Postgres Conference

Silicon Valley 2022

San Jose / United Statues

April 07-08, 2022

Deep Dive Into PostgreSQL Indexes

Technical Breakout

Ibrar Ahmed
Senior Database Architect
Percona LLC

Who am I?

@ibrar_ahmad

https://www.facebook.com/ibrar.ahmed

https://www.linkedin.com/in/ibrarahmed74/

Software Career

- Software industries since 1998.

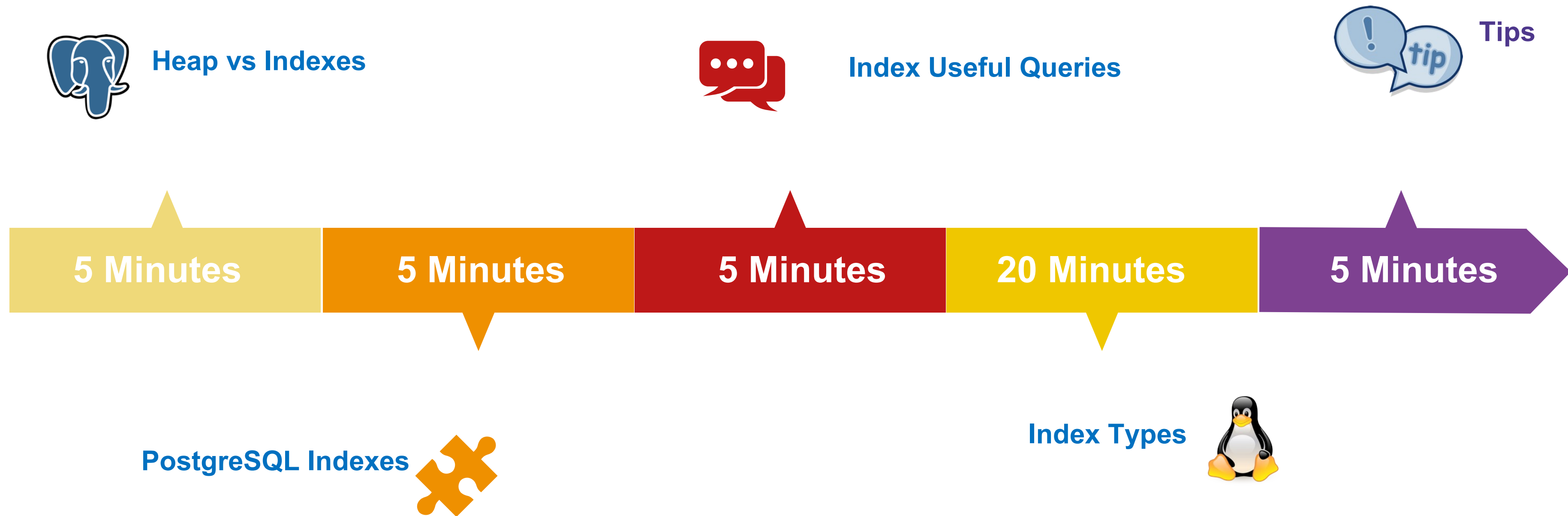
PostgreSQL Career

- Working on PostgreSQL Since 2006.
- EnterpriseDB (Associate Software Architect core Database Engine) 2006-2009
- EnterpriseDB (Software Architect core Database Engine) 2011 - 2016
- EnterpriseDB (Senior Software Architect core Database Engine) 2016 – 2018
- Percona (Senior Software Architect core Database Engine) 2018 – Present

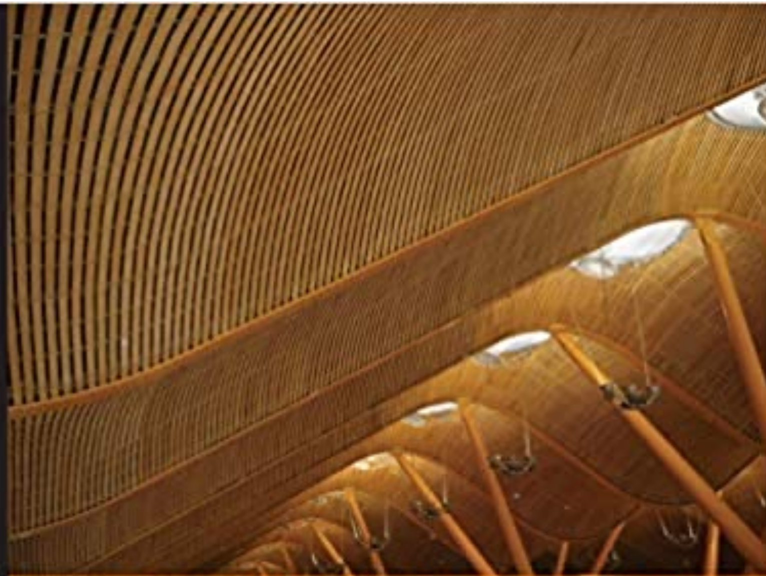
PostgreSQL Books

- PostgreSQL Developer's Guide
- PostgreSQL 9.6 High Performance

Timeline



Heap / Index



about 178
using 178, 179
PQsetdbLogin function
about 178
using 178
prepared statements, executing
PQexecPrepared, using 186, 187
PQprepare, using 185, 186

Q

query, executing
about 184
PQexecParams, using 185
PQexec, using 184
query optimization
about 137
configuration parameters 153
cost parameters 141
EXPLAIN command 138
hints 151
query planning
about 149, 150
window functions 150, 151
query tree 102

R

range partition
about 124
constraint exclusion, enabling 129
index, creating on child tables 127
master table, creating 124, 125
range partition table, creating 125, 126
trigger, creating on master table 127, 128
rank() function
about 114
calling 114
row-level trigger 87
row_number() function
about 113
calling 113
rules
about 85
versus triggers 103, 104

S

schema_name parameter 225
self join 120
semi join 148
sequential scan 138, 142
set of variables, TriggerData
NEW 90
OLD 90
TG_OP 90
TG_TABLE_NAME 90
TG_WHEN 90
shared_buffers parameter 155
single-column index
about 69
creating 69, 70
SQL commands, running
about 203
dynamic SQL 206
host variables, using 205
values, obtaining from SQL 205
values, passing to SQL 205
SQL Communication Area (sqlca)
about 210
using 210-212
SQL file 216
SQL/MED (SQL/Management of External Data) 213
start up cost 138
statement-level trigger 87
status functions
PQresStatus, using 196
PQresultStatus, using 195
using 195

T

table partition
creating 123
TriggerData
about 86
set of variables 90
trigger function
about 85, 86
creating, with PL/pgSQL 90-92
defining 86
triggers
about 86
creating, in PL/Perl 96-98

PostgreSQL comes with two main types of triggers: **row-level trigger** and **statement-level trigger**. These are specified with `FOR EACH ROW` (row-level triggers) and `FOR EACH STATEMENT` (statement-level triggers). The two can be differentiated by how many times the trigger is invoked and at what time. This means that if an `UPDATE` statement is executed that affects 10 rows, the row-level trigger will be invoked 10 times, whereas the statement-level trigger defined for a similar operation will be invoked only once per SQL statement.

