Beyond the Basics: Mastering PostgreSQL Index Performance

Federico Campoli

PostgreSQL DBA, because freaking miracle worker is not a job title

PGDay Austria, Vienna, 4th September 2025

Few words about the speaker

KAMEDATA
DEVOPS AND DATA ENGINEERING

- Born in 1972
- Passionate about IT since 1985
- In love with PostgreSQL since 2006
- PostgreSQL and FreeBSD tattoo on the right shoulder
- Amateur Jazz Guitarist
- Amateur Road and MTB Cyclist
- Freelance DBA at Kamedata

Getting in touch

- **Linkedin:** https://www.linkedin.com/in/federicocampoli/
- **Codeberg:** https://codeberg.com/the4thdoctor
- **Bluesky:** https://bsky.app/profile/pgdba.org
- **Mastodon:** https://fosstodon.org/@4thdoctor_scarf
- **Blog:** https://pgdba.org

Table of contents

Grand départ

From Wikipedia:

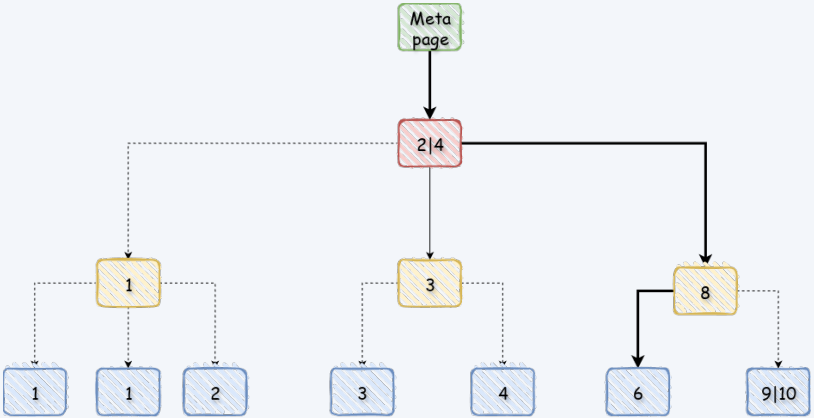
A database index is a data structure that improves the speed of data retrieval.
Indexes are used to quickly locate data without having to search every row in a database table every time said table is accessed.

La forme

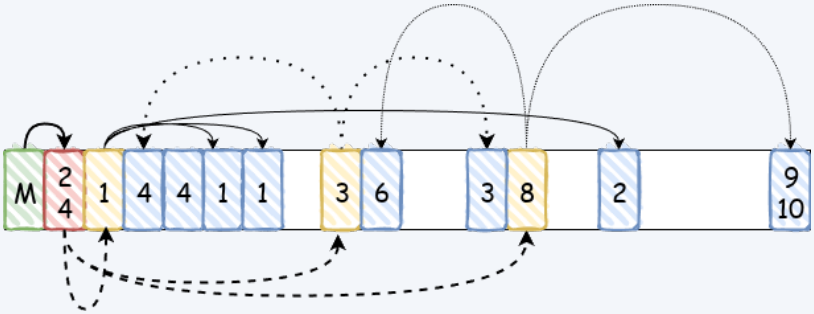
- Nowadays everything is cloud based
- It's just an hallucination though
- The physical storage still fights for the users

La forme

- After a quick dive in the physical world...
- we'll see how an index can improve the query performance
- ... or make things worse



The logical representation of a b-tree is apparently very linear and easy to understand. However "Not all that glitters is gold".



In the real world an index appears like a mess of interconnections.


Insta360 Ace Pro
Like
MugelloCircuit
mugello

The database directories

Within the data area into the subdirectory $PGDATA/base there are folders with a numerical name.
Each folder contains the data files belonging to the corresponding database in the instance.

The data files

The data files have also a numerical name.
When the relation is created then the file name is set to the value of the relation's object identifier (OID).
The relation's filename is stored into the column relfilenode of pg_class and can change under particular conditions (VACUUM FULL, REINDEX).

The data files

A data file's max size is 1 GB.
If the data exceeds the first file then a new file with the same name and a numerical suffix is created.
The first segment doesn't have suffix though.

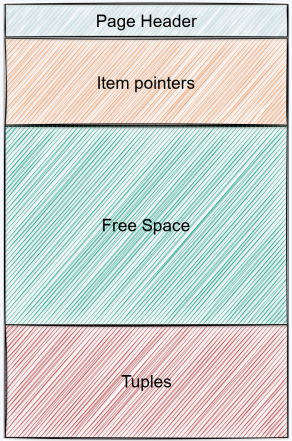
The data pages

Either table and index segments are then organised in "pages".
The page size is set at compile time.
The defacto standard is to have 8kb pages.

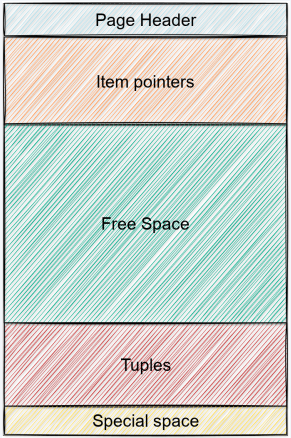
The data pages

The heap pages, belonging to the table's datafiles, and index pages have almost the same structure with one tiny but fundamental difference.

