Postgres

Best practices / useful blog posts

- https://pglocks.org/
  - https://github.com/AdmTal/PostgreSQL-Query-Lock-Explainer
  - https://postgresisenough.dev/
  - https://wiki.postgresql.org/wiki/Lock_Monitoring
  - https://wiki.postgresql.org/wiki/Don%27t_Do_This
  - Checks for potentially unsafe migrations
  - https://www.crunchydata.com/postgres-tips
  - https://blog.sequinstream.com/keyset-cursors-not-offsets-for-postgres-pagination/
  - EXPLAIN Glossary
  - Avoid using nested transactions and select for share
    - Notes on some PostgreSQL implementation details
  - Partitioning
  - Safely renaming a table
  - Notes on Postgres WAL LSNs
  - https://gitlab.com/postgres-ai/postgresql-consulting/postgres-howtos
  - https://rachbelaid.com/introduction-to-postgres-physical-storage/
  - Set a lower fillfactor for tables which receive a lot of update operations
    - See also: https://www.crunchydata.com/blog/postgres-performance-boost-hot-updates-and-fill-factor#fill-factor
      - A fill factor of 70%, 80% or 90% provides a good tradeoff to hit more HOT updates at the cost of some extra storage
      - Stats should probably be reset after fill factors are tweaked to better keep track of the percentage of HOT updates going forward
  - Consider using **plan_cache_mode = force_custom_plan** to prevent Postgres from caching bad plans
  - Configure **random_page_cost** to a better value (e.g. 1.1) if using SSDs
    - https://www.postgresql.org/docs/current/runtime-config-query.html#GUC-RANDOM-PAGE-COST
  - Configure **effective_cache_size** to 50-75% of your total

RAM
- **create extension pg_proctab** + https://pg_top.gitlab.io/ for Postgres process-level monitoring
  - https://docs.crunchybridge.com/insights-metrics/pgproctab
- increase **maintenance_io_concurrency** to 20-50 or higher
  - https://www.credativ.de/en/blog/postgresql-en/quick-benchmark-analyze-vs-maintenance_io_concurrency/
- Set **default_toast_compression** to **lz4**
- https://x.com/gwenshap/status/1993114834435817786
- set aggressively low timeouts
  - **statement_timeout**
  - **idle_in_transaction_session_timeout**
  - **transaction_timeout**
- https://tembo.io/blog/optimizing-memory-usage
  - **shared_buffers**
    - 25%-30% of total RAM is a good starting value
  - **work_mem**
    - (80% of total RAM - **shared_buffers**) / (**max_connections** * max plan nodes)
    - So if we have 16GB of RAM, 4GB of shared buffers, and 100 max connections, we'd end up with about 88MB available per session. We would divide this value by the average number of query plan nodes to obtain a good setting for **work_mem**
  - **maintenance_work_mem**
    - 1-2 GB
  - **autovacuum_max_workers**
    - each worker may use up to **maintenance_work_mem**
    - there is a **autovacuum_work_mem** parameter if we want a separate autovacuum **work_mem** value
    - **Note:** Postgres autovacuum workers cannot use more than 1GB
  - rough estimate of the maximum memory a Postgres instance might allocate to user sessions for basic queries
    - **work_mem** * **max_connections** * 5
  - https://www.reddit.com/r/PostgreSQL/comments/1d0cjv3/comment/l61y8ba/
- **pg_dump dbname | lz4 | rclone cat storagebox:dst/path**

Extensions

- https://github.com/cybertec-postgresql/pg_squeeze
  - https://github.com/reorg/pg_repack
  - https://github.com/dventimisupabase/pg_flight_recorder
  - https://github.com/NikolayS/pgque
  - pg_ash: Active Session History for PostgreSQL — wait event sampling with zero bloat (pg_cron + PGQ-style partition rotation)
  - https://github.com/kylorend3r/pg-reindexer
  - https://github.com/vitabaks/pg_auto_reindexer
  - https://github.com/dataegret/pgcompacttable
  - https://github.com/HypoPG/hypopg
    - A hypothetical -- or virtual -- index is an index that doesn't really exist, and thus doesn't cost CPU, disk or any resource to create. They're useful to know if specific indexes can increase performance for problematic queries, since you can know if PostgreSQL will use these indexes or not without having to spend resources to create them.
  - https://github.com/VADOSWARE/pg_idkit
  - https://github.com/pg-uint/pg-uint128
  - https://github.com/postgrespro/pg_wait_sampling
    - https://akorotkov.github.io/blog/2016/03/25/wait_monitoring_9_6/
    - https://andyatkinson.com/blog/2024/07/23/postgresql-extension-pg_wait_sampling
  - https://github.com/CrystallineCore/Biscuit
    - High-Performance Pattern Matching Index (for queries using LIKE)

