```
$ psql
SELECT(*) FROM slides
WHERE presntation_id = 42\gx
```

loading...

How to optimize queries

1. Find a slow query
2. Run `` `EXPLAIN ANALYZE` ``
3. Add index
4. …
5. Developer goes brr

Real life example

- over 10M JS projects
- project files stored in jsonb column
- business wants to search projects by their node dependencies

Real life example

```
SELECT *
FROM projects
WHERE (
  (dependencies->>'package.json')::jsonb->'dependencies' ? 'react'
  OR legacy_dependencies->...
  )
AND deleted_at IS NULL
ORDER BY forks_count DESC
LIMIT 10
```

```
CREATE TABLE project_dependencies (id bigserial primary key, data jsonb, project_id ...
CREATE TRIGGER ...
```

Real life example

```sql
SELECT p.*
FROM projects p
JOIN project_dependencies pd ON p.id = pd.project_id
WHERE pd.data->'packages' ? 'react'
AND p.deleted_at IS NULL
ORDER BY p.forks_count DESC
LIMIT 10;
```

```sql
CREATE INDEX proj_forks_count_idx ON projects USING (forks_count);
CREATE INDEX deps_data_idx ON project_dependencies USING GIN (data);
```

Real life example

- `react` => 1-200ms
- `nanoid` => 10-1300ms
- `next` => 40s-60s

Real life example

```
EXPLAIN ANALYZE SELECT ...

Limit (cost=0.87..7979.33 rows=10 width=12)
      (actual time=25.362..39896.966 rows=10 loops=1)
  ->  Nested Loop (cost=0.87..98463773.75 rows=123412 width=12)
                  (actual time=25.360..39896.704 rows=10 loops=1)
        ->  Index Scan Backward using proj_forks_count_idx on projects
                  (cost=0.43..14777766.90 rows=10029488 width=12)
                  (actual time=2.711..18215.382 rows=19180 loops=1)
        ->  Index Scan using proj_deps_projs_idx on project_dependencies
                  (cost=0.43..8.34 rows=1 width=8)
                  (actual time=1.129..1.129 rows=0 loops=19180)
              Index Cond: (project_id = projects.id)
              Filter: ((data -> 'packages'::text) ? 'next'::text)
              Rows Removed by Filter: 1
Planning Time: 27.642 ms
Execution Time: 39897.861 ms
```

Advanced PostgreSQL Query Optimization

Life after `CREATE INDEX`

Svyatoslav Kryukov
Evil Martians

About me

- work in Evil Martians
- sometimes write giant posts for Martian's blog
- started career as a self-taught PHP developer

When the grass was greener

- deploy via FileZilla
- no frameworks
- no composer
- no git
- language runtime is a black box

Mindful approach

- learn language features
- read libraries code
- discover language runtime
- delve into the specifics of OS and networking
- …

Know your tools

Database
is your main tool

PostgreSQL

The World’s Most Advanced Open Source Relational Database

Things we won’t talk about today

- transactions
- locks and MVCC
- cache and WAL
- DDL & DML (gem strong_migrations)
- different types of indexes

Plan for today

- PostgreSQL Planner
- data access
- combining relations
- statistics
- optimization techniques

SQL

- declarative
- based on relational theory

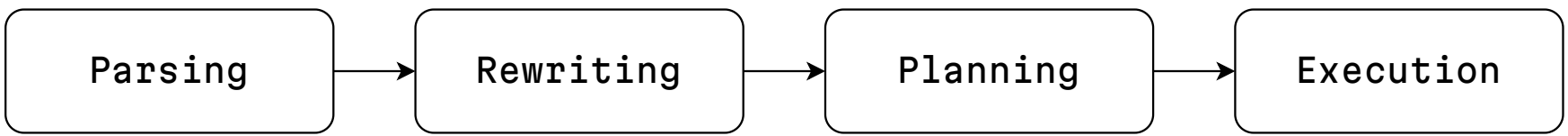
Relational operations

- selection (`WHERE`)
- projection (`SELECT`)
- production (`FULL JOIN`)
- all other operations can be expressed through these three

Math vs Real life

- PostgreSQL won't use logical operations as is — it's too expensive
- it must transform a declarative query to a sequence of physical operations

Query Execution Lifecycle



PostgreSQL Planner

- finds plans
- selects the best plan (using cost)
- does not support hints (kinda)

`EXPLAIN`

```
EXPLAIN SELECT * FROM users WHERE created_at > '2021-01-01';
```

```
Bitmap Heap Scan on users (cost=3753.91..24070.21 rows=177224 width=366)
  Recheck Cond: (created_at > '2021-01-01'::timestamp)
  ->  Bitmap Index Scan on idx_users_created_at (cost=0.00..3709.61 rows=177224 width=0)
        Index Cond: (created_at > '2021-01-01'::timestamp)
```

- reads from the inside out
- `width=366` — average width of rows output (in bytes)
- `rows=177224` — estimated number of rows
- `cost=3753.91..24070.21` — start-up and total costs

