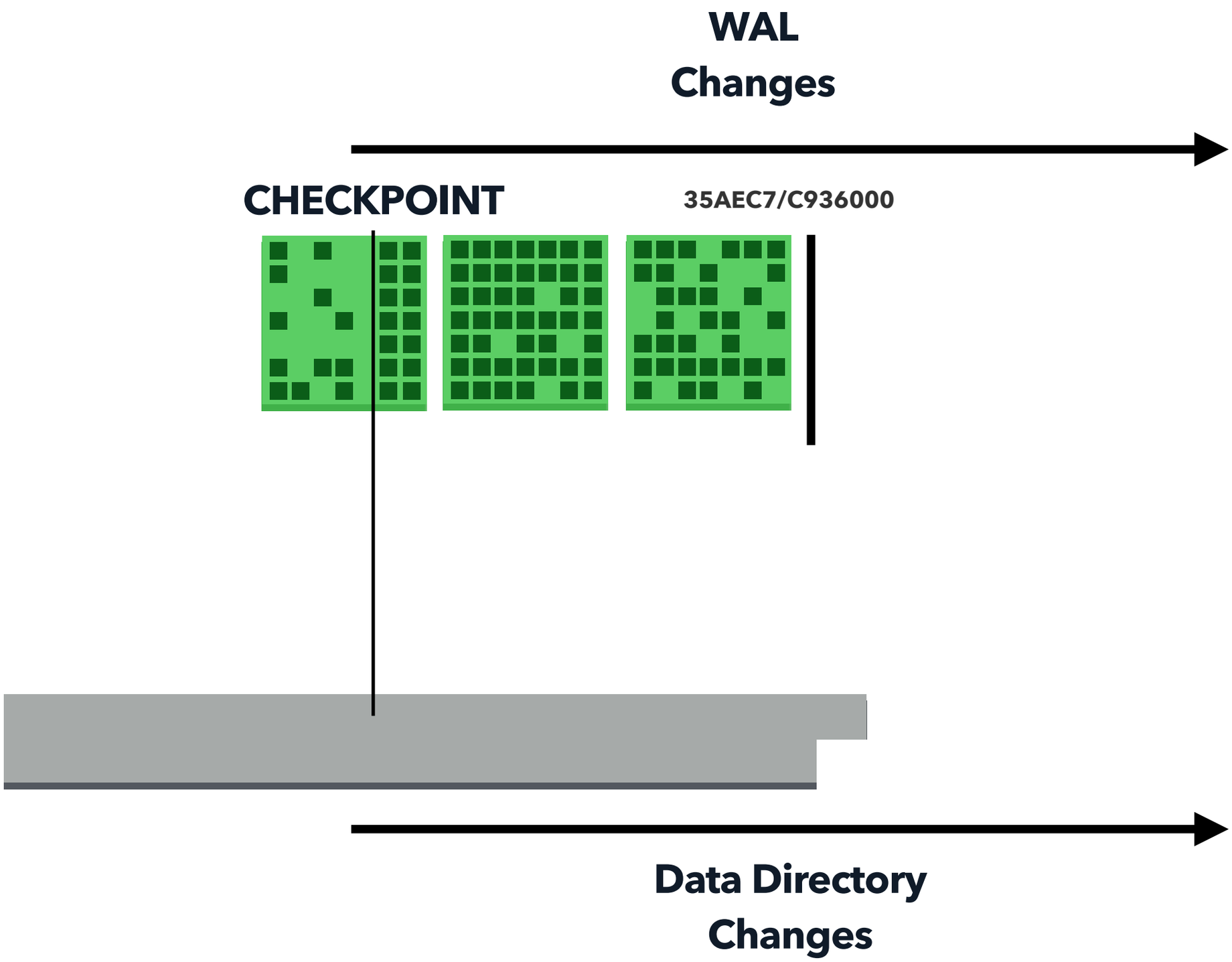
Full Page Writes

Full Page Images (FPIs) are big

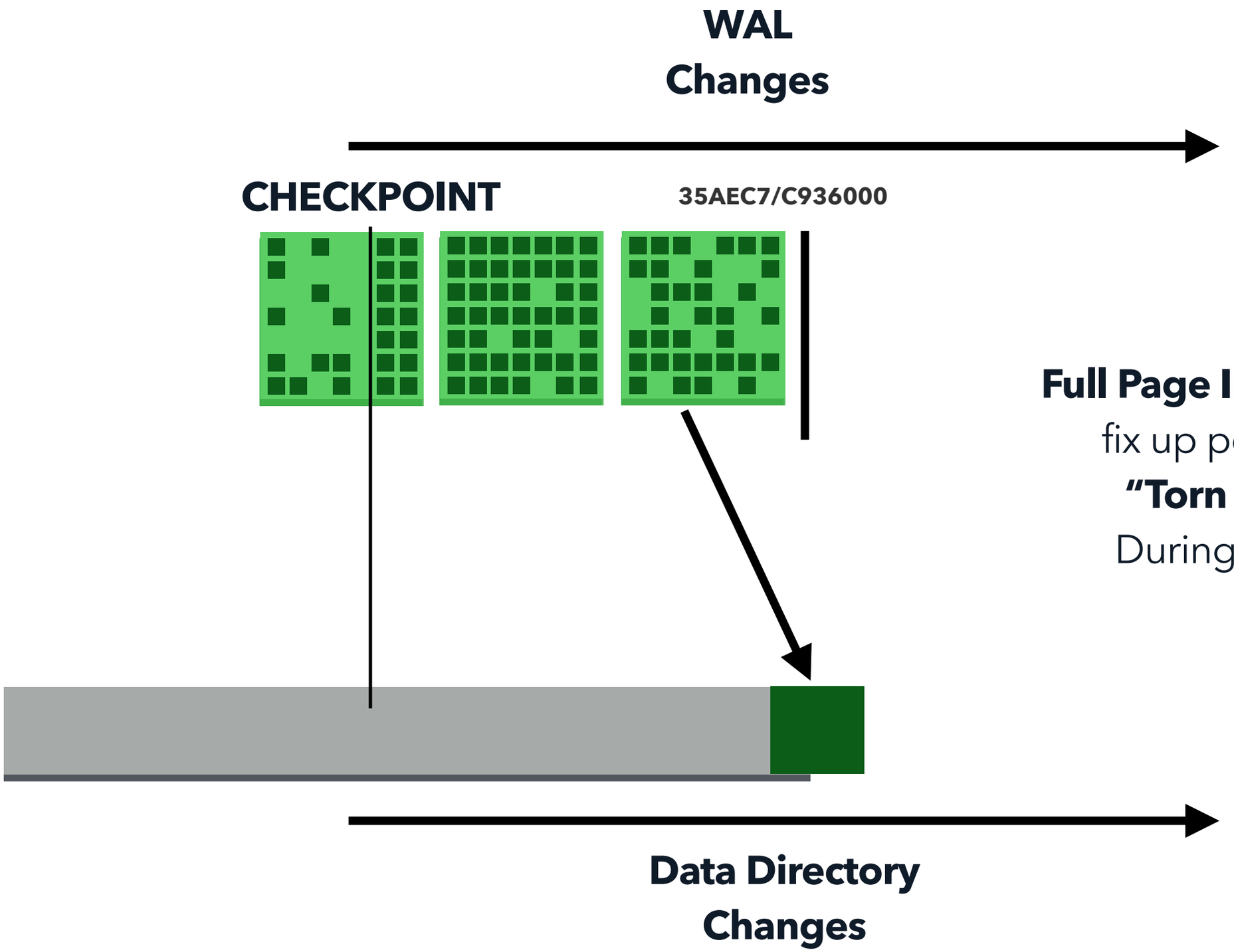
(a full 8kb page copied into the WAL)


WAL
Changes
CHECKPOINT
35AEC7/C936000
Data Directory
Changes

pganalyze

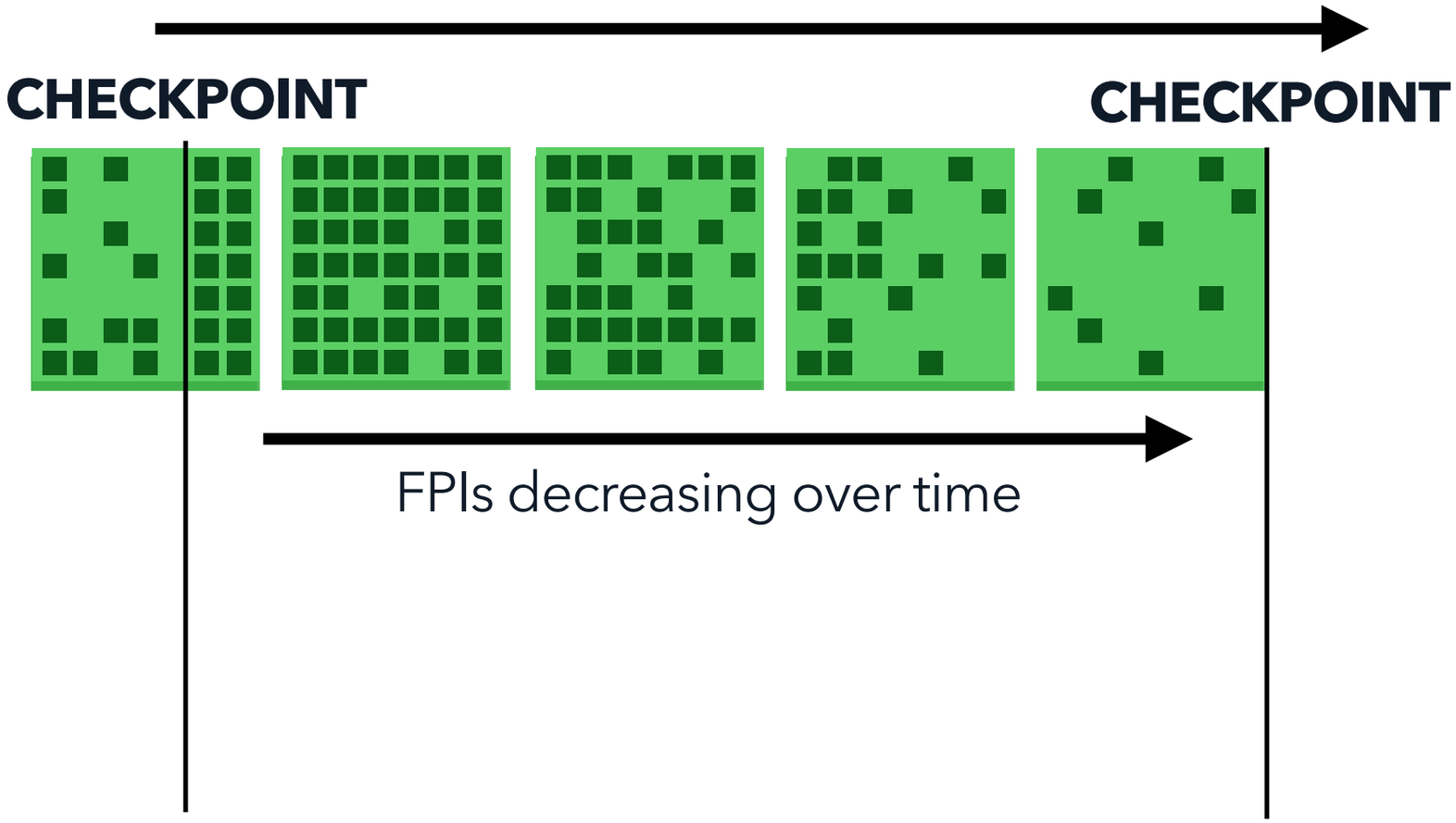

WAL
Changes
CHECKPOINT
35AEC7/C936000
Data Directory
Changes

pganalyze

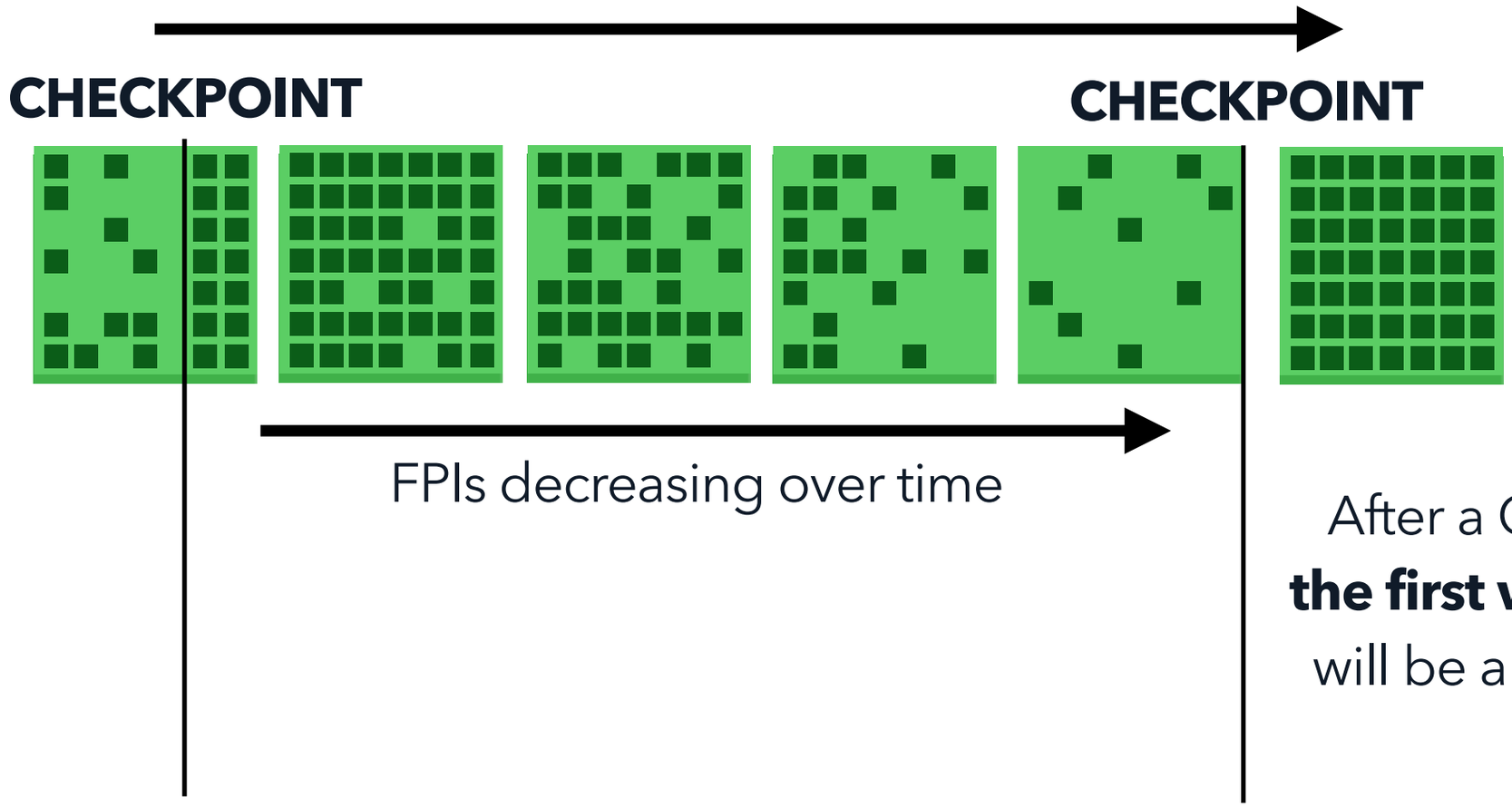


Full Page Images (FPI)
fix up potentially
“Torn Pages”
During a Crash

pganalyze



pganalyze



After a CHECKPOINT,
the first write to a page
will be a full page write

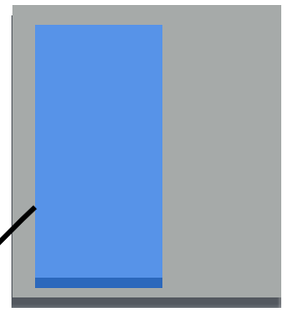
pganalyze

wal_compression = lz4 (with PG15+)

- Compresses FPIs in WAL with lz4 compression
- Reduces impact of having lots of full page writes after a checkpoint
- Before PG15 only pglz format was supported (high CPU overhead)

shared_buffers are not (just) a cache, they are essential for writes.

1. Remember changed page in shared_buffers



2. Remember changed rows (or full page) in wal_buffers

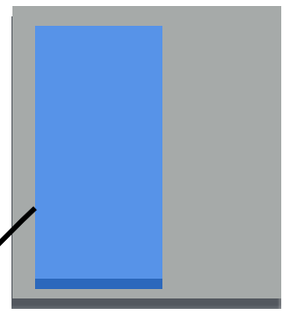
3. Persist to WAL

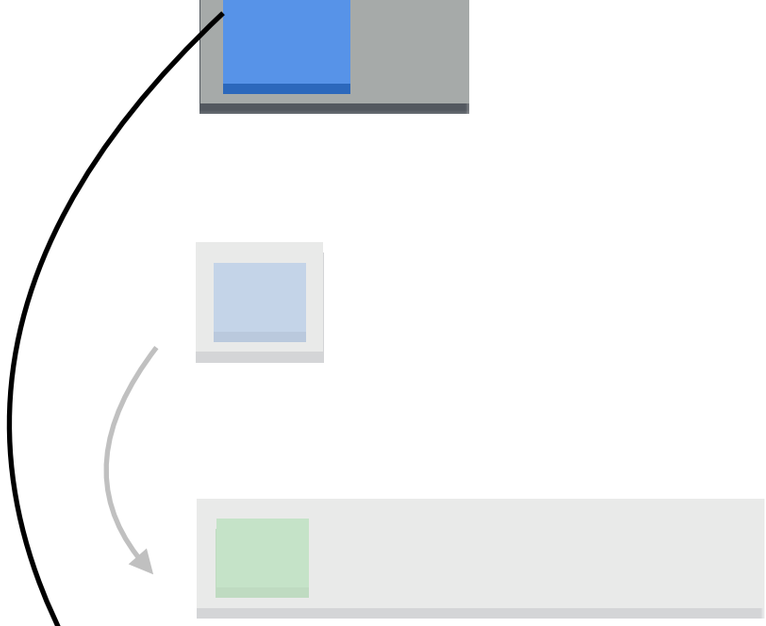
4. Persist to Data Directory

pganalyze

shared_buffers are not (just) a cache, they are essential for writes.

1. Remember changed page in shared_buffers

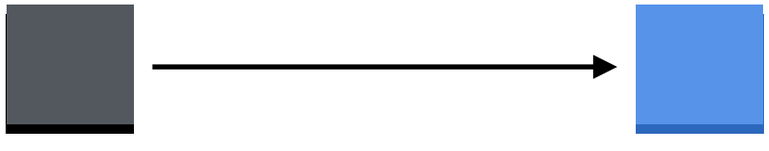




4. Persist to Data Directory

pganalyze

Postgres Shared Buffers



Pinned Buffer

Dirty Buffer

What?