Triggers can be attached to both tables and views. Triggers can be fired for tables before or after any `INSERT`, `UPDATE`, or `DELETE` operation; they can be fired once per affected row, or once per SQL statement. Triggers can be executed for the `TRUNCATE` statements as well. When a trigger event occurs, the trigger function is invoked to make the appropriate changes as per the logic you have defined in the trigger function.

The triggers defined with `INSTEAD OF` are used for `INSERT`, `UPDATE`, or `DELETE` on the views. In the case of views, triggers fired before or after `INSERT`, `UPDATE`, or `DELETE` can only be defined at the statement level, whereas triggers that fire `INSTEAD OF` on `INSERT`, `UPDATE`, or `DELETE` will only be defined at the row level.

Triggers are quite helpful where your database is being accessed by multiple applications, and you want to maintain complex data integrity (this will be difficult with available means) and monitor or log changes whenever a table data is being modified.

The next topic is a concise explanation of tricky trigger concepts and behaviors that we discussed previously. They can be helpful in a database design that involves triggers.

Tricky triggers

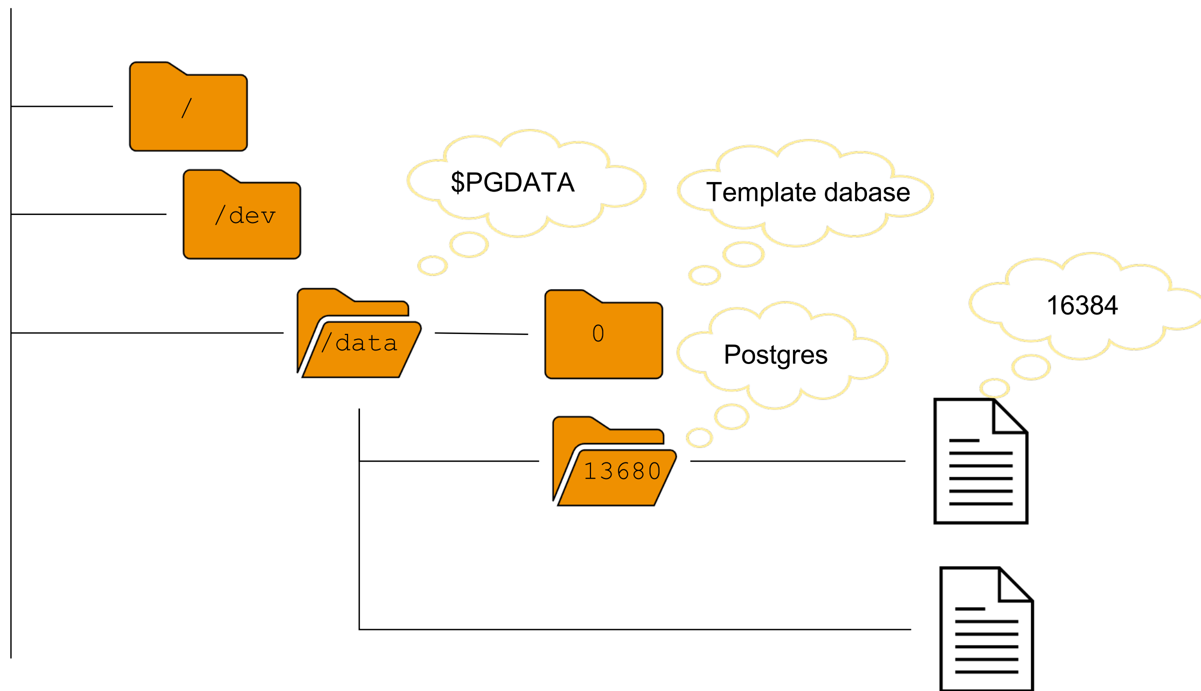
In `FOR EACH ROW` triggers, function variables contain table rows as either a `NEW` or `OLD` record variable, for example, in the case of `INSERT`, the table rows will be `NEW`, for `DELETE`, it is `OLD`, and for `UPDATE`, it will be both. The `NEW` variable contains the row after `UPDATE` and `OLD` variable holds the row state before `UPDATE`.

Hence, you can manipulate this data in contrast to `FOR EACH STATEMENT` triggers. This explains one thing clearly, that if you have to manipulate data, use `FOR EACH ROW` triggers.

The next question that strikes the mind is how to choose between row-level `AFTER` and `BEFORE` triggers.

PostgreSQL Tables (Heap)

- Rows / Tuples stored in a table.
- Every table in PostgreSQL has physical disk file(s)*
- The physical files on disk can be seen in the $PGDATA directory
- Tuple stored in a table does not have any order.
- Rows can be accessed in sequential order.



SQL

```
CREATE TABLE admin(id int, name text, dt date);
SELECT relfilenode FROM pg_class WHERE relname
LIKE 'admin';
relfilenode
-------------
16384
```

Bash

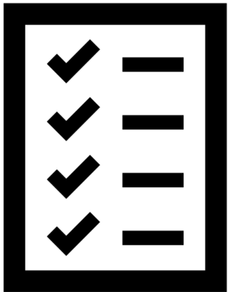
```
$ ls $PGDATA/base/13680/16384
$PGDATA/base/13680/16384
```

PostgreSQL Tables (Heap)

- Select whole table, must be a sequential scan.
- Select table's rows where id is 1200, it should not be a sequential scan.

```
EXPLAIN ANALYZE SELECT name FROM admin;
                                          QUERY PLAN
-------------------------------------------------------------------------------------
 Seq Scan on admin
 Planning Time: 0.038 ms
 Execution Time: 34185.803 ms
(3 rows)
```

Make sense?