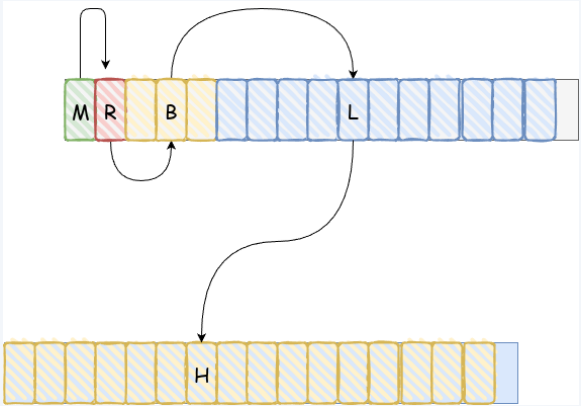
Heap pages



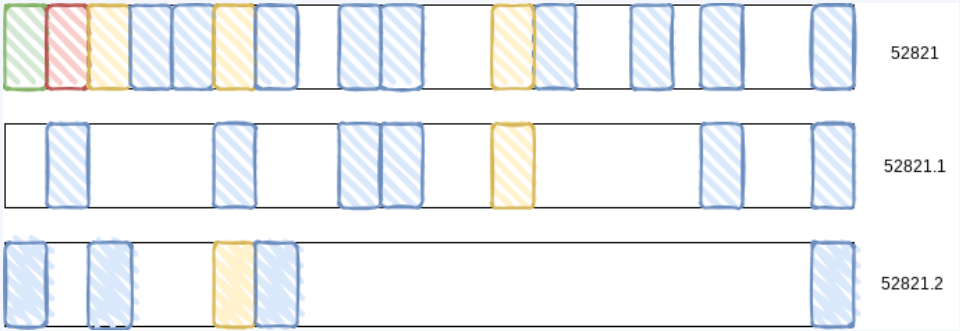
Index pages



A freshly created index



A slightly bloated index file



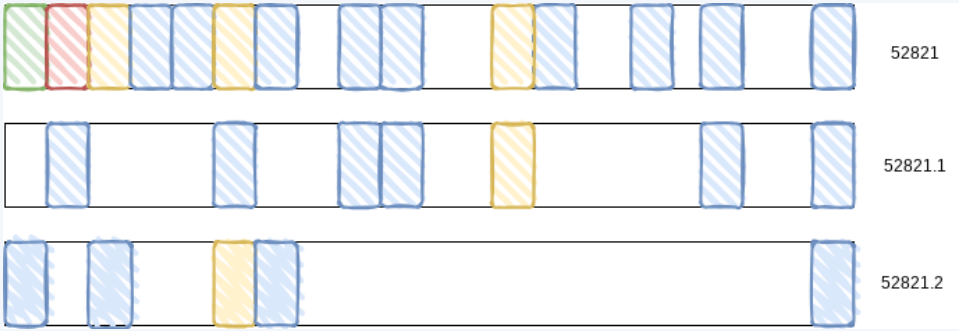
Bloating the index

- Inserts and delete update the index entries accordingly
- The update in PostgreSQL is an insert/delete within a single transaction
- It may not upate the index if the new row version doesn't change page
- If the row change the page then the index is updated
- If the index page is full then a new page is allocated
- The index dead rows are cleaned by VACUUM as well
- however...

Looking after the index

- VACUUM is not so effective with the indices
- The free space can be used when the tuple's data fits in the index structure
- Index pages that contains all dead rows can be reused only after two VACUUM runs
- The first VACUUM marks the index page as deleted with the XID which follows VACUUM's XID
- The second VACUUM returns the deleted page to the free space
- The real index maintenance is therefore the REINDEX (CONCURRENTLY?)

A slightly bloated index file




```
SELECT oid,oid::regclass AS relname,relfilenode,relkind from pg_class where oid
    =52821;

  oid  | relname  | relfilenode | relkind
-------+----------+-------------+---------
 52821 | index_01 |       52821 | i
(1 row)
```

After the reindex


```
REINDEX INDEX index_01;
REINDEX
SELECT oid,oid::regclass AS relname,relfilenode,relkind from pg_class where oid
    =52821;

  oid  | relname  | relfilenode | relkind
-------+----------+-------------+---------
 52821 | index_01 |       54058 | i
(1 row)
```

Attention

REINDEX is a blocking procedure.
The index's table is set in read only during the reindex and any query that would use the index will have to wait for the reindex to finish.
It's possible to REINDEX CONCURRENTLY but beware of unexpected impact on the database as described in this presentation.
LFMF: how a CREATE INDEX CONCURRENTLY led to a 6 hour downtime

Contre-la-montre individuel

Setup environment

Data area on Samsung SSD 870 EVO 1TB with partition EXT4 luks encrypted

```
SELECT version();
                                  version
-------------------------------------------------------------------------------
 PostgreSQL 17.6 on x86_64-pc-linux-gnu, compiled by gcc (GCC) 11.2.0, 64-bit
(1 row)

SHOW shared_buffers ;
 shared_buffers
----------------
 1GB
(1 row)
```