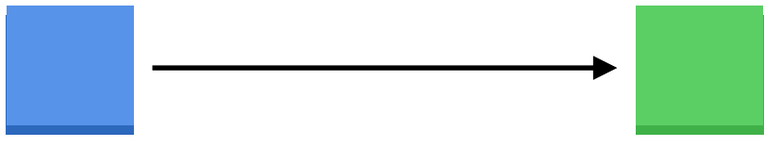
Backend (or other processes) changed the data in the buffer (= “dirtied” it) and that data now needs to be persisted to disk.

Who can do it?

1. Individual backends trying to write data
2. VACUUM cleaning up a table

pganalyze

Postgres Shared Buffers



Dirty Buffer Unpinned Buffer

What?

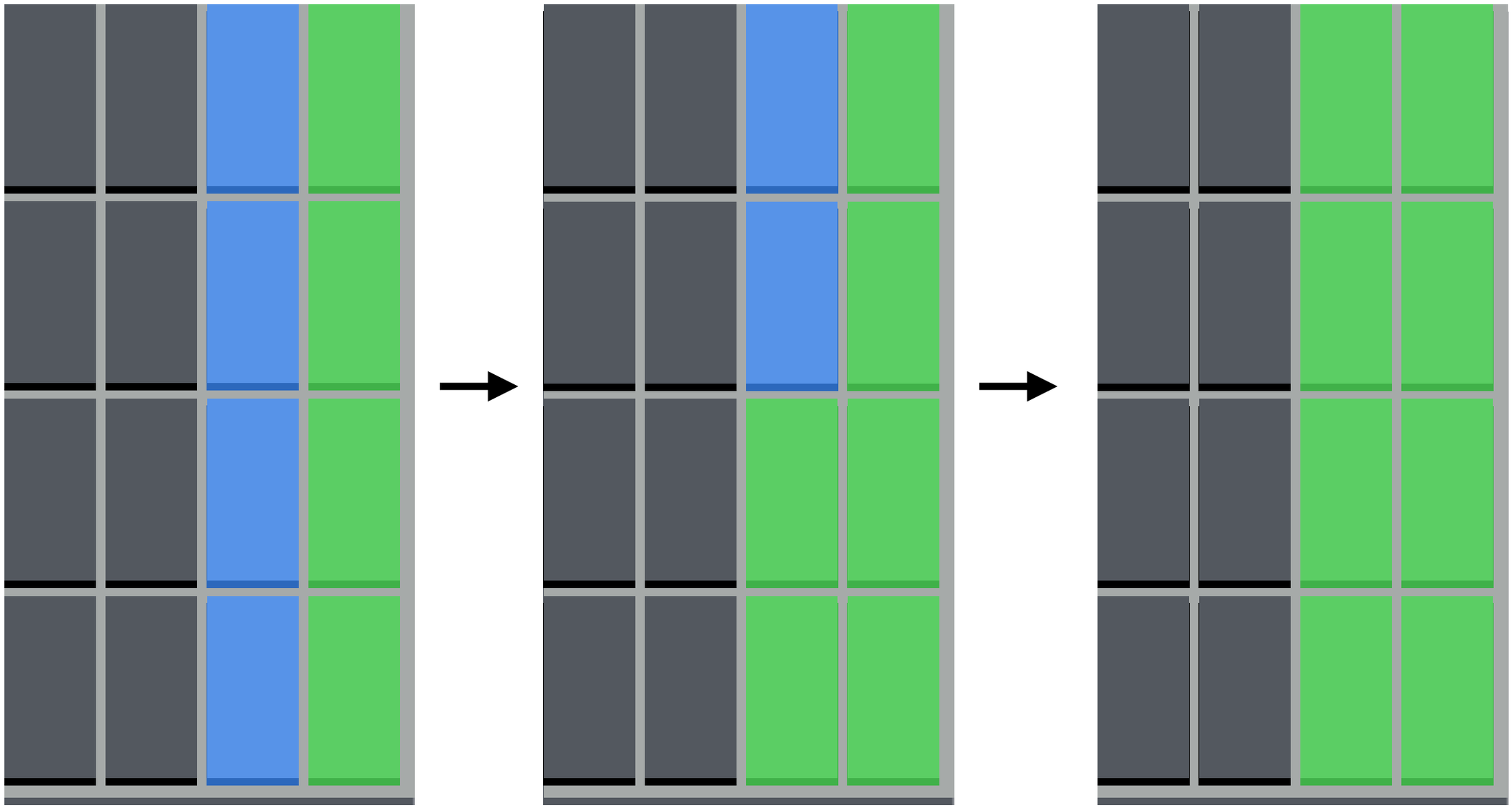
Write data from memory to OS page cache, and potentially flush OS page cache to disk.

Who can do it?

1. Background writer (bgwriter)
2. Checkpointer
3. Individual backend that needs an unpinned buffer 🚨

pganalyze

Background Writer



pg_stat_io (Postgres 16+)

Which part of Postgres wrote (flushed) to disk?

```
SELECT backend_type, object, context, writes FROM pg_stat_io WHERE
writes <> 0 AND context = 'normal';

    backend_type     |  object  | context |   writes
---------------------+----------+---------+-------------
 autovacuum worker   | relation | normal  |           9
 client backend      | relation | normal  |       90103
 background writer   | relation | normal  |   401710668
 checkpointer        | relation | normal  | 65848285326
 standalone backend | relation | normal  |        1037
```

pg_stat_io (Postgres 16+)

Which part of Postgres wrote (flushed) to disk?

```
   backend_type    | io_object | io_context |  reads   |    hits    |  writes
-------------------+-----------+------------+----------+------------+----------
 client backend    | relation  | normal     | 99844392 | 321091023 | 18120752
 background writer | relation  | normal     |          |           | 20100354
 checkpointer      | relation  | normal     |          |           |   753214
(3 rows)
```

Melanie Plageman: Additional I/O Observability with pg_stat_io

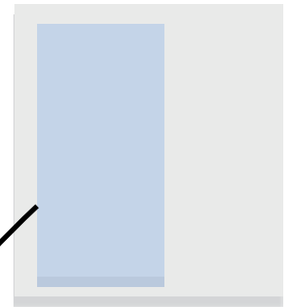
Background Writer

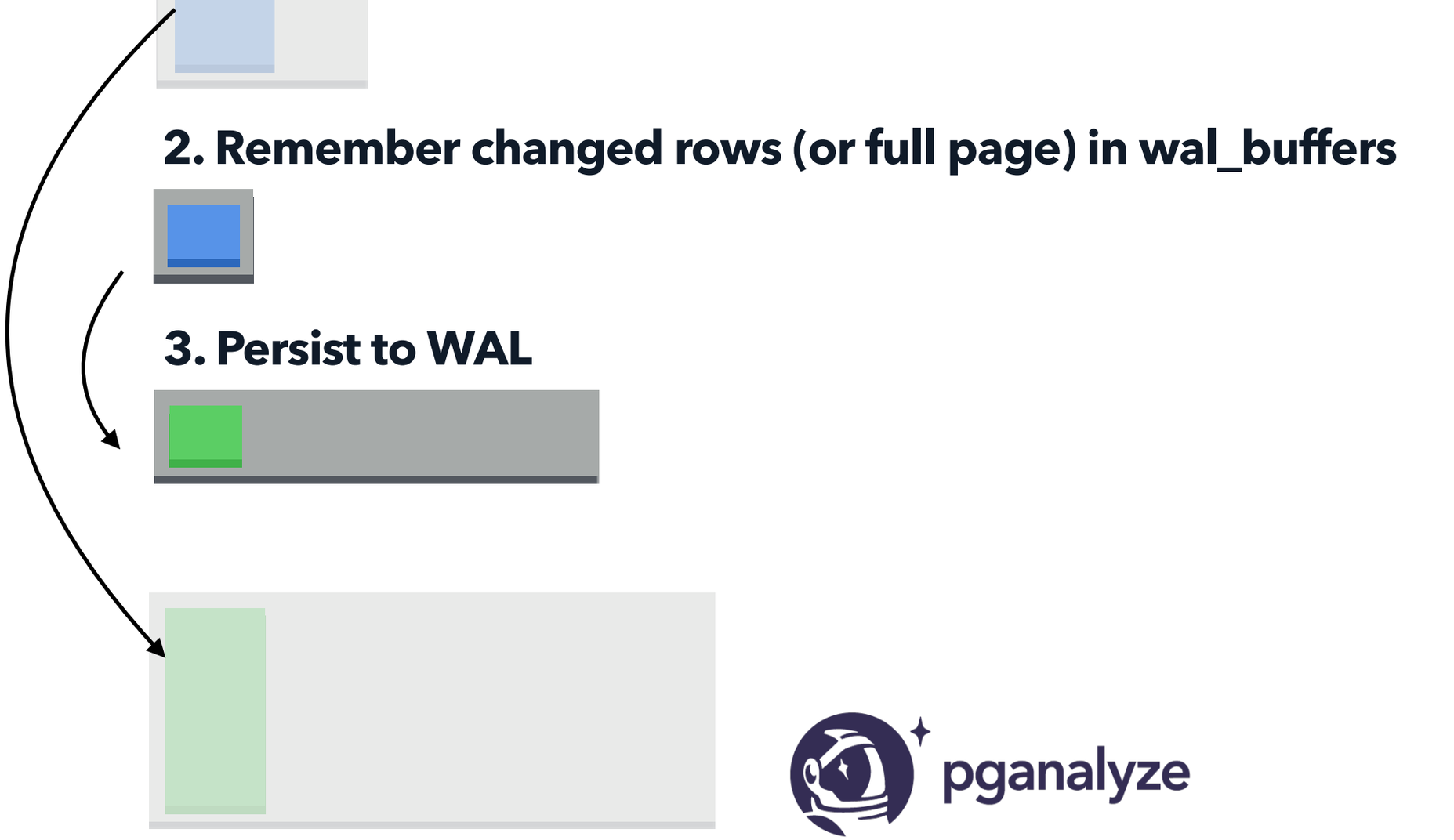
Tune to run more often for busy workloads
=> **reduce bgwriter_delay**

If background worker doesn't do its job in time,
Individual queries might write dirty buffers before they can read.
(pg_stat_statements.shared_blks_written)

But: If it runs too often, you'll create additional I/O.
(a dirty page can be "re-used" for a write within the same checkpoint)

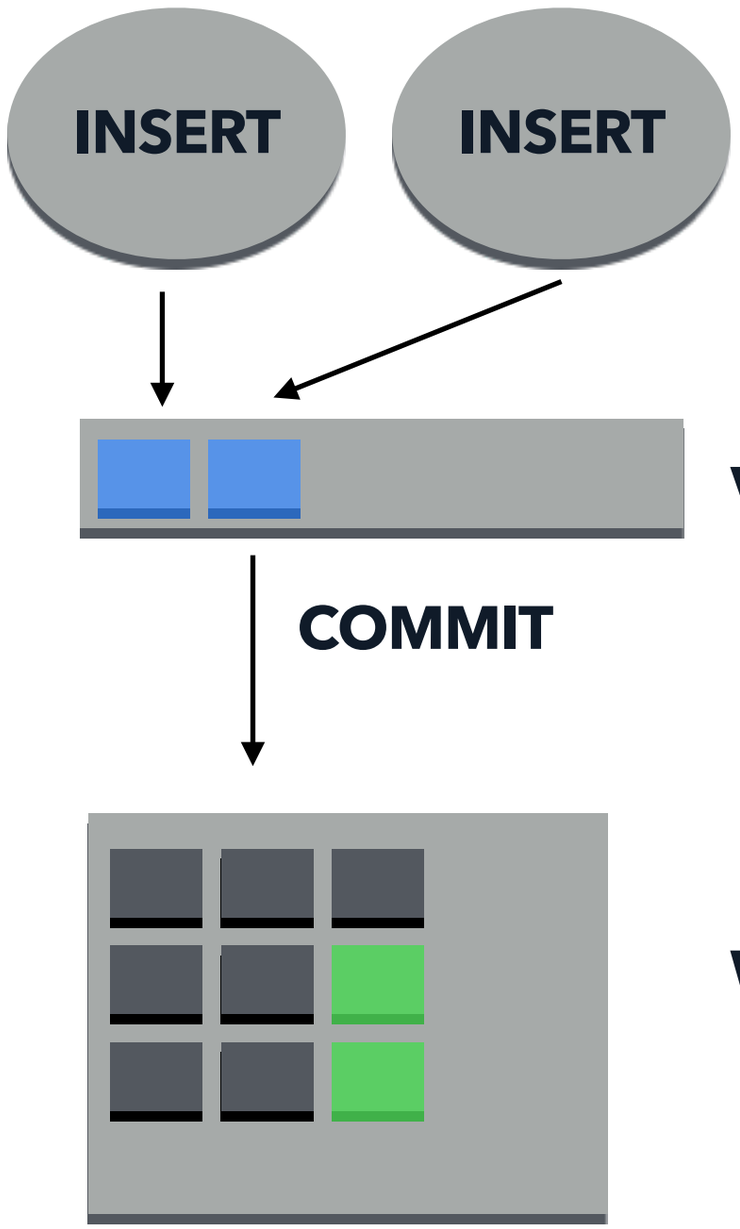
Persisting Changes to the WAL





pganalyze

wal_buffers



wal_buffers (e.g. 16 MB)

WAL file (e.g. 64 MB)

pganalyze

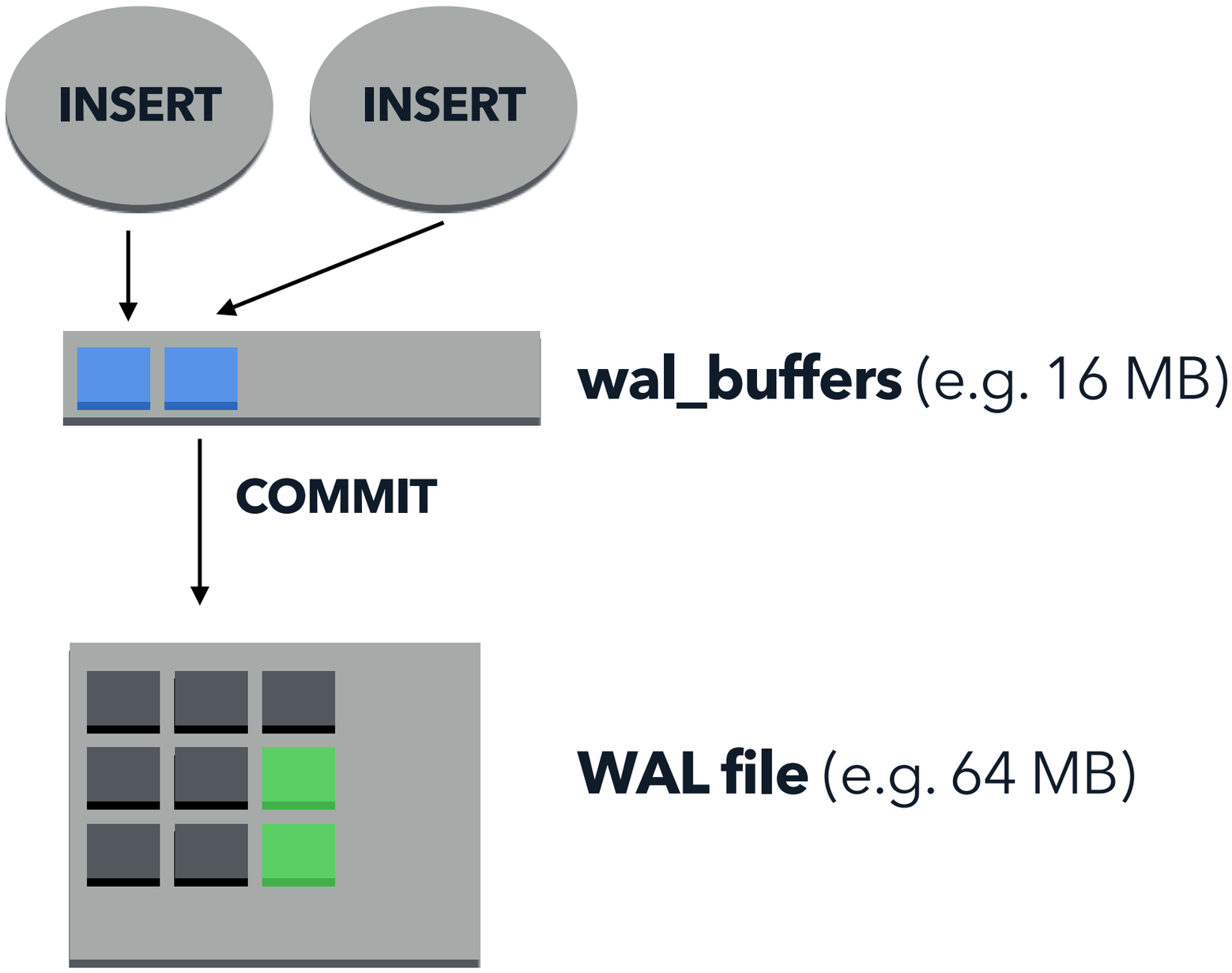
wal_buffers = -1

- Default is automatically set to max(shared_buffers / 32, wal_segment_size)
- If you have very big transactions (or are using synchronous_commit = off), increasing wal_buffers to a multiple of wal_segment_size can be helpful

synchronous_commit = off

(sometimes)

synchronous_commit = on

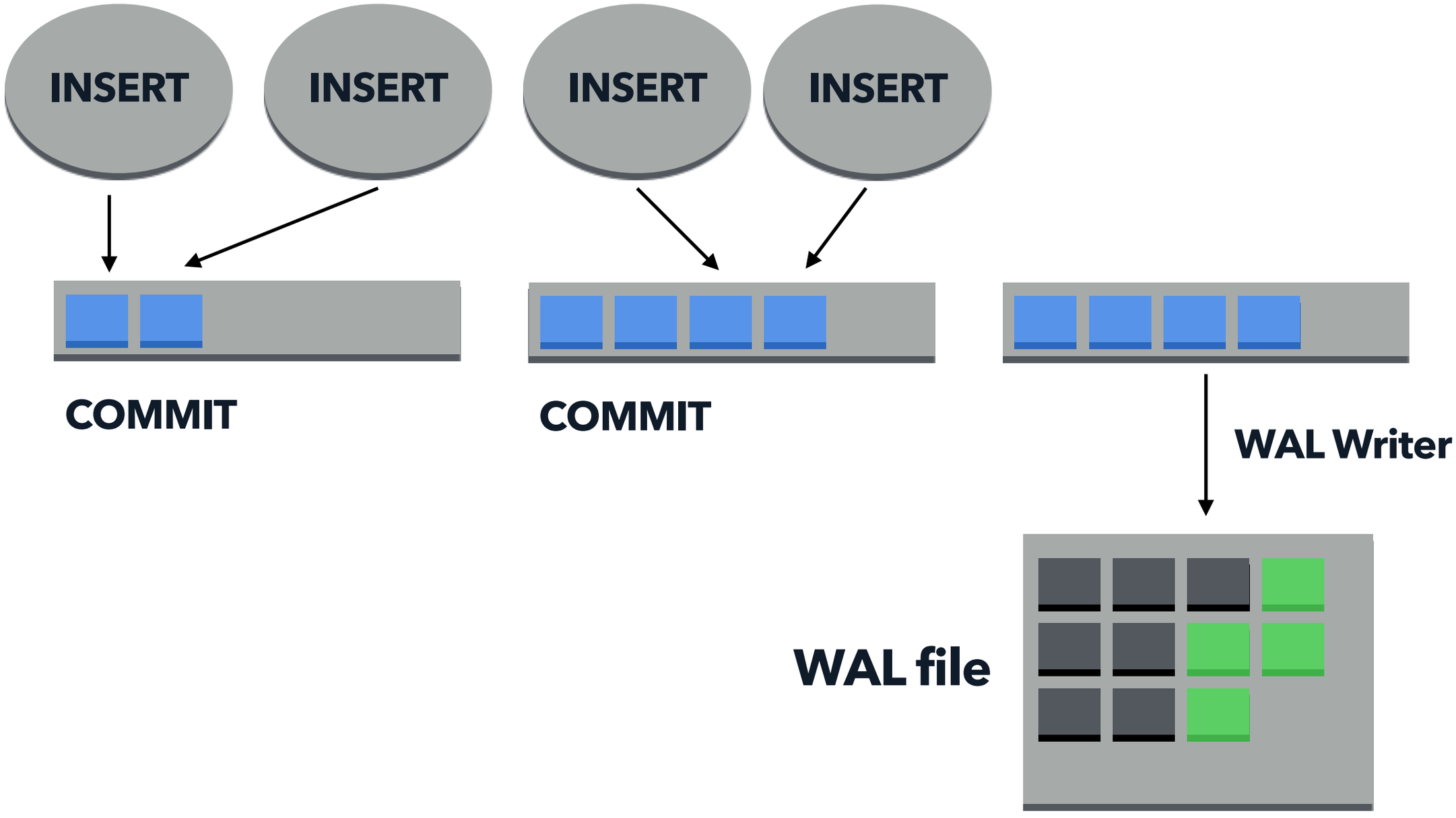


wal_buffers (e.g. 16 MB)