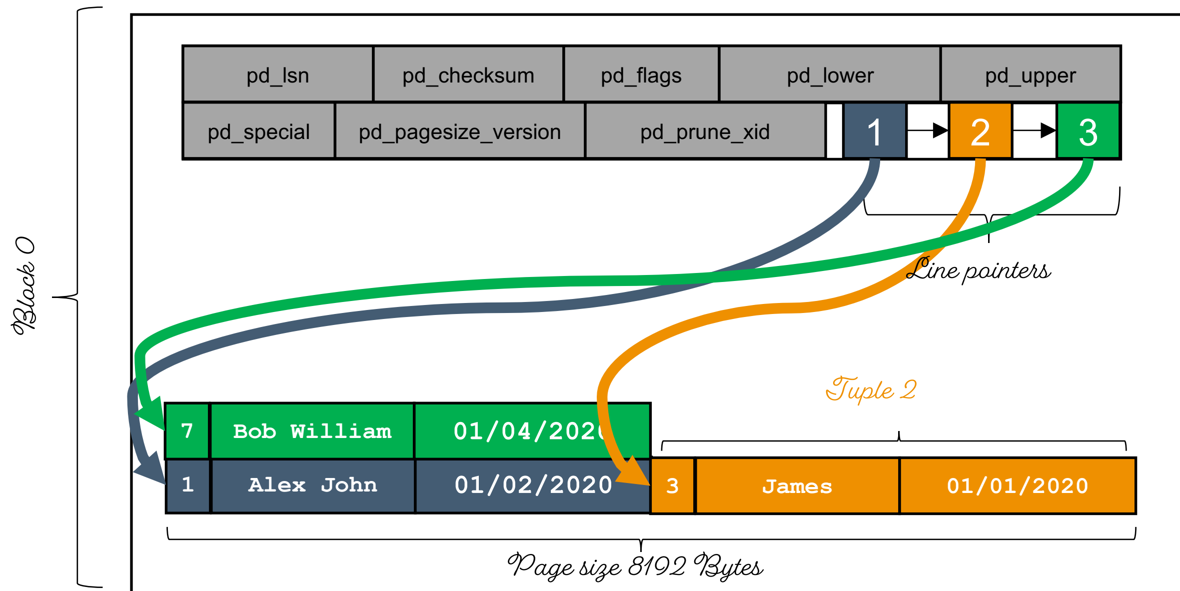
```
EXPLAIN ANALYZE SELECT name FROM admin WHERE id = 1200;
                                               QUERY PLAN
--------------------------------------------------------------------------------------
 Seq Scan on admin
   Filter: (id = 1200)
   Rows Removed by Filter: 99999812
 Planning Time: 0.111 ms
 Execution Time: 26026.162 ms
(5 rows)
```

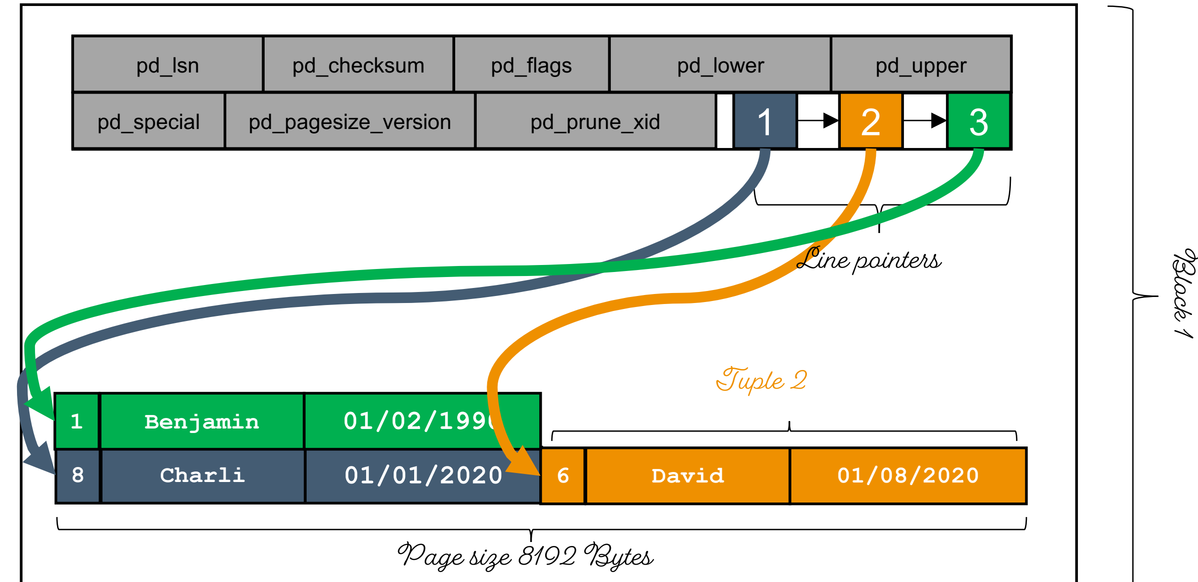
Why?

Sequential Scan

```
SELECT * FROM admin WHERE dt < '2021/04/01';
 id  |     name      |      dt
-----+--------------+------------
  3 | James         | 2020-01-01
  1 | Alex Johns    | 2020-01-02
  7 | Bob William   | 2020-01-04
  8 | Charli        | 2020-01-01
  6 | David         | 2020-08-02
  9 | Benjamin      | 1990-01-02
```

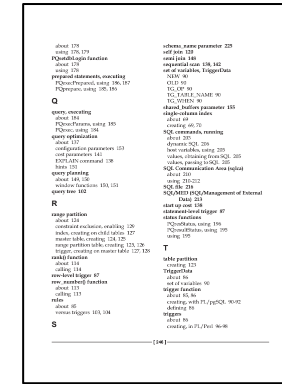
```
SELECT ctid, * FROM admin WHERE id = 8;

 ctid   | id | name
-------+----+------
 (1,1)  | 8  | Charli
(1 rows)
```





Why Index?

- Indexes are entry points for tables
- Index used to locate the tuples in the table
- The sole reason to have an index is performance
- Index is stored separately from the table's main storage (PostgreSQL Heap)
- More storage required to store the index along with original table



about 178
using 178, 179
PQsetdbLogin function
about 178
using 178
prepared statements, executing
PQexecPrepared, using 186, 187
PQprepare, using 185, 186

Q

query, executing
about 184
PQexecParams, using 185
PQexec, using 184
query optimization
about 137
configuration parameters 153
cost parameters 141
EXPLAIN command 138
hints 151
query planning
about 149, 150
window functions 150, 151
query tree 102

R

range partition
about 124
constraint exclusion, enabling 129
index, creating on child tables 127
master table, creating 124, 125
range partition table, creating 125, 126
trigger, creating on master table 127, 128
rank() function
about 114
calling 114
row-level trigger 87
row_number() function
about 113
calling 113
rules
about 85
versus triggers 103, 104

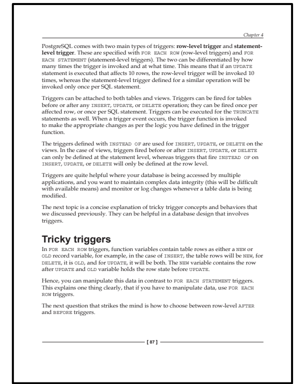
S

schema_name parameter 225
self join 120
semi join 148
sequential scan 138, 142
set of variables, TriggerData
NEW 90
OLD 90
TG_OP 90
TG_TABLE_NAME 90
TG_WHEN 90
shared_buffers parameter 155
single-column index
about 69
creating 69, 70
SQL commands, running
about 203
dynamic SQL 206
host variables, using 205
values, obtaining from SQL 205
values, passing to SQL 205
SQL Communication Area (sqlca)
about 210
using 210-212
SQL file 216
SQL/MED (SQL/Management of External Data) 213
start up cost 138
statement-level trigger 87
status functions
PQresStatus, using 196
PQresultStatus, using 195
using 195