Physical operations

- data access
- combining relations

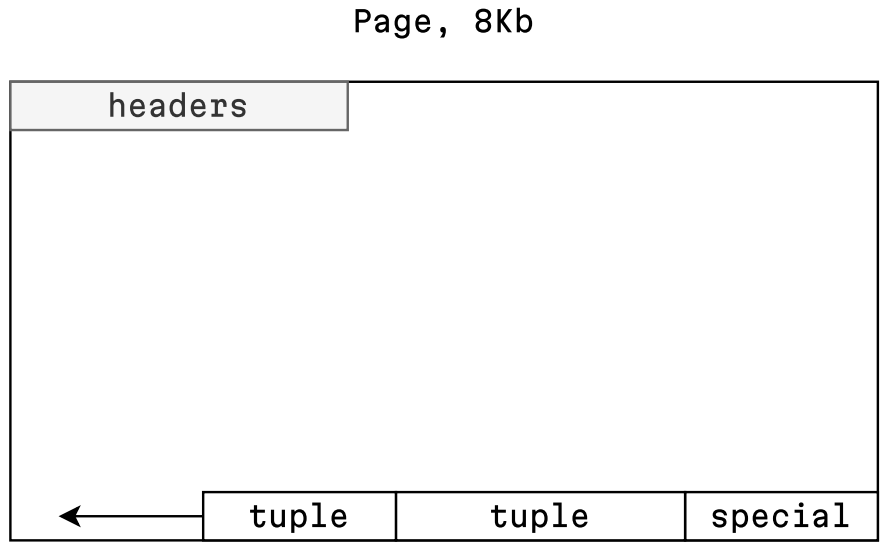
Data Access

- sequential scan
- index scan
- index only scan
- bitmap scan

Warning: no parallel versions in this talk

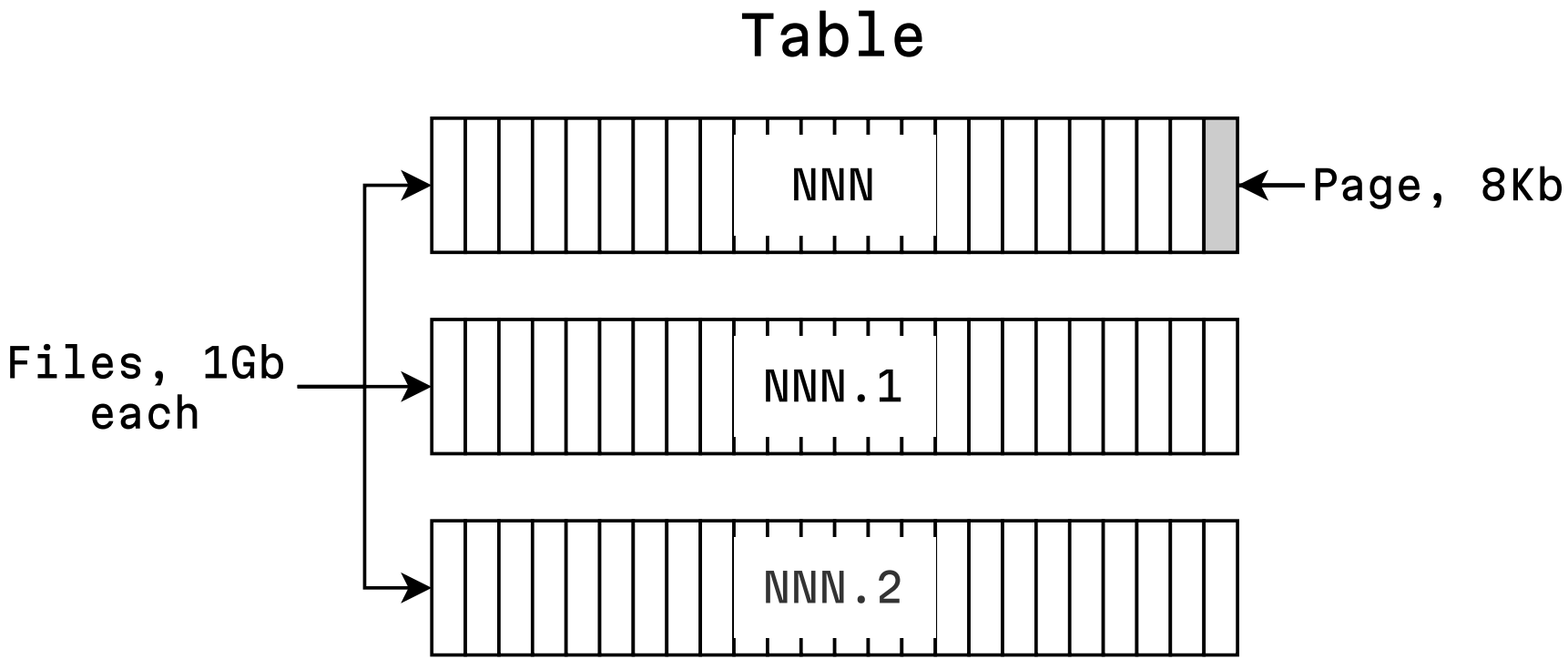
Sequential scan

Physical storage. Pages

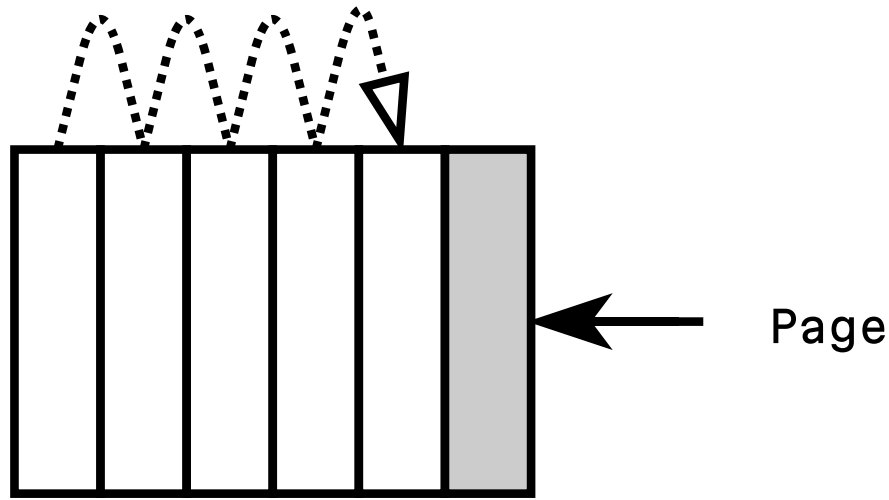


- tuple — a version of a row
- big values are stored in separate TOAST tables

Physical storage. Files



Sequential scan

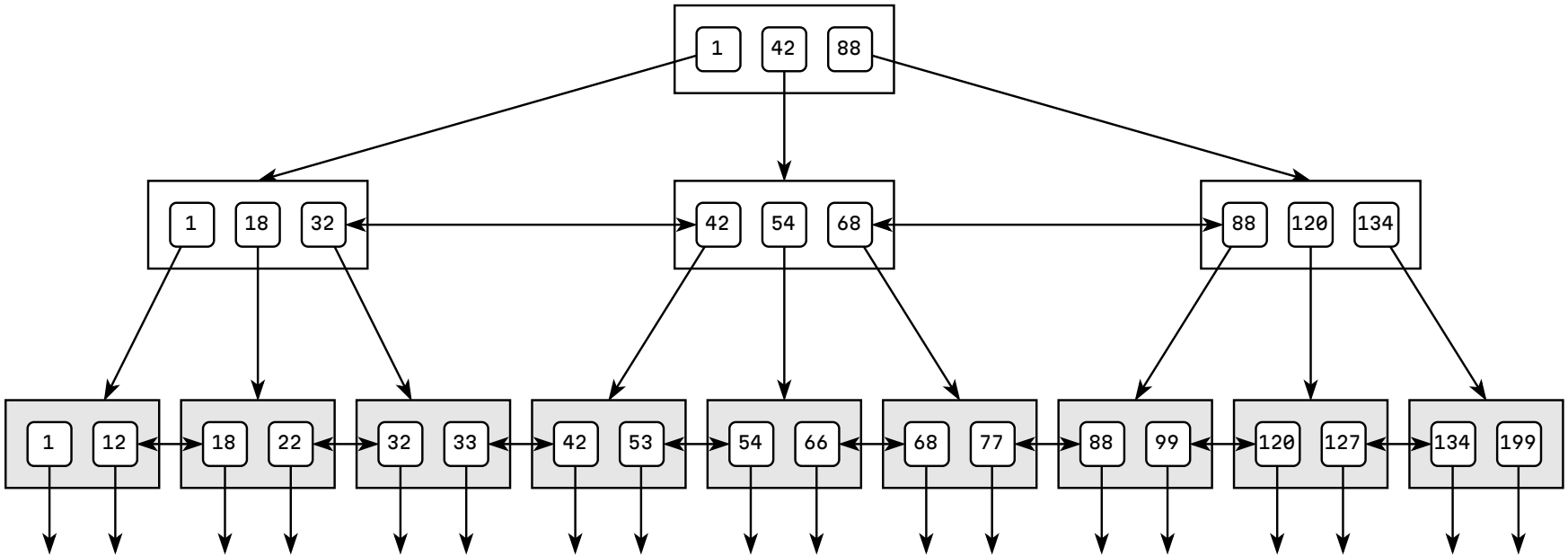


Sequential scan

- effective with low selectivity
- sequentially reads every page (`seq_page_cost = 1`)