WAL file (e.g. 64 MB)

pganalyze

synchronous_commit = off



pganalyze

synchronous_commit = off

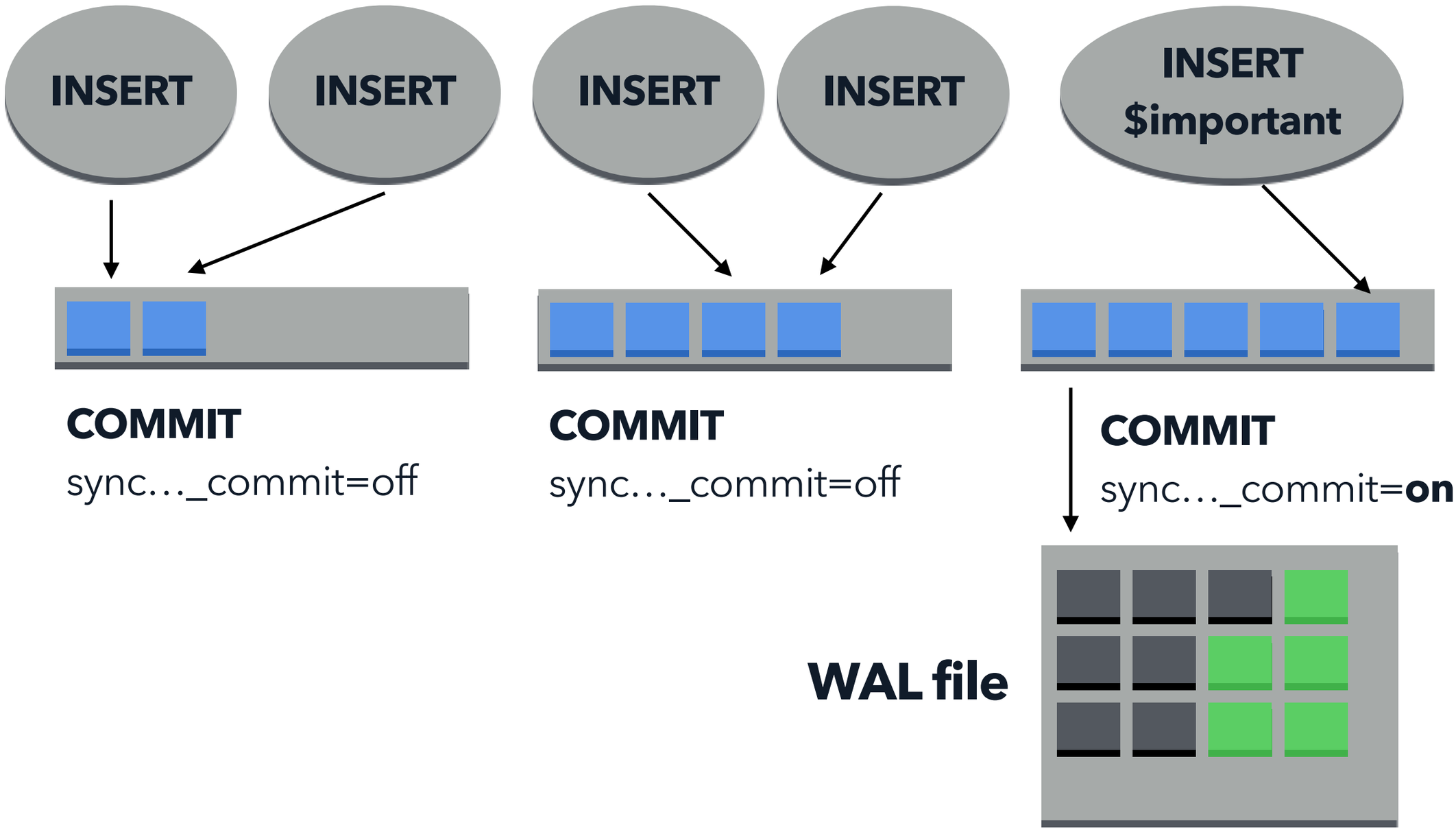
! You may loose data not yet persisted to WAL in a crash.

The database will be consistent
(just missing that most recent data),
no risk of corruption.

"This parameter **can be changed at any time**; the behavior for any one transaction is determined by the setting in effect when it commits.

It is therefore possible, and useful, to **have some transactions commit synchronously and others asynchronously**."

synchronous_commit = [SET per transaction]



pganalyze

If you make heavy use of
synchronous_commit = off ...

- Consider lowering **wal_writer_delay**
  (to write WAL more frequently, avoiding flushes during individual commits)
- Increase **wal_buffers** to a multiple of wal_segment_size

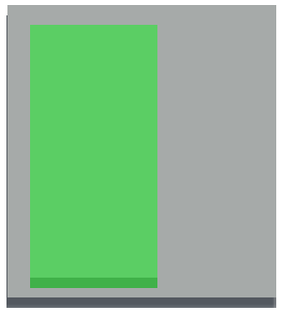
wal_segment_size & min_wal_size

- Default of 16 MB is too small (too much churn with new files)
- Consider at least 64 MB
- If going for a much larger value, ensure min_wal_size is sufficient
  (to keep old files around, avoiding expensive initialization of new files)

Amazon Aurora is different

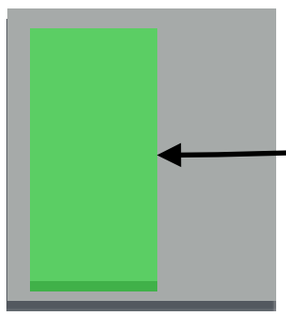
Amazon Aurora Is Different

1. Remember changed page in shared_buffers



2. Write out changed rows to WAL

3. Refresh Replicas
(their shared_buffers)



Aurora Log Write

pganalyze

Amazon Aurora Is Different

No Full Page Writes

"No Checkpoints"

(Heavily Modified Checkpointer

+ Background Writer)

Aurora Is Not Always Better

Both Read and Write IOPS are charged extra

Read I/Os are always charged per 8kB disk page

Read I/Os will be slower (sometimes)

Write I/Os are always charged per 4kB log record

Write I/O with synchronous_commit=on will be slower

SLRU Caches

SLRU = "Simple Least Recently Used"

SLRU caches provide

Storage management and buffered access for 7 kinds of critical transaction-related data

commit_timestamp: Tracks commit timestamps (if enabled)
multixact_(member|offset): Row-level FOR SHARE locks
notify: LISTEN/NOTIFY queue
serializable: Tracks information related to SERIALIZABLE
subtransaction: Tracks subtransaction (SAVEPOINT) parents
transaction: Tracks transaction commit/abort status

SLRUs have corresponding on-disk storage

```
-rw-------   1 lukasfittl  staff      3 Mar  1 20:18 PG_VERSION
drwx------   8 lukasfittl  staff    256 Mar 18 13:33 base
drwx------  65 lukasfittl  staff   2080 Mar 18 13:40 global
drwx------   2 lukasfittl  staff     64 Mar  1 20:18 pg_commit_ts
drwx------   2 lukasfittl  staff     64 Mar  1 20:18 pg_dynshmem
-rw-------   1 lukasfittl  staff   5721 Mar  1 20:18 pg_hba.conf
-rw-------   1 lukasfittl  staff   2640 Mar  1 20:18 pg_ident.conf
drwx------   5 lukasfittl  staff    160 Mar 18 17:57 pg_logical
drwx------   4 lukasfittl  staff    128 Mar  1 20:18 pg_multixact
drwx------   2 lukasfittl  staff     64 Mar  1 20:18 pg_notify
drwx------   2 lukasfittl  staff     64 Mar  1 20:18 pg_replslot
drwx------   2 lukasfittl  staff     64 Mar  1 20:18 pg_serial
drwx------   2 lukasfittl  staff     64 Mar  1 20:18 pg_snapshots
drwx------   2 lukasfittl  staff     64 Mar 16 00:06 pg_stat
drwx------   3 lukasfittl  staff     96 Mar 16 00:06 pg_stat_tmp
drwx------   3 lukasfittl  staff     96 Mar 17 16:47 pg_subtrans
```

You can track SLRU cache use to understand your workload:

```
SELECT name, blks_hit, blks_read, blks_written FROM pg_stat_slru;

      name       |   blks_hit   | blks_read  | blks_written
-----------------+--------------+------------+--------------
 CommitTs        |            0 |          0 |            0
 MultiXactMember |            0 |          0 |            1
 MultiXactOffset |            1 |          7 |            8
 Notify          |         1720 |          0 |            0
 Serial          |            0 |          0 |            0
 Subtrans        |          127 |          0 |     16410858
 Xact            | 147228675344 | 2285819909 |      1005967
 other           |            0 |          0 |            0
(8 rows)
```

"We noticed the database on GitLab.com would mysteriously stall for minutes, which would lead to users seeing 500 errors during this time"

GitLab: Why we spent the last month eliminating PostgreSQL subtransactions

"We saw our query latency to our Postgres database spiking precipitously, and our throughput plummeting virtually to 0; the database would almost completely lock up and make virtually no progress at all for a while"

Nelson Elhage: Notes on some PostgreSQL implementation details

Postgres 17 improves SLRUs significantly:

- Cache sizes are configurable
- Locks are partitioned

Action to take: Understand SLRU usage ahead of time, so you can modify your workload. If you use certain SLRU caches heavily and are not on 17, upgrade.

Replication

Replication Lag is a function of the amount of WAL

Less WAL writes = Less replica lag

Less Full Page Images = Less replica lag

The dilemma of hot_standby_feedback

hot_standby_feedback = on
Inform the primary of activity on the replica (e.g. slow query), so that it does not remove information still needed. Reduces replication lag, but impacts regular operations on the primary (e.g. can block VACUUM).

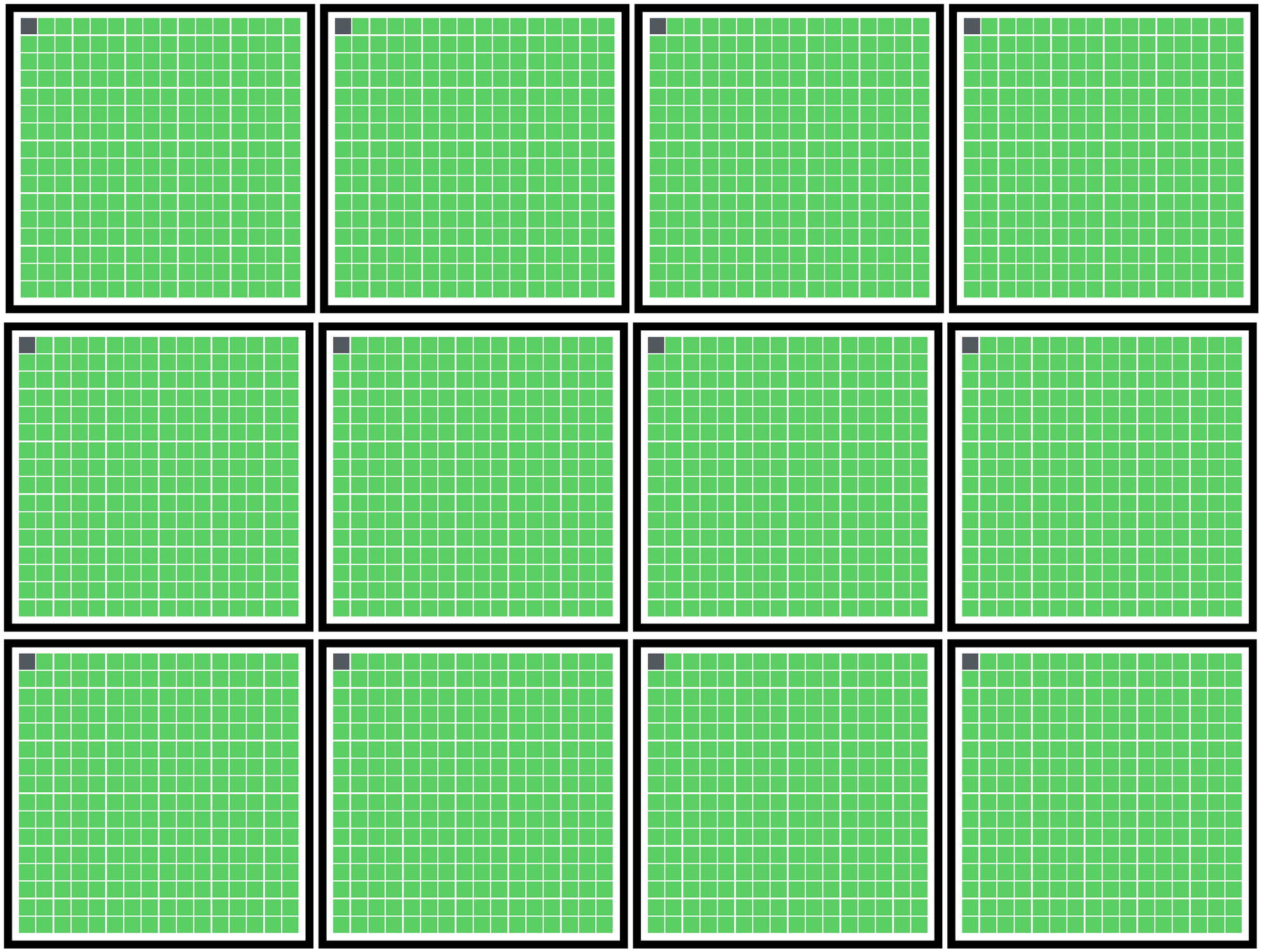
hot_standby_feedback = off
Increases replication lag, but avoids impact on the primary.

=> Consider splitting data warehousing replicas out and using hot_standby_feedback = off and an increased max_standby_*_delay there.

pganalyze

Table and Index Maintenance

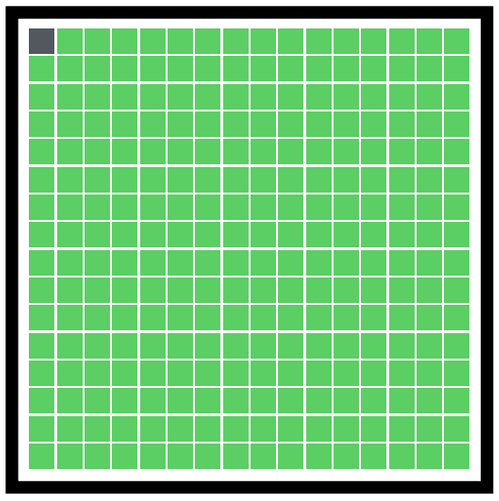
What is "Bloat"?