T

table partition
creating 123
TriggerData
about 86
set of variables 90
trigger function
about 85, 86
creating, with PL/pgSQL 90-92
defining 86
triggers
about 86
creating, in PL/Perl 96-98

[246]



Chapter 4

PostgreSQL comes with two main types of triggers: **row-level trigger** and **statement-level trigger**. These are specified with `FOR EACH ROW` (row-level triggers) and `FOR EACH STATEMENT` (statement-level triggers). The two can be differentiated by how many times the trigger is invoked and at what time. This means that if an `UPDATE` statement is executed that affects 10 rows, the row-level trigger will be invoked 10 times, whereas the statement-level trigger defined for a similar operation will be invoked only once per SQL statement.

Triggers can be attached to both tables and views. Triggers can be fired for tables before or after any `INSERT`, `UPDATE`, or `DELETE` operation; they can be fired once per affected row, or once per SQL statement. Triggers can be executed for the `TRUNCATE` statements as well. When a trigger event occurs, the trigger function is invoked to make the appropriate changes as per the logic you have defined in the trigger function.

The triggers defined with `INSTEAD OF` are used for `INSERT`, `UPDATE`, or `DELETE` on the views. In the case of views, triggers fired before or after `INSERT`, `UPDATE`, or `DELETE` can only be defined at the statement level, whereas triggers that fire `INSTEAD OF` on `INSERT`, `UPDATE`, or `DELETE` will only be defined at the row level.

Triggers are quite helpful where your database is being accessed by multiple applications, and you want to maintain complex data integrity (this will be difficult with available means) and monitor or log changes whenever a table data is being modified.

The next topic is a concise explanation of tricky trigger concepts and behaviors that we discussed previously. They can be helpful in a database design that involves triggers.

Tricky triggers

In `FOR EACH ROW` triggers, function variables contain table rows as either a `NEW` or `OLD` record variable, for example, in the case of `INSERT`, the table rows will be `NEW`, for `DELETE`, it is `OLD`, and for `UPDATE`, it will be both. The `NEW` variable contains the row after `UPDATE` and `OLD` variable holds the row state before `UPDATE`.

Hence, you can manipulate this data in contrast to `FOR EACH STATEMENT` triggers. This explains one thing clearly, that if you have to manipulate data, use `FOR EACH ROW` triggers.

The next question that strikes the mind is how to choose between row-level `AFTER` and `BEFORE` triggers.

[87]

```
EXPLAIN ANALYZE SELECT name FROM admin WHERE id = 1200;
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Seq Scan on admin
   Filter: (id = 1200)
   Rows Removed by Filter: 99999812
 Planning Time: 0.111 ms
 Execution Time: 26026.162 ms
```

```
CREATE INDEX idx_id ON admin(id);
```

```
EXPLAIN ANALYZE SELECT name FROM admin WHERE id = 1200;
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Index Scan using idx_id on admin  (cost=0.57..8.59 rows=1 width=14) (actual time=2.231..2.233 rows=1 loops=1)
   Index Cond: (id = 1200)
 Planning Time: 0.288 ms
 Execution Time: 2.256 ms
```

01

PostgreSQL Indexes

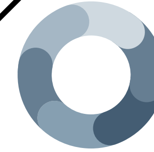
PostgreSQL Indexes

Index

How to create indexes in PostgreSQL

Concurrent Index

How to avoid locking while creating Indexes

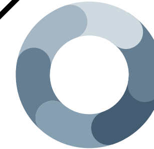


Partial Index

How to save space while creating index on big tables

Expression Index

How to create index on expressions



Index Methods

B-Tree, Hash Index
GIN, GIST, BRIN

Creating Index

- Index based on single column of the table

"admin" is a table and "id" is column

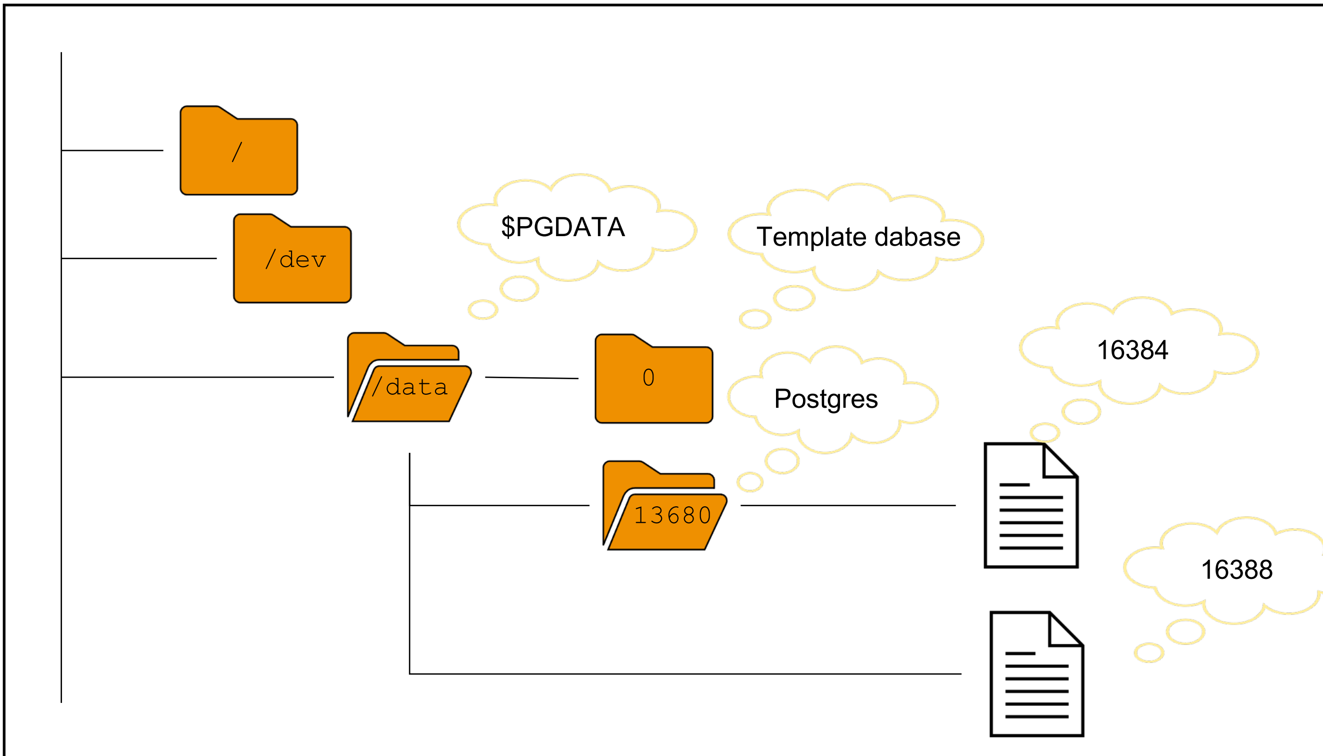
```
postgres=# CREATE INDEX idx_id ON admin(id);
```

```
EXPLAIN ANALYZE SELECT name FROM admin WHERE id = 1200;
                                 QUERY PLAN
----------------------------------------------------------------------
 Index Scan using idx_id on admin
   Index Cond: (id = 1200)
 Planning Time: 0.288 ms
 Execution Time: 2.256 ms
```