Index scan

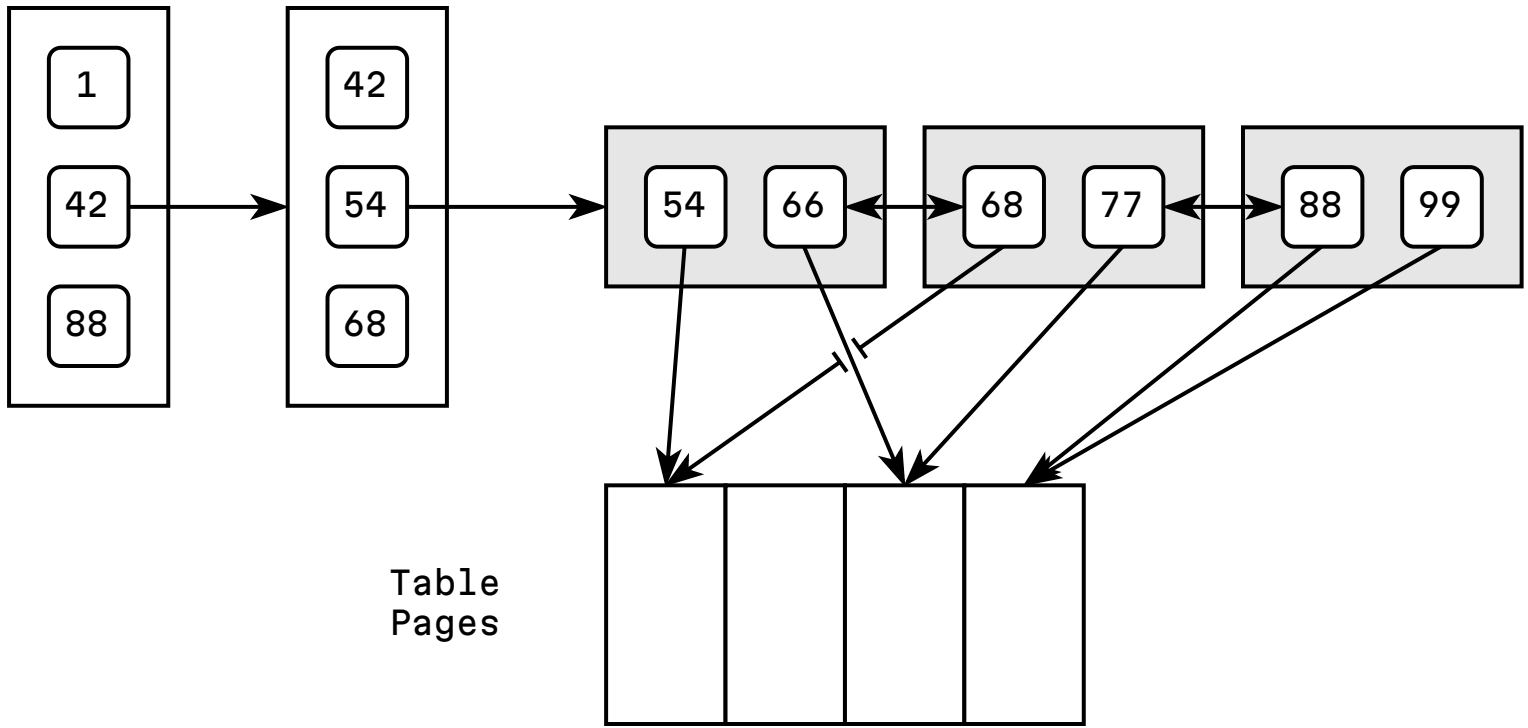
B-Tree index



Note: each page contains about 500 elements on average

Index scan

```
SELECT * FROM users WHERE id BETWEEN 50 AND 100
```

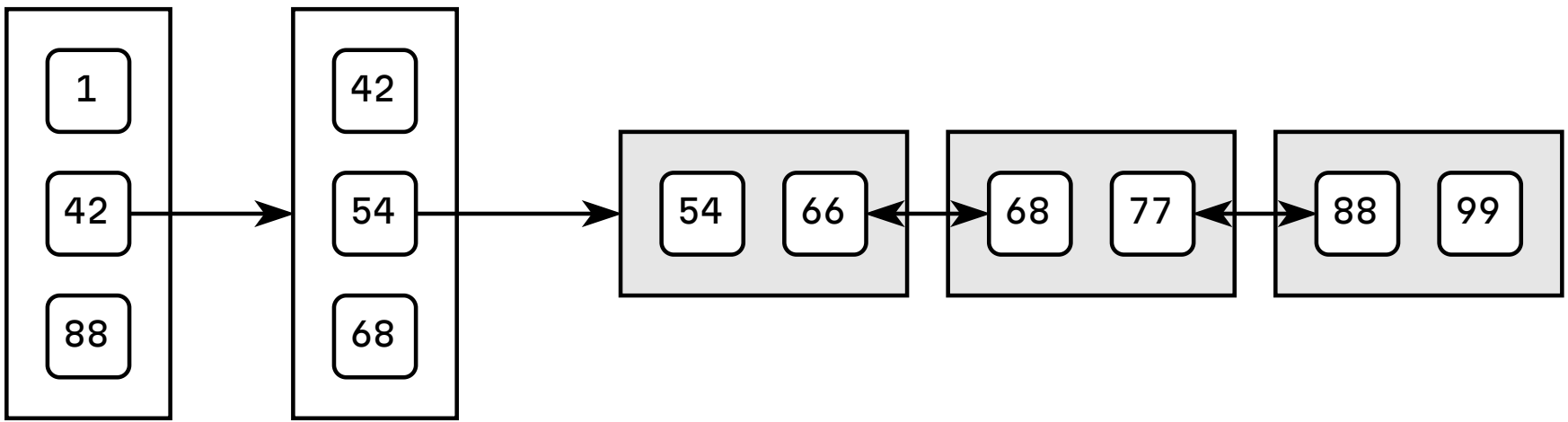


Index scan

- effective with high selectivity
- uses random access to read pages (`random_page_cost = 4`)
- returns sorted values

Index only scan

```
SELECT id FROM users WHERE id BETWEEN 50 AND 100
```



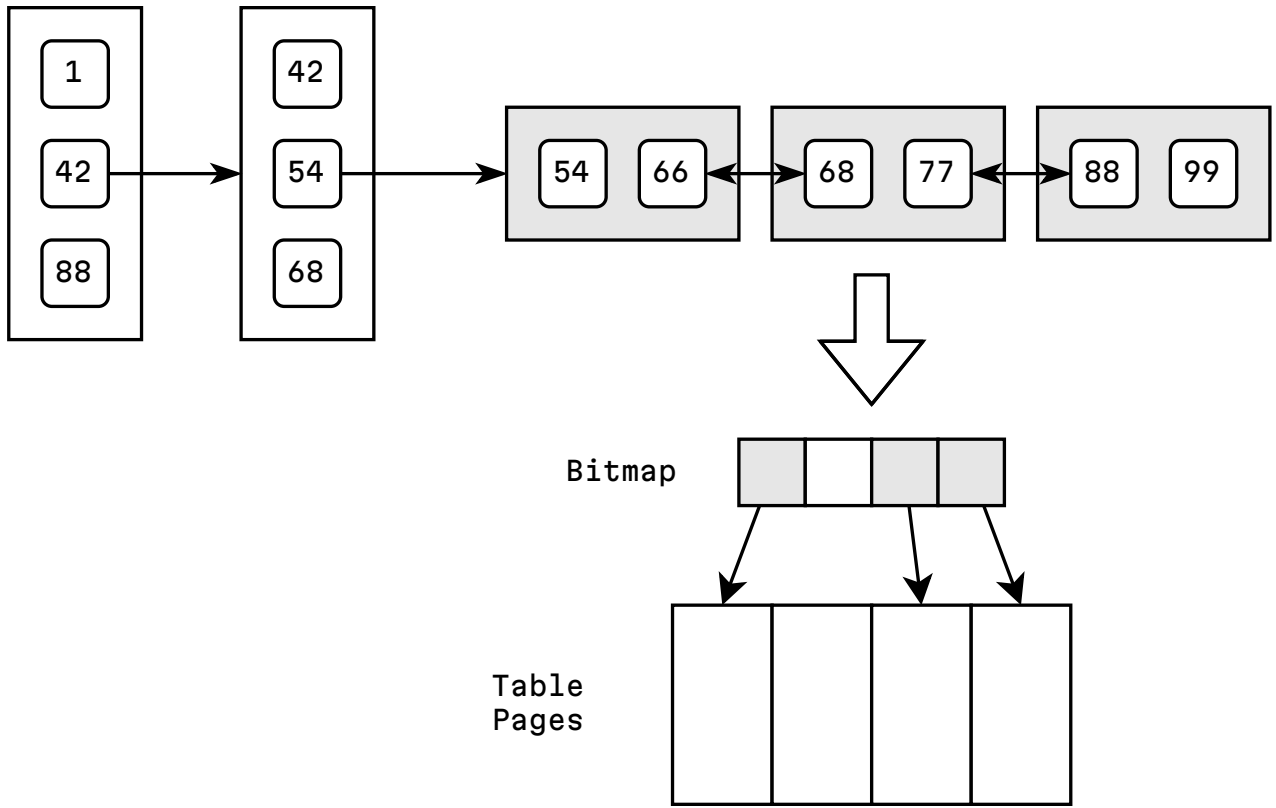
Index only scan

- the best data access strategy
- depends on visibility map
- can be used more often with covering indexes:

```
SELECT y FROM tab WHERE x = 'key';

CREATE INDEX tab_x_y ON tab(x) INCLUDE (y);
```

