B-Tree Indexes PostgreSQL

Created	@August 16, 2023 6:54 PM
Owner	Din Lester
Tags	databases postgresql
Status	Done

Contents

Resources:

- https://www.enterprisedb.com/postgres-tutorials/overview-postgresql-indexes
- https://youtu.be/fsG1XaZEa78
- https://youtu.be/-qNSXK7s7_w
- https://hakibenita.com/postgresql-hash-index
- https://habr.com/ru/articles/276973/
- https://pganalyze.com/docs/explain/scan-nodes/bitmap-heap-scan

Type of indexes in PostgreSQL

PostgreSQL server provides following types of indexes, which each uses a different algorithm:

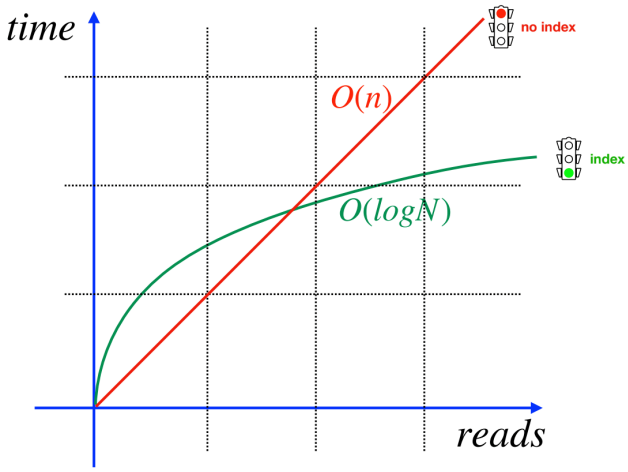
- B-tree
- Hash
- GiST
- SP-GiST
- GIN
- BRIN

Not all types of indexes are the best fit for every environment, so you should choose the one you use carefully. How you decide will depend upon your requirements.