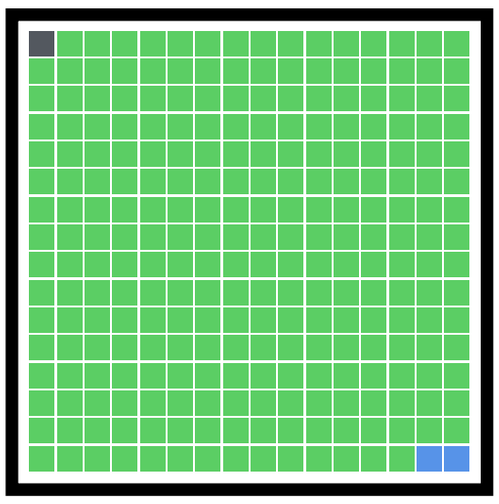


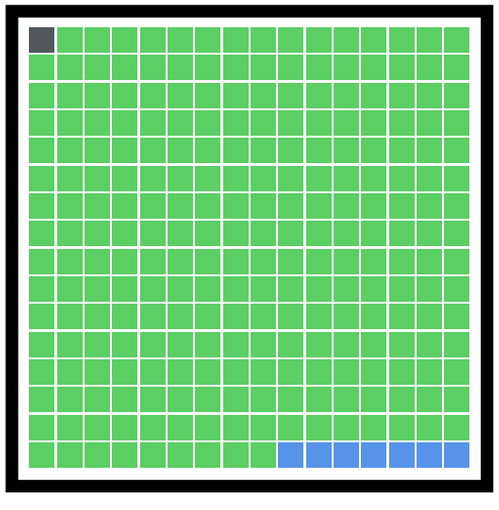
~100kB

Each ▪ is
32 bytes

Each □ is
8kB
(1 page)



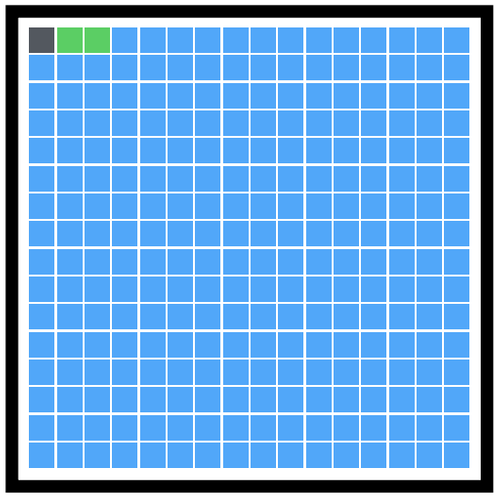




CREATE TABLE test(id SERIAL, data text);

INSERT INTO test VALUES ('short');

INSERT INTO test VALUES (‘longlonglonglonglong longlonglonglonglonglonglonglonglonglonglong longlonglonglonglonglonglonglonglonglonglong longlonglonglonglong');



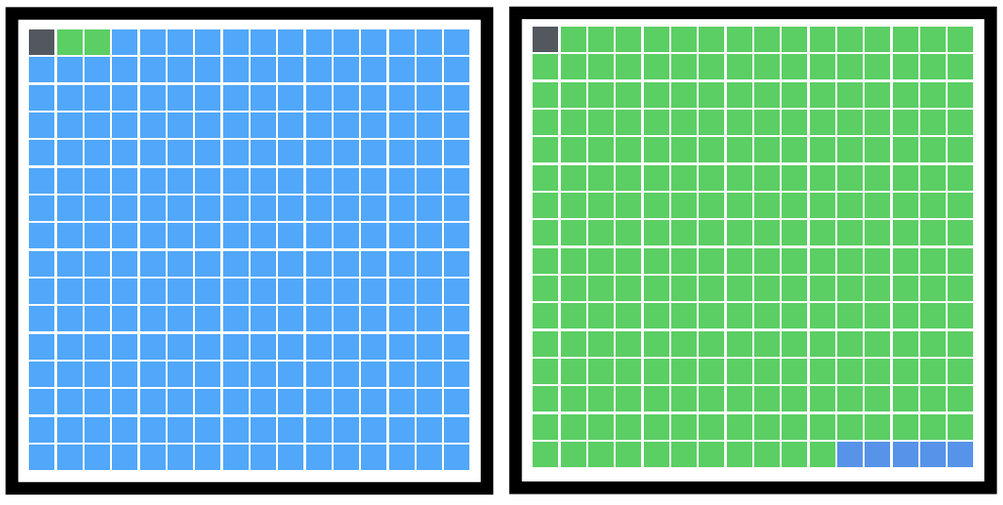
INSERT INTO test VALUES ('longlonglonglong longlonglonglonglonglonglonglonglonglong longlonglonglonglonglonglonglonglonglong longlonglonglonglong');

48 more times

We now have 50 live tuples...

```
=# SELECT * FROM pgstattuple('test');
-[ RECORD 1 ]------+------
table_len          | 8192
tuple_count        | 50
tuple_len          | 7874
tuple_percent      | 96.12
dead_tuple_count   | 0
dead_tuple_len     | 0
dead_tuple_percent | 0
free_space         | 84
free_percent       | 1.03
```

live tuples / page density = 50

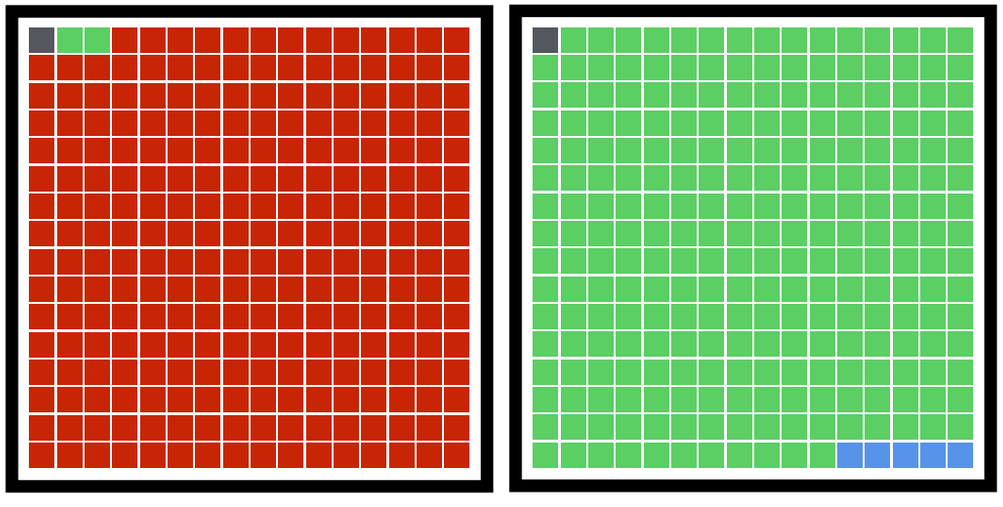


INSERT INTO test VALUES ('longlonglonglong longlonglonglonglonglonglonglonglonglong longlonglonglonglonglonglonglonglonglong longlonglonglonglong');

We now have 51 live tuples.

```
=# SELECT * FROM pgstattuple('test');
-[ RECORD 1 ]------+------
table_len          | 16384
tuple_count        | 51
tuple_len          | 8034
tuple_percent      | 49.04
dead_tuple_count   | 0
dead_tuple_len     | 0
dead_tuple_percent | 0
free_space         | 8084
free_percent       | 49.34
```

live tuples / page density = 25.5

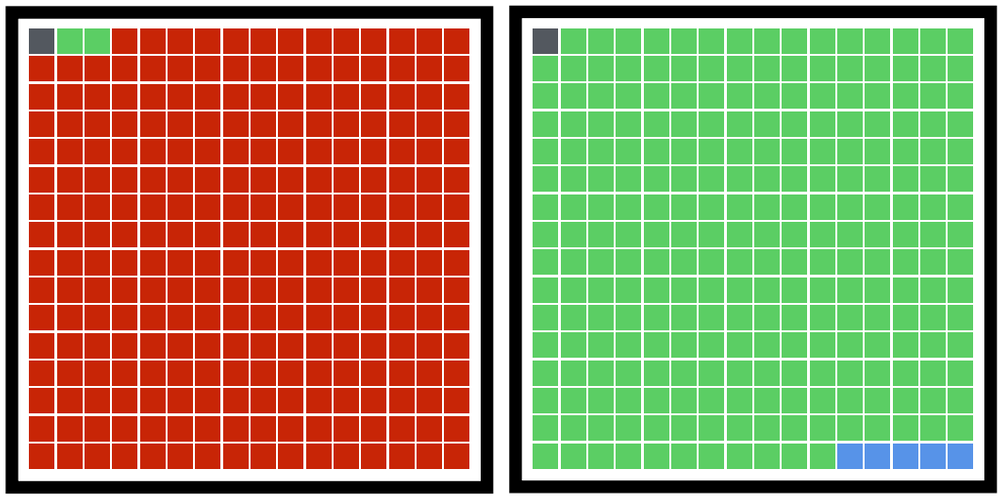


DELETE FROM test WHERE id < 51

We now have 1 live tuple and 50 dead tuples.

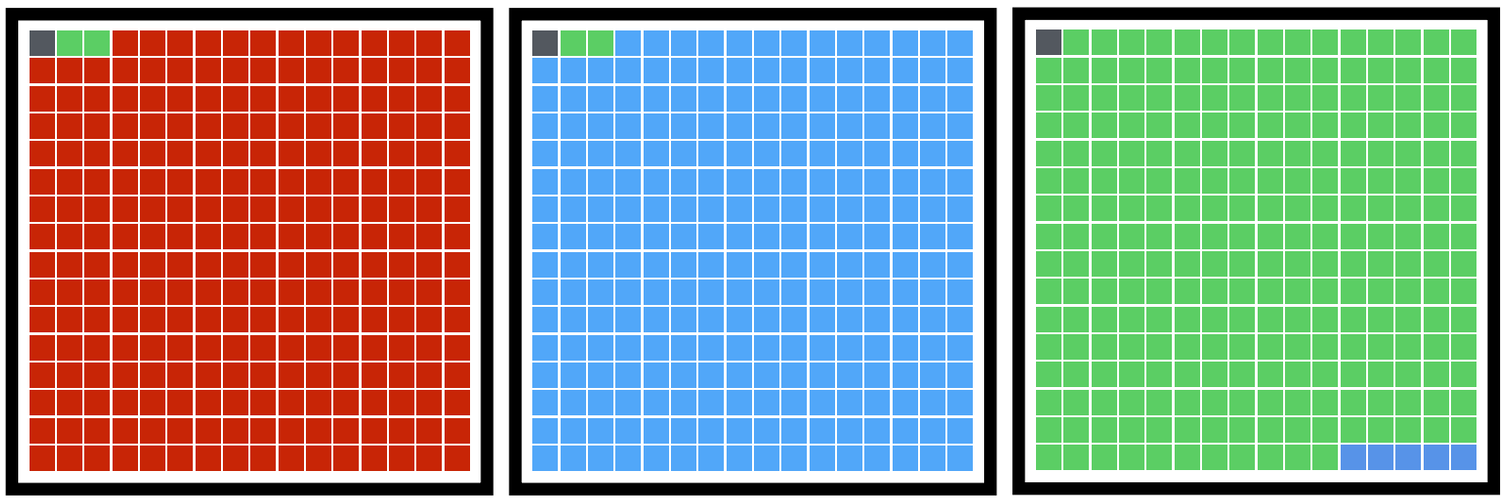
```
=# SELECT * FROM pgstattuple('test');
-[ RECORD 1 ]------+------
table_len          | 16384
tuple_count        | 1
tuple_len          | 160
tuple_percent      | 0.98
dead_tuple_count   | 50
dead_tuple_len     | 7874
dead_tuple_percent | 48.06
free_space         | 8084
free_percent       | 49.34
```

live tuples / page density = 0.5

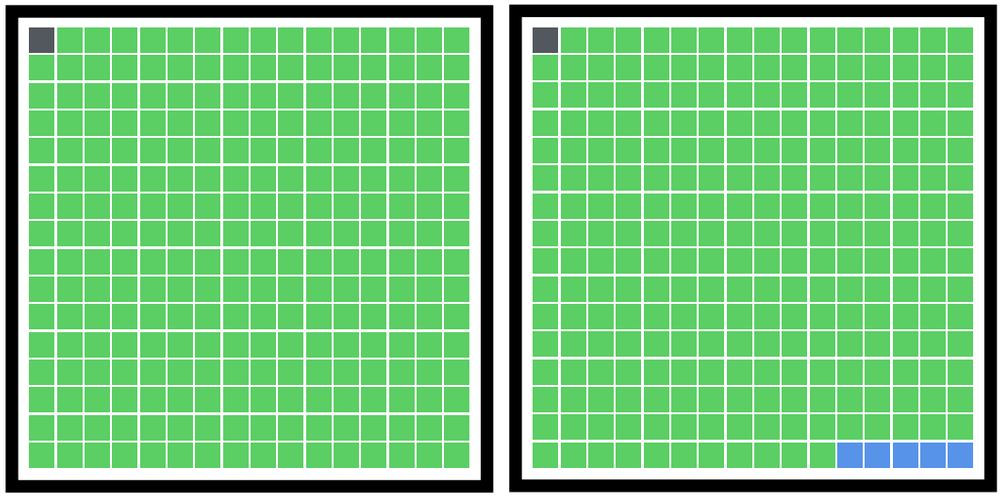


Do we get VACUUMed before the next 50 inserts?

If not, we keep growing the table...



...

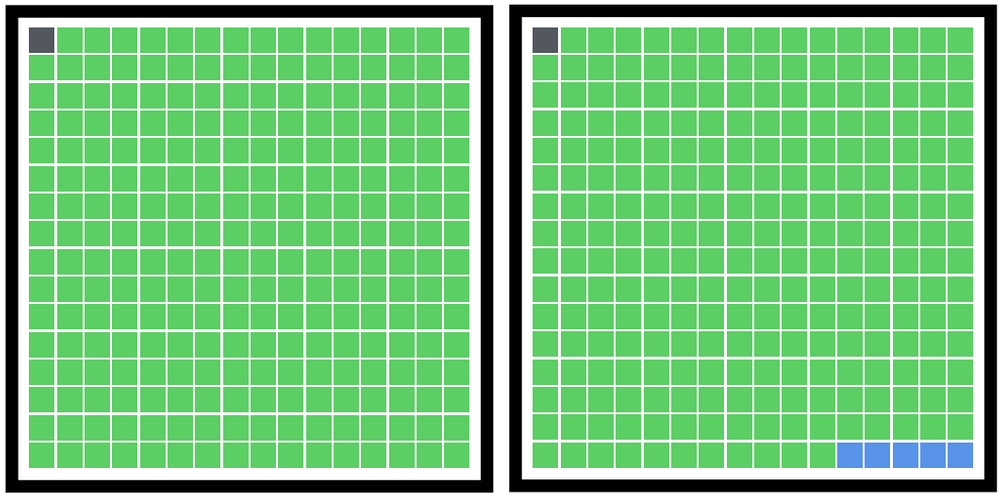


VACUUM test;

We now have 1 live tuple.

```
=# SELECT * FROM pgstattuple('test');
-[ RECORD 1 ]------+------
table_len          | 16384
tuple_count        | 1
tuple_len          | 160
tuple_percent      | 0.98
dead_tuple_count   | 0
dead_tuple_len     | 0
dead_tuple_percent | 0
free_space         | 16160
free_percent       | 98.63
```

live tuples / page density = 0.5



VACUUM test;

We now have 1 live tuple.

```
=# SELECT * FROM pgstattuple('test');
-[ RECORD 1 ]------+------
table_len          | 16384
tuple_count        | 1
tuple_len          | 160
tuple_percent      | 0.98
dead_tuple_count   | 0
dead_tuple_len     | 0
dead_tuple_percent | 0
free_space         | 16160
free_percent       | 98.63
```

live tuples / page density = 0.5

Bloat!

A bad bloat problem on our own database, produced by a bug that caused a lot of UPDATEs (that were not HOT).

```
=> SELECT * FROM pgstattuple('issue_references');
-[ RECORD 1 ]------+-------------
table_len          | 501153767424
tuple_count        | 22117518
tuple_len          | 8623158256
tuple_percent      | 1.72
dead_tuple_count   | 676
dead_tuple_len     | 262123
dead_tuple_percent | 0
free_space         | 476997629164
free_percent       | 95.18
```

1.7% of data, in a 500GB table...

Table bloat =
A less than optimal “page density”

(i.e. how many live tuples are on each page,
vs how many could potentially fit there)

Detecting Table Bloat

ioguix / pgsql-bloat-estimation Public

Code Issues 2 Pull requests 2 Actions Projects Wiki Security Insights

master pgsql-bloat-estimation / table / table_bloat.sql

63 lines (63 loc) · 3.32 KB

```
1   /* WARNING: executed with a non-superuser role, the query inspect only tables and materialized view (9.3+) you are granted to read.
2   * This query is compatible with PostgreSQL 9.0 and more
3   */
4   SELECT current_database(), schemaname, tblname, bs*tblpages AS rea
5     (tblpages-est_tblpages)*bs AS extra_size,
6     CASE WHEN tblpages > 0 AND tblpages - est_tblpages > 0
7       THEN 100 * (tblpages - est_tblpages)/tblpages::float
8       ELSE 0
9     END AS extra_pct, fillfactor,
10    CASE WHEN tblpages - est_tblpages_ff > 0
11      THEN (tblpages-est_tblpages_ff)*bs
12      ELSE 0
13    END AS bloat_size,
14    CASE WHEN tblpages > 0 AND tblpages - est_tblpages_ff > 0
15      THEN 100 * (tblpages - est_tblpages_ff)/tblpages::float
16      ELSE 0
17    END AS bloat_pct, is_na
18    -- , tpl_hdr_size, tpl_data_size, (pst).free_percent + (pst).dead_tuple_percent AS real_frag -- (DEBUG INFO)
19  FROM (
20    SELECT ceil( reltuples / ( (bs-page_hdr)/tpl_size ) ) + ceil( toasttuples / 4 ) AS est_tblpages,
21      ceil( reltuples / ( (bs-page_hdr)*fillfactor/(tpl_size*100) ) ) + ceil( toasttuples / 4 ) AS est_tblpages_ff,
22      tblpages, fillfactor, bs, tblid, schemaname, tblname, heappages, toastpages, is_na
```

Bloat Estimation Queries

The idea is simple.

Based on the:

- Number of tuples on the table (per ANALYZE / VACUUM)
- Average column width (as collected by ANALYZE)
- Alignment of columns (as defined by the platform)
- FILLFACTOR (as configured on the table)

What is the **optimal size of the table**,
if all pages were packed efficiently?
(and how does current table size compare to that?)

pganalyze

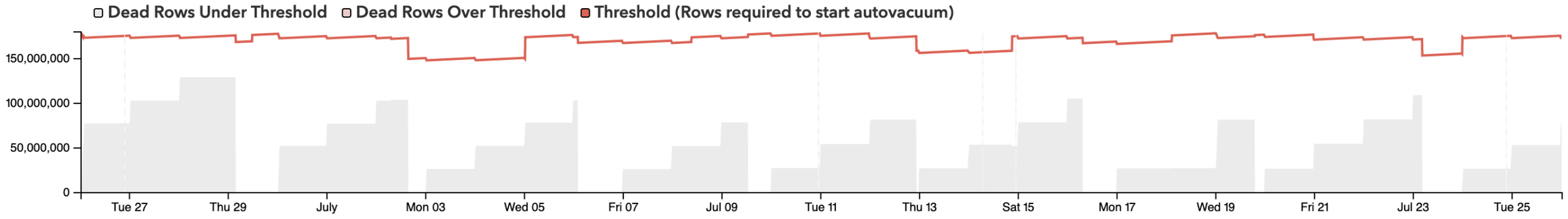
pganalyze VACUUM Advisor

Table: public.snapshot_benchmarks

Statistics | Partitions | Queries | Columns | Indexes | Constraints | VACUUM/ANALYZE Activity

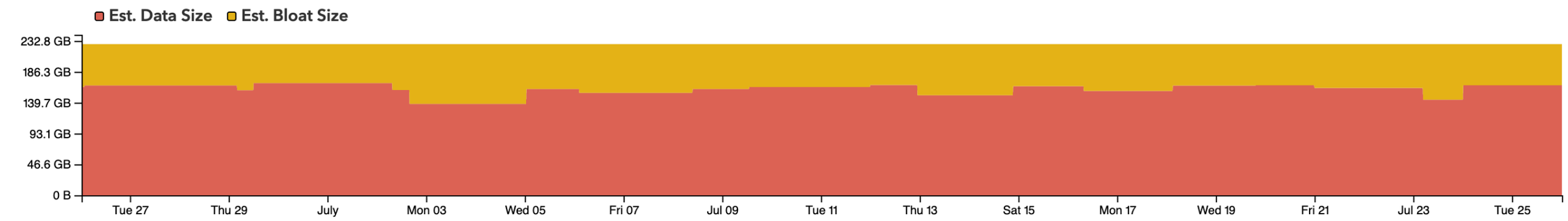
VACUUM Activity

Avg. VACUUM Duration	An hour	**Number of VACUUMs / day**	0
Autovacuum Enabled	Yes	**Autovacuum Freeze Max Age**	400,000,000 / 400,000,000
Autovacuum Cost Delay	2 ms	**Autovacuum Cost Limit**	1,800
Autovacuum Threshold	50	**Autovacuum Scale Factor**	10%



Estimated Table Bloat

Estimated Data Size	165.9 GB	**Estimated Table Bloat**	62.2 GB



Best practices for bloat estimation:

- Start with fast estimates, use pgstattuple to verify
- Run ANALYZE before running bloat estimation queries
- Avoid if you have very large columns (e.g. JSONB), since the average column width is likely to be incorrect
- Pay close attention to the math behind index bloat estimation, it can often times be completely off
  (better to test a REINDEX on a copy of the database)

Fixing Table Bloat

Using VACUUM on a bloated table does not fix the bloat.