Bitmap scan

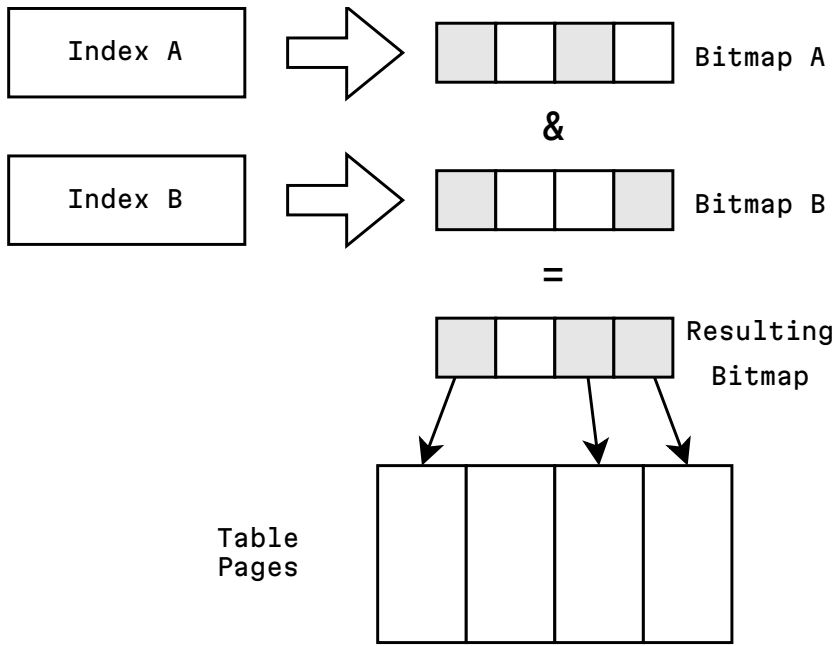


Bitmap scan

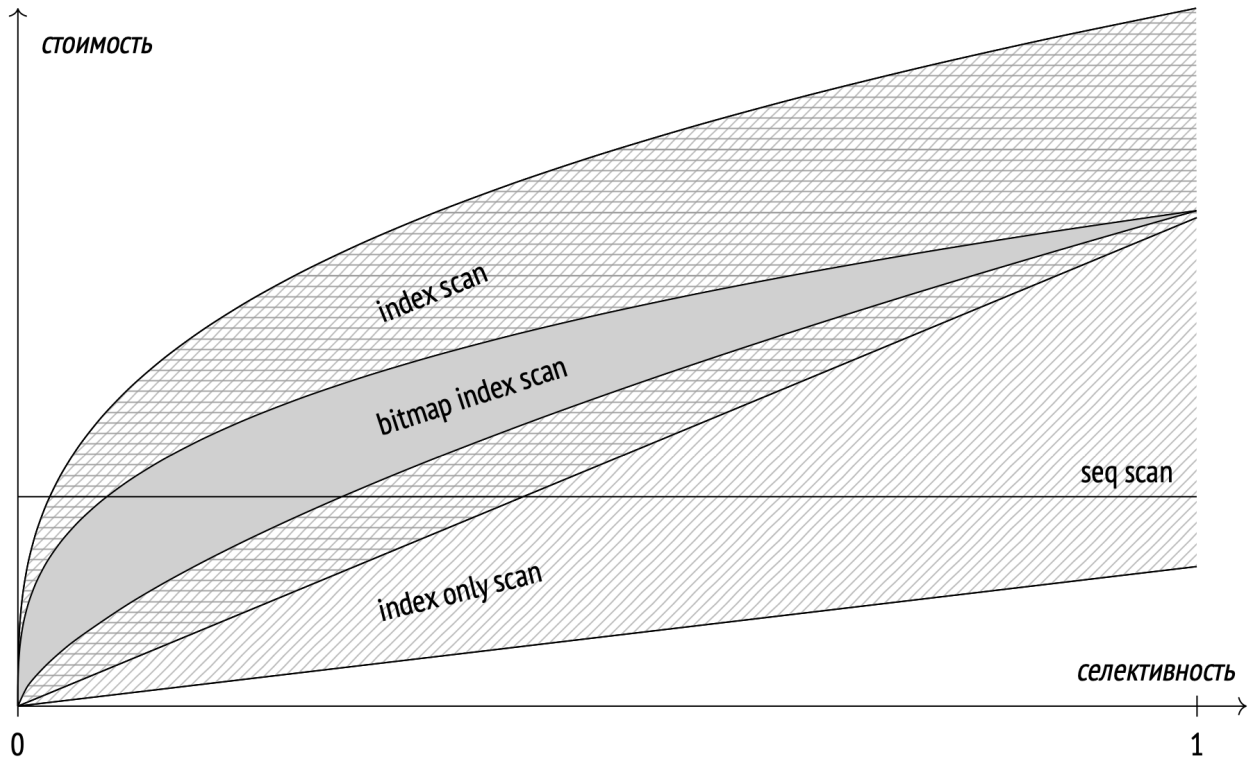
- effective with medium selectivity
- eliminates repeated reading of pages
- values are unsorted
- refers to specific tuples, or to pages when too many values (`Recheck Cond`)
- two step process: `Bitmap Index Scan` & `Bitmap Heap Scan`
- requires building a complete map before returning rows

Bitmap scan

Multiple indexes



Data Access. Comparison



Рогов Е. В., PostgreSQL 15 изнутри

Data Access. Comparison

```
EXPLAIN (ANALYZE, COSTS OFF) SELECT * FROM projects WHERE preset = 'angular' LIMIT 100;
```

```
Limit (actual time=0.017..5.186 rows=100 loops=1)
  ->  Seq Scan on projects (actual time=0.016..5.169 rows=100 loops=1)
        Filter: ((preset)::text = 'angular'::text)
        Rows Removed by Filter: 129
Planning Time: 10.113 ms
Execution Time: 5.230 ms
```

```
SET enable_seqscan = off;
```

```
Limit (actual time=4.169..77.817 rows=100 loops=1)
  ->  Index Scan using index_projects_on_preset on projects
                    (actual time=4.167..77.795 rows=100 loops=1)
        Index Cond: ((preset)::text = 'angular'::text)
Planning Time: 0.142 ms
Execution Time: 77.947 ms
```