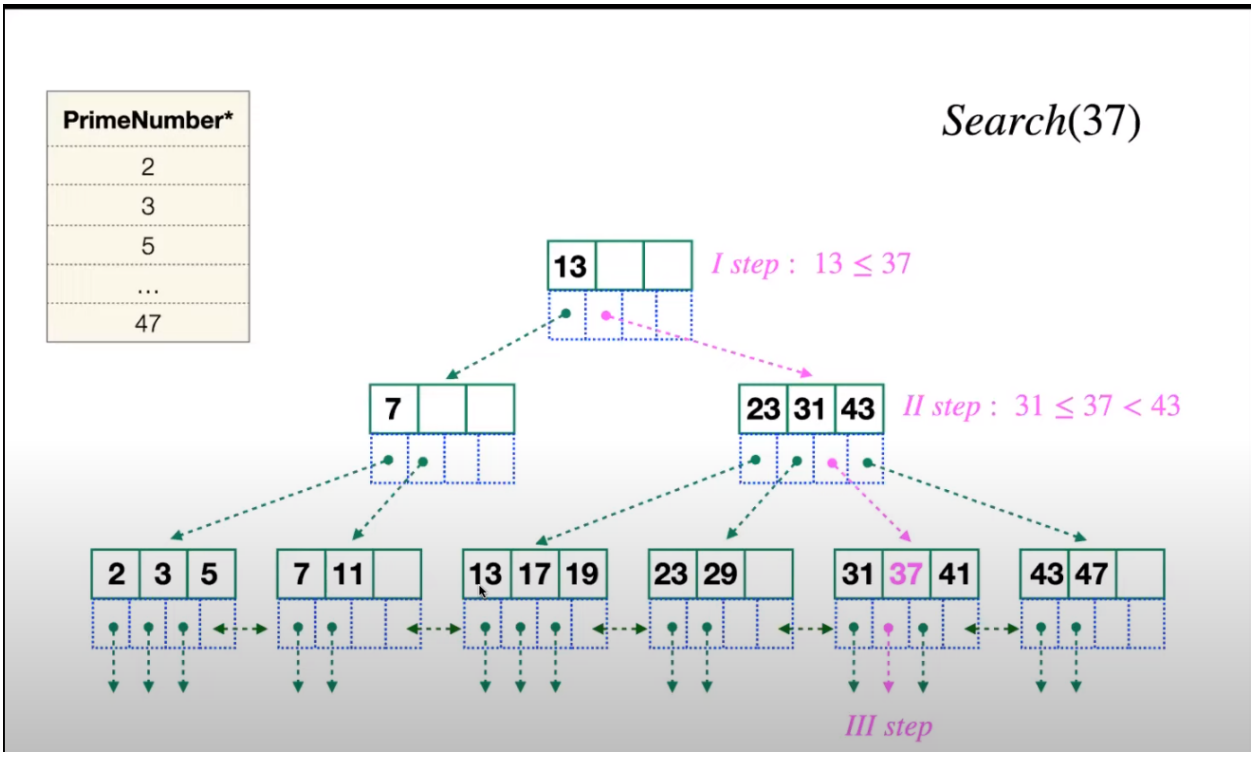
Logical schema of B-Tree index

First of all, need to mention that this type of index is made on idea of *Lehman and Yao Algorithm* in 1981.

What is the complexity of B-Tree index in compare to sequence scan ?

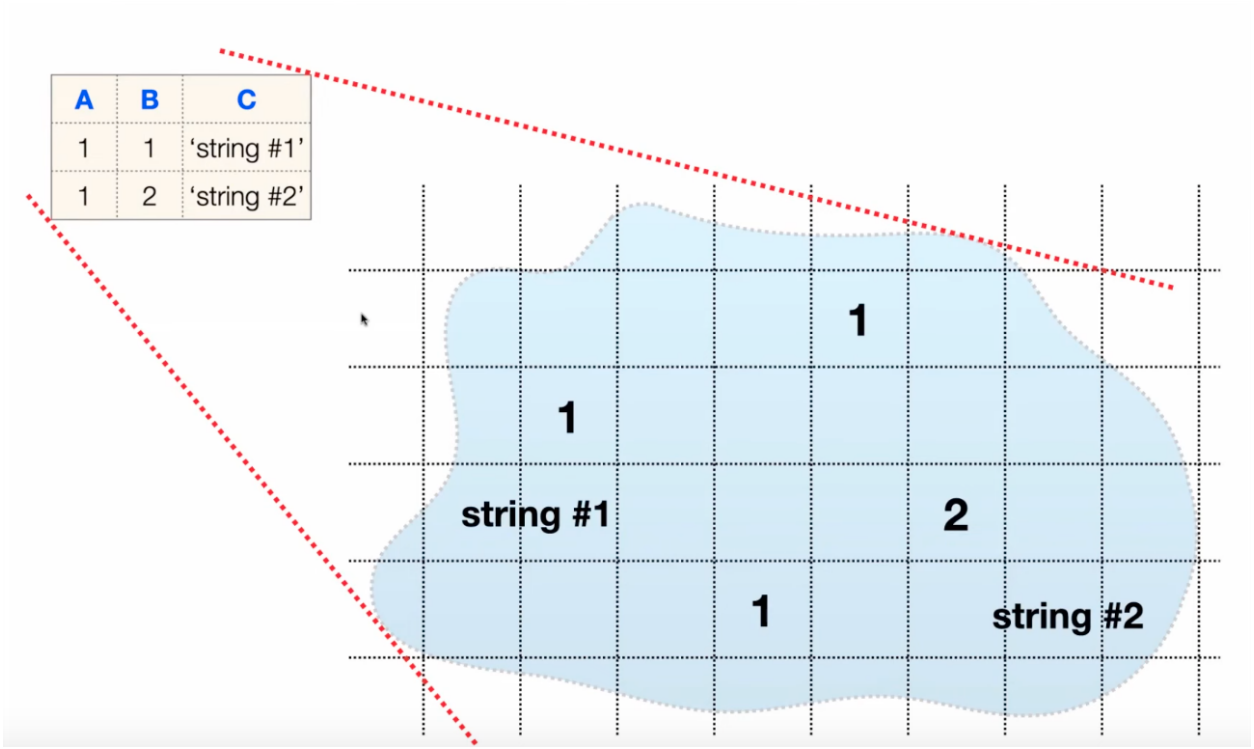


B-Tree is a shorter name of balanced tree, which means that the distance between each node and root on levels are the same.