Table without any primary key or indices

KAMEDATA
DEVOPS AND DATA ENGINEERING

```
CREATE TABLE t_benchmark
(
    id bigint,
    text_value text,
    hashed_value text
);
```

Add a function to build random sentences

KAMEDATA
DEVOPS AND DATA ENGINEERING

```
CREATE OR REPLACE FUNCTION random_sentence()
    RETURNS TEXT as $$
WITH dictionary AS (
    SELECT ARRAY[
        'postgres', 'database', 'query', 'sql', 'generate', 'random',
        'series', 'function', 'array', 'select', 'string', 'aggregate',
        'table', 'index', 'performance', 'window', 'cluster', 'schema',
        'elephant', 'slonik', 'postgresql', 'amazing', 'wal'
    ] AS words
)
SELECT
    (
        SELECT
            string_agg(word, ' ') AS random_text
        FROM (
            -- for each number in the inner series, pick one random word from our dictionary
            -- using array_length to determine the upper bound for random.
            SELECT
                d.words[random(1,array_length(d.words, 1))::int] AS word
            FROM
                dictionary d,
                -- Generate a series for a random sentence length (e.g., 5 to 30 words).
                generate_series(1, (5 + random(1,30))::int)
        ) AS random_words
    )
;
$$ LANGUAGE sql;
```

Function to build random sentences

```
SELECT random_sentence();
                              random_sentence
-------------------------------------------------------------------
 select aggregate elephant array slonik performance elephant database wal window
(1 row)

SELECT random_sentence();
                                                  random_sentence
-----------------------------------------------------------------------------------------------
 string amazing select wal series aggregate cluster random select performance performance aggregate index
      function
(1 row)
```

Store the 10 million rows into a separate table

KAMEDATA
DEVOPS AND DATA ENGINEERING

```
CREATE TABLE t_random_data
(
    id bigint GENERATED ALWAYS AS IDENTITY,
    text_value text,
    hashed_value text
);
\timing

INSERT INTO t_random_data (text_value, hashed_value)
SELECT
        random_sentence,
        md5(random_sentence)
FROM
(
    SELECT random_sentence() AS random_sentence
    FROM generate_series(1,10000000)
) r
;
INSERT 0 10000000
Time: 145022.300 ms (02:25.022)
```

Insert in table without index


```
INSERT INTO t_benchmark (id,text_value,hashed_value)
    SELECT
        id,
        text_value,
        hashed_value FROM t_random_data;
INSERT 0 10000000
Time: 29844.131 ms (00:29.844)


\dt+ t_benchmark
                                    List of relations
 Schema |    Name     | Type  |   Owner   | Persistence | Access method |  Size   | Description
--------+-------------+-------+-----------+-------------+---------------+---------+-------------
 public | t_benchmark | table | thedoctor | permanent   | heap          | 2220 MB |
(1 row)
```

Reading from the table without any index

KAMEDATA DEVOPS AND DATA ENGINEERING

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT
        *
FROM
        t_benchmark
WHERE
        text_value='aggregate window generate sql amazing performance';

                                                                QUERY PLAN
--
     ------------------------------------------------------------------------------------------------------------------------------------------

 Gather  (cost=1000.00..337163.80 rows=1 width=198) (actual time=552.198..566.989 rows=1 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   Buffers: shared hit=130627 read=153450
   ->  Parallel Seq Scan on t_benchmark  (cost=0.00..336163.70 rows=1 width=198) (actual time=402.865..547.145
        rows=0 loops=3)
         Filter: (text_value = 'aggregate window generate sql amazing performance'::text)
         Rows Removed by Filter: 3333333
         Buffers: shared hit=130627 read=153450
 Planning Time: 0.194 ms
 Execution Time: 567.077 ms
(10 rows)

Time: 567.762 ms
```

Trying to increase the parallel workers

```
SET max_parallel_workers_per_gather=4;
SET
Time: 0.468 ms
EXPLAIN (ANALYZE, BUFFERS)
SELECT
        *
FROM
        t_benchmark
WHERE
        text_value='aggregate window generate sql amazing performance';
                                                              QUERY PLAN
--
     ------------------------------------------------------------------------------------------------------------------------------------------------

 Gather  (cost=1000.00..316329.12 rows=1 width=198) (actual time=523.142..526.241 rows=1 loops=1)
   Workers Planned: 4
   Workers Launched: 4
   Buffers: shared hit=130584 read=153493
   ->  Parallel Seq Scan on t_benchmark  (cost=0.00..315329.02 rows=1 width=198) (actual time=411.615..490.389
        rows=0 loops=5)
         Filter: (text_value = 'aggregate window generate sql amazing performance'::text)
         Rows Removed by Filter: 2000000
         Buffers: shared hit=130584 read=153493
 Planning Time: 0.200 ms
 Execution Time: 526.296 ms
(10 rows)

Time: 527.062 ms
```

Adding an index on the field text_value

KAMEDATA
DEVOPS AND DATA ENGINEERING