Index

- PostgreSQL standard way to create a index (https://www.postgresql.org/docs/current/sql-createindex.html)

```sql
CREATE INDEX idx_btree ON bar(id);
```



SQL

```sql
CREATE TABLE admin(id int, name text, dt date);
SELECT relfilenode FROM pg_class WHERE relname
LIKE 'admin_idx';
relfilenode
-------------
16388
```

Bash

```
$ ls $PGDATA/base/13680/16388
$PGDATA/base/13680/16388
```

Creating Index (CONCURRENTLY)

- PostgreSQL locks the table when creating index

```
CREATE INDEX idx_btree ON admin USING BTREE(id);
CREATE INDEX
Time: 12303.172 ms (00:12.303)
```

- CONCURRENTLY option creates the index without locking the table

```
CREATE INDEX CONCURRENTLY idx_btree ON admin USING BTREE(id);
CREATE INDEX
Time: 23025.372 ms (00:23.025)
```

Expression Index

Index based on Column	Index based on Expression

Index based on Column

```
EXPLAIN ANALYZE SELECT * FROM admin
WHERE name LIKE 'David';
                      QUERY PLAN
-----------------------------------------------------------
Seq Scan on admin
   Filter: (name ~~ 'David'::text)
   Rows Removed by Filter: 99999812
 Planning Time: 0.068 ms
 Execution Time: 24721.398 ms
```

```
CREATE INDEX idx_name ON admin (name);
```

```
EXPLAIN ANALYZE SELECT * from admin
WHERE lower(name) LIKE 'david';
                        QUERY PLAN
-----------------------------------------------------------
Seq Scan on admin
   Filter: (lower(name) ~~ 'david'::text)
   Rows Removed by Filter: 99999812
 Planning Time: 0.255 ms
 Execution Time: 71892.784 ms
```

Index based on Expression

```
EXPLAIN ANALYZE SELECT * FROM admin
WHERE lower(name) LIKE 'david';
                        QUERY PLAN
-----------------------------------------------------------
Seq Scan on admin
   Filter: (lower(name) ~~ 'david'::text)
   Rows Removed by Filter: 99999812
 Planning Time: 0.118 ms
 Execution Time: 80422.699 ms
```

```
CREATE INDEX idx_name_exp ON admin (lower(name));
```

```
ANALYZE SELECT * FROM admin
WHERE lower(name) LIKE 'david';
                        QUERY PLAN
-----------------------------------------------------------
Index Scan using idx_name_exp on admin
   Index Cond: (lower(name) = 'david'::text)
   Filter: (lower(name) ~~ 'david'::text)
 Planning Time: 0.087 ms
 Execution Time: 1.157 ms
```

Expression Index 2/2

```
postgres=# EXPLAIN SELECT * FROM bar WHERE (dt + (INTERVAL '2 days'))  < now();
                          QUERY PLAN
----------------------------------------------------------------
 Seq Scan on bar  (cost=0.00..238694.00 rows=3333333 width=40)
   Filter: ((dt + '2 days'::interval) < now())
```

```
postgres=# CREATE INDEX idx_math_exp ON bar((dt + (INTERVAL '2 days')));
```

```
postgres=# EXPLAIN SELECT * FROM bar WHERE (dt + (INTERVAL '2 days'))  < now();
                                     QUERY PLAN
-------------------------------------------------------------------------------------
 Bitmap Heap Scan on bar  (cost=62449.77..184477.10 rows=3333333 width=40)
   Recheck Cond: ((dt + '2 days'::interval) < now())
   ->  Bitmap Index Scan on idx_math_exp  (cost=0.00..61616.43 rows=3333333 width=0)
         Index Cond: ((dt + '2 days'::interval) < now())
```

Partial Index

Index

```
CREATE INDEX idx_full ON bar(id);
EXPLAIN SELECT * FROM bar
                WHERE id  < 1000
                AND name LIKE 'text1000';
                                QUERY PLAN
--------------------------------------------------------------------
-------
 Bitmap Heap Scan on bar  (cost=61568.60..175262.59 rows=16667
width=40)
   Recheck Cond: (id < 1000)
   Filter: ((name)::text ~~ 'text1000'::text)
   ->  Bitmap Index Scan on idx_full  (cost=0.00..61564.43
rows=3333333 width=0)
         Index Cond: (id < 1000)

SELECT pg_size_pretty(pg_total_relation_size('idx_full'));
 pg_size_pretty
----------------
 214 MB
(1 row)
```

Look at the size of the index

Partial Index

```
CREATE INDEX idx_part ON bar(id) where id < 1000;
EXPLAIN SELECT * FROM bar
            WHERE id  < 1000
            AND name LIKE 'text1000';
                          QUERY PLAN
-------------------------------------------------------------
-------------
 Bitmap Heap Scan on bar  (cost=199.44..113893.44
rows=16667 width=40)
   Recheck Cond: (id < 1000)
   Filter: ((name)::text ~~ 'text1000'::text)
   ->  Bitmap Index Scan on idx_part  (cost=0.00..195.28
rows=3333333 width=0)
         Index Cond: (id < 1000)

SELECT pg_size_pretty(pg_total_relation_size('idx_part'));
 pg_size_pretty
----------------
 240 kB
(1 row)
```

Index where id < 1000 only

Why create full index if we don't need that.