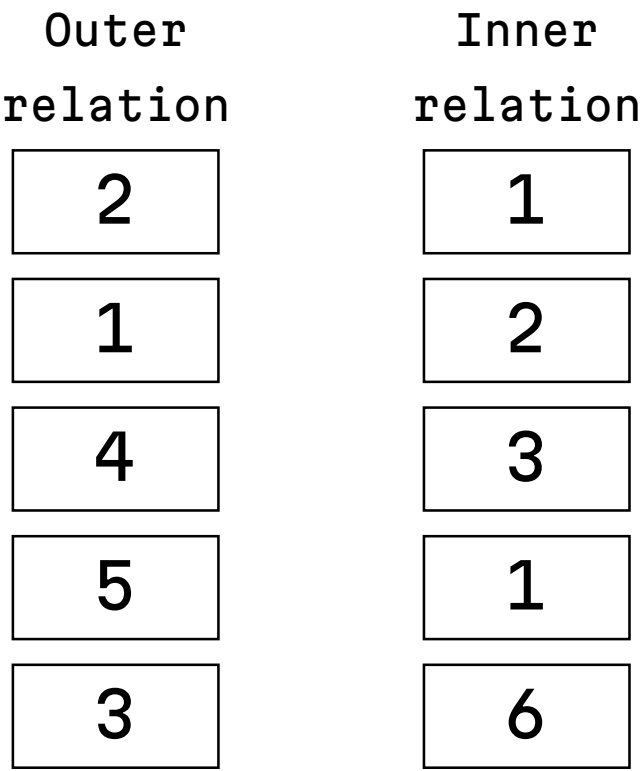
```
RESET enable_seqscan;
```

See Planner Method Configuration

Combining relations

- nested loop
- hash join
- merge join

Nested loop

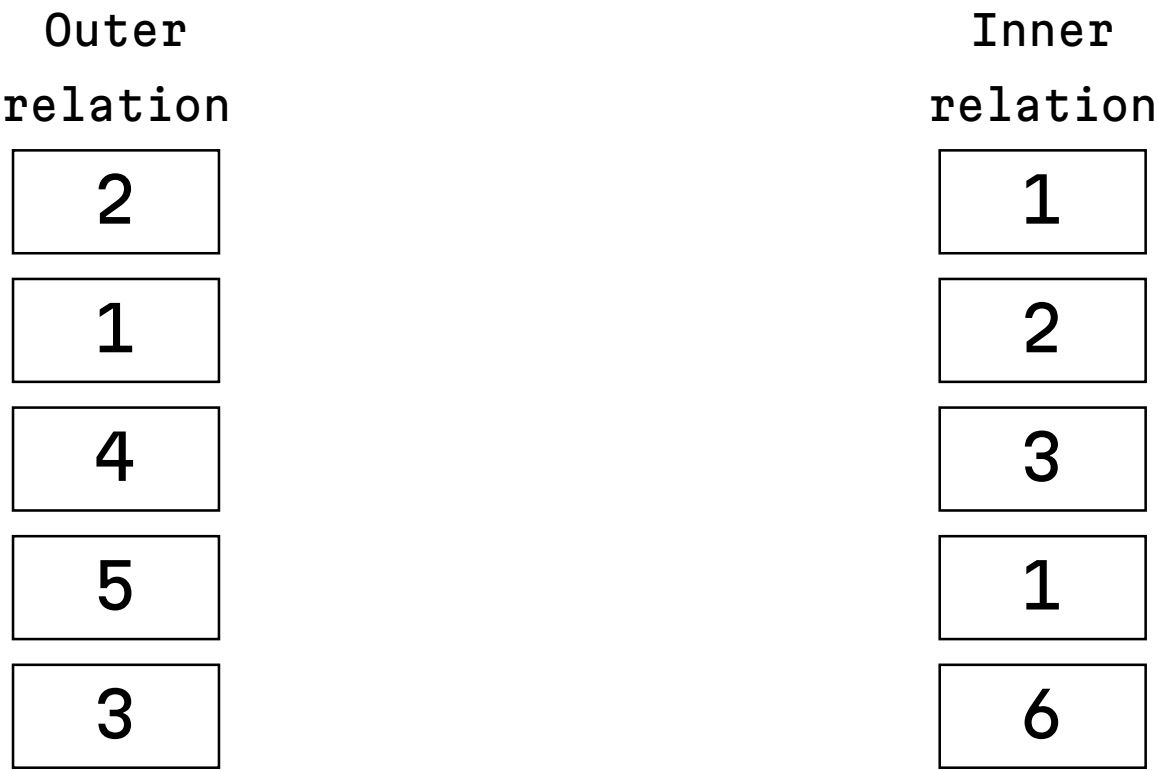



```ruby
outer.each do |i|
  inner.each do |j|
    i.col == j.col
  end
end
```

Nested loop

- effective only for a small number of rows
- no preps required
- supports joins by any condition

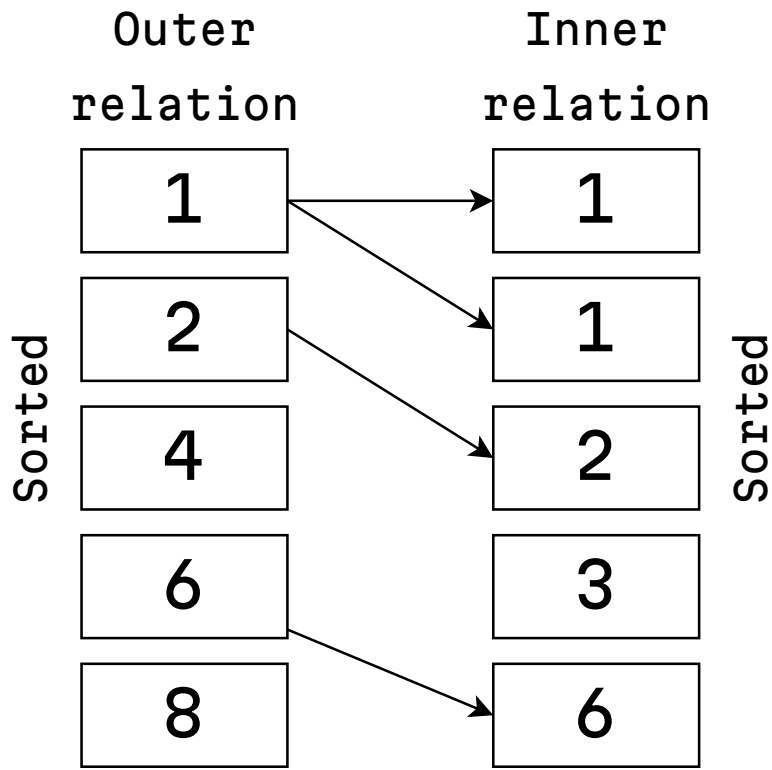
Hash join



Hash join

- effective for a large number of rows
- requires preparation of hash table
- starts using swap when there is not enough memory
- supports joins only by `=`

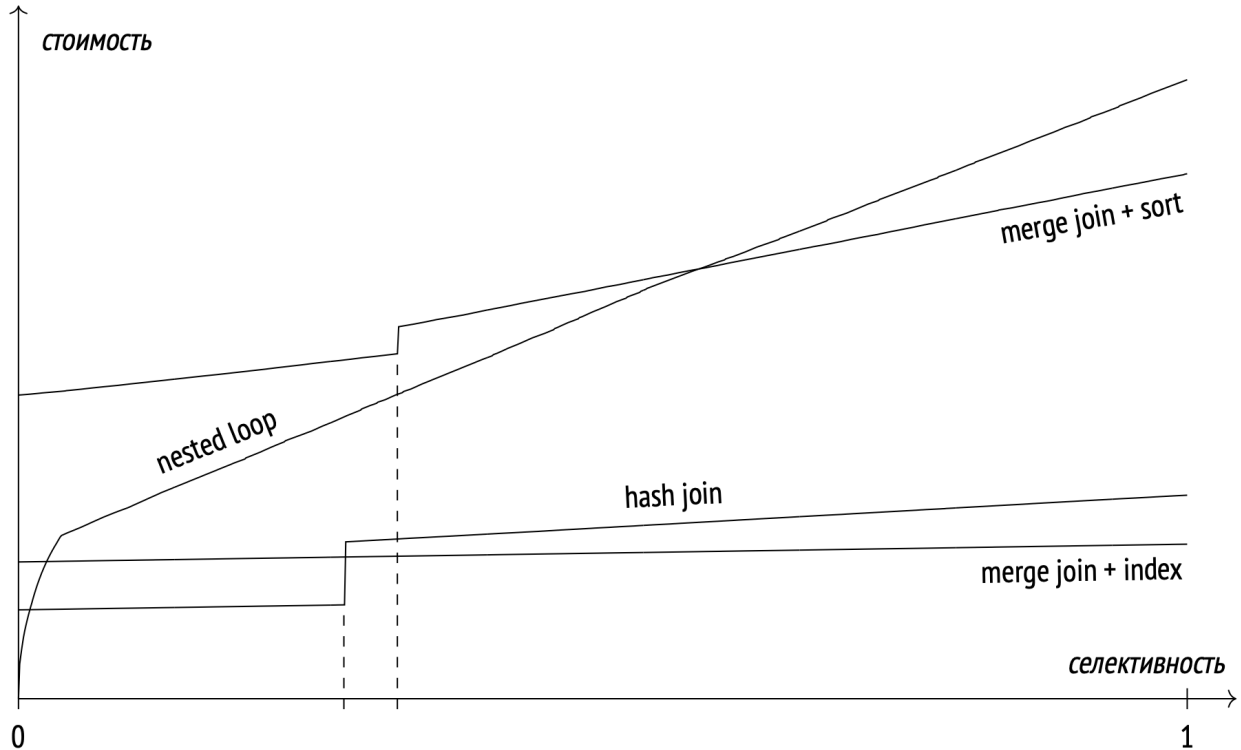
Merge join



Merge join

- effective for a large number of rows
- requires preparation unless rows already sorted
- supports joins only by `=`