```
CREATE INDEX idx_text_value ON t_benchmark USING btree(text_value);
CREATE INDEX
Time: 27178.512 ms (00:27.179)
EXPLAIN (ANALYZE, BUFFERS)
SELECT
        *
FROM
        t_benchmark
WHERE
        text_value='aggregate window generate sql amazing performance';

                                                                QUERY PLAN
--
     ------------------------------------------------------------------------------------------------------------------------------------------

 Index Scan using idx_text_value on t_benchmark  (cost=0.69..8.70 rows=1 width=198) (actual time=0.645..0.646
      rows=1 loops=1)
   Index Cond: (text_value = 'aggregate window generate sql amazing performance'::text)
   Buffers: shared hit=1 read=5
 Planning:
   Buffers: shared hit=18 read=1 dirtied=2
 Planning Time: 0.267 ms
 Execution Time: 0.663 ms
(7 rows)

Time: 1.370 ms
```

Index's impact on insert

KAMEDATA
DEVOPS AND DATA ENGINEERING

```
TRUNCATE TABLE t_benchmark ;
TRUNCATE TABLE
Time: 717.027 ms

INSERT INTO t_benchmark (id,text_value,hashed_value)
    SELECT
        id,
        text_value,
        hashed_value FROM t_random_data;
INSERT 0 10000000
Time: 287194.719 ms (04:47.195)
```

The index is also bloated compared to the previous one

KAMEDATA
DEVOPS AND DATA ENGINEERING

```
\di+ idx_text_value
                                          List of relations
 Schema |      Name      | Type  |   Owner   |    Table     | Persistence | Access method |  Size   | Description
--------+----------------+-------+-----------+--------------+-------------+---------------+---------+-------------
 public | idx_text_value | index | thedoctor | t_benchmark | permanent   | btree         | 2550 MB |
(1 row)

REINDEX
Time: 31603.499 ms (00:31.603)

\di+ idx_text_value
                                          List of relations
 Schema |      Name      | Type  |   Owner   |    Table     | Persistence | Access method |  Size   | Description
--------+----------------+-------+-----------+--------------+-------------+---------------+---------+-------------
 public | idx_text_value | index | thedoctor | t_benchmark | permanent   | btree         | 1941 MB |
(1 row)
```

What about the primary key?

KAMEDATA
DEVOPS AND DATA ENGINEERING

```
DROP INDEX idx_text_value;
Time: 374.797 ms
ALTER TABLE t_benchmark ADD CONSTRAINT pk_t_benchmark PRIMARY KEY(id);
ALTER TABLE
Time: 6929.552 ms (00:06.930)

\di+ pk_t_benchmark
                                                List of relations
 Schema |      Name      | Type  |   Owner   |    Table    | Persistence | Access method |  Size  | Description
--------+----------------+-------+-----------+-------------+-------------+---------------+--------+-------------
 public | pk_t_benchmark | index | thedoctor | t_benchmark | permanent   | btree         | 214 MB |
(1 row)
```

Select using the primary key

KAMEDATA
DEVOPS AND DATA ENGINEERING

```
EXPLAIN (ANALYZE,BUFFERS)
    SELECT
        *
    FROM
        t_benchmark
    WHERE
        id=19999841;
                                                          QUERY PLAN
--
     ------------------------------------------------------------------------------------------------------------------------------------------

 Index Scan using pk_t_benchmark on t_benchmark  (cost=0.43..8.45 rows=1 width=199) (actual time=0.071..0.076
      rows=1 loops=1)
   Index Cond: (id = 19999841)
   Buffers: shared hit=4
 Planning Time: 0.307 ms
 Execution Time: 0.175 ms
(5 rows)

Time: 1.151 ms
```

Primary key impact on insert

KAMEDATA
DEVOPS AND DATA ENGINEERING

```
TRUNCATE TABLE t_benchmark ;
TRUNCATE TABLE
Time: 470.087 ms

INSERT INTO t_benchmark (id,text_value,hashed_value)
    SELECT
        id,
        text_value,
        hashed_value FROM t_random_data;
INSERT 0 10000000
Time: 46028.772 ms (00:46.029)

\di+ pk_t_benchmark
                                          List of relations
 Schema |      Name      | Type  |   Owner   |    Table    | Persistence | Access method |  Size  | Description
--------+----------------+-------+-----------+-------------+-------------+---------------+--------+-------------
 public | pk_t_benchmark | index | thedoctor | t_benchmark | permanent   | btree         | 292 MB |
(1 row)
```

Redundant indices

```
CREATE INDEX idx_id ON t_benchmark USING btree(id);
CREATE INDEX
Time: 4651.104 ms (00:04.651)
```

```
                                                List of relations
 Schema |       Name       | Type  |   Owner   |     Table     | Persistence | Access method |  Size   | Description
--------+------------------+-------+-----------+---------------+-------------+---------------+---------+-------------
 public | idx_id           | index | thedoctor | t_benchmark | permanent   | btree         | 214 MB |
 public | pk_t_benchmark | index | thedoctor | t_benchmark | permanent   | btree         | 292 MB |
(2 rows)
```

Redundant indices

KAMEDATA
DEVOPS AND DATA ENGINEERING