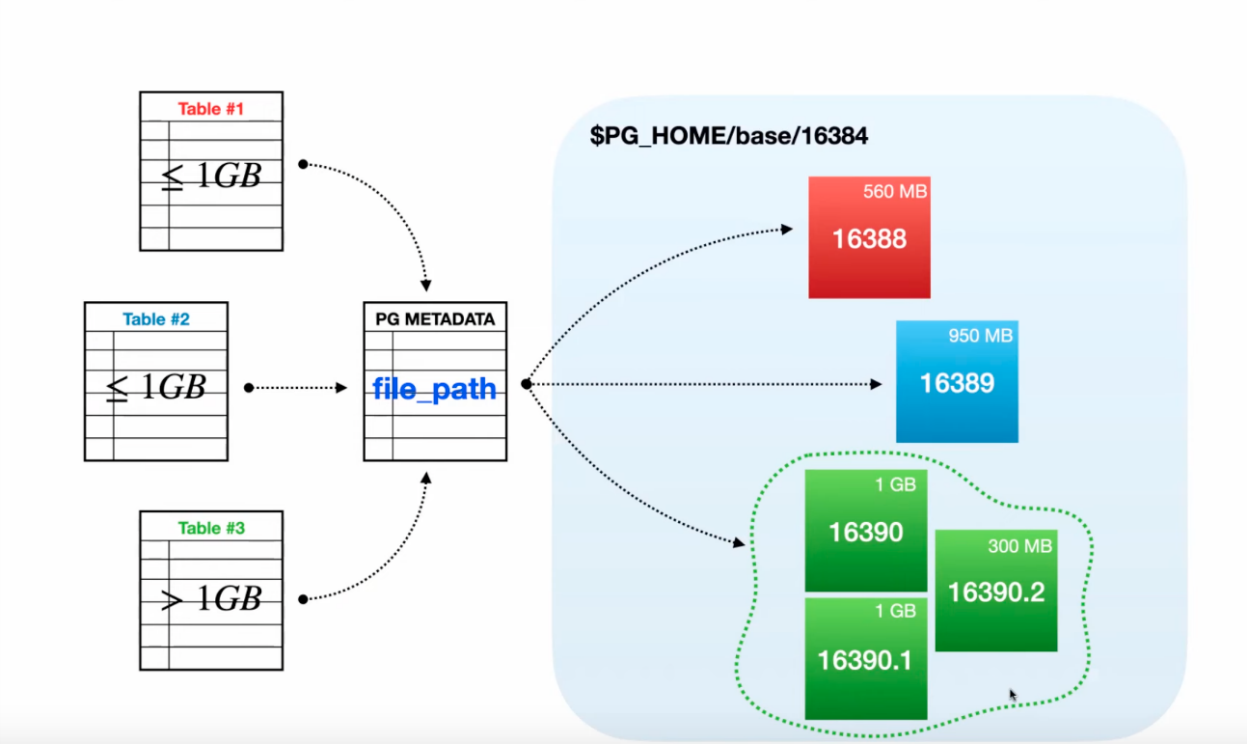


[Short recap] How postgreSQL stores data ?

Data stores on a disc not like ordered table view. It rather an unordered chaos - heap.