Tooling

- Custom psqlrc
    - See aliased queries here
  - db<>fiddle
  - readyset: Readyset is a MySQL and Postgres wire-compatible caching layer that sits in front of existing databases to speed up queries and horizontally scale read throughput. Under the

hood, ReadySet caches the results of cached select statements and incrementally updates these results over time as the underlying data changes.
  - https://www.pgmustard.com/docs/explain
  - https://github.com/ankane/pghero
  - https://github.com/kwent/pgbouncerhero
  - https://www.pgbouncer.org/
  - https://github.com/postgresml/pgcat
  - https://github.com/pgdogdev/pgdog
  - https://github.com/dimitri/pgcopydb
  - Explore what commands issue which kinds of locks
  - https://leontrolski.github.io/pglockpy.html
    - https://github.com/leontrolski/pglockpy/tree/eda985ff87dbc8c30197e190aa1d8af777f8d349
    - https://github.com/leontrolski/leontrolski.github.io/blob/2d05dcb52b3b90f70a4a3ef9f2cf178b94440cf8/pglockpy.html
  - https://github.com/AdmTal/PostgreSQL-Query-Lock-Explainer
  - https://github.com/NikolayS/postgres_dba
  - https://pgbarman.org/
  - https://pgbackrest.org/
  - https://github.com/CrunchyData/postgres-operator
  - https://github.com/zalando/postgres-operator
  - https://www.postgresql.org/docs/17/pgtestfsync.html
    - https://tanelpoder.com/posts/using-pg-test-fsync-for-testing-low-latency-writes/
    - Great tool to test your disk for low latency writes
  - https://github.com/wublabdubdub/PDU-PostgreSQLDataUnloader (Rescue tool)
    - https://pduzc.com/blog/dropscan_recovery
    - https://pduzc.com/blog/recovery-1tb-data
    - https://pduzc.com/blog/4hour-core-data
    - https://pduzc.com/blog/instant_recovery
  - https://github.com/DmitryNFomin/pg_wait_tracer

Postgres pgbouncer

- https://devcenter.heroku.com/articles/best-practices-pgbouncer-configuration
  - https://jpcamara.com/2023/04/12/pgbouncer-is-useful.html

- https://www.crunchydata.com/blog/postgres-at-scale-running-multiple-pgbouncers
- https://www.2ndquadrant.com/en/blog/running-multiple-pgbouncer-instances-with-systemd/

Transaction pooling mode

- Use **set local** over **set** or **set session**
  - Connect directly to the database when running DDL commands (database migrations) to avoid:
    - Not being able to use **create index concurrently**
    - **set lock_timeout**
  - Don't use session level advisory locks (**pg_advisory_lock**, **pg_advisory_unlock**)
  - Configure pgbouncer prepared statements (or turn prepared statements off on your ORM)
    - Set **max_prepared_statements**, 100 or 200 is a decent starting number
    - Don't forget to set **plan_cache_mode** to **force_custom_plan** on the pgbouncer connections
  - **listen** does not work, but **notify** does
    - You can connect your listener directly to Postgres instead of having it go through pgbouncer
  - Don't use **pg_dump** through a pgbouncer connection
    - https://github.com/pgbouncer/pgbouncer/issues/452

Tricks

Vacuuming a table as quickly as possible (12+)

INDEX_CLEANUP can also be set to **OFF** to force **VACUUM** to always skip index vacuuming, even when there are many dead tuples in the table. This may be useful when it is necessary to make **VACUUM** run as quickly as possible to avoid imminent transaction ID wraparound (see Section 24.1.5). However, the wraparound failsafe mechanism controlled by **vacuum_failsafe_age** will generally trigger automatically to avoid transaction ID wraparound failure, and should be preferred. If index cleanup is not performed regularly, performance may suffer, because as the table is modified indexes will accumulate dead tuples and the table itself will accumulate dead line pointers that cannot be removed until index cleanup is completed.

— https://www.postgresql.org/docs/current/sql-vacuum.html#:~:text=the%20partitioned%20table.-,INDEX_CLEANUP,-Normally%2C%20VACUUM

```
vacuum (index_cleanup false) table;
```

Only use this to prevent an impending (and catastrophic) transaction ID wraparound!