VACUUM does not move around rows in a table

(it just marks dead row space as reusable,
and pages as all-visible or all-frozen)

To fix bloat, use pg_repack/pg_squeeze!

It’s like VACUUM FULL, but made for people who are running production databases :-)

CREATE EXTENSION pg_repack; on your database
+ install pg_repack client side utilities on a virtual machine.

```
pg_repack -k -j 5
-h mydb.myaccount.us-east-1.rds.amazonaws.com -d mydb -U myuser
-t issue_references
```

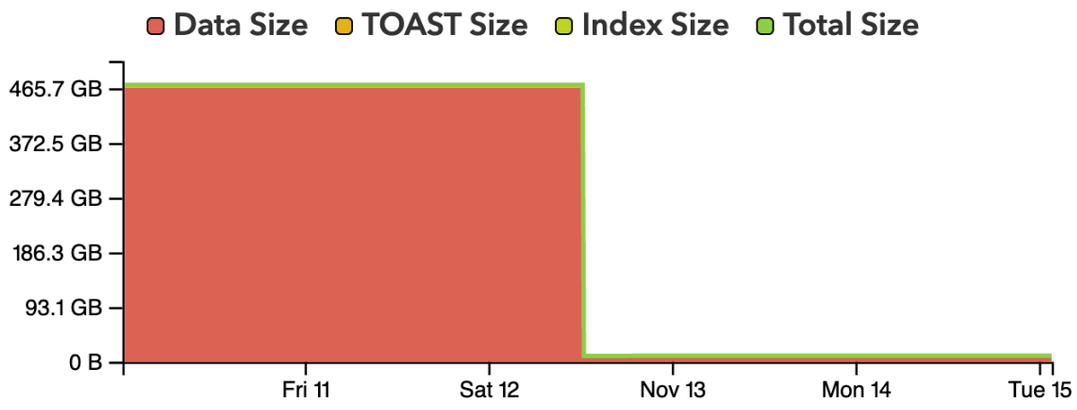
This is an online operation,
except for a short exclusive lock at the end
(similar to REINDEX CONCURRENTLY)

When VACUUM didn't do its job,

you probably need pg_repack/pg_squeeze to fix it.

```
-[ RECORD 1 ]------+-------------
table_len          | 501153767424
tuple_count        | 22117518
tuple_len          | 8623158256
tuple_percent      | 1.72
dead_tuple_count   | 676
dead_tuple_len     | 262123
dead_tuple_percent | 0
free_space         | 476997629164
free_percent       | 95.18
```

```
-[ RECORD 1 ]------+-----------
table_len          | 9559777280
tuple_count        | 22629063
tuple_len          | 9047624099
tuple_percent      | 94.64
dead_tuple_count   | 166446
dead_tuple_len     | 132483269
dead_tuple_percent | 1.39
free_space         | 183796600
free_percent       | 1.92
```

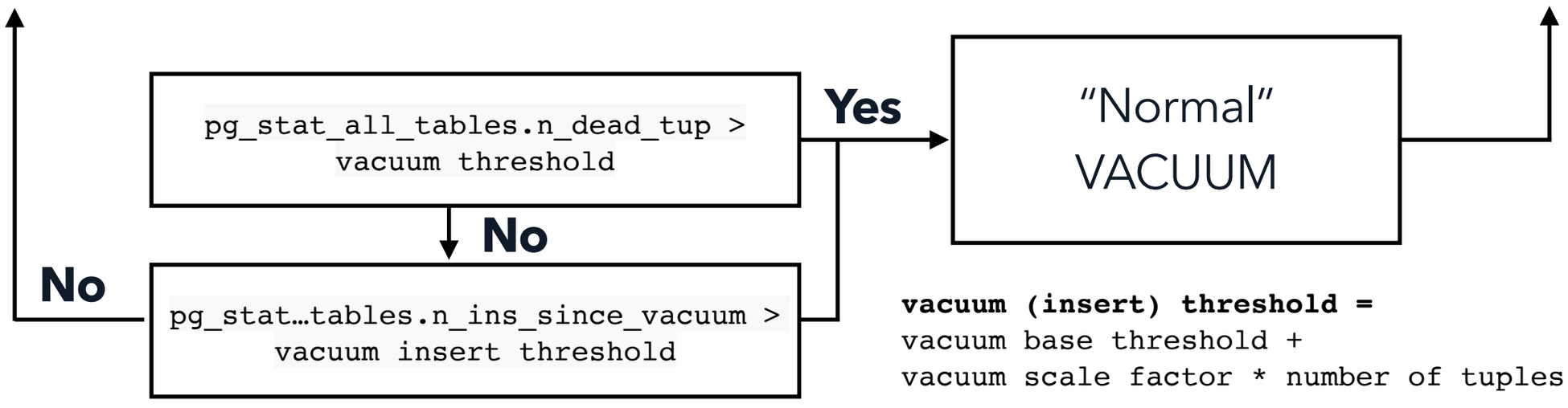


pganalyze

Preventing New Table Bloat

Autovacuum Scheduling

Thresholds



vacuum (insert) threshold =
vacuum base threshold +
vacuum scale factor * number of tuples

Understanding autovacuum scheduling

pganalyze

Server: prod-db-main (Amazon RDS) | Database: pgaweb | Last 7 days

ORGANIZATION: pganalyze

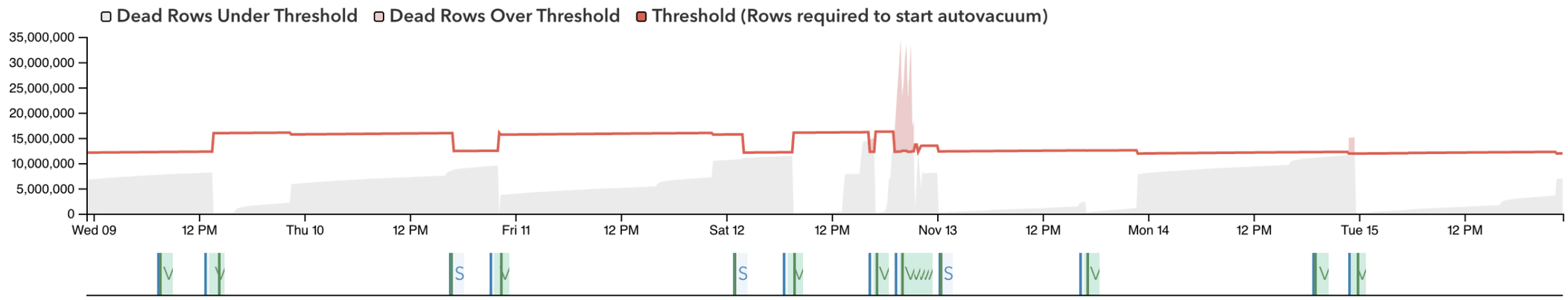
- Dashboard
- Query Performance
- Index Advisor
- EXPLAIN Plans
- Schema Statistics
- Log Insights
- Connections
- VACUUM Activity
- Config Tuning
- System
- Alerts & Check-Up
- Settings

Table: public.queries

Statistics | Partitions | Queries | Columns | Indexes | Constraints | VACUUM/ANALYZE Activity

VACUUM Activity

Avg. VACUUM Duration	39 minutes	**Number of VACUUMs / day**	2
Autovacuum Enabled	Yes	**Autovacuum Freeze Max Age**	200,000,000 / 400,000,000
Autovacuum Cost Delay	2 ms	**Autovacuum Cost Limit**	1,800
Autovacuum Threshold	50	**Autovacuum Scale Factor**	10%



ID	Started By	TOAST	Time Started	Time Finished
16684947100040816	autovacuum	No	Nov 14 10:45:09pm PST	Nov 14 11:42:21pm PST
16684800400037688	autovacuum	Yes	Nov 14 06:40:39pm PST	Nov 14 06:52:41pm PST
16683845710046576	autovacuum	No	Nov 13 04:09:30pm PST	Nov 13 04:57:10pm PST
16683268450026426	autovacuum	Yes	Nov 13 12:07:24am PST	Nov 13 12:16:30am PST
16683181230017307	autovacuum	No	Nov 12 09:42:03pm PST	Nov 12 10:01:50pm PST

Understanding autovacuum scheduling

pganalyze

Server: prod-db-main (Amazon RDS) | Database: pgaweb | Last 7 days

ORGANIZATION: pganalyze

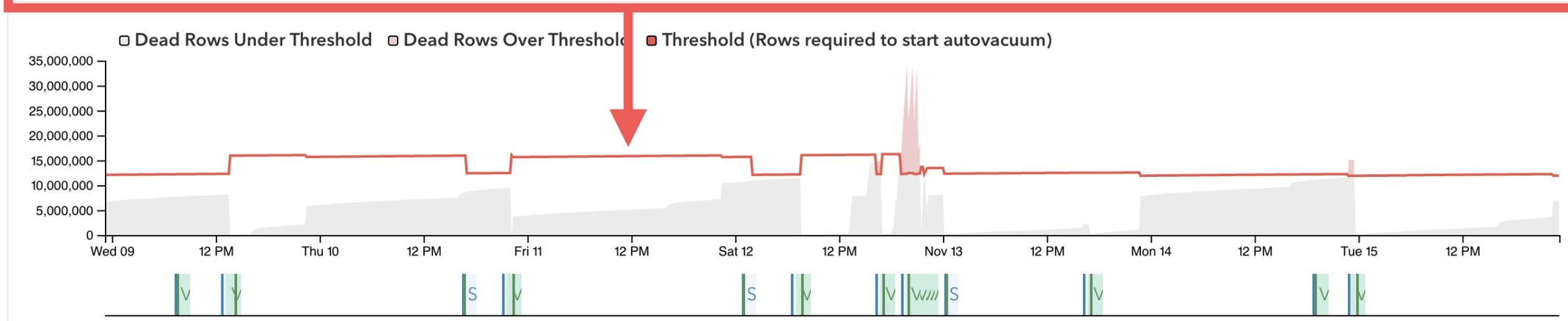
- Dashboard
- Query Performance
- Index Advisor
- EXPLAIN Plans
- Schema Statistics
- Log Insights
- Connections
- VACUUM Activity
- Config Tuning
- System
- Alerts & Check-Up
- Settings

Table: public.queries

Statistics | Partitions | Queries | Columns | Indexes | Constraints | VACUUM/ANALYZE Activity

VACUUM Activity

Avg. VACUUM Duration	39 minutes	**Number of VACUUMs / day**	2
Autovacuum Enabled	Yes	**Autovacuum Freeze Max Age**	200,000,000 / 400,000,000
Autovacuum Cost Delay	2 ms	**Autovacuum Cost Limit**	1,800
Autovacuum Threshold	50	**Autovacuum Scale Factor**	10%



ID	Started By	TOAST	Time Started	Time Finished
16684947100040816	autovacuum	No	Nov 14 10:45:09pm PST	Nov 14 11:42:21pm PST
16684800400037688	autovacuum	Yes	Nov 14 06:40:39pm PST	Nov 14 06:52:41pm PST
16683845710046576	autovacuum	No	Nov 13 04:09:30pm PST	Nov 13 04:57:10pm PST
16683268450026426	autovacuum	Yes	Nov 13 12:07:24am PST	Nov 13 12:16:30am PST
16683181230017307	autovacuum	No	Nov 12 09:42:03pm PST	Nov 12 10:01:50pm PST

Understanding autovacuum scheduling

pganalyze

Server: prod-db-main (Amazon RDS) | Database: pgaweb | Last 7 days

ORGANIZATION: pganalyze

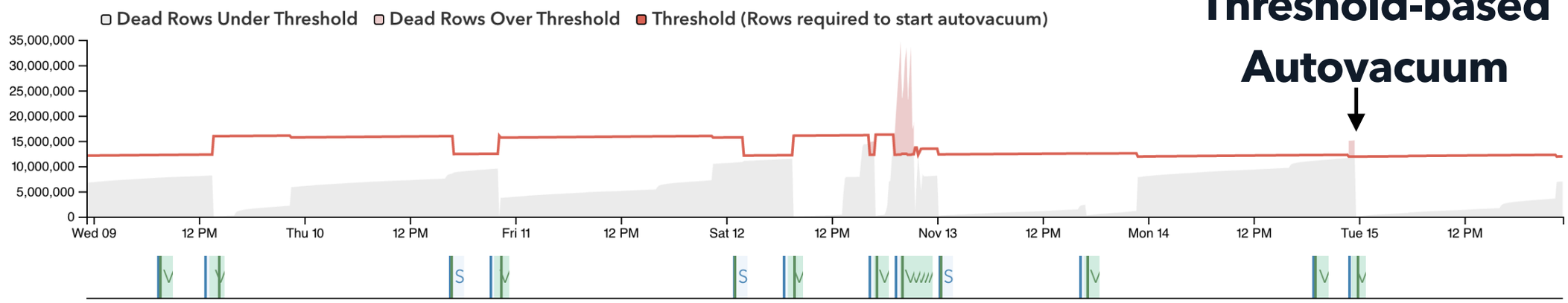
- Dashboard
- Query Performance
- Index Advisor
- EXPLAIN Plans
- Schema Statistics
- Log Insights
- Connections
- VACUUM Activity
- Config Tuning
- System
- Alerts & Check-Up
- Settings

Table: public.queries

Statistics | Partitions | Queries | Columns | Indexes | Constraints | VACUUM/ANALYZE Activity

VACUUM Activity

Avg. VACUUM Duration	39 minutes	**Number of VACUUMs / day**	2
Autovacuum Enabled	Yes	**Autovacuum Freeze Max Age**	200,000,000 / 400,000,000
Autovacuum Cost Delay	2 ms	**Autovacuum Cost Limit**	1,800
Autovacuum Threshold	50	**Autovacuum Scale Factor**	10%



Threshold-based Autovacuum

Frozen XID-based Autovacuum

ID	Started By	TOAST	Time Started	Time Finished
16684947100040816	autovacuum	No	Nov 14 10:45:09	2:21pm PST
16684800400037688	autovacuum	Yes	Nov 14 06:40:39	2:41pm PST
16683845710046576	autovacuum	No	Nov 13 04:09:30	7:10pm PST
16683268450026426	autovacuum	Yes	Nov 13 12:07:24	6:30am PST
16683181230017307	autovacuum	No	Nov 12 09:42:03pm PST	Nov 12 10:01:50pm PST

You can tune autovacuum thresholds

on a per-table basis

```
ALTER TABLE test SET
(autovacuum_vacuum_scale_factor = 0.05);
```

Tuning Tip:

If you have a **large table with a small active portion**, turn off the % based scale factor, and only use the threshold setting.

```
ALTER TABLE test SET
(autovacuum_vacuum_scale_factor = 0,
autovacuum_vacuum_threshold = 10000);
```

pganalyze

pganalyze VACUUM Advisor

Issue #138: VACUUM: Bloat - Insufficient VACUUM Frequency

Overview

Severity
Info

Check Frequency
Daily

Last Updated
2021-03-22 04:00:00pm UTC

State
Acknowledged

Re-open

Description
Insufficient vacuuming is causing avoidable growth on table my_table

Guidance

```
ALTER TABLE my_schema.my_table SET
 (autovacuum_vacuum_scale_factor = 0.01);
```

Copy ALTER TABLE command

Avoidable table growth due to insufficient VACUUM frequency was detected. You can adjust the frequency of autovacuum by changing relevant config settings. Lowering `autovacuum_vacuum_threshold` or `autovacuum_vacuum_scale_factor` will increase the frequency of autovacuum in general.

› How to apply changes

› What to check following ALTER TABLE

To confirm the impact of changes, it is also recommended to run pg_repack to reclaim disk space. You can check out a graph of estimated bloat over time on the VACUUM/ANALYZE Activity page.

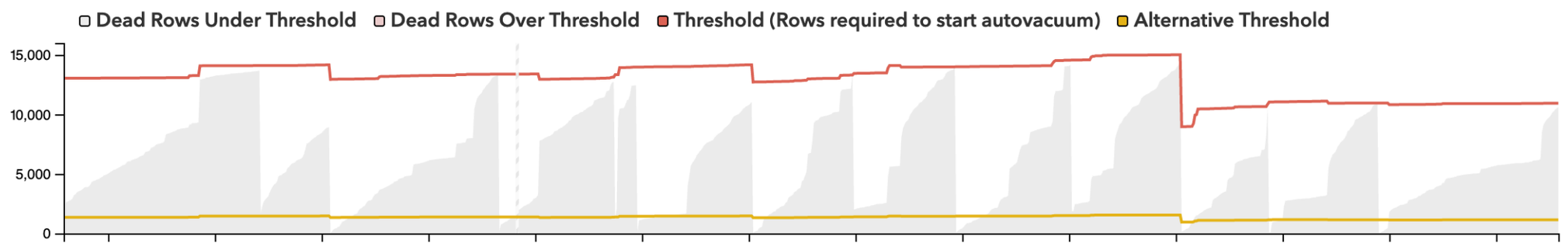
Recommendation

	Current	Recommended
autovacuum_vacuum_threshold	50	50
autovacuum_vacuum_scale_factor	0.10	0.01

Estimated Improvement

	Current	Estimate
Total autovacuum count	7	68
Avoidable growth		
Rows	58,527	30,966
Percentage	25.9%	13.7%
Bytes	11.1 MB	5.8 MB

VACUUM Activity

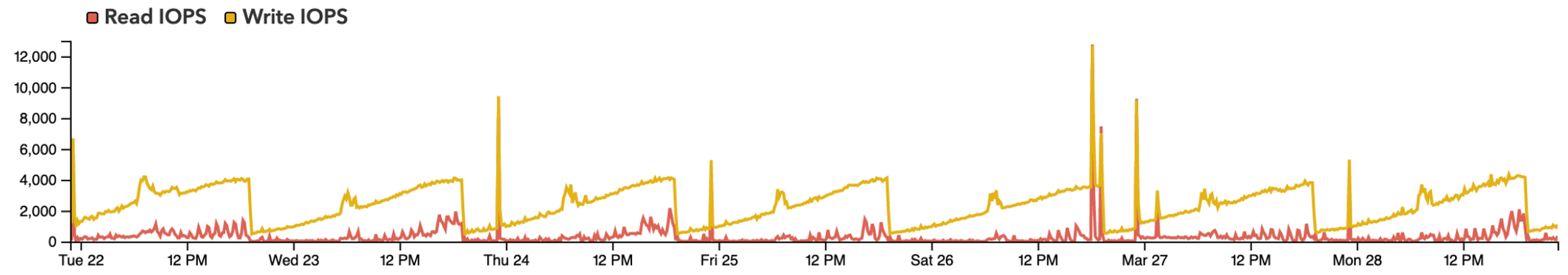


Index Bloat

Bad Index Structure =

Lots Of Unnecessary Index Pages (and I/O!)