```
EXPLAIN (ANALYZE,BUFFERS)
    SELECT
        *
    FROM
        t_benchmark
    WHERE
        id=19999841;
                                                        QUERY PLAN
--
     ------------------------------------------------------------------------------------------------------------------------------------------

 Index Scan using idx_id on t_benchmark  (cost=0.43..8.45 rows=1 width=199) (actual time=0.047..0.052 rows=1
      loops=1)
   Index Cond: (id = 19999841)
   Buffers: shared hit=4
 Planning Time: 0.258 ms
 Execution Time: 0.099 ms
(5 rows)

Time: 1.071 ms
```

Redundant indices

```
TRUNCATE TABLE t_benchmark ;
TRUNCATE TABLE
Time: 544.275 ms
db_bench=# INSERT INTO t_benchmark (id,text_value,hashed_value)
    SELECT
        id,
        text_value,
        hashed_value FROM t_random_data;
INSERT 0 10000000
Time: 53654.549 ms (00:53.655)
```

Multi column indices

KAMEDATA
DEVOPS AND DATA ENGINEERING

```
CREATE INDEX idx_hash_text_value ON  t_benchmark USING btree(hashed_value,text_value);
CREATE INDEX
Time: 26409.935 ms (00:26.410)

EXPLAIN (ANALYZE, BUFFERS)
SELECT
        *
FROM
        t_benchmark
WHERE
        text_value='aggregate window generate sql amazing performance';

                                                                QUERY PLAN
--
     ------------------------------------------------------------------------------------------------------------------------------------------

 Gather  (cost=1000.00..337160.43 rows=1 width=198) (actual time=624.561..635.386 rows=1 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   Buffers: shared hit=103885 read=180192
   ->  Parallel Seq Scan on t_benchmark  (cost=0.00..336160.33 rows=1 width=198) (actual time=610.662..610.664
        rows=0 loops=3)
         Filter: (text_value = 'aggregate window generate sql amazing performance'::text)
         Rows Removed by Filter: 3333333
         Buffers: shared hit=103885 read=180192
 Planning Time: 0.308 ms
 Execution Time: 635.425 ms
(10 rows)
```

Multi column indices

KAMEDATA
DEVOPS AND DATA
ENGINEERING

```
SET enable_seqscan=off;
SET
Time: 0.130 ms
EXPLAIN (ANALYZE, BUFFERS)
SELECT
        *
FROM
        t_benchmark
WHERE
        text_value='aggregate window generate sql amazing performance';
                                                                    QUERY PLAN
--
     ------------------------------------------------------------------------------------------------------------------------

 Index Scan using idx_hash_text_value on t_benchmark  (cost=0.56..1240544.57 rows=1 width=198) (actual time
      =1186.822..1558.407 rows=1 loops=1)
   Index Cond: (text_value = 'aggregate window generate sql amazing performance'::text)
   Buffers: shared read=288738
 Planning Time: 0.281 ms
 Execution Time: 1558.455 ms
(5 rows)

Time: 1559.254 ms (00:01.559)
```

Index with the correct column order

KAMEDATA
DEVOPS AND DATA ENGINEERING

```
SET enable_seqscan =on;
CREATE INDEX id_text_hash_value ON  t_benchmark USING btree(text_value,hashed_value);
CREATE INDEX
Time: 34139.106 ms (00:34.139)

EXPLAIN (ANALYZE, BUFFERS)
SELECT
        *
FROM
        t_benchmark
WHERE
        text_value='aggregate window generate sql amazing performance';

                                                                QUERY PLAN
--
     ------------------------------------------------------------------------------------------------------------------------------------------

 Index Scan using id_text_hash_value on t_benchmark  (cost=0.69..8.70 rows=1 width=198) (actual time=6.912..6.920
       rows=1 loops=1)
   Index Cond: (text_value = 'aggregate window generate sql amazing performance'::text)
   Buffers: shared hit=1 read=5
 Planning:
   Buffers: shared hit=19 read=2 dirtied=2
 Planning Time: 1.079 ms
 Execution Time: 6.974 ms
(7 rows)

Time: 9.162 ms
```

Size

```
REINDEX TABLE t_benchmark ;
REINDEX
Time: 75646.467 ms (01:15.646)