Combining relations. Comparison



Рогов Е. В., PostgreSQL 15 изнутри

`EXPLAIN` VS `EXPLAIN ANALYZE`

```
EXPLAIN ANALYZE SELECT * FROM users WHERE created_at > '2021-01-01';
```

```
Bitmap Heap Scan on users (cost=3753.91..24070.21 rows=177224 width=366)
                  (actual time=48.260..13430.283 rows=175130 loops=1)
  Recheck Cond: (created_at > '2021-01-01'::timestamp)
  Heap Blocks: exact=13966
  ->  Bitmap Index Scan on users_created_at (cost=0.00..3709.61 rows=177224 width=0)
                                  (actual time=44.396..44.400 rows=195678 loops=1)
        Index Cond: (created_at > '2021-01-01'::timestamp)
Planning Time: 0.197 ms
Execution Time: 13443.429 ms
```

- `actual time=48.260..13430.283` — milliseconds of real time
- `actual rows=175130` — actual number of rows
- `Planning Time: 0.197 ms` — time to generate the query plan
- `Execution Time: 13443.429 ms` — time of the plan execution

Statistics

- metadata for tables and columns
- changes over time (different plan can be chosen for the same query)

Ways to update statistics

- Automatic (`autovacuum_enabled = on`)
- Manual `ANALYZE` / `VACUUM ANALYZE`

Table(-ish) statistics (`pg_class`)

- `reltuples` — estimated number of live rows
- `relpages` — estimated size of the on-disk pages
- `relallvisible` — number of pages that are marked all-visible in the table's visibility map
- PG uses `300 × default_statistics_target (default: 100)` random rows for this analysis

Table(-ish) statistics (`pg_class`)

```
SELECT reltuples, relpages, relallvisible
FROM pg_class WHERE relname = 'projects';
```

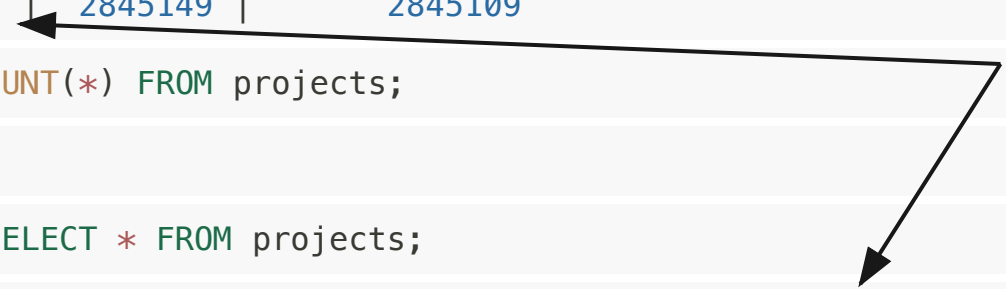
```
reltuples | relpages | relallvisible
----------+----------+--------------
 10726556 |  2845149 |       2845109
```

```
SELECT COUNT(*) FROM projects;
```

```
10860693
```

```
EXPLAIN SELECT * FROM projects;
```

```
Seq Scan on projects  (cost=0.00..2952414.56 rows=10726556 width=1505)
```



Column(-ish) statistics (`pg_stats`)

- `null_frac` — fraction of rows with `NULL` values
- `avg_width` — average width in bytes of column's entries
- `n_distinct` — estimated number of distinct values
- `correlation` — correlation between physical row ordering and logical ordering (Bitmap vs Index scan)

Column(-ish) statistics (`pg_stats`)

```
SELECT attname, avg_width, null_frac, n_distinct, correlation
FROM pg_stats
WHERE tablename = 'projects'
  AND attname IN ('slug', 'deleted_at');
```

```
 attname    | avg_width | null_frac | n_distinct | correlation
      slug |        18 |         0 |         -1 | -0.02839662
deleted_at |         8 |    0.9272 | -0.0727999 | -0.10090815
```

- what is the `correlation` value for the `created_at` column?

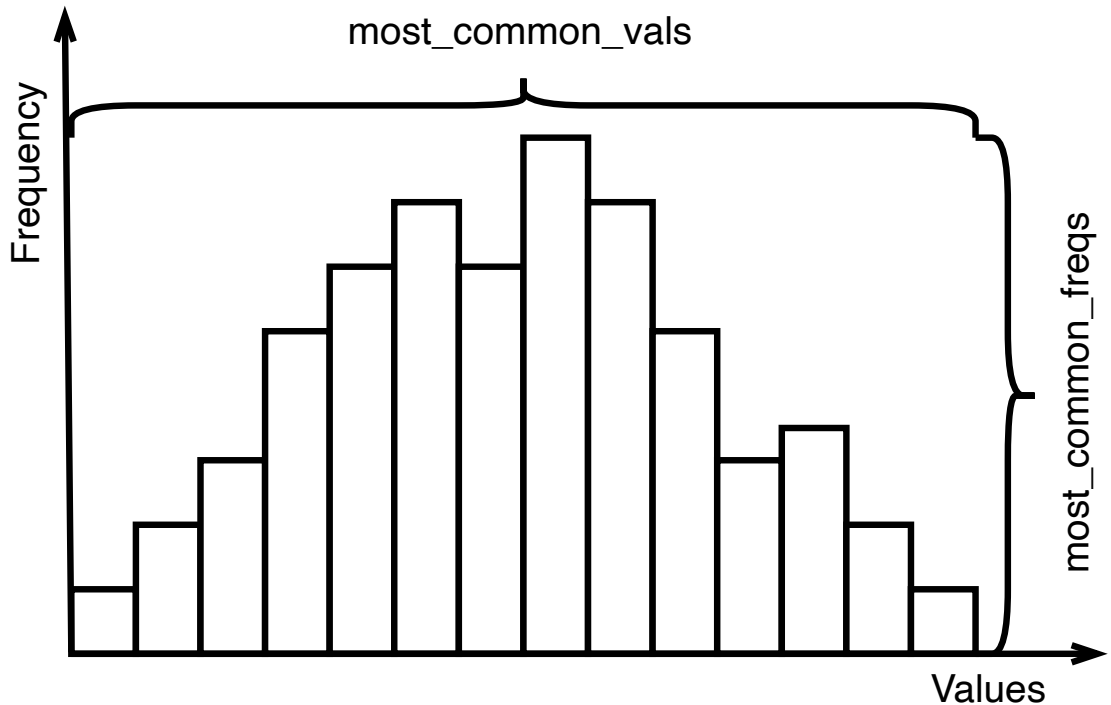
```
 attname    | avg_width | null_frac | n_distinct | correlation
created_at |         8 |         0 |         -1 | -0.02188248
```

- Why? Because of fragmentation (see MVCC)

Column(-ish) statistics (`pg_stats`)

- `most_common_vals`, `most_common_freqs` — common values statistics
- `histogram_bounds` — common values histogram
- `most_common_elems`, `most_common_elem_freqs`, `elem_count_histogram` — statistics for arrays, tsvector, etc.

`pg_stats.most_common_vals` & `pg_stats.most_common_freqs`



`pg_stats.most_common_vals` & `pg_stats.most_common_freqs`

- length of arrays is `default_statistics_target`
- can be altered `ALTER TABLE ... ALTER COLUMN ... SET STATISTICS ...;`

`pg_stats.most_common_vals` & `pg_stats.most_common_freqs`

```
SELECT attname, most_common_vals AS mcv, most_common_freqs AS mcf
FROM pg_stats
WHERE tablename = 'projects' AND attname = 'visibility';
```