Tuple shuffling with CTEs

https://www.crunchydata.com/blog/tuple-shuffling-postgres-ctes-for-update-and-delete-table-data

Deleting rows and inserting to another table:

```
with deleted as (
  delete from table_a
  where last_modified < now () - interval '1 year'
returning *
)
insert into archive
select *
from deleted;
```

Disable synchronous commits within session/transaction

It's possible to disable synchronous commits when insert/updating analytical, log, or otherwise non-critical data within a session or transaction.

https://www.postgresql.org/docs/17/runtime-config-wal.html#GUC-SYNCHRONOUS-COMMIT

```
begin;
set local synchronous_commit to off;
insert into table (column) select i from
generate_series(1, 10000) as i;
commit;

set synchronous_commit to off;
insert into table (column) select i from
generate_series(1, 10000) as i;
```

```
set synchronous_commit to on;
```

Alternative sorting (e.g. numeric) using collations

- https://dbfiddle.uk/qrCsRe5E
  - https://www.postgresql.org/docs/17/collation.html#ICU-CUSTOM-COLLATIONS
  - https://www.postgresql.eu/events/pgconfeu2022/sessions/session/4040/slides/337/Collations%20in%20PostgreSQL%20-%20The%20good,%20the%20bad%20and%20the%20ugly%20-%20Tobias%20Bussmann.pdf
    - Be aware that using collations in indexes can potentially result in corrupted indexes when glibc or ICU versions are upgraded!

```
create table demo (
  id bigint generated always as identity primary
key,
  str text
);

insert into demo (str) select i::text from
generate_series(1, 1000) as i;

-- https://www.postgresql.org/docs/17/
collation.html#ICU-CUSTOM-COLLATIONS
-- https://www.postgresql.eu/events/pgconfeu2022/
sessions/session/4040/slides/337/
Collations%20in%20PostgreSQL%20-%20The%20good,
%20the%20bad%20and%20the%20ugly%20-
%20Tobias%20Bussmann.pdf
create collation numeric (provider = icu,
deterministic = true, locale = 'en-u-kn-true');

select str from demo order by str collate "numeric"
asc limit 10;
```

Limited count

```
select count(*) from (
  select 1
```

```
  from table_name
  where foo = 'bar'
  limit 101
);
```

Exists at least N

```
select id
from foo f
where exists (
  select 1
  from bar b
  where b.foo_id = f.id
  offset 4
  limit 1
);
```

Distinct only on some fields

```
select distinct on (url) url, request_duration
from logs
order by url, timestamp desc
```

Running an alter table without lengthy locks

```
set lock_timeout = 50;
alter table test add column whatever2 int4;
```

Repeat, infinitely or up to a certain limit of tries, if timeout was reached.

Simple loop:

```
create procedure execute_with_minimal_locking( in
p_sql text) language plpgsql as $$
begin
    set local lock_timeout = 10;
    loop
        begin
            execute p_sql;
            exite;
        exception when lock_not_available then
            perform pg_sleep(random() * 5 + 2);
```

```
        end;
    end loop;
    reset lock_timeout;
end;
$$;
-- call execute_with_minimal_locking('alter table z
add column test text');
```

Loop with limit of attempts and exponential backoff:

```
do $do$
declare
  lock_timeout constant text := '50ms';
  max_attempts constant int := 30;
  ddl_completed boolean := false;
  cap_ms bigint := 60000;
  base_ms bigint := 10;
  delay_ms bigint := NULL;
begin
  perform set_config('lock_timeout', lock_timeout,
false);

  for i in 1..max_attempts loop
    begin
      alter table test add column whatever2 int4;

      ddl_completed := true;

      exit;
    exception when lock_not_available then
      raise warning 'attempt %/% to lock table
"test" failed', i, max_attempts;

      delay_ms := round(random() * least(cap_ms,
base_ms * 2 ^ i));
      raise debug 'delay %/%: % ms', i,
max_attempts, delay_ms;

      perform pg_sleep(delay_ms::numeric / 1000);
    end;
  end loop;

  if ddl_completed then
```

```
    raise info 'table "test" successfully altered';
  else
    raise exception 'failed to alter table "test"';
  end if;
end $do$;
```

Gapless sequences (insert w/ on conflict vs insert w/ anti-join)

```
-- Will leave gaps
insert into t (val) values ('abc'), ('def') on
conflict do nothing;

-- Won't leave gaps
insert into t (val)
select *
from (values ('abc'), ('def')) as tmp (val)
where not exists (select 1 from t where t.val =
tmp.val);
```