 \di+
                                                 List of relations
 Schema |        Name          | Type  |   Owner    |    Table     | Persistence | Access method |  Size   |
      Description
--
     ------+----------------------+-------+------------+--------------+-------------+---------------+---------+--------

 public | id_text_hash_value   | index | thedoctor | t_benchmark | permanent   | btree         | 2313 MB |
 public | idx_hash_text_value | index | thedoctor | t_benchmark | permanent   | btree         | 2276 MB |
 public | idx_id               | index | thedoctor | t_benchmark | permanent   | btree         | 214 MB  |
 public | pk_t_benchmark       | index | thedoctor | t_benchmark | permanent   | btree         | 214 MB  |
(4 rows)
```

Flamme rouge

Like and the btree indices

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT
        *
FROM
        t_benchmark
WHERE
        text_value LIKE 'aggregate window generate sql amazing%';
                                                                    QUERY PLAN
--
     ----------------------------------------------------------------------------------------------------------------
 Index Scan using id_text_hash_value on t_benchmark  (cost=0.69..8.71 rows=1000 width=198) (actual time
      =1.463..1.467 rows=1 loops=1)
   Index Cond: ((text_value >= 'aggregate window generate sql amazing'::text) AND (text_value < 'aggregate window
         generate sql amazinh'::text))
   Filter: (text_value ~~ 'aggregate window generate sql amazing%'::text)
   Buffers: shared hit=1 read=5
 Planning:
   Buffers: shared hit=23 read=12
 Planning Time: 4.891 ms
 Execution Time: 1.516 ms
(8 rows)

Time: 8.081 ms
```

like searching whithin the string doesn't use the index

KAMEDATA DEVOPS AND DATA ENGINEERING

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT
        *
FROM
        t_benchmark
WHERE
        text_value LIKE '%window generate sql amazing%';
                                                                QUERY PLAN
--
     ------------------------------------------------------------------------------------------------------------------------------------------------

 Gather  (cost=1000.00..337260.33 rows=1000 width=198) (actual time=3.625..1983.764 rows=640 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   Buffers: shared hit=321 read=283756
   ->  Parallel Seq Scan on t_benchmark  (cost=0.00..336160.33 rows=417 width=198) (actual time=27.151..1965.301
         rows=213 loops=3)
          Filter: (text_value ~~ '%window generate sql amazing%'::text)
          Rows Removed by Filter: 3333120
          Buffers: shared hit=321 read=283756
 Planning Time: 0.622 ms
 Execution Time: 1983.924 ms
(10 rows)

Time: 1985.053 ms (00:01.985)
```

ilike performs even worse

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT
        *
FROM
        t_benchmark
WHERE
        text_value ILIKE 'window generate sql amazing%';
                                                                QUERY PLAN
--
     ------------------------------------------------------------------------------------------------------------------------------------------------

 Gather  (cost=1000.00..337260.33 rows=1000 width=198) (actual time=122.868..9663.216 rows=31 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   Buffers: shared hit=513 read=283564
   ->  Parallel Seq Scan on t_benchmark  (cost=0.00..336160.33 rows=417 width=198) (actual time=245.826..9653.818
         rows=10 loops=3)
         Filter: (text_value ~~* 'window generate sql amazing%'::text)
         Rows Removed by Filter: 3333323
         Buffers: shared hit=513 read=283564
 Planning:
   Buffers: shared hit=2
 Planning Time: 1.099 ms
 Execution Time: 9663.253 ms
(12 rows)
```

ilike performs even worse

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT
        *
FROM
        t_benchmark
WHERE
        text_value ILIKE '%window generate sql amazing%';

                                                                QUERY PLAN
--
     ------------------------------------------------------------------------------------------------------------------------------------------

 Gather  (cost=1000.00..337260.33 rows=1000 width=198) (actual time=25.715..11792.322 rows=640 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   Buffers: shared hit=609 read=283468
   ->  Parallel Seq Scan on t_benchmark  (cost=0.00..336160.33 rows=417 width=198) (actual time=85.812..11776.941
         rows=213 loops=3)
         Filter: (text_value ~~* '%window generate sql amazing%'::text)
         Rows Removed by Filter: 3333120
         Buffers: shared hit=609 read=283468
 Planning Time: 2.111 ms
 Execution Time: 11792.577 ms
(10 rows)

Time: 11795.431 ms (00:11.795)
```

The extension pg_trgm can help

KAMEDATA
DEVOPS AND DATA ENGINEERING

```
CREATE EXTENSION pg_trgm ;


CREATE INDEX idx_ilike_trgrm ON t_benchmark USING GIN (text_value gin_trgm_ops);

CREATE INDEX
Time: 302416.295 ms (05:02.416)

db_bench=# \di+ idx_ilike_trgrm
                                                 List of relations
 Schema |       Name       | Type  |   Owner    |    Table    | Persistence | Access method |  Size  | Description
--------+------------------+-------+------------+-------------+-------------+---------------+--------+-------------
 public | idx_ilike_trgrm | index | thedoctor | t_benchmark | permanent   | gin           | 933 MB |
(1 row)
```

With the index the performance are worse

KAMEDATA
DEVOPS AND DATA ENGINEERING

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT
        *
FROM
        t_benchmark
WHERE
        text_value LIKE '%window generate sql amazing%';
                                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on t_benchmark  (cost=85065.94..88896.71 rows=1000 width=198) (actual time=3929.346..8813.597
      rows=640 loops=1)
   Recheck Cond: (text_value ~~ '%window generate sql amazing%'::text)
   Rows Removed by Index Recheck: 8371828
   Heap Blocks: exact=52517 lossy=230412
   Buffers: shared hit=14758 read=305262
   ->  Bitmap Index Scan on idx_ilike_trgrm  (cost=0.00..85065.69 rows=1000 width=0) (actual time
        =3917.411..3917.412 rows=1413339 loops=1)
         Index Cond: (text_value ~~ '%window generate sql amazing%'::text)
         Buffers: shared hit=14758 read=22333
 Planning:
   Buffers: shared read=1
 Planning Time: 1.009 ms
 Execution Time: 8813.974 ms
(12 rows)

Time: 8815.473 ms (00:08.815) - Sequential scan time 2 seconds
```