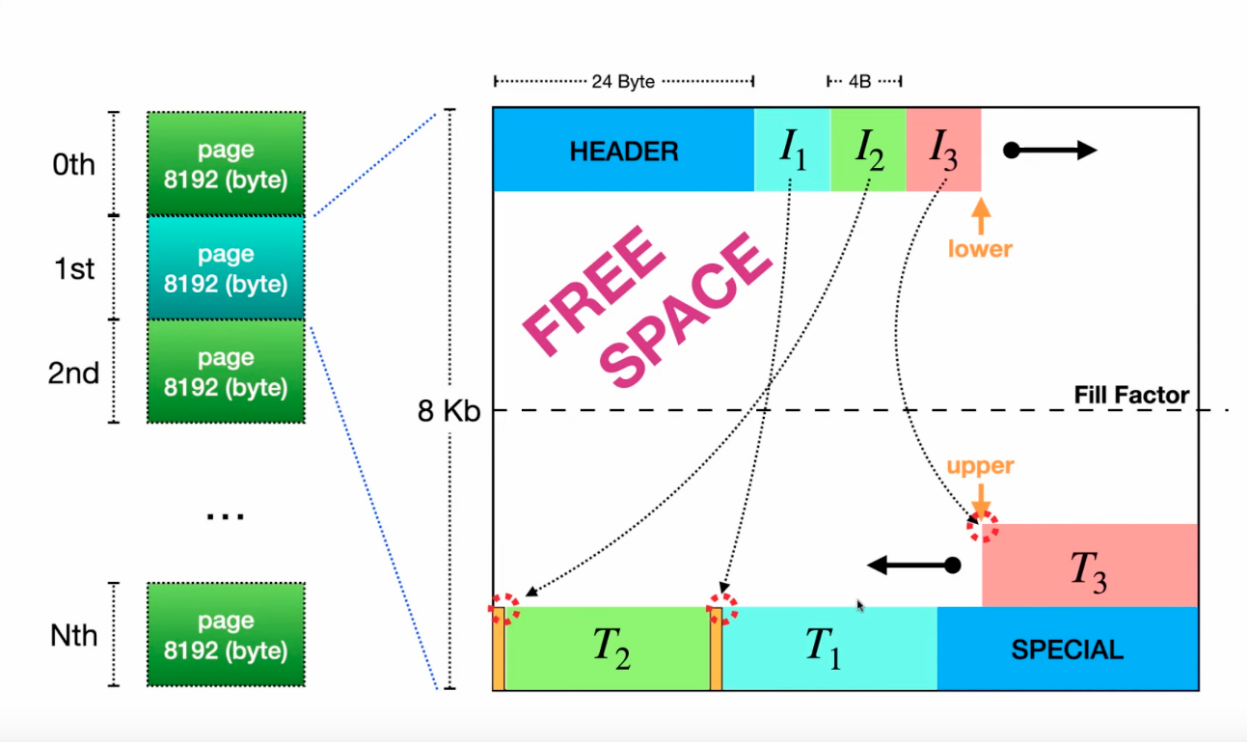


When size of a table is bigger than 1 GB, Postgres split it in files adding suffix.



Also, need to mention that every object in Postgres has its unique identifier - OID. So, our files named like OID of a table.

Let's take a closer look on a page.



How B-Tree index is implemented in PostgreSQL

Let's assume we have next table:

```
CREATE TABLE IF NOT EXISTS user(
  id int,
  name varchar
);
```

For investigation of under the hood processes we will use pageinspect extension.

> ***Important*** PRIMARY KEY and UNIQUE constraints create index by default.

```
CREATE EXTENSION IF NOT EXISTS pageinspect;
TRUNCATE TABLE user;

INSERT INTO user VALUES (1, 'Row #1');

SELECT *
FROM bt_metap('idx_user');
```

magic	version	root	level	fastroot	fastlevel	oldest_xact	last_cleanup_num_tuples
340322	3	1	0	1	0	0	-1

```
SELECT *
FROM bt_page_stats('idx_user', 1);
```

blkno	type	live_items	dead_items	avg_item_size	page_size	free_size	btpo_prev	btpo_next	btpo	btpo_flags
1	L	1	0	16	8192	8128	0	0	0	3

Let's look on the items in this tree leaf.

```
SELECT *
FROM bt_page_items('idx_user', 1);
```

1 live item

itemoffset	ctid	itemlen	nulls	vars	data
1	(0,1)	16	false	false	01 00 00 00 00 00 00 00

type **bigint** = 8 bytes

Now let's create multiple records:

```
TRUNCATE TABLE user;
```

```
INSERT INTO "user"
SELECT i, 'Row #'||i::VARCHAR
  FROM generate_series(1,1000) AS k(i)

SELECT *
FROM bt_metap('idx_user');
```

magic	version	root	level	fastroot	fastlevel	oldest_xact	last_cleanup_num_tuples
340322	3	3	1	**3**	**1**	0	-1

Let's look into first node:

blkno	type	live_items	dead_items	avg_item_size	page_size	free_size	btpo_prev	btpo_next	btpo	btpo_flags
3	R	3	0	13	8192	8096	0	0	1	2

Now, type is not a leaf but root.