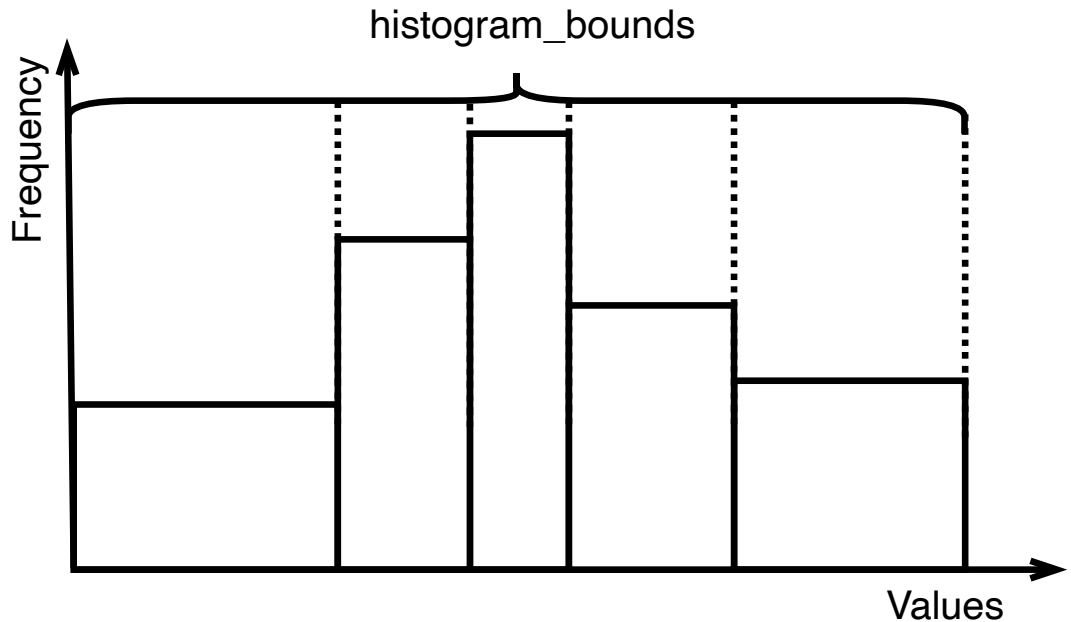
```
attname    | mcv                        | mcf
visibility | {public,private,secret} | {0.99647,0.00270,0.00083}
```

```
EXPLAIN SELECT * FROM projects WHERE visibility = 'private';
```

```
Seq Scan on projects  (cost=0.00..2979230.95 rows=28962 width=1505)
  Filter: (visibility = 'private'::project_visibility)
```

- private: 0.00270 * 10,726,556 = 28,962 rows

`pg_stats.histogram_bounds`



- number of buckets ≤ `default_statistics_target`

`pg_stats.histogram_bounds`

```
SELECT attname,
       array_length(histogram_bounds, 1) as bounds_count,
       left(histogram_bounds::text,60) || '...' AS hist_bounds
FROM pg_stats s WHERE s.tablename = 'projects' AND s.attname = 'created_at';
```

```
attname    | bounds_count | hist_bounds
-----------+--------------+------------------------------
created_at |          101 | {"2017-08-03 06:37:15.351392","2017-11-23 05:51:46.556693",".
```

- 10,726,556 / 100 = 107,266 rows in one bucket

```
EXPLAIN SELECT * FROM projects WHERE created_at <= '2017-11-23 05:51:46.556693';
```

```
Index Scan using projects_created_at on projects
      (cost=0.43..428434.82 rows=107266 width=1505)
  Index Cond: (created_at <= '2017-11-23 05:51:46.556693'::timestamp without time zone)
```

```
SELECT COUNT(*) FROM projects WHERE created_at <= '2017-11-23 05:51:46.556693';
```

```
113158
```

Cardinality of function

- Planner doesn’t know what result the function will return and uses the default cardinality: 0.5%

Cardinality of function

```
EXPLAIN SELECT * FROM projects
WHERE created_at BETWEEN '2017-01-01 00:00:00' AND '2017-12-31 23:59:59';
```

```
Index Scan using projects_created_at on projects
        (cost=0.43..632257.71 rows=158385 width=1505)
  Index Cond: ((created_at >= '2017-01-01 00:00:00') AND (created_at <= '2017-12-31 23:59
```

- ~ 158,385 rows

```
SELECT count(*) FROM projects
WHERE created_at between '2017-01-01 00:00:00' and '2017-12-31 23:59:59';
```

- 165,490 rows

Cardinality of function

```
EXPLAIN SELECT * FROM projects WHERE EXTRACT(YEAR FROM created_at) = 2017;
```

```
Seq Scan on projects  (cost=0.00..3006047.34 rows=53633 width=1505)
  Filter: (date_part('year'::text, created_at) = '2017'::double precision)
```

- ~ 53,633 rows (0.5% of all rows)
- What result will the planner return for `2029`?

```
EXPLAIN SELECT * FROM projects WHERE EXTRACT(YEAR FROM created_at) = 2029;
```

```
Seq Scan on projects  (cost=0.00..3008059.40 rows=54303 width=1507)
  Filter: (date_part('year'::text, created_at) = '2029'::double precision)
```

- ~ 53,633 rows as well

Cardinality of function

```
CREATE INDEX ON projects(extract(year FROM created_at));
```

```
ANALYZE projects;
```

```
EXPLAIN SELECT * FROM projects WHERE EXTRACT(YEAR FROM created_at) = 2017;
```

```
Index Scan using projects_date_part_idx on projects
        (cost=0.43..574284.93 rows=150579 width=1508)
  Index Cond: (date_part('year'::text, created_at) = '2017'::double precision)
```

- ~ 150,579 rows

Cardinality of function

```
SELECT n_distinct FROM pg_stats WHERE tablename = 'projects_date_part_idx';
```

```
n_distinct
------------
6
```

```
ALTER INDEX index_name ALTER COLUMN name SET STATISTICS 42;
```

Cardinality of function

PostgreSQL ≥ 14

```
CREATE STATISTICS created_at_year ON (extract(year FROM created_at)) FROM projects;
```

```
ALTER STATISTICS statistics_name SET STATISTICS 42;
```

- statistics data stored in `pg_statistic_ext` & `pg_statistic_ext_data`

Multivariate statistics

PostgreSQL ≥ 10

- It's possible to collect statistics covering multiple columns:
  - functional dependency

```
CREATE STATISTICS idx_dep(dependencies) ON gender, job_title FROM profs;
```

  - n-distinct statistics

```
CREATE STATISTICS idx_nd(ndistinct) ON gender, job_title FROM profs;
```

  - most-common values lists

```
CREATE STATISTICS idx_mcv(mcv) ON gender, job_title FROM profs;
```

Statistics. Summary

- predicts cardinality (rows)
- predicts average width of rows
- predicts number of pages to read
- helps planner to decide what **physical operations** to use

Real life example

```sql
SELECT p.*
FROM projects p
JOIN project_dependencies pd ON p.id = pd.project_id
WHERE pd.data->'packages' ? 'react'
AND p.deleted_at IS NULL
ORDER BY p.forks_count DESC
LIMIT 10;
```

Real life example

- `react` => 1-200ms
- `nanoid` => 10-1300ms
- `next` => 40s-60s

Real life example

```
EXPLAIN ANALYZE SELECT ...

Limit (cost=0.87..7979.33 rows=10 width=12)
      (actual time=25.362..39896.966 rows=10 loops=1)
  ->  Nested Loop (cost=0.87..98463773.75 rows=123412 width=12)
                  (actual time=25.360..39896.704 rows=10 loops=1)
        ->  Index Scan Backward using proj_forks_count_idx on projects
                  (cost=0.43..14777766.90 rows=10029488 width=12)
                  (actual time=2.711..18215.382 rows=19180 loops=1)
        ->  Index Scan using proj_deps_projs_idx on project_dependencies
                  (cost=0.43..8.34 rows=1 width=8)
                  (actual time=1.129..1.129 rows=0 loops=19180)
              Index Cond: (project_id = projects.id)
              Filter: ((data -> 'packages'::text) ? 'next'::text)
              Rows Removed by Filter: 1
Planning Time: 27.642 ms
Execution Time: 39897.861 ms
```

How to optimize

Different strategies for two query types:

- "short" queries (must use index scan)
- "long" queries (have to use seq scan)

"Short" queries

"Short" queries. Selectivity

- column `status` with 3 values in it: `new`, `active`, `complete`
- should we create an index?
- no, it won't be used because of low selectivity

"Short" queries. Selectivity

- do not create indexes for columns with low selectivity
- consider creating partial indexes:

```
CREATE INDEX tickets_status ON tickets(status) WHERE status = 'new';
```

- but remember that you might break HOT updates (see MVCC)

"Short" queries. Multiple vs Multicolumn indexes

- PostgreSQL can use multiple indexes using bitmaps
- but, those bitmaps might be gigantic:

```
EXPLAIN SELECT * FROM projects WHERE forked_from_id = 1
ORDER BY updated_at DESC LIMIT 10
```

```
Limit  (cost=2816580.03..2816580.05 rows=10 width=1507)
  ->  Sort  (cost=2816580.03..2820760.49 rows=1672185 width=1507)
        Sort Key: updated_at DESC
        ->  Bitmap Heap Scan on projects
                    (cost=91151.99..2780444.71 rows=1672185 width=1507)
              Recheck Cond: (forked_from_id = 1)
              ->  Bitmap Index Scan on index_projects_on_forked_from_id
                      (cost=0.00..90733.95 rows=1672185 width=0)
                    Index Cond: (forked_from_id = 1)
```