With the index the performance are worse

KAMEDATA
DEVOPS AND DATA ENGINEERING

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT
        *
FROM
        t_benchmark
WHERE
        text_value ILIKE 'window generate sql amazing%';
                                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on t_benchmark  (cost=91872.77..95703.55 rows=1000 width=198) (actual time=4313.177..22102.506
      rows=31 loops=1)
   Recheck Cond: (text_value ~~* 'window generate sql amazing%'::text)
   Rows Removed by Index Recheck: 8372437
   Heap Blocks: exact=52517 lossy=230412
   Buffers: shared hit=16154 read=307058
   ->  Bitmap Index Scan on idx_ilike_trgrm  (cost=0.00..91872.52 rows=1000 width=0) (actual time
        =4158.437..4158.437 rows=1413339 loops=1)
         Index Cond: (text_value ~~* 'window generate sql amazing%'::text)
         Buffers: shared hit=16154 read=24129
 Planning:
   Buffers: shared read=4 dirtied=1
 Planning Time: 1.993 ms
 Execution Time: 22102.589 ms
(12 rows)
Time: 22105.433 ms (00:22.105)  - Sequential scan time 9 seconds
```

With the index the performance are worse

KAMEDATA
DEVOPS AND DATA ENGINEERING

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT
        *
FROM
        t_benchmark
WHERE
        text_value ILIKE '%window generate sql amazing%';
```

```
                                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on t_benchmark  (cost=85065.94..88896.71 rows=1000 width=198) (actual time=3834.949..25408.550
      rows=640 loops=1)
   Recheck Cond: (text_value ~~* '%window generate sql amazing%'::text)
   Rows Removed by Index Recheck: 8371828
   Heap Blocks: exact=52517 lossy=230412
   Buffers: shared hit=14801 read=305219
   ->  Bitmap Index Scan on idx_ilike_trgrm  (cost=0.00..85065.69 rows=1000 width=0) (actual time
        =3816.419..3816.419 rows=1413339 loops=1)
         Index Cond: (text_value ~~* '%window generate sql amazing%'::text)
         Buffers: shared hit=14801 read=22290
 Planning:
   Buffers: shared read=1
 Planning Time: 2.250 ms
 Execution Time: 25409.042 ms
(12 rows)

Time: 25412.065 ms (00:25.412)  - Sequential scan time 8 seconds
```

Attention when using GIN indices

From the PostgreSQL documentation

Updating a GIN index tends to be slow because of the intrinsic nature of inverted indexes: ... GIN is capable of postponing much of this work by inserting new tuples into a temporary, unsorted list of pending entries. When the table is vacuumed or autoanalyzed, or when gin_clean_pending_list function is called, or if the pending list becomes larger than gin_pending_list_limit, the entries are moved to the main GIN data structure using the same bulk insert techniques used during initial index creation.

https://www.postgresql.org/docs/current/gin.html#GIN-FAST-UPDATE

What about the GiST indices?

KAMEDATA
DEVOPS AND DATA ENGINEERING

```
DROP INDEX idx_ilike_trgrm;
CREATE INDEX idx_ilike_trgrm ON t_benchmark USING GIST (text_value gist_trgm_ops);
CREATE INDEX
Time: 473976.926 ms (07:53.977)

\di+ idx_ilike_trgrm
                                                List of relations
 Schema |      Name       | Type  |   Owner    |    Table    | Persistence | Access method |  Size   | Description
--
      ------+-----------------+-------+------------+-------------+-------------+---------------+---------+-------------

 public | idx_ilike_trgrm | index | thedoctor | t_benchmark | permanent   | gist          | 4546 MB |
(1 row)
```

With the index the performance are worse

KAMEDATA
DEVOPS AND DATA ENGINEERING

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT
        *
FROM
        t_benchmark
WHERE
        text_value LIKE '%window generate sql amazing%';
```

```
                                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on t_benchmark  (cost=244.17..4074.94 rows=1000 width=198) (actual time=4927.537..17000.926
      rows=640 loops=1)
   Recheck Cond: (text_value ~~ '%window generate sql amazing%'::text)
   Rows Removed by Index Recheck: 8358313
   Heap Blocks: exact=51594 lossy=231335
   Buffers: shared hit=13713 read=570460 written=48889
   ->  Bitmap Index Scan on idx_ilike_trgrm  (cost=0.00..243.92 rows=1000 width=0) (actual time
        =4916.416..4916.417 rows=1413339 loops=1)
         Index Cond: (text_value ~~ '%window generate sql amazing%'::text)
         Buffers: shared hit=13713 read=287531 written=48889
 Planning Time: 0.116 ms
 Execution Time: 17001.402 ms
(10 rows)
```