Let's take a look into items

3 live items

itemoffset	ctid	itemlen	nulls	vars	data	
1	(1,0)	8	false	false	*<null>*	
2	(2,38)	16	false	false	6f 01 00 00 00 00 00 00	=367 (0x16f)
3	(4,90)	16	false	false	dd 02 00 00 00 00 00 00	=733 (0x2dd)

```
SELECT *
from bt_page_stats('idx_user', 1);
```

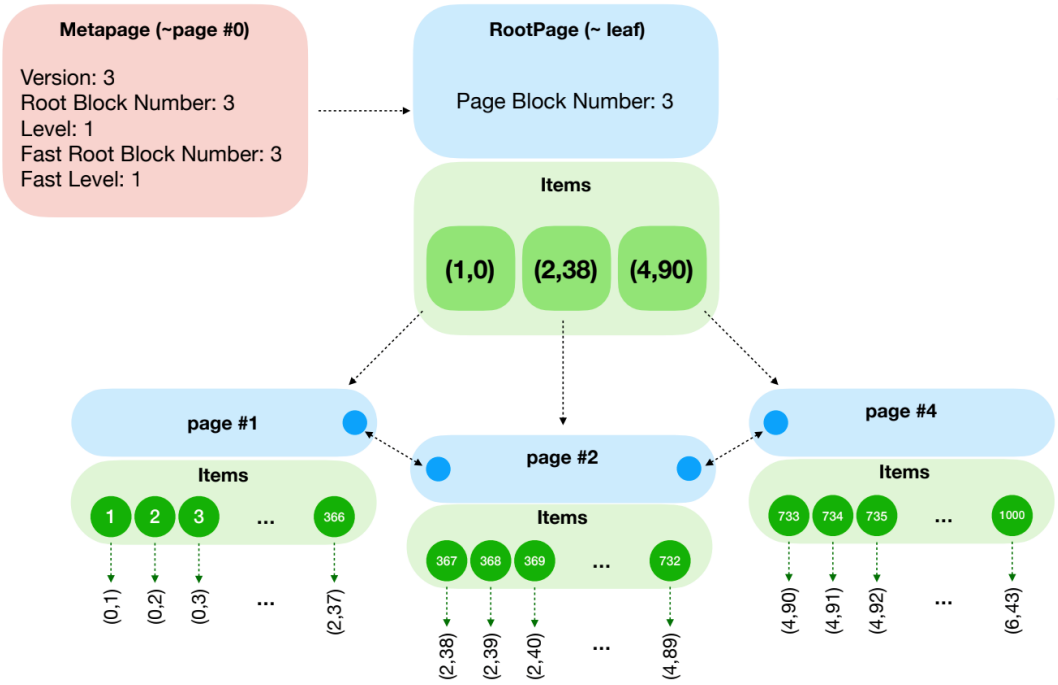
blkno	type	live_items	dead_items	avg_item_size	page_size	free_size	btpo_prev	btpo_next	btpo	btpo_flags
1	**L**	367	0	16	8192	808	**0**	**2**	0	1
							previous leaf page	**next leaf page**		

```
SELECT *
FROM bt_page_items('idx_user', 1);
```

high key

367 live items

itemoffset	ctid	itemlen	nulls	vars	data	
1	(2,38)	16	false	false	6f 01 00 00 00 00 00 00	=367 (0x16f)
2	(0,1)	16	false	false	01 00 00 00 00 00 00 00	=1 (0x01)
3	(0,2)	16	false	false	02 00 00 00 00 00 00 00	=2 (0x02)
4	(0,3)	16	false	false	03 00 00 00 00 00 00 00	=3 (0x03)
5	(0,4)	16	false	false	04 00 00 00 00 00 00 00	=4 (0x04)
6	(0,5)	16	false	false	05 00 00 00 00 00 00 00	=5 (0x05)
7	(0,6)	16	false	false	06 00 00 00 00 00 00 00	=6 (0x06)
…	…	…	…	…	…	…
367	(2,37)	16	false	false	6e 01 00 00 00 00 00 00	=366 (0x16e)