"Short" queries. Multiple vs Multicolumn indexes

- in such cases multicolumn index might help:

```
CREATE INDEX ON projects(forked_from_id, updated_at DESC);
```

```
Limit  (cost=40.08..79.73 rows=10 width=1507)
  ->  Index Scan using projects_forked_from_id_updated_at_idx
                on projects (cost=0.43..6629764.44 rows=1672185 width=1507)
        Index Cond: (forked_from_id = 1)
```

"Short" queries. Multiple vs Multicolumn indexes

- multicolumn indexes are bigger
- `(X, Y, Z)` index can be used for queries by `X`, `XY`, `XYZ`, or even `XZ`, but not `Y`/`YZ`
- `DESC`/`ASC` does matter in multicolumn indexes
- think of indexes as of **sorted lists**

"Short" queries. Works `like` a charm

- PostgreSQL will use Btree index for `like` operator

Unless it won't:

- `email like '%@gmail.com'` won't work
- `lower(username) like 'palk%'` won't work
- only `username like 'palk%'`
- if `LC_COLLATE` != `C`, use operator classes `text_pattern_ops`:

```
CREATE INDEX users_last_name ON users (last_name text_pattern_ops);
```

"Short" queries. Redundant filters

- you can help planner by providing redundant filtering criterias

```
SELECT * FROM profs
WHERE address = 'Lavrushinsky Ln, 10';
```

```
SELECT * FROM profs
WHERE address = 'Lavrushinsky Ln, 10' AND city = "Moscow";
```

- unless functional dependency will make it worse (`CREATE STATISTICS`)

"Short" queries. Jsonb

- jsonb is awesome
- for storing data, but not for querying
- big jsonb's stored in TOAST
- extract data from jsonb to a column to fix condition rechecks speed
- do not `` `SELECT *` `` (TOAST + serialization on the backend)

"Short" queries. Create or not create indexes

Data always changing:

- do not optimize prematurely
- watch `pg_stat_all_indexes` (see Index Maintenance)
- more indexes — slower updates
- bigger indexes — more memory used

"Long" queries

"Long" queries. Joins

- aim for Hash join instead of Nested loop (when possible)
- first use Semi-joins (`EXISTS` / `IN` / `EXCEPT`)
  and Anti-joins (`NOT EXISTS` / `NOT IN` / `INTERSECT`)

"Long" queries. Joins

- PostgreSQL stops reordering joins after `join_collapse_limit (default: 8)` is exceeded

```
EXPLAIN
SELECT *
  FROM projects p
  JOIN project_dependencies pd ON p.id = pd.project_id
  WHERE p.id IN (SELECT project_id FROM collection_projects WHERE collection_id = 42)
```

```
Nested Loop (cost=0.87..18.67 rows=1 width=1911)
  ...
```

```
SET join_collapse_limit = 1;
```

```
Hash Join (cost=852104.51..5676891.77 rows=1 width=1911)
  ...
```

"Long" queries. Help planner to use existing statistics

- `EXCEPT` instead of `NOT EXISTS` / `NOT IN`
- `INTERSECT` instead of `EXISTS` / `IN`
- `UNION` instead of `OR`

"Long" queries. Watch buffers to avoid multiple seq scans

- use `EXPLAIN (ANALYZE, BUFFERS)` to check the number of fetched pages

```
EXPLAIN (ANALYZE, BUFFERS, COSTS FALSE, TIMING FALSE, SUMMARY FALSE)
SELECT * FROM projects WHERE slug LIKE 'slidev%' LIMIT 20;
```

```
Limit (actual rows=20 loops=1)
  Buffers: shared hit=15 read=3907
  I/O Timings: read=219.956
  ->  Seq Scan on projects (actual rows=20 loops=1)
        Filter: ((slug)::text ~~ 'slidev%'::text)
        Rows Removed by Filter: 10033
        Buffers: shared hit=15 read=3907
        I/O Timings: read=219.956
```

"Long" queries. Views

- prepared and saved query, quite readable
- folded by PostgreSQL
- but it’s imposible to inject logic into the view’s query

"Long" queries. Materialized Views

- views with stored results
- have statistics, can be used with indexes, etc — helpful to the planner
- but they must be updated manually via `REFRESH`
- materialized views with `ORDER BY` => `correlation = 1`

"Long" queries. `WITH` Queries (CTE)

- makes requests more readable:

```
WITH sub_query AS (
  SELECT * FROM big_table -- ...
)
SELECT * FROM sub_query WHERE key = 123;
```

- folded by default (in PostgreSQL ≥ 12)
- materialized if CTE is called multiple times
- can be fixed via `NOT MATERIALIZED`:

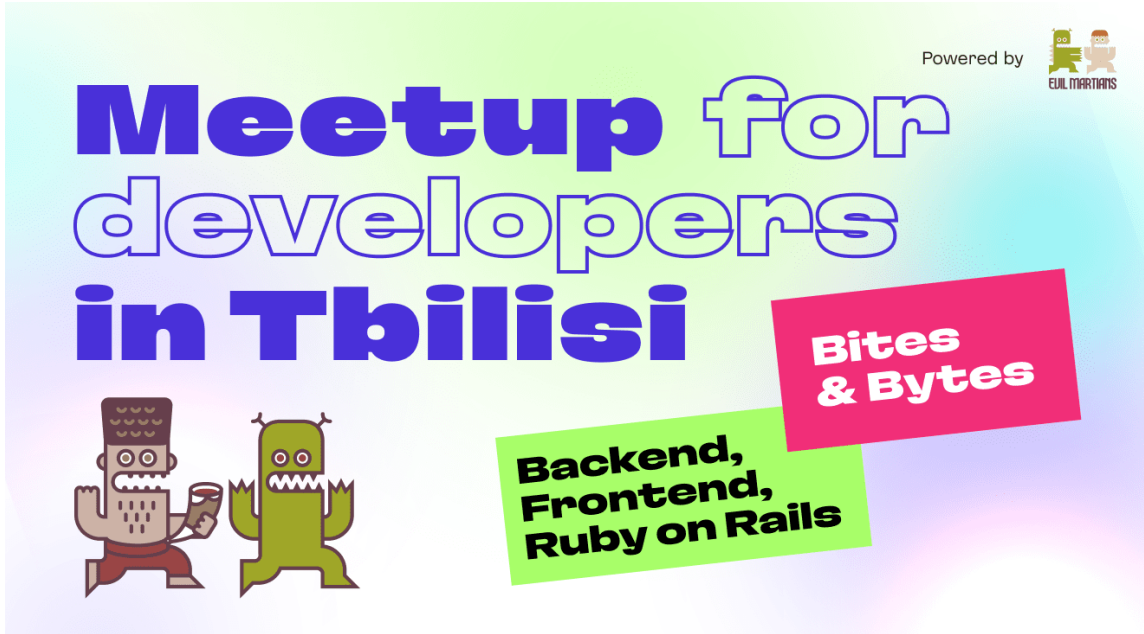
```
WITH sub_query AS NOT MATERIALIZED (
  SELECT * FROM big_table -- ...
)
SELECT * FROM sub_query AS sq1
  JOIN sub_query AS sq2 ON sq1.key = sq2.ref WHERE sq2.key = 123;
```

"Long" queries. Partitioning

- partitioning reduces the amount of data and makes data access less expensive
- read more in Martian blog

Stay in touch ❤️

- Vacancies: You X Martians
- Our blog: Martian Chronicles
- Thinknetica: Best Ruby On Rails Courses



March 30, 2023: Meetup in Geogia

Recap

- PostgreSQL can fetch and join data with a number of physical methods
- statistics help PostgreSQL choose the right method
- we can help PostgreSQL by providing better statistics
- make PostgreSQL use indexes for "short" queries
- optimize buffer reads for "long" queries

Know your tools