Group by rollup

https://www.crunchydata.com/blog/easy-totals-and-subtotals-in-postgres-with-rollup-and-cube#group-by-rollup

```
select
    to_char(date_trunc('month', order_date),
'FMMonth YYYY') as month,
    category,
    count(*) as total_orders,
    sum(total_amount) as total_amount
from
    orders
group by
    rollup (date_trunc('month', order_date),
category)
order by
    date_trunc('month', order_date), category;

--      month      |  category   | total_orders |
total_amount
-- ----------------+-------------+--------------
+--------------
```

```
--  October 2021   | Books       |            3 |
2375.73
--  October 2021   | Clothing    |           18 |
13770.09
--  October 2021   | Computers   |           17 |
13005.87
--  October 2021   | Electronics |           25 |
16358.96
--  October 2021   |             |           63 |
45510.65
```

Group by cube

https://www.crunchydata.com/blog/easy-totals-and-subtotals-in-postgres-with-rollup-and-cube#group-by-cube

```
select
    to_char(date_trunc('month', order_date),
'FMMonth YYYY') as month,
    category,
    count(*) as total_orders,
    sum(total_amount) as total_amount
from
    orders
group by
    cube (date_trunc('month', order_date), category)
order by
    date_trunc('month', order_date), category;

--       month      |  category   | total_orders |
total_amount
-- ----------------+-------------+--------------
+--------------
--  October 2024   | Books       |            9 |
5574.92
--  October 2024   | Clothing    |           19 |
11856.80
--  October 2024   | Computers   |           22 |
13002.10
--  October 2024   | Electronics |           50 |
34251.83
--  October 2024   |             |          100 |
```

```
64685.65
--                  | Books       |          521 |
328242.79
--                  | Clothing    |         1133 |
739866.25
--                  | Computers   |         1069 |
680817.70
--                  | Electronics |         2709 |
1707713.80
--                  |             |         5432 |
3456640.54
```

CTID paginated deletion

https://www.shayon.dev/post/2024/303/using-ctid-based-pagination-for-data-cleanups-in-postgresql/

```
page_size = 200_000  # Number of pages to process at
once
current_page = 0
cutoff_date = '2023-04-01'
deleted_count = 0

ApplicationRecord.with_statement_timeout(TIMEOUT) do
  loop do
    delete_sql = <<~SQL
      WITH to_delete AS (
        SELECT ctid
        FROM large_table
        WHERE ctid BETWEEN '(#{current_page *
page_size},0)'::tid
                       AND '(#{(current_page + 1) *
page_size},0)'::tid
          AND created_at < '#{cutoff_date}'
        FOR UPDATE OF large_table SKIP LOCKED
      )
      DELETE FROM large_table
      WHERE ctid IN (SELECT ctid FROM to_delete)
      RETURNING id;
    SQL

    result =
```

```
ActiveRecord::Base.connection.exec_query(delete_sql)
    deleted_count += result.rows.size

    current_page += 1

    # Check if there are any rows in next page range
    check_sql = <<~SQL
      SELECT EXISTS (
        SELECT 1
        FROM large_table
        WHERE ctid BETWEEN '(#{current_page *
page_size},0)'::tid
                       AND '(#{(current_page + 1) *
page_size},0)'::tid
        LIMIT 1
      );
    SQL

    has_more_rows =
ActiveRecord::Base.connection.exec_query(check_sql).
rows[0][0]
    break unless has_more_rows
  end
end
```

Temporarily drop an index

```
begin;
set lock_timeout = '50ms';
drop index index_name;
explain analyze select id from table_name where
table_name.reference_id = '3bc3de7d-8428-475e-
a66a-8b173d5f8a58' limit 2;
rollback;
```

Row constructor comparison

With row constructor comparisons you can force certain queries to use an Index Cond instead of a Filter.

```
select posts.*, channels.team_id
from posts
```

```
left join channels on posts.channel_id = channels.id
where (posts.created_at, posts.id) > (?1, ?2) --
lexicographical comparison
order by posts.created_at asc, posts.id asc
limit ?3;
```

Structural table copy

```
create table table_copy (like original_table
including defaults including constraints including
indexes);
```

Unique hash index (via constraint)

Caveats:

- **on conflict** statements must explicitly name the constraint
  - exclude constraints can't be added as not valid, this means you can't add the constraint to an existing table without issuing an exclusive lock
  - **TODO:** I wonder if it'd be possible to move to an exclude constraint backed by a hash index (without exclusively locking the table) when you already have a unique index backed by a btree