IOPS



pganalyze

```
CREATE TABLE public.log_line_stats_7d (
    log_line_id uuid DEFAULT public.gen_random_uuid() NOT NULL,
    server_id uuid NOT NULL,
    occurred_at_10min timestamp without time zone NOT NULL,
    occurred_at_1h timestamp without time zone NOT NULL,
    log_classification integer NOT NULL,
    database_id bigint,
    postgres_role_id uuid,
    schema_table_id bigint,
    query_fingerprint bytea
)
PARTITION BY RANGE (occurred_at_10min);
```

Use ULIDs for logs to reduce index write overhead #1942

Edit | Code

Merged **seanlinsley** merged 4 commits into `main` from `logs-ulid` on Apr 22

Conversation 14 | Commits 4 | Checks 4 | Files changed 5 | +182 −9

seanlinsley commented on Apr 11

https://brandur.org/nanoglyphs/026-ids#ulids
https://crates.io/crates/ulid

In the future we could further optimize storage by embedding `collected_at` inside the ULID, then extracting it with a SQL function when generating indexes and when reading from the table.

I'm not confident this will significantly improve the IO situation because there are much larger indexes than these, which one would expect take up more IO.

Indexes

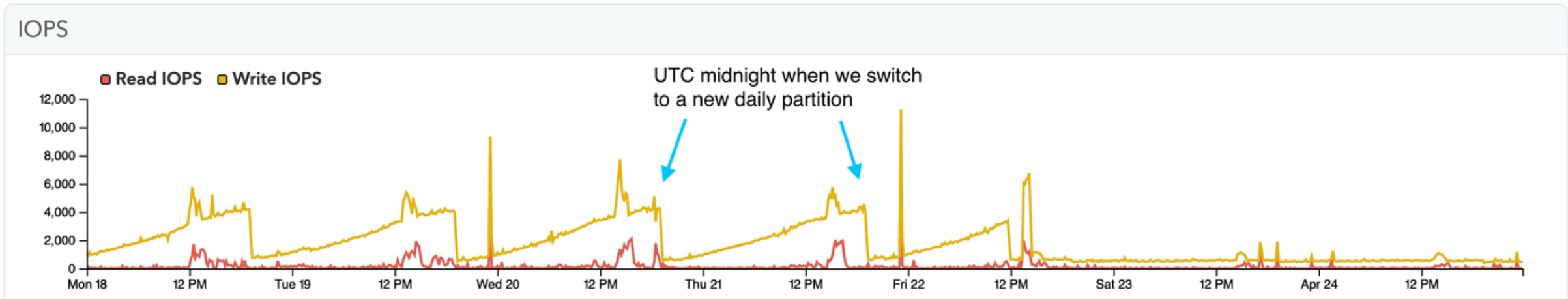
Name	Definition	Constraint	Valid?	First Seen	Size
log_lines_7d_20220407_pkey	btree (log_line_id)	PRIMARY KEY (log_line_id)	VALID	6 days ago	13.7 GB

Reviewers
lfittl
msakrejda

Assignees
No one—assign yourself

Labels
None yet

Projects



6,000 Write IOPS peak => ~1,000 Write IOPS peak

pganalyze

Or, sometimes, doing

REINDEX CONCURRENTLY

on a schedule (e.g. monthly)

is a good idea

Unused Indexes

Unused (non-UNIQUE) Indexes =

Added write I/O without any benefit

Index Advisor

Overview | Missing Indexes | Unused Indexes

Total Data Size
6.5 TB
▲ 157.6 GB

Total Index Size
6 GB

Table Writes
852,414
per minute

Avg. Index Write Overhead
0.55
index bytes per table byte

Opportunities for Database (20)

Search...

IMPACT	TABLE	INDEX	INDEX SIZE	LAST USED	TABLE WRITES / MIN	INDEX WRITE OVERHEAD
▮▮▮▮▮	public.indexing_engine_runs_35d	indexing_engine_runs_35d_20220815_table_id_run_at_idx (+3…	37.3 MB	2022-08-17	0.000	-0.89
▮▮▯▯▯	public.snapshots	index_snapshots_on_snapshot_id	52.8 GB	2021-10-27	17,047.690	-0.26
▮▮▯▯▯	public.query_stats_35d	query_stats_35d_20220827_collected_at_idx (+1 more)	6.8 GB	2022-09-04	0.000	-0.23
▮▯▯▯▯	public.schema_table_options	index_schema_table_options_on_invalidated_at_snapshot_id	23.3 MB	2022-03-16	3.800	-0.17
▮▯▯▯▯	public.replication_stats	index_replication_stats_on_snapshot_id	1.1 GB	2022-08-29	225.997	-0.16
▮▯▯▯▯	public.replication_follower_stats	index_replication_follower_stats_on_snapshot_id	427.1 MB	2022-08-29	139.795	-0.15
▮▯▯▯▯	public.schema_columns	index_schema_columns_on_invalidated_at_snapshot_id	3.3 GB	2022-08-23	1,534.581	-0.13
▮▯▯▯▯	public.postgres_settings	index_postgres_settings_on_invalidated_at_snapshot_id	39.5 MB	2022-08-29	7.556	-0.12
▮▯▯▯▯	public.issue_states	index_issue_states_on_user_id	1.2 GB	2022-07-22	10.669	-0.10
▮▯▯▯▯	public.system_snapshots	index_system_snapshots_on_snapshot_id (+1 more)	381.3 MB	2021-10-27	0.000	-0.09
▮▯▯▯▯	public.invoices	index_invoices_on_organization_id	88 kB	2022-09-02	0.000	-0.07

pganalyze

Postgres 17 adds a timestamp
to pg_stat_user_indexes
when an index was last used:

`idx_scan bigint` Number of index scans initiated on this index
`last_idx_scan timestamp with time zone` The time of the last scan on this index, based on the most recent transaction stop time

Other things to consider:

- Vacuuming more often can reduce I/O spikes

- Support HOT Updates by avoiding indexes on updated columns

- If you use GIN indexes watch out for the pending list

pganalyze

Optimizing Queries, Connections & Wait Events

The 101 of Tracking Query Performance

- **Use pg_stat_statements**

  to track aggregate statistics on finished queries over time

The 101 of Tracking Query Performance

- **Use pg_stat_statements**
  to track aggregate statistics on finished queries over time
- **Use pg_stat_activity**
  to track currently running queries and current lock issues

The 101 of Tracking Query Performance

- **Use pg_stat_statements**
  to track aggregate statistics on finished queries over time
- **Use pg_stat_activity**
  to track currently running queries and current lock issues
- **Use the Postgres logs**
  to track timed out and cancelled queries and historic locks

The 101 of Tracking Query Performance

- **Use pg_stat_statements**
  to track aggregate statistics on finished queries over time
- **Use pg_stat_activity**
  to track currently running queries and current lock issues
- **Use the Postgres logs**
  to track timed out and cancelled queries and historic locks
- **Use auto_explain**
  to track query plan outliers and identify bad query plans

Connection Pooling

The age-old problem:

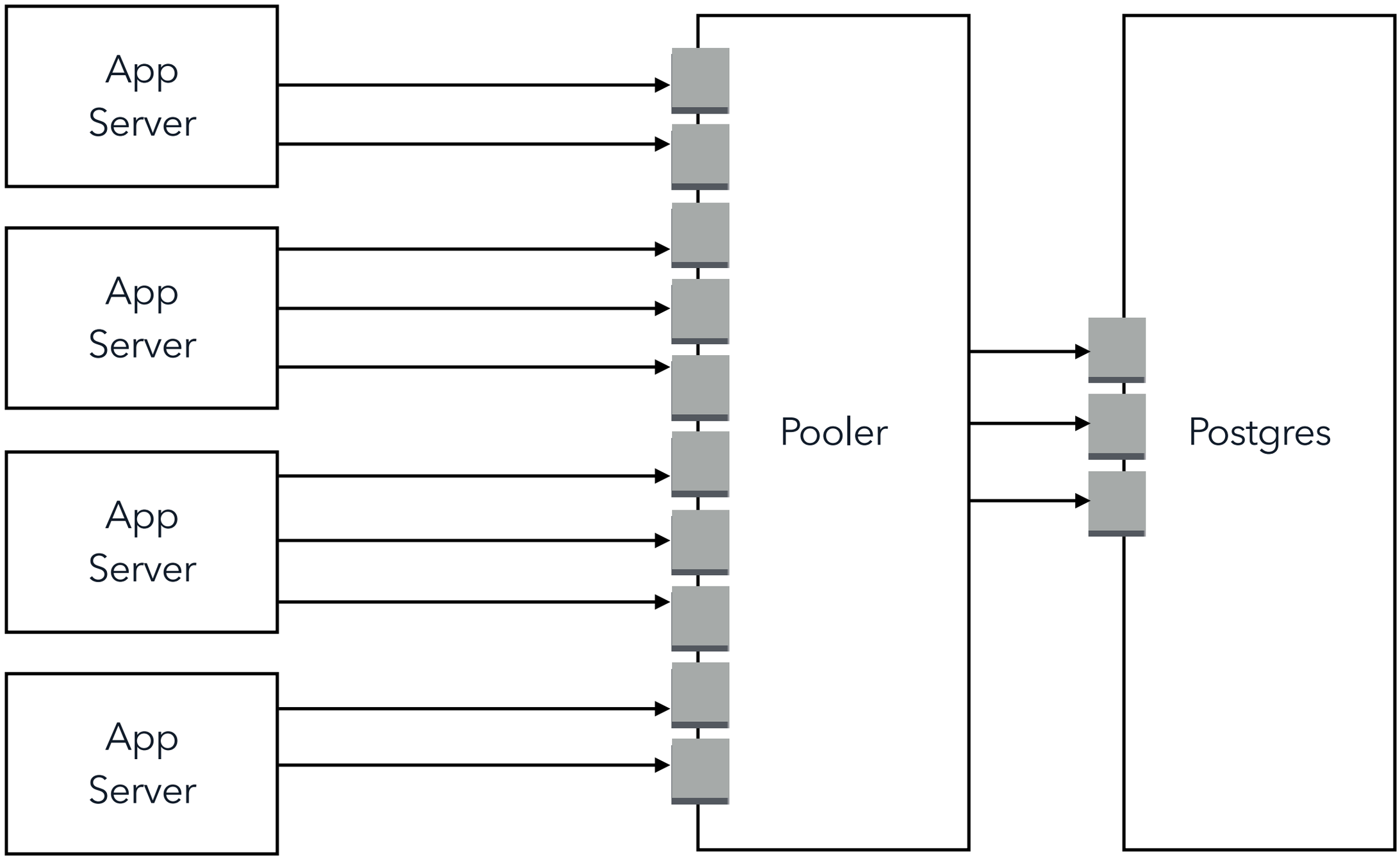
Postgres connections are expensive, because they are processes (not threads)

This is still true today, but the overhead of individual connection processes has been reduced in the last releases.

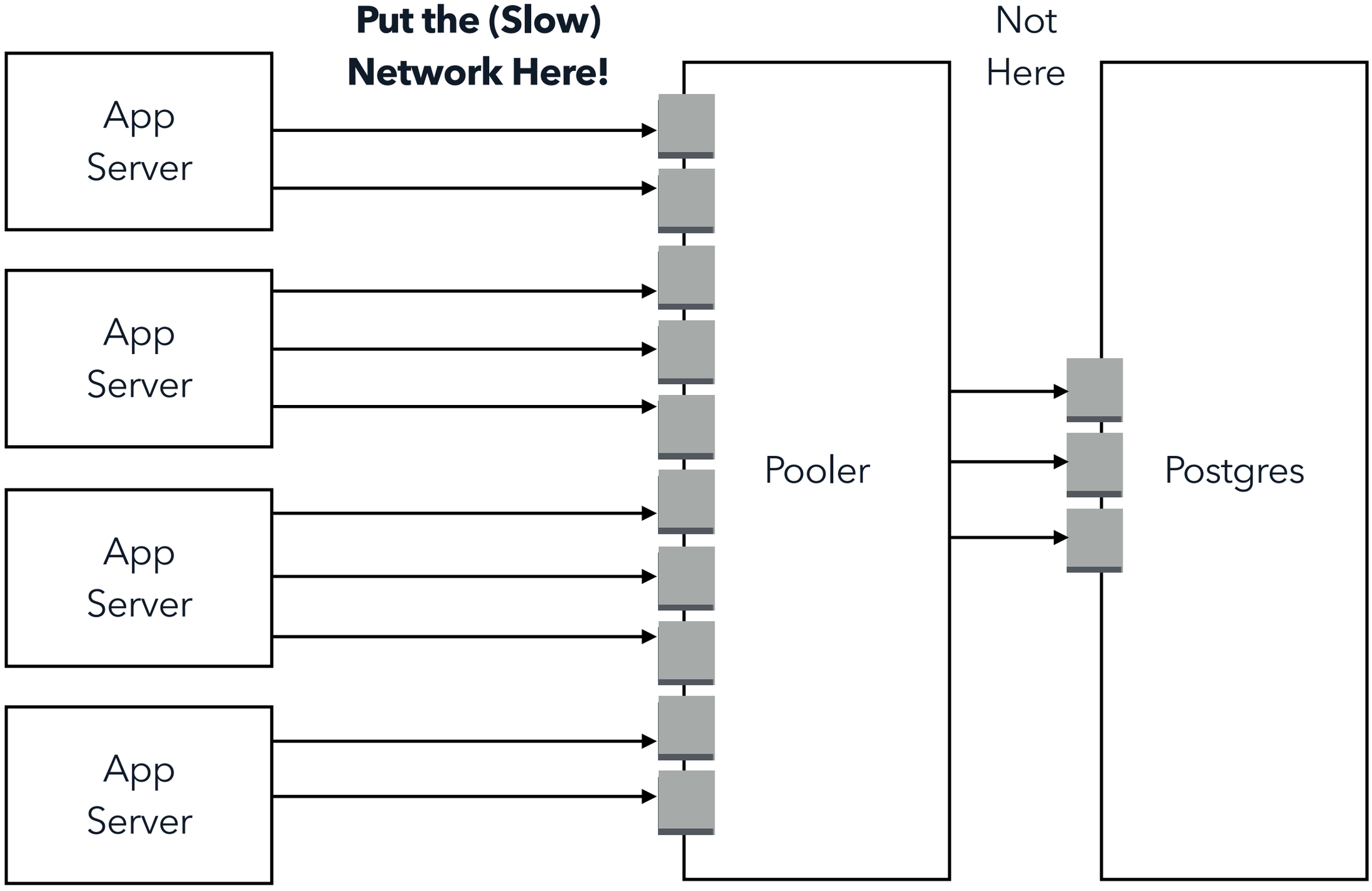
And there is an active effort towards multi-threading in Postgres.

If you have less than ~ 1000 max. idle connections, you don't need to worry (much).

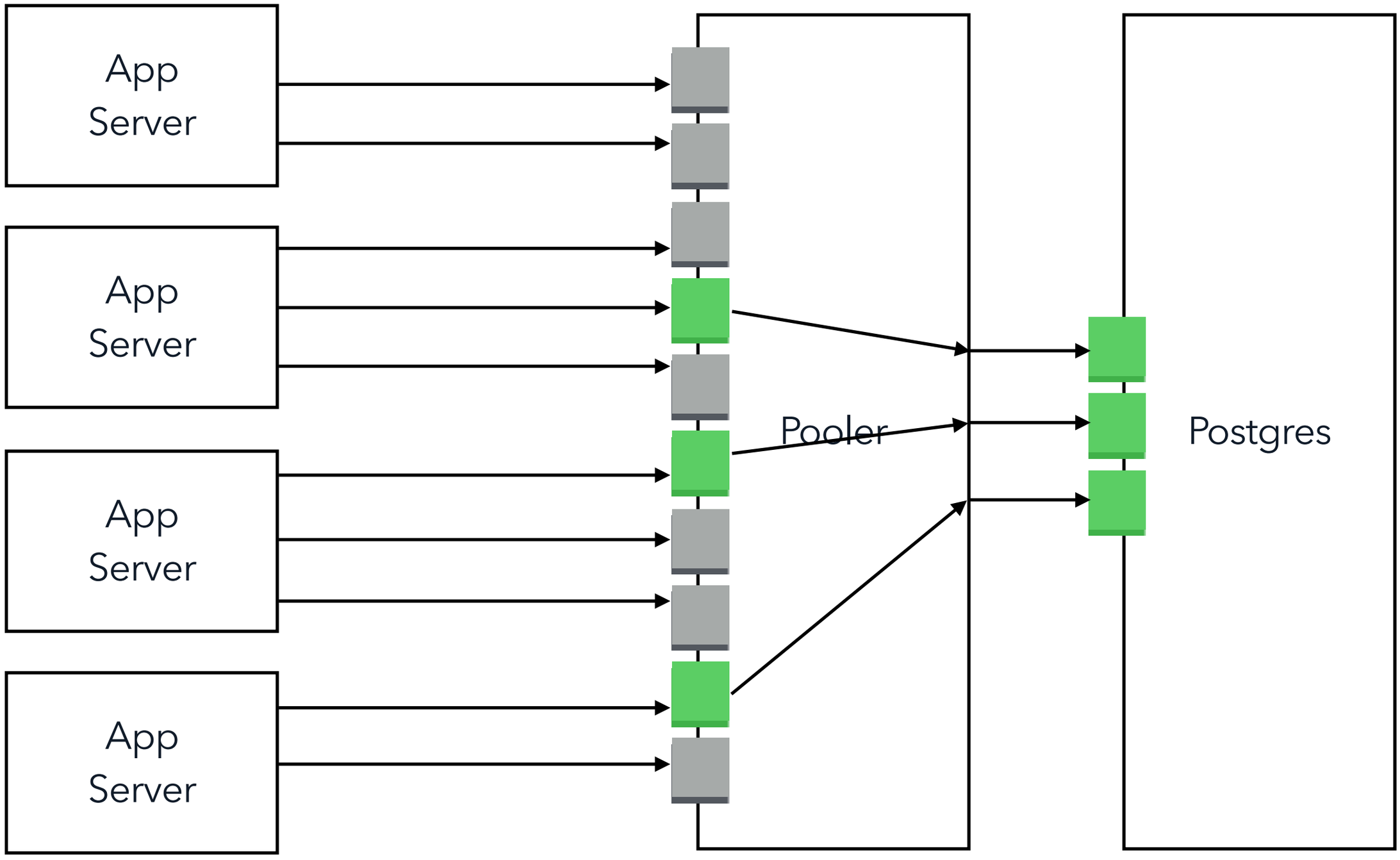
Beyond that, pgbouncer (or pgCat, etc) is essential.