02

Index Methods

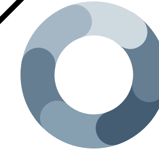
PostgreSQL Index Methods

B-Tree

PostgreSQL default index
Based on B-Tree

Hash

PostgreSQL Index Method
Based on Hasing



Brin

Block Range Index

GIST

Generalized Search Tree

GIN

Generalized Inverted
Index

B-Tree Index

- What is a B-Tree index?
- Supported Operators
  - Less than <
  - Less than equal to <=
  - Equal =
  - Greater than equal to >=
  - Greater than >

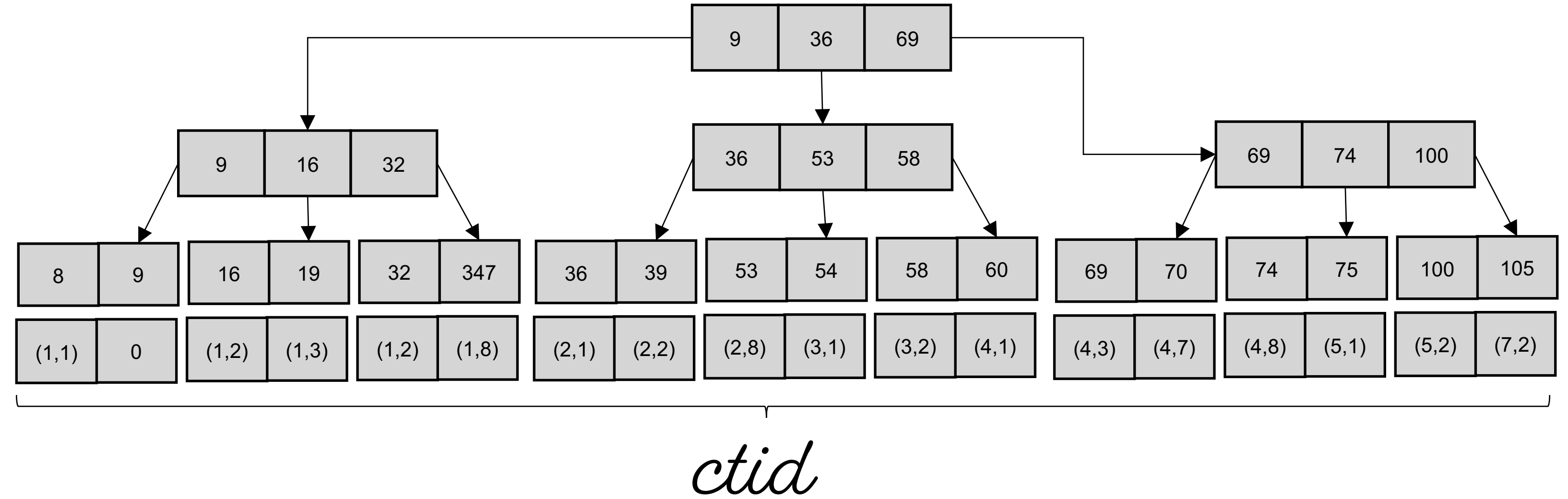
Wikipedia: (https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree)
In computer science, a self-balancing (or height-balanced) binary search tree is any node-based binary search tree that automatically keeps its height small in the face of arbitrary item insertions and deletions.

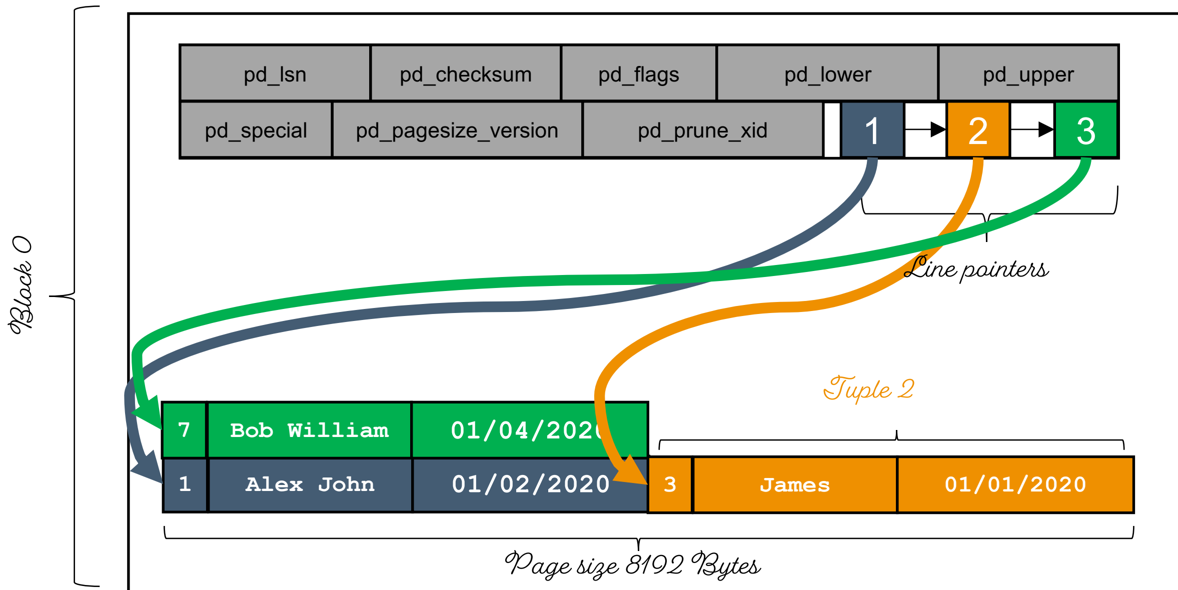
```
CREATE INDEX idx_btree ON admin USING BTREE (name);
```

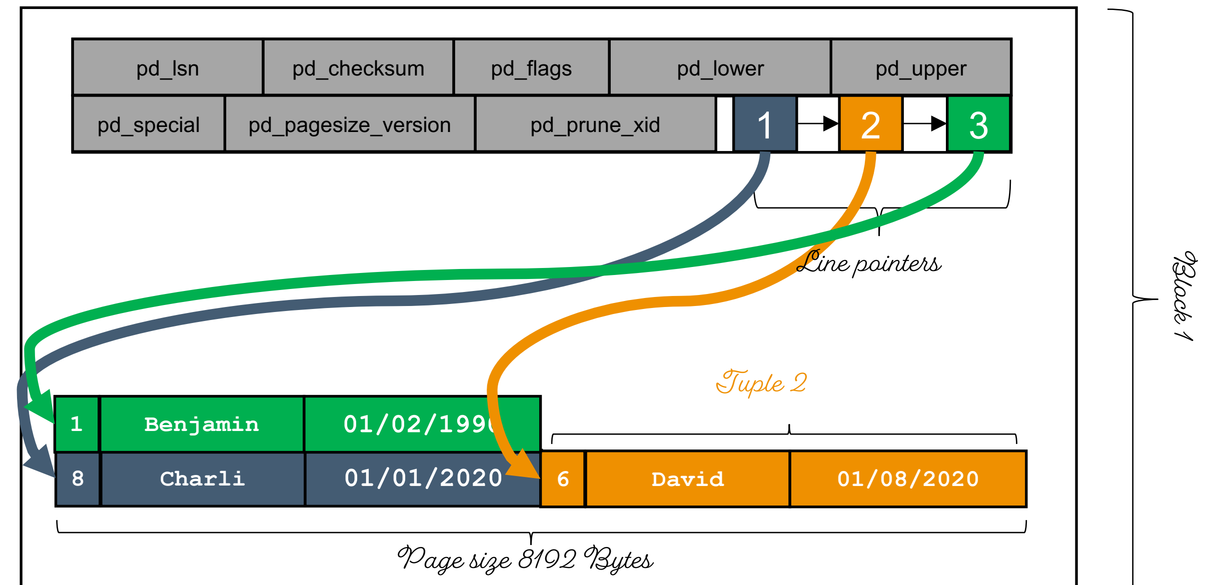
```
postgres=# EXPLAIN ANALYZE SELECT * FROM admin WHERE name = 'text%';
                              QUERY PLAN
------------------------------------------------------------------------------------------
 Index Scan using idx_btree on admin  (cost=0.43..8.45 rows=1 width=19) (actual time=0.015..0.015 rows=0 loops=1)
   Index Cond: ((name)::text = 'text%'::text)
 Planning Time: 0.105 ms
 Execution Time: 0.031 ms
(4 rows)
```