Example showing it handling collisions correctly:

```
create table demo (
  id bigint generated always as identity primary
key,
  value int not null,
  constraint unique_value exclude using hash(value
with =)
);

insert into demo (value)
  select unnest(array_agg(i))
  from generate_series(1,(2^16)::int4) i
  group by hashint4(i)
  having count(*) > 1
  limit  2;

select
  opc.opcname as operator_class_name,
```

```
  amproc.amproc as access_method_procedure
from
  pg_class t
  join pg_index ix on t.oid = ix.indrelid
  join pg_class i on i.oid = ix.indexrelid
  join pg_attribute a on a.attrelid = t.oid and
a.attnum = any(ix.indkey)
  join pg_opclass opc on opc.oid = ix.indclass[0]
  join pg_amproc amproc on amproc.amprocfamily =
opc.opcfamily
where
  i.relname = 'unique_value' and amproc.amprocnum =
1; -- HASHSTANDARD_PROC = 1

select *, hashint4(value), hashint8(value) from
demo;

insert into demo (value)
  select unnest(array_agg(i))
  from generate_series(1,(2^16)::int4) i
  group by hashint4(i)
  having count(*) > 1
  limit  1;

insert into demo (value)
  select unnest(array_agg(i))
  from generate_series(1,(2^16)::int4) i
  group by hashint4(i)
  having count(*) > 1
  limit  1
on conflict (value) do nothing;

insert into demo (value)
  select unnest(array_agg(i))
  from generate_series(1,(2^16)::int4) i
  group by hashint4(i)
  having count(*) > 1
  limit  1
on conflict on constraint unique_value do nothing;
```

Insert sample data

```
do $$
  begin loop
    insert into http_request (
      site_id, ingest_time, url, request_country,
      ip_address, status_code, response_time_msec
    ) values (
      trunc(random()*32), clock_timestamp(),
      concat('[http://example.com](https://t.co/
arRaQ8EOz0)', md5(random()::text)),
      ('{China,India,USA,Indonesia}'::text[])
[ceil(random()*4)],
      concat(
        trunc(random()*250 + 2), '.',
        trunc(random()*250 + 2), '.',
        trunc(random()*250 + 2), '.',
        trunc(random()*250 + 2)
      )::inet,
      ('{200,404}'::int[])[ceil(random()*2)],
      5+trunc(random()*150)
    );
    commit;
    perform pg_sleep(random() * 0.05);
  end loop;
end $$;
```

Bulk insert

```
insert into authors(id, name, bio)
select *
from unnest(
  array[1, 2]::bigint[],
  array['John', 'Mary']::text[],
  array['Just a person.', 'Just a person.']::text[]
);
```

Bulk update

```
update authors
set name = tmp.name, bio = tmp.bio
from (
  select *
```

```
  from unnest(
    array[1, 2]::bigint[],
    array['John', 'Mary']::text[],
    array['Just a person.', 'Just a
person.']::text[]
  ) as t(id, name, bio)
) as tmp
where authors.id = tmp.id;
```

Bulk select

```
select *
from authors
where (name, bio) in (
  select *
  from unnest(
    array['John', 'Mary']::text[],
    array['Just a person.', 'Just a
person.']::text[]
  )
);

-- Or, if you need more control:

select authors.*
from authors
inner join (
  select *
  from unnest(
    array['John', 'Mary']::text[],
    array['Just a person.', 'Just a
person.']::text[]
  ) as t(name, bio)
) as unnest_query
on (
  lower(authors.name) = lower(unnest_query.name) and
lower(authors.bio) = lower(unnest_query.bio)
);
```

Bulk delete

```
delete from authors
```

```
where (name, bio) in (
  select *
  from unnest(
    array['John', 'Mary']::text[],
    array['Just a person.', 'Just a
person.']::text[]
  )
);
```

Array union

```
update table_name
set column_name = (
  select array_agg(distinct elem)
  from unnest(coalesce(column_name, array[]::text[])
|| array['foobar']) as elem
);
```

Conditional insert

```
insert into possible_problematic_domains (domain,
created_at, updated_at)
select $1, current_timestamp, current_timestamp
where exists (
  select 1
  from companies
  where discarded_at is null
  and domain = $1
  having (count(*) >= 5)
  limit 1
) and not (
  exists (
    select 1
    from problematic_domains
    where domain = $1
    limit 1
  )
)
on conflict (domain) do nothing
returning id;
```

Snippets

Query execution times

```
select round(total_exec_time*1000)/1000 as
total_exec_time,
       round(total_plan_time*1000)/1000 as
total_plan_time,
       query
from pg_stat_statements
order by total_exec_time desc
limit 2;
select total_exec_time,
       mean_exec_time as avg_ms,
       calls,
       query
from pg_stat_statements
order by mean_exec_time desc
limit 10;
```

Check long running queries

```
select pid,
       now() - pg_stat_activity.query_start as
duration,
       query,
       state,
       wait_event,
       wait_event_type,
       pg_blocking_pids(pid)
from pg_stat_activity
where (now() - pg_stat_activity.query_start) >
interval '1 minutes'
  and state = 'active';
```