App
Server
App
Server
App
Server
App
Server
Pooler
Postgres

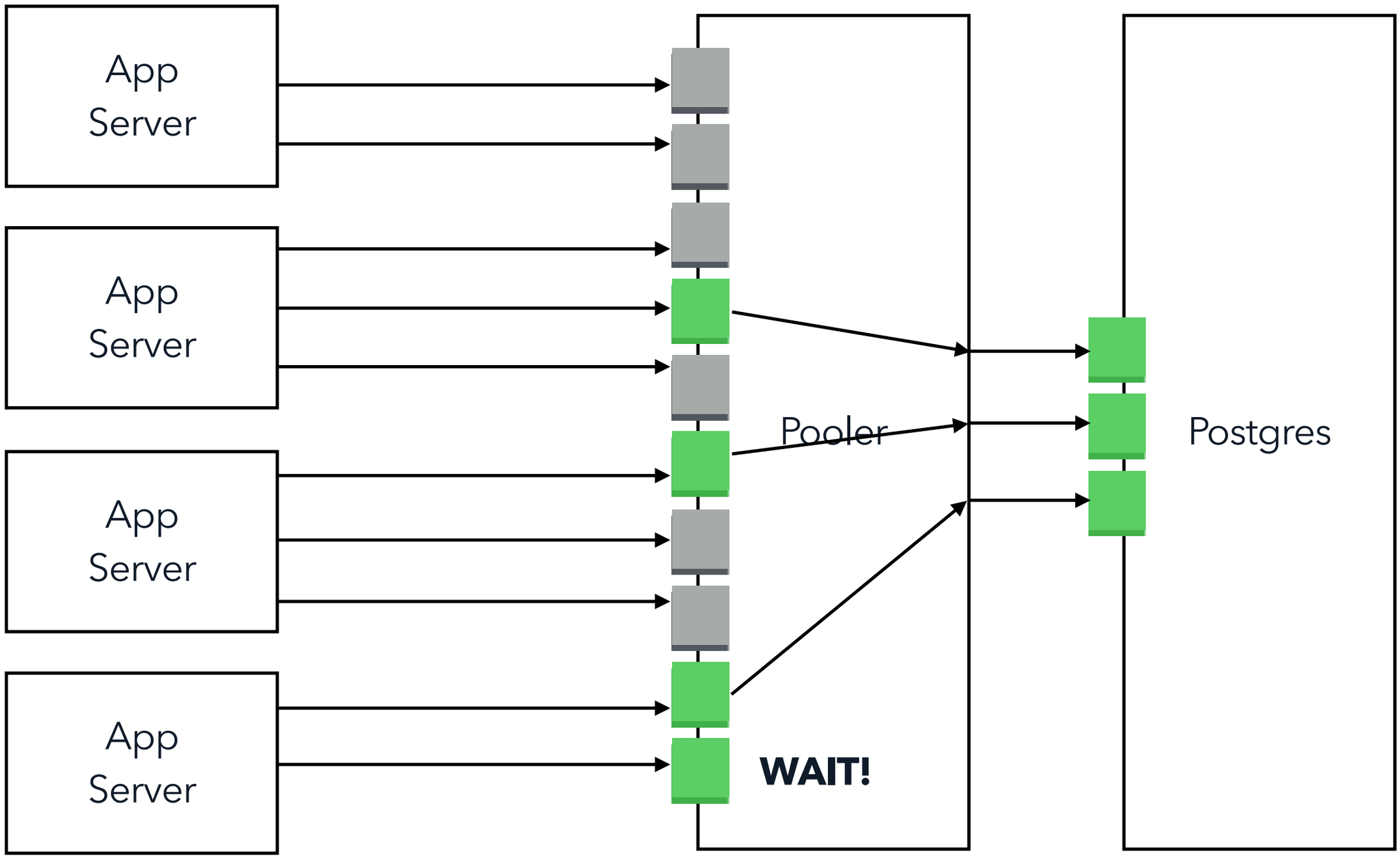
pganalyze


Put the (Slow) Network Here!
Not Here
App Server
App Server
App Server
App Server
Pooler
Postgres

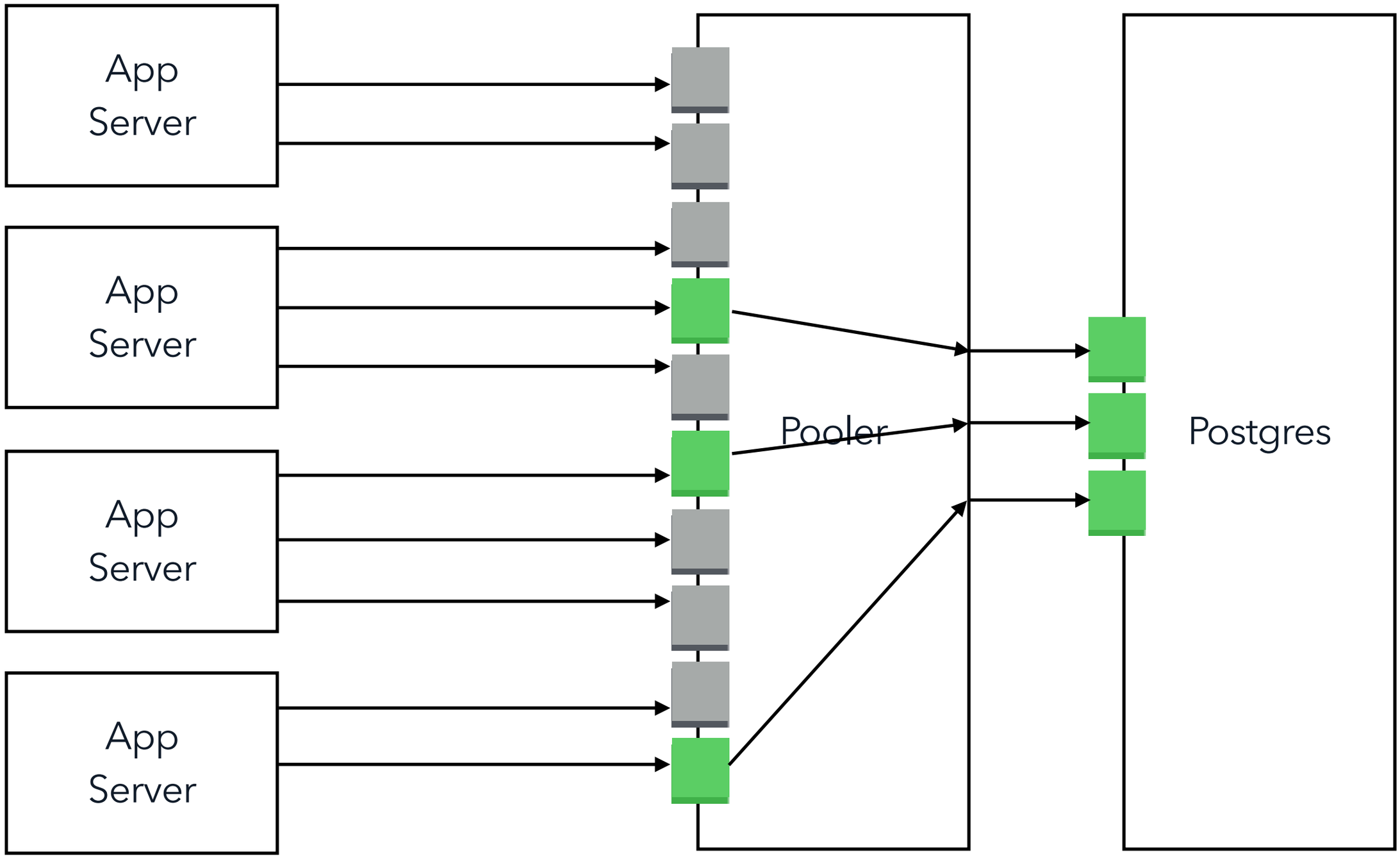
pganalyze


App
Server
App
Server
App
Server
App
Server
Pooler
Postgres

pganalyze


App
Server
App
Server
App
Server
App
Server
Pooler
WAIT!
Postgres

pganalyze

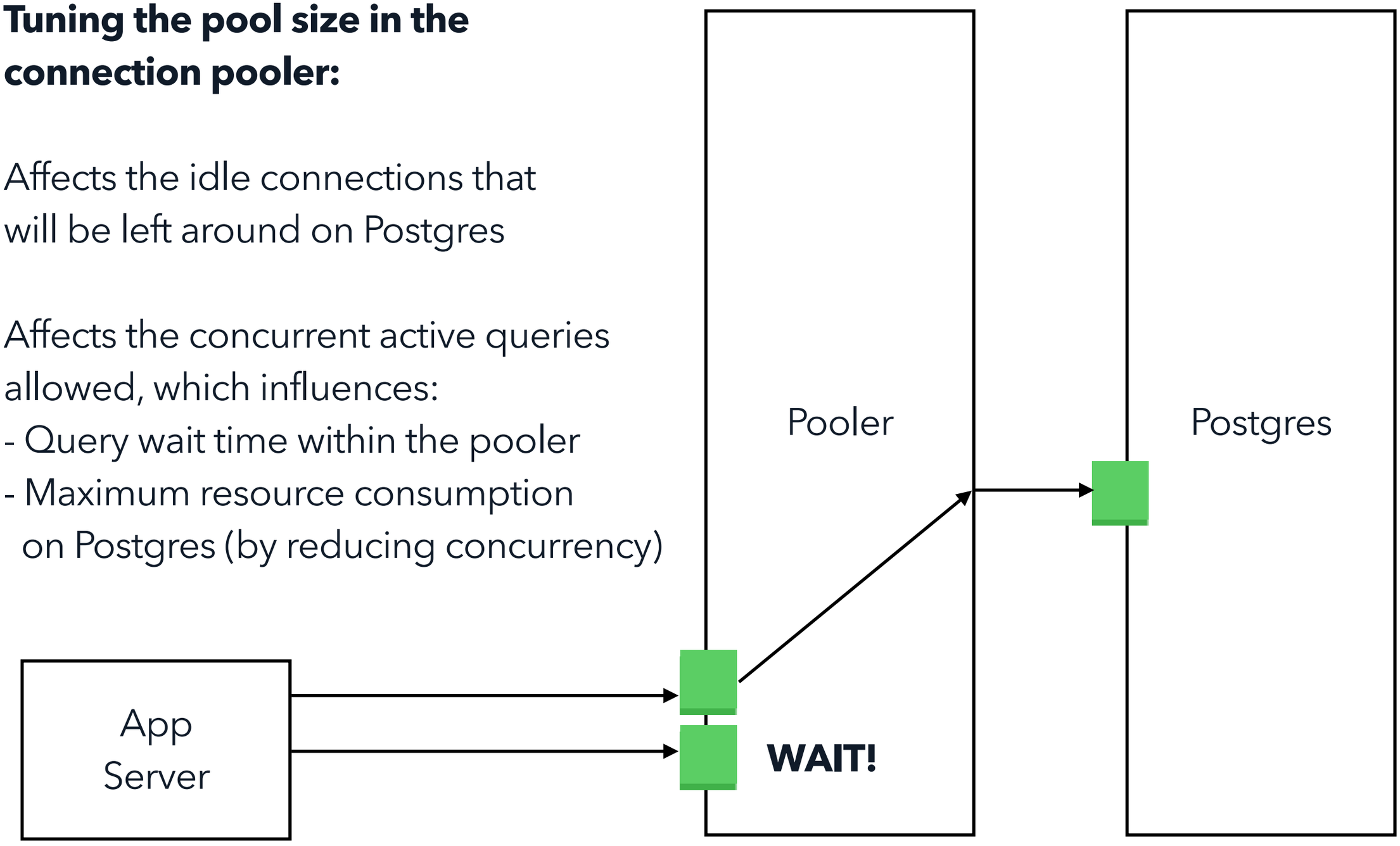

App
Server
App
Server
App
Server
App
Server
Pooler
Postgres

pganalyze

Tuning the pool size in the connection pooler:

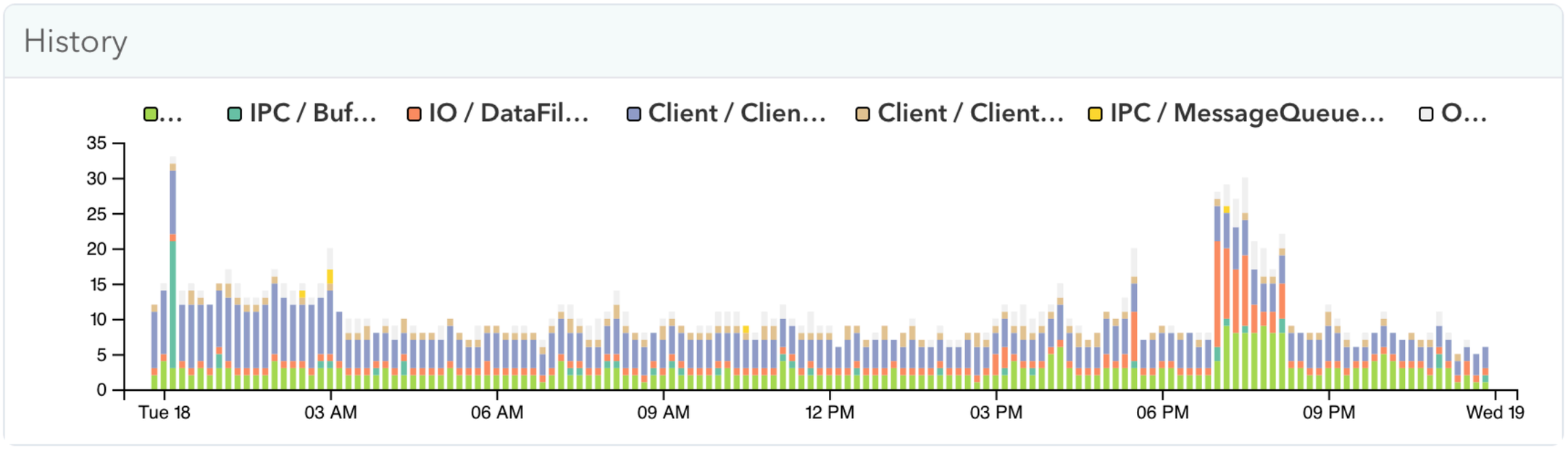
Affects the idle connections that will be left around on Postgres

Affects the concurrent active queries allowed, which influences:
- Query wait time within the pooler
- Maximum resource consumption on Postgres (by reducing concurrency)



pganalyze

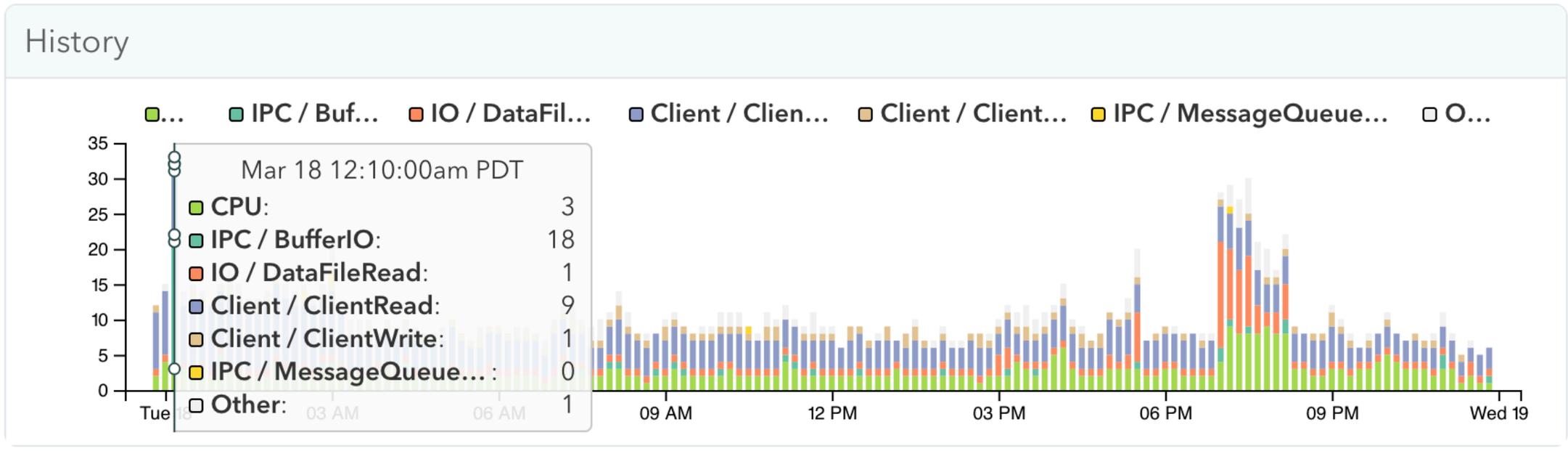
Wait Events



Wait Events are the best source to tell you where you have a bottleneck.

Sample pg_stat_activity over time to get this data.

pganalyze

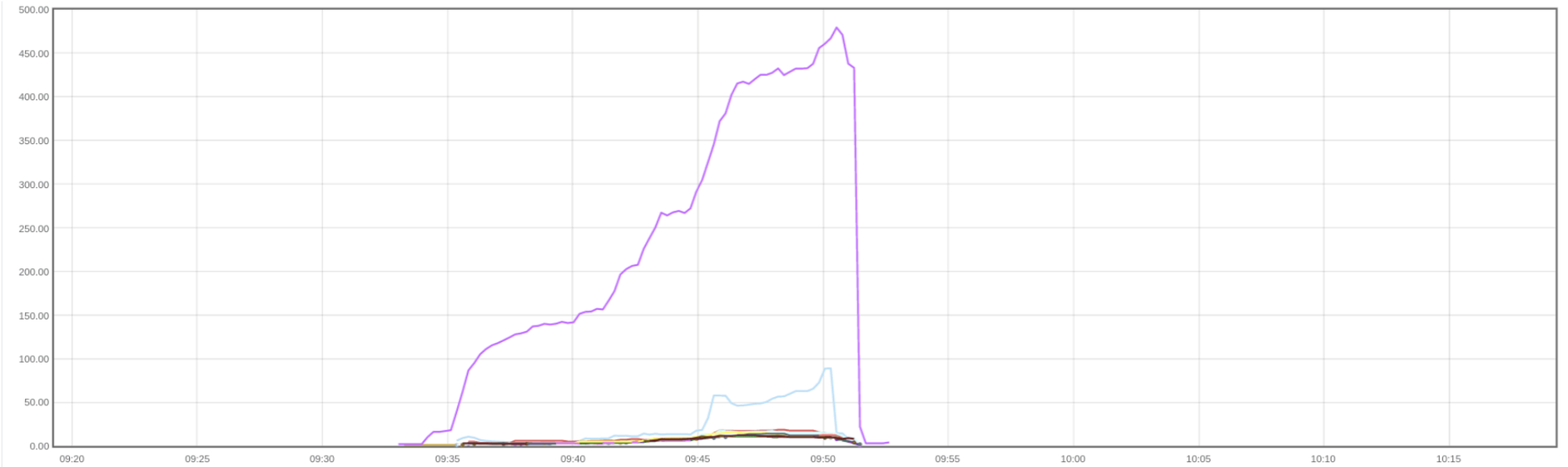


IPC / BufferIO =

Waiting for Concurrent Page Reads

(shared buffers too small / working set too large)

pganalyze



{**endpoint**="AuthorizedProjectUpdate::UserRefreshFromReplicaWorker", **wait_event**="SubtransControlLock", **wait_event_type**="LWLock"}
{**endpoint**="BuildHooksWorker", **wait_event**="SubtransControlLock", **wait_event_type**="LWLock"}
{**endpoint**="ExpirePipelineCacheWorker", **wait_event**="SubtransControlLock", **wait_event_type**="LWLock"}
{**endpoint**="Explore::ProjectsController#index", **wait_event**="SubtransControlLock", **wait_event_type**="LWLock"}
{**endpoint**="GET /api/:version/events", **wait_event**="SubtransControlLock", **wait_event_type**="LWLock"}
{**endpoint**="GET /api/:version/projects", **wait_event**="SubtransControlLock", **wait_event_type**="LWLock"}
{**endpoint**="GET /api/:version/projects/:id/jobs", **wait_event**="SubtransControlLock", **wait_event_type**="LWLock"}
{**endpoint**="GET /api/:version/projects/:id/merge_requests/:merge_request_iid/approvals", **wait_event**="SubtransControlLock", **wait_event_type**="LWLock"}
{**endpoint**="GET /api/:version/repos/:namespace/:project/issues/:id/comments", **wait_event**="SubtransControlLock", **wait_event_type**="LWLock"}
{**endpoint**="GET /api/:version/users", **wait_event**="SubtransControlLock", **wait_event_type**="LWLock"}
{**endpoint**="GET /api/:version/users/:id/events", **wait_event**="SubtransControlLock", **wait_event_type**="LWLock"}
{**endpoint**="GraphqlController#execute", **wait_event**="SubtransControlLock", **wait_event_type**="LWLock"}
{**endpoint**="PATCH /api/:version/jobs/:id/trace", **wait_event**="SubtransControlLock", **wait_event_type**="LWLock"}

LWLock / SubtransControlLock = Waiting to access Subtrans SLRU Cache

(too many subtransactions / old Postgres version)

GitLab: Why we spent the last month eliminating PostgreSQL subtransactions

pganalyze

Every Wait Event has a different meaning,
and not all are a problem (e.g. Client / ClientRead).