So, the physical representation of B-Tree index in Postgres looks like this:

How to create indexes

```
CREATE INDEX idx_user ON user (id) TABLESPACE ts_ssd;

CREATE UNIQUE INDEX idx_user ON user USING BTREE (id);---only unique values in index

CREATE INDEX IF NOT EXISTS idx_user ON user (id, name);

CREATE INDEX idx_user ON user (name DESC NULLS FIRST);

CREATE INDEX idx_user ON user (name) WITH (FILLFACTOR = 70);

CREATE INDEX CONCURRENTLY idx_user ON user (id, name);--- don't block data

CREATE UNIQUE INDEX idx_user ON ONLY user (id);---only on cerain partitioned table

DROP INDEX CONCURRENTLY idx_user;

REINDEX (VERBOSE) { INDEX | TABLE | SCHEMA | DATABASE | SYSTEM } name
```

Other types of indexes

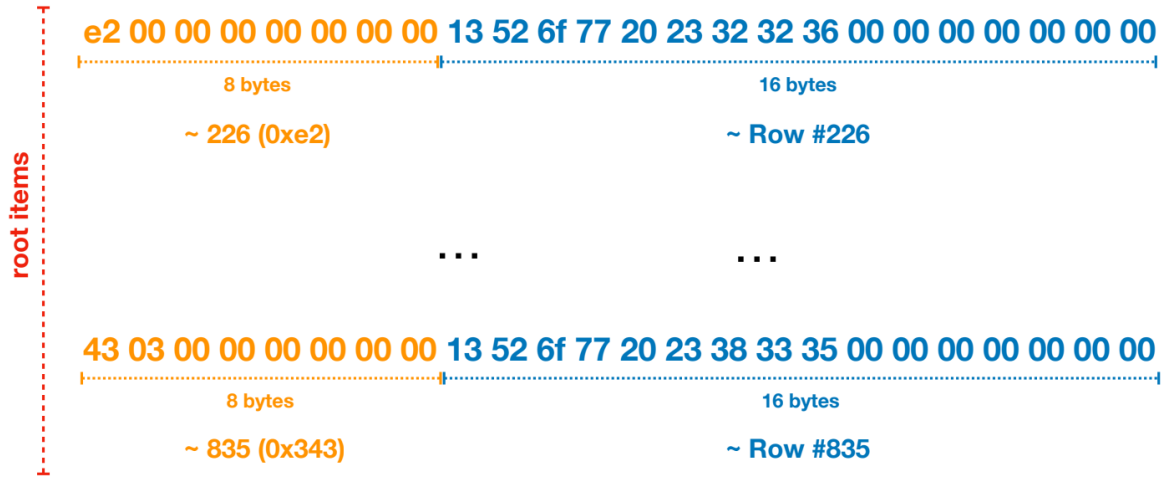
```
--- FUNCTIONAL INDEX
CREATE INDEX idx_user ON user(UPPER(name)));

SELECT id WHERE user WHERE name = 'BOB'; --- WRONG
SELECT id WHERE user WHERE UPPER(name) = 'BOB'; --- CORRECT

--- CONDITIONAL INDEX
CREATE UNIQUE INDEX idx_user ON user(name)
WHERE id BETWEEN 0 AND 100;

SELECT id WHERE user WHERE name = 'BOB'; --- WRONG
SELECT id WHERE user WHERE UPPER(name) = 'BOB' AND id BETWEEN 20 AND 40; --- CORRECT

--- INCLUDE INDEXES
```

```
CREATE INDEX id_user ON user(id) INCLUDE(name);
select name where id =2;
```