Blockers of queries (ALTER TABLE)

```
select blockers.pid,
       blockers.usename,
       blockers.query_start,
       blockers.query
from pg_stat_activity blockers
```

```
inner join
  (select pg_blocking_pids(pid) blocking_pids
   from pg_stat_activity
   where pid != pg_backend_pid()
     and query ilike 'alter table%' ) my_query on
blockers.pid = any(my_query.blocking_pids);
```

Blockers of queries (blocked query + blocking query)

```
select a1.pid,
       a1.usename,
       (now() - a1.query_start) as running_time,
       pg_blocking_pids(a1.pid) as blocked_by,
       a1.query as blocked_query,
       a2.query as blocking_query
from pg_stat_activity as a1
inner join pg_stat_activity as a2 on (a2.pid =
(pg_blocking_pids(a1.pid)::integer[])[1])
where cardinality(pg_blocking_pids(a1.pid)) > 0;
```

Kill query

```
select pg_cancel_backend(pid);
select pg_terminate_backend(pid);
```

Kill all autovacuums

```
select pg_terminate_backend(pid),
       query,
       now() - pg_stat_activity.query_start as
duration
from pg_stat_activity
where query ilike 'autovacuum:%';
```

Check ongoing vacuums

```
select p.pid,
       now() - a.xact_start as duration,
       coalesce(wait_event_type ||'.'|| wait_event,
'f') as waiting,
       case
           when a.query ~*'^autovacuum.*to prevent
```

```
wraparound' then 'wraparound'
           when a.query ~*'^vacuum' then 'user'
           else 'regular'
       end as mode,
       p.datname as database,
       p.relid::regclass as table,
       p.phase,
       pg_size_pretty(p.heap_blks_total *
current_setting('block_size')::int) as table_size,
       pg_size_pretty(pg_total_relation_size(relid))
as total_size,
       pg_size_pretty(p.heap_blks_scanned *
current_setting('block_size')::int) as scanned,
       pg_size_pretty(p.heap_blks_vacuumed *
current_setting('block_size')::int) as vacuumed,
       round(100.0 * p.heap_blks_scanned /
p.heap_blks_total, 1) as scanned_pct,
       round(100.0 * p.heap_blks_vacuumed /
p.heap_blks_total, 1) as vacuumed_pct,
       p.index_vacuum_count,
       round(100.0 * p.num_dead_tuples /
p.max_dead_tuples, 1) as dead_pct
from pg_stat_progress_vacuum p
join pg_stat_activity a using (pid)
order by now() - a.xact_start desc;
```

Estimate row count

```
select reltuples::numeric as estimate_count
from pg_class
where relname = 'table_name';
```

Estimate query row count

```
create function row_estimator(query text) returns
bigint language plpgsql as $$declare
   plan jsonb;
begin
   execute 'explain (format json) ' || query into
plan;

   return (plan->0->'Plan'->>'Plan Rows')::bigint;
```

```
end;$$;
```

Check table sizes

```
select nspname || '.' || relname as "relation",
       pg_size_pretty(pg_total_relation_size(c.oid))
as "total_size"
from pg_class c
left join pg_namespace n on (n.oid = c.relnamespace)
where nspname not in ('pg_catalog',
                      'information_schema')
  and c.relkind <> 'i'
  and nspname !~ '^pg_toast'
order by pg_total_relation_size(c.oid) desc
limit 40;
```

Check table size

```
select
pg_size_pretty(pg_relation_size('table_name'));
```

Check unused indexes

```
select schemaname || '.' || relname as table,
       indexrelname as index,

pg_size_pretty(pg_relation_size(i.indexrelid)) as
"index size",
       idx_scan as "index scans"
from pg_stat_user_indexes ui
join pg_index i on ui.indexrelid = i.indexrelid
where not indisunique
  and idx_scan < 50
  and pg_relation_size(relid) > 5 * 8192
order by pg_relation_size(i.indexrelid) /
nullif(idx_scan, 0) desc nulls first,
         pg_relation_size(i.indexrelid) desc;
```

Check which tables are aging

```
select c.oid::regclass,
       age(c.relfrozenxid),
```

```
       pg_size_pretty(pg_total_relation_size(c.oid))
from pg_class c
join pg_namespace n on c.relnamespace = n.oid
where relkind in ('r',
                  't',
                  'm')
  and n.nspname not in ('pg_toast')
order by 2 desc
limit 20;
```

Connection counts

```
select count(*),
       state
from pg_stat_activity
group by state;
```