If a critical workload keeps being bottlenecked
at critical times with the same wait event,
prioritize understanding it.

In conclusion:

Understand which kind of workload you have (and what it does to Postgres), ahead of time.

Wait events will remind you of those characteristics, but you may be in a hurry to address the bottleneck.

Monitoring high-level metrics can help identify problematic aspects of a workload (e.g. heavy use of subtransactions).

Detailed per-table/per-query metrics are often necessary to take action, or implement recurring practices such as a repack or REINDEX.

Thank you!

Get a free trial of pganalyze

PGANALYZE.COM

Get free pganalyze eBooks and Postgres blog posts

PGANALYZE.COM/RESOURCES PGANALYZE.COM/BLOG PGANALYZE.COM/NEWSLETTER

Additional Q&A

Additional Q&A

What is the one major reason to use pganalyze versus Datadog or SolarWinds? My employer has these tools now. I like pganalyze but must make a case for purchasing it.
Question by Richard K.

pganalyze is purpose-built for Postgres. If you have SolarWinds or Datadog in place but are still seeing Postgres performance issues, one of the problems can be that the insights from those tools do not go deep enough.

Some features that can make a difference are specific advice for query plans with EXPLAIN Insights, Query Tuning Workbooks to iterate on improving a slow query, automatic filters for common Postgres log events, or recommendations for indexes.

pganalyze will be more specific and deliver better results for Postgres databases.

Our sales team would be happy to discuss the business case with pganalyze with you in detail: https://pganalyze.com/contact

Additional Q&A

What type of queries can cause client backend writes instead of checkpointer/bgwriter?
Question by Eric S.

The client backends (i.e. regular connections) making writes is not necessarily a function of the query itself, but rather the state of shared buffers at the time. Basically if there is no available unpinned, non-dirty shared buffer available to read data into, Postgres will have to write out a dirty buffer to disk to be able to perform the read. If you have a query that reads lots of data that will of course be a bigger issue, but typically it makes sense to instead tweak the background writer or checkpointer settings, so the dirty buffers get written out before the query even runs.

You mentioned tuning bgwriter_delay to improve background writer flushing dirty pages, what about tuning bgwriter_lru_maxpages to a higher value?
Question by Somdyuti P.

If your systems have sufficient I/O and you observe the background writer not doing sufficient work it does indeed make sense to raise bgwriter_lru_maxpages in addition (or potentially an alternative to) decreasing bgwriter_delay. I would recommend monitoring pg_stat_io data closely as you change these values, to ensure it has the intended effect.

Additional Q&A

Is there any development effort to have a storage engine in PostgreSQL where the dead rows are not kept in the table itself? Something like OrioleDB.
Question by Rajorshi S.

OrioleDB is the main effort I'm familiar with that is actively working on this in the broader Postgres community. Alternatives like zheap are unfortunately no longer maintained at this point, and there is currently no active core efforts that I'm familiar with to add UNDO-type logs to Postgres.

What is meant by subtransactions?
Question by Kumar D.

Subtransactions are typically created when using the SAVEPOINT command in Postgres, as well as exception handling in stored procedures: https://www.postgresql.org/docs/current/subxacts.html

Is it valid to set synchronous_commit=off for time series data? Like GPS or other sensor data
Question by Ruslan K.

It depends on whether its acceptable to loose the last few data points in the case of a database crash. If that is acceptable, yes, synchronous_commit=off will provide a noticable benefit for write-heavy workloads such as timeseries data.

Additional Q&A

If you have a highly heterogeneous workload (changing over the course of a business day), what kind of configurations would you bias towards?
Question by Curran M.

I would focus on understanding where the workload currently bottlenecks (e.g. based on wait event data), even if not yet impacting user queries. Even if the specific workload is changing, I would at a high-level try to understand whether its write-heavy vs read-heavy. Assuming a write-heavy workload, many of the discussed topics (e.g. checkpoints, index maintenance) apply to different workloads equally, and so its more a matter of identifying those inefficiencies (even if they only occur for a few hours a day), and tackling them through specific individual maintenance (e.g. REINDEX/etc), or by tweaking parameters so you have more headroom.

Is it worth changing commit_delay to improve commits?
Question by Somdyuti P.

We didn't discuss this today due to timing constraints, but yes, commit_delay can be a worthwhile setting to consider on a highly concurrent system with many writes. Laurenz Albe did a write up on this recently: https://www.cybertec-postgresql.com/en/commit_delay-performance-postgresql-benchmark/

Additional Q&A

Why is wal_compression only designed to compress full page images, rather than compressing all WAL content? Couldn't compressing normal WAL records also reduce WAL disk I/O?
Question by CC H.

My assumption is that this does not make sense for small WAL records, since compression would add a lot of complexity for little gains. There is a recent proposal to do compression for larger non-FPI WAL records: https://www.postgresql.org/message-id/flat/4DC38068-976E-4A84-8EE6-4EFACBBD927A%40yandex-team.ru

What is the implication of wal_log_hints setting ? Why does a read-only (counter intuitively) "select" generate lots of WAL ?
Question by David M.

Hint bits serve an important function in Postgres, to reduce the overhead of checking transaction status for a given row version (see https://wiki.postgresql.org/wiki/Hint_Bits). Because this is dependent on the other sessions/ transactions that are active, hint bits will be set at a later point after the initial transaction committed/aborted.

SELECTs, just like other database activity on a given row (e.g. autovacuum), will update hint bits in order to reduce the overhead of future reads of a given row version. The wal_log_hints setting influences whether hint bit updates cause a full page image to be produced when its the first modification to that page after a checkpoint. When data checksums are enabled (very common these days) this is always required to ensure the integrity of the checksum.

Additional Q&A

What can happen if we change max_wal_size to 64 MB from 16 MB in AWS Aurora? Does this affect anything else?
Question by Travis P.

Its worth noting that there are two different settings you may be referencing: max_wal_size, which controls how much WAL can be retained at most before a checkpoint needs to occur, and wal_segment_size which controls the size of WAL files (default 16MB). You cannot change the wal_segment_size on RDS or Aurora. You can modify max_wal_size (default 2GB), and as discussed it can often times make sense to increase this if you see a lot of WAL-based checkpoints, so Postgres can use time-based checkpoints instead (and smooth out I/O over time). Note that Aurora's I/O system and checkpointing works differently than standard Postgres, and so I would recommend reaching out to AWS directly for specific tuning advice regarding max_wal_size on Aurora.

Does specifying a larger wal_segment_size help for WAL I/O performance?
Question by CC H.

Yes, from my experience, it can often make sense to increase from the default WAL size (16 MB) to a larger setting (e.g. 64 MB) since that will be more efficient to work with for Postgres, and ultimately result in less I/O. But I would recommend doing benchmarking to confirm for your workload, especially when going beyond 64 MB.

Additional Q&A

In a Rails app where JSONB fields are updated frequently, does this impact WAL size? How can we minimize that impact?
Question by Raj B.

Yes, JSONB updates would be WAL logged, and due to how UPDATEs are handled in Postgres could have significant impact on the amount of WAL being generated. I would recommend pulling out heavily updated fields from the JSONB data, and storing them as a separate fields. Its worth referencing this part of the Postgres documentation, since large JSONB values are very likely to be in TOAST, and so pulling out the heaviliy updated fields would avoid that TOAST data from being replicated in the WAL stream: https://www.postgresql.org/docs/current/storage-toast.html#STORAGE-TOAST-ONDISK

I have a database with TimescaleDB tables. The checkpoint is well behaved except when TimescaleDB does compression. Then the database checkpoints every few seconds. The max_wal_size is 1 GB. What is a reasonable upper limit for this setting? Are there any special considerations for TimescaleDB?
Question by Warren S.

The main downside of increasing max_wal_size is that you increase the crash recovery time. 1GB is a very small value in my experience, and unless you have very strict RTO targets you can likely increase this to 5GB or 10GB without any concerns. Its worth noting that having infrequent WAL-based checkpoints is not necessarily a problem, it just means that you will see I/O spikes occur during that time (and if it coincides with a regular user-facing workload, that could result in a user-visible performance regression). I don't have any specific recommendations regarding TimescaleDB, but since TimescaleDB tables are WAL logged all of the regular recommendations should apply.

Additional Q&A

If one sets up a readonly hot standby with restore_command that consumes WAL generated with pg_receivewal command, together with an empty primary_conninfo setting. Can such a configuration save from the dilemma of hot_standby_feedback?
Question by CC H.

In my understanding in that situation you are basically choosing something that's similar to hot_standby_feedback=off, because the replica would not have any way to tell the primary to preserve rows, and as such replication lag would occur if there is an active query on your replica.

How would you diagnose whether you have too many full-page writes in a large DB that is experiencing significant replication lag?
Question by Jerad G.

You can monitor the number of full page images being written using high-level statistics (pg_stat_wal), as well as some more recently added counters in other views (e.g. pg_stat_statements).

Additional Q&A

Have you seen CPU bound postgres and not I/O bound? what could cause this?
Question by Suchal J.

Yes, and there are many potential causes of CPU utilization. If you're on a hosted provider the best tool are usually wait events, and reviewing the plans of queries that take a lot of time (e.g. if data is in cache, but a lot of shared buffers are being used by a plan, that can cause CPU utilization). If you have direct access to the machine running Postgres I'd recommend looking at "perf" profiles: https://wiki.postgresql.org/wiki/Profiling_with_perf

How can we monitor locks efficiently and prevent stuff like deadlocks?
More specific example, let's say it's mostly `select for update`s workloads that contribute most to the problem, and there are some weird deadlocks here and there.
Question by Fira R.

For lock monitoring I would recommend combining the "log_lock_waits" setting in Postgres (which will log slow lock acquisitions to the Postgres logs, together with the involved PIDs), with sampling the lock information in pg_stat_activity and using the pg_blocking_pids(..) function. You can also use tools like pganalyze to capture this information: https://pganalyze.com/blog/postgres-lock-monitoring

Additional Q&A

Why does detaching a partition from the table require a table lock? Can you advise regarding strategy for pruning old partitions?
Question by Nikolay K.

Its worth noting that DETACH PARTITION has a CONCURRENTLY option (https://www.postgresql.org/docs/current/sql-altertable.html#SQL-ALTERTABLE-DETACH-PARTITION). As described in the documentation that uses reduced locking, at the expense of taking a bit longer (similar to other CONCURRENTLY commands). I don't have any additional insights beyond the documentation of that option, but would recommend looking at the mailinglist archives (usually linked in the commit that added it to Postgres) for why those specific locks are still used for CONCURRENTLY.

Are manual maintenance task bad practice? For example, could periodically vacuuming heavily used tables go wrong due to the VACUUM locks?
Question by Jahaziel N.

It depends a lot on the workload. I know many larger successful Postgres deployments that have manual practices like a monthly REINDEX, or a weekly VACUUM on specific tables. The big benefit of doing manual, or ideally semi-automatic, maintenance yourself is that you know better when its good to do more I/O on the database because its outside of business hours. If you are concerned about locking you could consider using lock_timeout, but that won't help with e.g VACUUM blocking a DDL statement (you'd have to do some other monitoring/scripting to detect that and cancel the VACUUM, to behave like an autovacuum would).

Additional Q&A

Do the Postgres shared buffers also contain bloat, if the table is bloated?
Question by Raj B.

Yes, in case you have bloated tables where you have only a few rows per 8kb page, and you read those pages, Postgres would have to read the whole 8kb page into the shared buffers. Reducing bloat would also reduce shared buffers requirements in such situations.

With the INDEX_CLEANUP parameter for the VACUUM command set to AUTO by default after PostgreSQL 14.5, is there a risk that this could cause bloat?
Question by Prakash R.

Whilst its theoretically possible, I have not seen any reports myself of that being a noticable cause of bloat.

In production we have a nightly cronjob that does VACUUM FULL at midnight. Would it be better to use pg_squeeze or pg_repack?
Question by Ruslan K.

If you can take the downtime on the tables (i.e. no read/write activity at night), then a VACUUM FULL is the best choice. The alternatives (pg_repack/pg_squeeze) mainly exist to avoid the exclusive lock that a VACUUM FULL would require whilst doing its work.

Additional Q&A

I was advised by someone to run VACUUM FULL after insertion of large number of rows to a table. Should we use pg_squeeze instead?
Question by Keerthimalan B.

If you are actively reading from that table, VACUUM FULL is typically not an option, since it will take an exclusive lock (all reads will be blocked by the VACUUM FULL). If your goal is to reduce existing bloat in the table, it can make sense to run pg_squeeze (you could do that before doing the inserts). Otherwise you could instead run a VACUUM FREEZE which will make sure the newly added rows are both marked all-visible and all-frozen, which will help reduce future autovacuum work.

Does pg_squeeze need the wal_level to be set in logical?
Question by Isaias S.
Yes, because pg_squeeze is using logical replication internally, wal_level needs to be set to logical.

What can I say to persuade my leadership to change autovacuum settings aggressively, or even to persuade running pg_repack. I believe autovacuuming 3 times a month wont hurt more than 1 time per month, correct?
Question by Prakash R.

Often times default settings are acceptable if your database is small. I would recommend starting with doing bloat estimation of large tables, especially those were you are doing lots of activity. A bloated table will slow down both reads and writes, and cause increased storage and I/O costs for the database. You could implement a recurring VACUUM as well, but typically autovacuum in Postgres will do a better job, and so I'd recommend looking at optimizing the settings for the large tables so your autovacuums reliably clean up dead rows. You are correct that increasing autovacuum frequency at that level is unlikely to have any impact.

Additional Q&A

How can I force parallelism for query and how to find out if a query is using parallelism?
Question by Raj R.

Whilst there are debug settings for forcing a parallel query plan, its typically not advisable to do so. Instead I would recommend reviewing the relevant planner cost settings, as well as making sure you have sufficient parallel workers available. You can monitor whether parallel query plans are used by logging auto_explain plans.

My EXPLAIN ANALYZE shows an index (unique) scan needed to read 2GB of data in order to retrieve 1 record, what could be causing the large index page I/O?
Question by Andrew P.

Its worth analyzing in detail whether the EXPLAIN actually showed 2GB of data read (shared buffers read), vs 2GB of data being used (shared buffers hit). Often times it will be the second case (hits), where the counters can be misleading, because they will count each repeated access to the same buffer page (i.e. its not distinct). I would also review whether you have many "Rows Removed By Filter" or similar on that plan node - the row count shown is the result after such filters are applied, so Postgres might have fetched a lot of rows from the table, but filtered many of them right afterwards.

Additional Q&A

Are there ways to reduce overhead for queries with IN clauses that have thousands of values?
Question by Ciprian U.

A good alternative to consider is using "column1 = ANY($1)" and pass an array as the parameter value. That is equivalent to using IN, but performs a bit better in terms of reducing parsing overhead and avoiding some inefficiencies in monitoring with pg_stat_statements.

pg_stat_monitor vs pg_stat_statements which one is better?
Question by Fikrat I.

It depends on which kind of tooling you use to integrate. pg_stat_statements has the benefit that is available everywhere (including managed providers), which makes it the universal tool to use. Its also a lot more production tested and many providers (e.g. Amazon RDS) turn it on by default. The main downside of pg_stat_statements is that you need an external tool (for example pganalyze) that makes snapshots of the data and provides a history of query performance over time. pg_stat_monitor has some built-in capabilities to that internally, but is also a lot more complex overall, so I would measure performance overhead/etc.

Additional Q&A

What would you recommend to select optimal amount of opened connections to postgres? When is it time to introduce pooler?
Question by Gennady M.

In terms of open connections, I would say up to 1000 connections on reasonably sized hardware, you don't need a pooler (but it will give you some memory savings if you do have one). Beyond that, it can make a lot of sense to add a pooler, especially when going to 10000 connections or beyond.

Is it possible to configure pgpool for load balancing and repmgr for auto failover in Postgres 17?
Question by Srinivas M.

To my knowledge you should be able to do this, yes, but its not directly connected to what we discussed in this webinar (which was more focused on performance bottlenecks, vs high availability).

Is it better to have many small pgbouncers vs one big pgbouncer?
Question by Silvio A.

For the purpose of optimizing connections on the Postgres side, it likely makes sense to have one (or a very small number) of pgbouncer deployments close to the database server.

Additional Q&A

Is there data that highlights the benefits of having a connection pooler in front of the DB layer instead of the app layer?
Question by Biagio P.

My recommendation for doing so comes from having run benchmarks in the past, but I don't have that data available currently. Basically if you have network latency of even a few milliseconds between the application server and the database, it can add noticable latency to each query being run, so the pooler has to pin the connection longer, and as such require more connections on the Postgres server itself.

Is UUID v7 safe to use if we need a random value?
Question by Isaias S.

The later parts of a UUIDv7 value have a randomness component (but less bits of randomness than e.g. UUIDv4). For most practical applications this is the right tradeoff, e.g. if you want a random ID for objects that makes it harder to know what the next value is (so you can use it in a URL) a UUIDv7 should be sufficient, and will provide much better index performance.

To find unused indexes, how can we make sure that indexes are not used across different replicas and writer instances?
Question by Arun K.

There is no built-in facility in Postgres to track this information across primary and replica, so you would have to build your own tooling or utilize something like postgres_fdw to query the replicas from the primary.

Additional Q&A

Is there a way to get the unused index at the cluster level instead of getting the instances level (writer/ reader) separately using pganalyze?
Question by Raj B.

We are working on adding this capability to pganalyze, and plan to have it available later this year.

Sometimes actual table size is not shown in pganalyze. Why?
Question by Prateek V.

pganalyze may not show table sizes when a server exceeds the number of table objects that can be monitored (5,000 tables on pganalyze Scale plan, higher on Enterprise plans). You can use the schema filter (https://pganalyze.com/docs/collector/settings#schema-filter-settings) setting to ignore tables to stay under that limit.