Concat indexes

```
TRUNCATE TABLE user;

DROP INDEX idx_user;
CREATE INDEX idx_name ON user(id, name);

INSERT INTO test."user"
SELECT i, 'Row #'||i::VARCHAR
  FROM generate_series(1,1000) AS k(i)

SELECT *
FROM bt_page_items('idx_user', 3);

where id and name
where id =
where name
```

itemoffset	ctid	itemlen	nulls	vars	data
1	(1,0)	8	false	false	*<null>*
2	(2,54)	32	false	true	e2 00 00 00 00 00 00 00 13 52 6f 77 20 23 32 32 36 00 00 00 00 00 00 00
3	(4,100)	32	false	true	ad 01 00 00 00 00 00 00 13 52 6f 77 20 23 34 32 39 00 00 00 00 00 00 00
4	(5,146)	32	false	true	78 02 00 00 00 00 00 00 13 52 6f 77 20 23 36 33 32 00 00 00 00 00 00 00
5	(6,35)	32	false	true	43 03 00 00 00 00 00 00 13 52 6f 77 20 23 38 33 35 00 00 00 00 00 00 00



Live example

Computer hardware:

memory	16GiB System memory
processor	Intel(R) Core(TM) i7-10750H CPU @ 2

```
CREATE TABLE IF NOT EXISTS public.person
(
  id integer NOT NULL,
  first_name character varying(255) COLLATE pg_catalog."default",
  last_name character varying(255) COLLATE pg_catalog."default",
  gender character varying(1) COLLATE pg_catalog."default",
  birthday date,
  CONSTRAINT person_pkey PRIMARY KEY (id)
);
```

Insert 40 000 000 rows . This table has

- 695 unique `first_names`
- 8736 unique `birthdays`
- 1584 unique `last_names`

Let's try to make some queries on clean not-indexed DB and indexed DB:

```
select id from person where first_name = 'Connie'; worked 928 msec.
```

With index:

```
CREATE INDEX person_first_name ON person(first_name);

select id from person where first_name = 'Dustin';

"Bitmap Heap Scan on person  (cost=324.14..84412.70 rows=28864 width=4) (actual time=12.632..170.082 rows=61070 loops=1)"
"  Recheck Cond: ((first_name)::text = 'Dustin'::text)"
"  Heap Blocks: exact=54846"
"  ->  Bitmap Index Scan on person_first_name  (cost=0.00..316.92 rows=28864 width=0) (actual time=4.775..4.776 rows=61070 loops=1)"
"        Index Cond: ((first_name)::text = 'Dustin'::text)"
"Planning Time: 0.083 ms"
"Execution Time: 172.262 ms"
```

```
select * from person where birthday = '1992-03-04'; worked 899 msec.
```

```
select * from person where birthday = '1992-03-02';

"Bitmap Heap Scan on person  (cost=56.61..16826.32 rows=4667 width=23) (actual time=1.371..17.529 rows=4607 loops=1)"
"  Recheck Cond: (birthday = '1992-03-02'::date)"
"  Heap Blocks: exact=4581"
"  ->  Bitmap Index Scan on person_birthday  (cost=0.00..55.44 rows=4667 width=0) (actual time=0.590..0.591 rows=4607 loops=1)"
"        Index Cond: (birthday = '1992-03-02'::date)"
"Planning Time: 0.070 ms"
"Execution Time: 17.804 ms"
```

Normal sample of experiments

```
DO $$
DECLARE
  n INTEGER := 1000;
  duration INTERVAL := 0;
  start TIMESTAMP;
  first_names TEXT[];
```

```
BEGIN
  -- Fetch random keys from the table
  SELECT ARRAY_AGG(first_name) INTO first_names
  FROM (
    SELECT first_name
    FROM person
    ORDER BY random()
    LIMIT n
  ) AS foo;

    FOR i IN array_lower(first_names, 1)..array_upper(first_names, 1) LOOP
      start := clock_timestamp();
        PERFORM * FROM person WHERE first_name = first_names[i];
      duration := duration + (clock_timestamp() - start);
  END LOOP;
  RAISE NOTICE '{TYPE OF SCAN HERE}: mean=%', extract('epoch' from duration) / n;
END;
$$;
```

First name

Sequence scan: mean=2.7447613120000000

B-Tree scan: mean=0.72053070000000000000

Birthdays here 100 select

Sequence scan: mean=1.1220963900000000

B-Tree scan: mean=0.01109847000000000000

What wrong with LIKE ???

```
explain analyze select * from person where first_name LIKE 'An%'
and birthday = '1980-04-13';
Parallel Seq Scan. Execution Time: 1739.075 ms

explain analyze select * from person where first_name > 'An'
and first_name < 'Ao' and birthday = '1980-04-13';
Index Scan. Execution Time: 90.549 ms
```