Check for index usage

```
select
    idstat.relname as table_name,
    indexrelname as index_name,
    idstat.idx_scan as index_scans_count,
    pg_size_pretty(pg_relation_size(indexrelid)) as
index_size,
    tabstat.idx_scan as table_reads_index_count,
    tabstat.seq_scan as table_reads_seq_count,
    tabstat.seq_scan + tabstat.idx_scan as
table_reads_count,
    n_tup_upd + n_tup_ins + n_tup_del as
table_writes_count,
    pg_size_pretty(pg_relation_size(idstat.relid))
as table_size
from
    pg_stat_user_indexes as idstat
join
    pg_indexes
    on
    indexrelname = indexname
    and
    idstat.schemaname = pg_indexes.schemaname
join
```

```
    pg_stat_user_tables as tabstat
    on
    idstat.relid = tabstat.relid
where
    indexdef !~* 'unique'
order by
    idstat.idx_scan desc,
    pg_relation_size(indexrelid) desc;
```

Check ongoing usage creation

```
select now(),
       query_start as started_at,
       now() - query_start as query_duration,
       format('[%s] %s', a.pid, a.query) as
pid_and_query,
       index_relid::regclass as index_name,
       relid::regclass as table_name,
       (pg_size_pretty(pg_relation_size(relid))) as
table_size,
       nullif(wait_event_type, '') || ': ' ||
wait_event as wait_type_and_event,
       phase,
       format('%s (%s of %s)', coalesce((round(100 *
blocks_done::numeric / nullif(blocks_total, 0),
2))::text || '%', 'N/A'),
coalesce(blocks_done::text, '?'),
coalesce(blocks_total::text, '?')) as
blocks_progress,
       format('%s (%s of %s)', coalesce((round(100 *
tuples_done::numeric / nullif(tuples_total, 0),
2))::text || '%', 'N/A'),
coalesce(tuples_done::text, '?'),
coalesce(tuples_total::text, '?')) as
tuples_progress,
       current_locker_pid,

  (select nullif(left(query, 150), '') || '...'
   from pg_stat_activity a
   where a.pid = current_locker_pid) as
current_locker_query,
       format('%s (%s of %s)', coalesce((round(100 *
```

```
lockers_done::numeric / nullif(lockers_total, 0),
2))::text || '%', 'N/A'),
coalesce(lockers_done::text, '?'),
coalesce(lockers_total::text, '?')) as
lockers_progress,
       format('%s (%s of %s)', coalesce((round(100 *
partitions_done::numeric / nullif(partitions_total,
0), 2))::text || '%', 'N/A'),
coalesce(partitions_done::text, '?'),
coalesce(partitions_total::text, '?')) as
partitions_progress,

  (select format('%s (%s of %s)',
coalesce((round(100 * n_dead_tup::numeric /
nullif(reltuples::numeric, 0), 2))::text || '%', 'N/
A'), coalesce(n_dead_tup::text, '?'),
coalesce(reltuples::int8::text, '?'))
   from pg_stat_all_tables t,
        pg_class tc
   where t.relid = p.relid
     and tc.oid = p.relid ) as table_dead_tuples
from pg_stat_progress_create_index p
left join pg_stat_activity a on a.pid = p.pid
order by p.index_relid;
```

See also: https://www.postgresql.org/docs/current/progress-reporting.html#CREATE-INDEX-PROGRESS-REPORTING