B-Tree Index

```
SELECT id, name FROM admin
WHERE id = 8;

 id    | name
 -----+------
   8   | Charli
(1 rows)
```







Index Only Scans

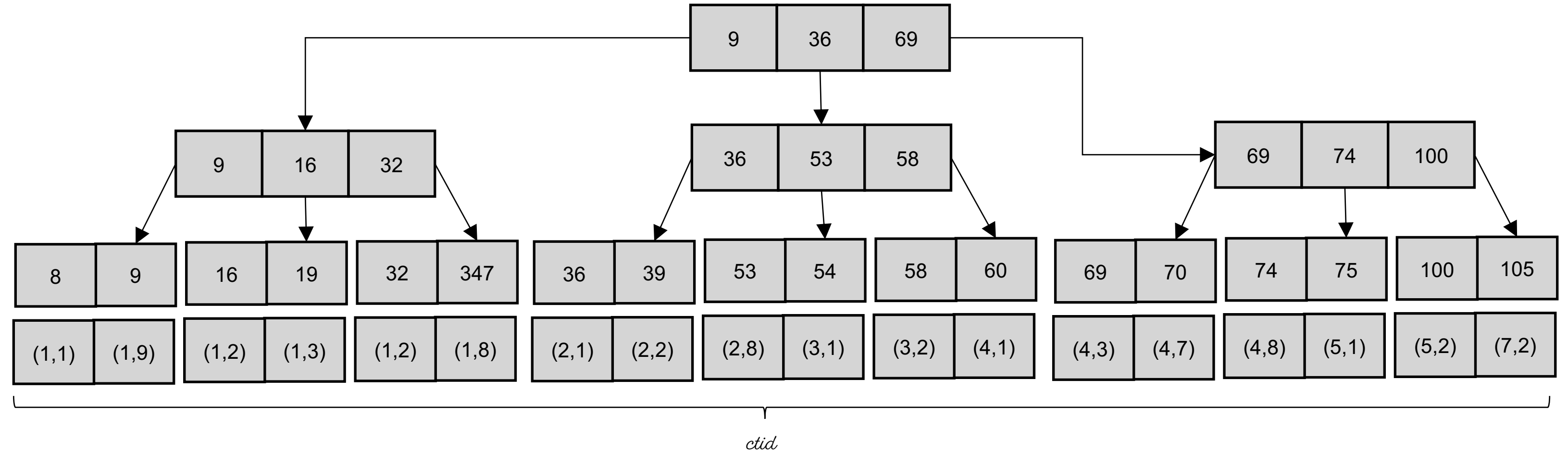
```
CREATE INDEX idx ON admin (id);
```

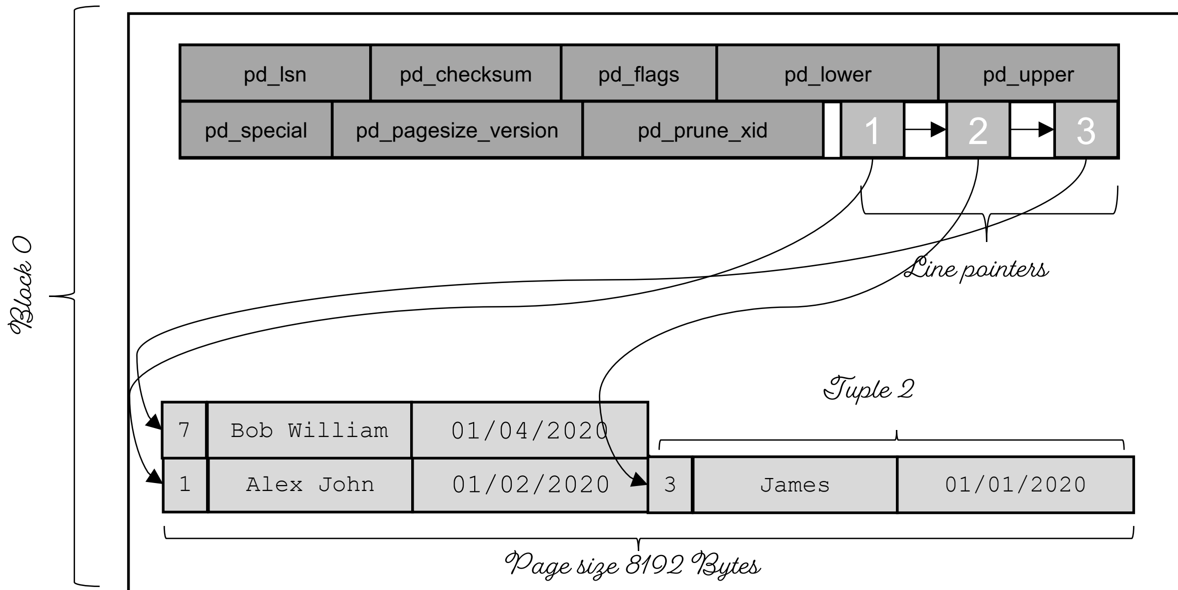
```
EXPLAIN SELECT id, name, dt FROM admin WHERE id > 100000 AND id <100010;
                              QUERY PLAN
Index Scan using idx on admin  (cost=0.56..99.20 rows=25 width=19)
   Index Cond: ((id > 100000) AND (id < 100010))
(2 rows)
```