```
Time: 17001.859 ms (00:17.002) - Sequential scan 2 seconds - GIN 8 seconds
```

With the index the performance are worse


```
EXPLAIN (ANALYZE, BUFFERS)
SELECT
        *
FROM
        t_benchmark
WHERE
        text_value ILIKE 'window generate sql amazing%';

                                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on t_benchmark  (cost=244.17..4074.94 rows=1000 width=198) (actual time=3885.294..22346.006
      rows=31 loops=1)
   Recheck Cond: (text_value ~~* 'window generate sql amazing%'::text)
   Rows Removed by Index Recheck: 8358922
   Heap Blocks: exact=51594 lossy=231335
   Buffers: shared read=584173
   ->  Bitmap Index Scan on idx_ilike_trgrm  (cost=0.00..243.92 rows=1000 width=0) (actual time
        =3732.545..3732.546 rows=1413339 loops=1)
         Index Cond: (text_value ~~* 'window generate sql amazing%'::text)
         Buffers: shared read=301244
 Planning:
   Buffers: shared read=3 dirtied=1
 Planning Time: 0.331 ms
 Execution Time: 22346.080 ms
(12 rows)

Time: 22346.750 ms (00:22.347) - Sequential scan 9 seconds - GIN 22 seconds
```

With the index the performance are worse

KAMEDATA
DEVOPS AND DATA ENGINEERING

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT
        *
FROM
        t_benchmark
WHERE
        text_value ILIKE '%window generate sql amazing%';
```

```
                                                                    QUERY PLAN
-----------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on t_benchmark  (cost=244.17..4074.94 rows=1000 width=198) (actual time=2387.155..22813.199
      rows=640 loops=1)
   Recheck Cond: (text_value ~~* '%window generate sql amazing%'::text)
   Rows Removed by Index Recheck: 8358313
   Heap Blocks: exact=51594 lossy=231335
   Buffers: shared read=584173
   ->  Bitmap Index Scan on idx_ilike_trgrm  (cost=0.00..243.92 rows=1000 width=0) (actual time
        =2374.898..2374.898 rows=1413339 loops=1)
         Index Cond: (text_value ~~* '%window generate sql amazing%'::text)
         Buffers: shared read=301244
 Planning Time: 0.298 ms
 Execution Time: 22813.623 ms
(10 rows)

Time: 22814.129 ms (00:22.814)  - Sequential scan 8seconds - GIN 25 seconds
```

Dernière étape, Champs-Élysées

Few words about the index on text fields

Strictly speaking about the b-tree the online documentation says the following:

From the PostgreSQL documentation

PostgreSQL includes an implementation of the standard btree (multi-way balanced tree) index data structure. Any data type that can be sorted into a well-defined linear order can be indexed by a btree index.
The only limitation is that an index entry cannot exceed approximately one-third of a page (after TOAST compression, if applicable).

https://www.postgresql.org/docs/17/btree.html

NO TOAST!

Source image https://www.flickr.com/photos/francoisroche/2584062428

T.O.A.S.T. Stands for
The Oversize Attribute Storage Tecnique.
The manaement routines are automatically triggered for the table's tuples when the row exceeds the TOAST_TUPLE_THRESHOLD (usually 2kb) and only for TOASTable data types (varlena).
TOAST can store rows up to 1 GB using one of the following tecniques.

- Compression
- Out of line storage
- Compression and out of line storage

Indices can't be TOASTed

As indices are not TOASTable, if the index entry after the eventual compression exceeds one third of the page then it will throw an error.
Adding an index on a text field doesn't mean that the index will always work.

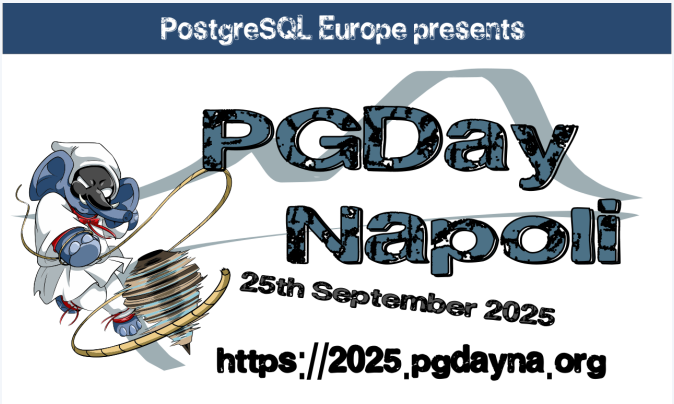
Wrap up

- PostgreSQL offers a constantly improving index infrastructure
- New ways are constantly opened for improving the performance...
- ...or crash it miserably!
- The rule of thumb is "test your assumptions, always"
- And remember that...

[1]

Y.M.M.V.

[1] Translation: no airbags. we die as heroes



https://2025.pgdaynapoli.org

That's all folks!

Thank you for listening!

Any questions?

Beyond the Basics: Mastering PostgreSQL Index Performance

Federico Campoli

PostgreSQL DBA, because freaking miracle worker is not a job title

PGDay Austria, Vienna, 4th September 2025