Show Analyze / Vacuum Statistics

```
with raw_data as (
  select
    pg_namespace.nspname,
    pg_class.relname,
    pg_class.oid as relid,
    pg_class.reltuples,
    pg_stat_all_tables.n_dead_tup,
    pg_stat_all_tables.n_mod_since_analyze,
    (select split_part(x, '=', 2) from
unnest(pg_class.reloptions) q (x) where x ~
'^autovacuum_analyze_scale_factor=' ) as
c_analyze_factor,
```

```
    (select split_part(x, '=', 2) from
unnest(pg_class.reloptions) q (x) where x ~
'^autovacuum_analyze_threshold=' ) as
c_analyze_threshold,
    (select split_part(x, '=', 2) from
unnest(pg_class.reloptions) q (x) where x ~
'^autovacuum_vacuum_scale_factor=' ) as
c_vacuum_factor,
    (select split_part(x, '=', 2) from
unnest(pg_class.reloptions) q (x) where x ~
'^autovacuum_vacuum_threshold=' ) as
c_vacuum_threshold,
    to_char(pg_stat_all_tables.last_vacuum, 'YYYY-
MM-DD HH24:MI:SS') as last_vacuum,
    to_char(pg_stat_all_tables.last_autovacuum,
'YYYY-MM-DD HH24:MI:SS') as last_autovacuum,
    to_char(pg_stat_all_tables.last_analyze, 'YYYY-
MM-DD HH24:MI:SS') as last_analyze,
    to_char(pg_stat_all_tables.last_autoanalyze,
'YYYY-MM-DD HH24:MI:SS') as last_autoanalyze
  from
    pg_class
  join pg_namespace on pg_class.relnamespace =
pg_namespace.oid
    left outer join pg_stat_all_tables on
pg_class.oid = pg_stat_all_tables.relid
  where
    n_dead_tup is not null
    and nspname not in ('information_schema',
'pg_catalog')
    and nspname not like 'pg_toast%'
    and pg_class.relkind = 'r'
), data as (
  select
    *,
    coalesce(raw_data.c_analyze_factor,
current_setting('autovacuum_analyze_scale_factor')):
:float8 as analyze_factor,
    coalesce(raw_data.c_analyze_threshold,
current_setting('autovacuum_analyze_threshold'))::fl
oat8 as analyze_threshold,
    coalesce(raw_data.c_vacuum_factor,
```

```
current_setting('autovacuum_vacuum_scale_factor'))::
float8 as vacuum_factor,
    coalesce(raw_data.c_vacuum_threshold,
current_setting('autovacuum_vacuum_threshold'))::flo
at8 as vacuum_threshold
  from raw_data
)
select
  relname,
  reltuples,
  n_dead_tup,
  n_mod_since_analyze,
  round(reltuples * vacuum_factor +
vacuum_threshold) as v_threshold,
  round(reltuples * analyze_factor +
analyze_threshold) as a_threshold,
  round(cast(n_dead_tup/(reltuples * vacuum_factor +
vacuum_threshold)*100 as numeric), 2) as v_percent,
  round(cast(n_mod_since_analyze/(reltuples *
analyze_factor + analyze_threshold)*100 as numeric),
2) as a_percent,
  last_vacuum,
  last_autovacuum,
  last_analyze,
  last_autoanalyze
from
  data
order by a_percent desc;
```

Check table statistics

```
select * from pg_stats pgs where pgs.tablename =
'?';
```

Prepared statements for psql operations

```
prepare add_flag_to_team(text, uuid) as insert into
flag_team (
    flag_id,
    team_id
) values (
    (select id from flag where name = $1),
```

```
    $2
) on conflict do nothing;

execute add_flag_to_team('<flag_name>',
'<team_id>');
```

Rows Without Overlapping Dates

Preventing e.g. multiple concurrent reservations for a meeting room is a complicated task because of race conditions. Without pessimistic locking by the application or careful planning, simultaneous requests can create room reservations for the exact timeframe or overlapping ones. The work can be offloaded to the database with an exclusion constraint that will prevent any overlapping ranges for the same room number. This safety feature is available for integer, numeric, date and timestamp ranges.

```
create table bookings (
  room_number int,
  reservation tstzrange,
  exclude using gist (room_number with =,
reservation with &&)
);
insert into meeting_rooms (
    room_number, reservation
) values (
  5, '[2022-08-20 16:00:00+00,2022-08-20
17:30:00+00]',
  5, '[2022-08-20 17:30:00+00,2022-08-20
19:00:00+00]',
);
```

Max per partition (50 rows, max 5 per tenant_id)

```
select "my_table"."id", "rank"
from (
  select *, row_number()
  over(
    partition by my_table.tenant_id
    order by greatest(
      my_table.contacted_at,
      my_table.exported_at,
```

```
      '-infinity'
    ) desc
  ) as rank
  from "my_table"
) my_table
where "my_table"."rank" between 1 and 5
limit 50;
```

Efficient schema for tags

```
create table products (
  id bigint,
  name text,
  tag_ids jsonb
);

create index idx_product_tags on products (tags
jsonb_path_ops);

select *
from products
where tag_ids @> '[1,2]' and not(tagids @> '[3]');
```

If you store the primary keys of a tags table in a JSON array, you can do efficient lookups for products having and not having specific tags. A traditional approach would need six joins, three times to the m:n table and three times to the tags table, while the JSON approach is only a simple condition. It's very effective to skip doing many complex joins, especially for filter-based queries that use many tags.

Disable an index

```
update pg_index set indisvalid = false where
indexrelid = 'index_name'::regclass;
```

indisvalid: If true, the index is currently valid for queries. False means the index is possibly incomplete: it must still be modified by INSERT/ UPDATE operations, but it cannot safely be used for queries. If it is unique, the uniqueness property is not guaranteed true either.
— https://www.postgresql.org/docs/17/catalog-pg-index.html#:~:text=indisvalid%20bool

Debugging local replication

debug_logical_replication_streaming (buffered by default, or immediate) forces logical replication to stream each in-progress change immediately rather than buffering, so the streaming code path gets tested even on small transactions.