Selectivity

Create index on gender

```
select * from person where gender = 'M';

"Seq Scan on person  (cost=0.00..782746.25 rows=20192657 width=23) (actual time=28.092..2909.585 rows=20309033 loops=1)"
"  Filter: ((gender)::text = 'M'::text)"
"  Rows Removed by Filter: 20298268"
"Planning Time: 0.167 ms"
"JIT:"
"  Functions: 2"
"  Options: Inlining true, Optimization true, Expressions true, Deforming true"
"  Timing: Generation 0.221 ms, Inlining 3.830 ms, Optimization 15.174 ms, Emission 9.072 ms, Total 28.298 ms"
"Execution Time: 3338.503 ms"
```

```
Index 1
create index person_first_name_birthday on person(first_name, birthday);
```

```
explain analyze select * from person where first_name='Angela'
and birthday = '1980-04-13';
Index Scan. Execution Time: 0.765 ms

explain analyze select * from person where first_name='Angela'
and birthday between '1980-02-13' and '2000-02-13';
Bitmap Index Scan. Execution Time: 1213.717 ms


explain analyze select * from person where starts_with(first_name, 'An')
and birthday = '1980-04-13';
Parallel Seq Scan. Execution Time: 3219.044 ms
```

```
Index 2
create index person_birthday_first_name on person(birthday, first_name);

Queries
explain analyze select * from person where first_name='Angela'
and birthday = '1980-04-13';
Index Scan. Execution Time: 0.150 ms

explain analyze select * from person where first_name='Angela'
and birthday between '1980-02-13' and '2000-02-13';
Parallel Seq Scan. Execution Time: 1967.308 ms


explain analyze select * from person where starts_with(first_name, 'An')
and birthday = '1980-04-13';
Bitmap Index Scan. Execution Time: 20.214 ms
```

Scan nodes

Across all our queries we saw different type of scans, so let’s take a closer look into each one:

- Index Only scan
- Index scan
- Bitmap Heap Scan & Bitmap Index Scan
- Sequential scan

Let's say your table has 100,000 pages (that's about 780MB). Bitmap Index Scan will create a bitmap, where each page of your table will have one bit. So, in this case, we will get a 100,000 bit ~ 12.5 kB block of memory. All of these bits will be set to 0. Then, the Bitmap Index Scan will set some bits to 1, depending on which page of the table the row to be returned might be on.

```
alter table test add column j int4 default random() * 1000000000;
ALTER TABLE
alter table test add column h int4 default random() * 1000000000;
ALTER TABLE
create index i2 on test (j);
CREATE INDEX
create index i3 on test (h);
CREATE INDEX
```

```
explain analyze select * from test where j < 50000000 and i < 50000000 and h > 950000000;
                                                      QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on test  (cost=280.76..323.61 rows=12 width=16) (actual time=2.295..2.352 rows=11 loops=1)
   Recheck Cond: ((h > 950000000) AND (j < 50000000) AND (i < 50000000))
   ->  BitmapAnd  (cost=280.76..280.76 rows=12 width=0) (actual time=2.278..2.278 rows=0 loops=1)
```

```
         ->  Bitmap Index Scan on i3  (cost=0.00..92.53 rows=4832 width=0) (actual time=0.546..0.546 rows=4938 loops=1)
               Index Cond: (h > 950000000)
         ->  Bitmap Index Scan on i2  (cost=0.00..93.76 rows=4996 width=0) (actual time=0.783..0.783 rows=5021 loops=1)
               Index Cond: (j < 50000000)
         ->  Bitmap Index Scan on i1  (cost=0.00..93.96 rows=5022 width=0) (actual time=0.798..0.798 rows=4998 loops=1)
               Index Cond: (i < 50000000)
 Total runtime: 2.428 ms
(10 rows)
```

> *Important note:* when you make bulk create/update/delete make vacuum analyze test;

More about VACUUM

Thank you for your attention.