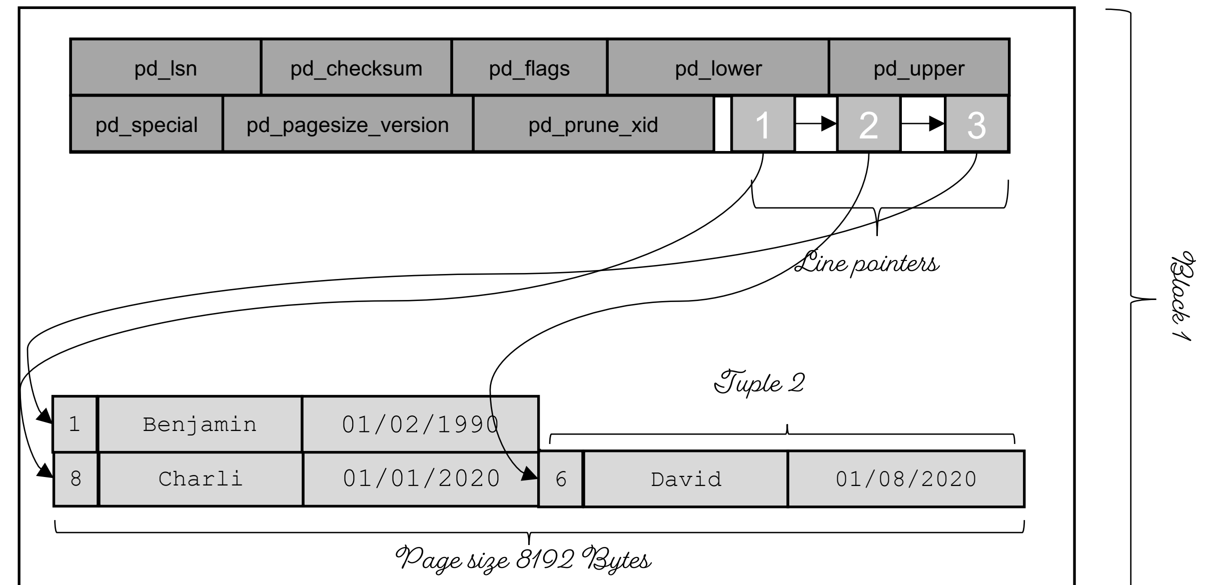
```
EXPLAIN SELECT id FROM admin WHERE id > 100000 AND id <100010;
                              QUERY PLAN
Index Only Scan using idx on admin  (cost=0.56..99.20 rows=25 width=15)
   Index Cond: ((id > 100000) AND (id < 100010))
(2 rows)
```

B-Tree Index (Index Only Scans)

```
SELECT id FROM admin WHERE
id = 8;

 id
-----
  8
(1 rows)
```

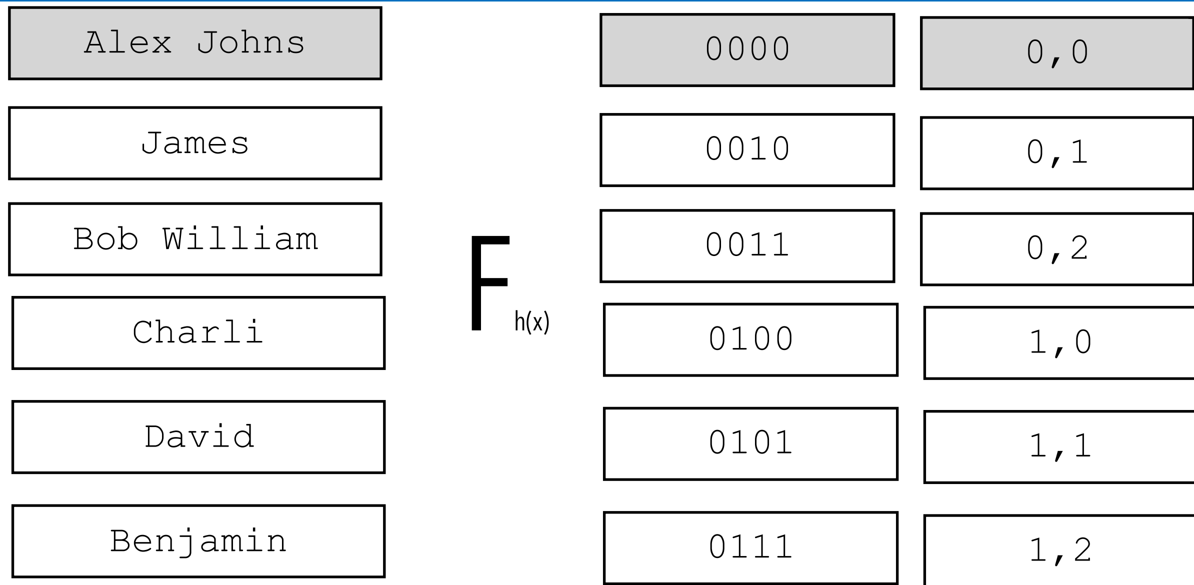


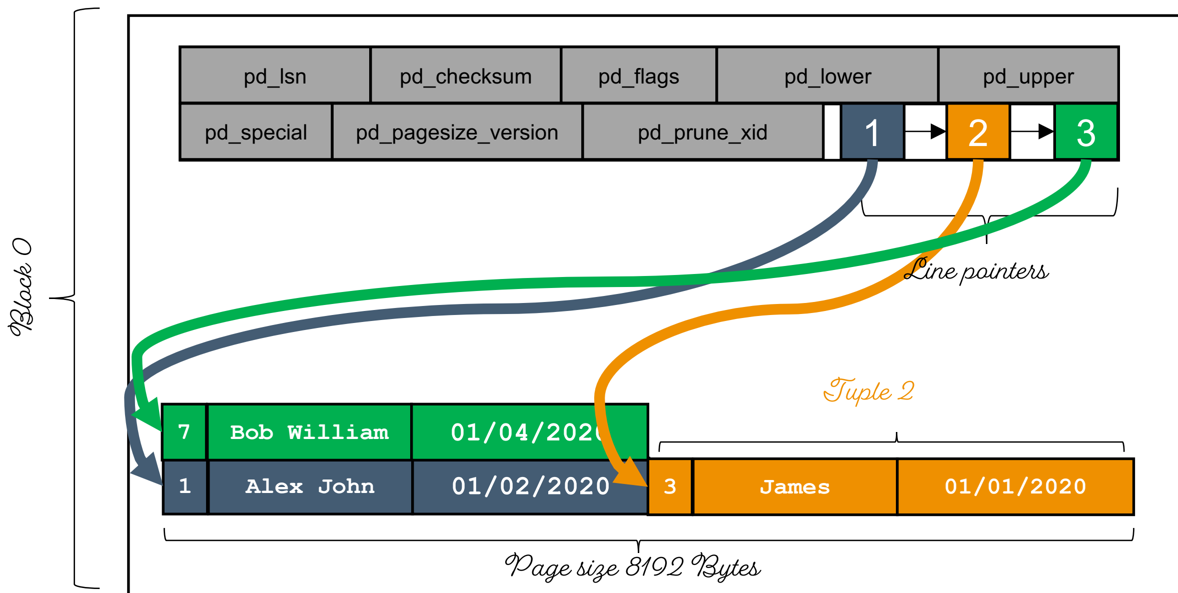


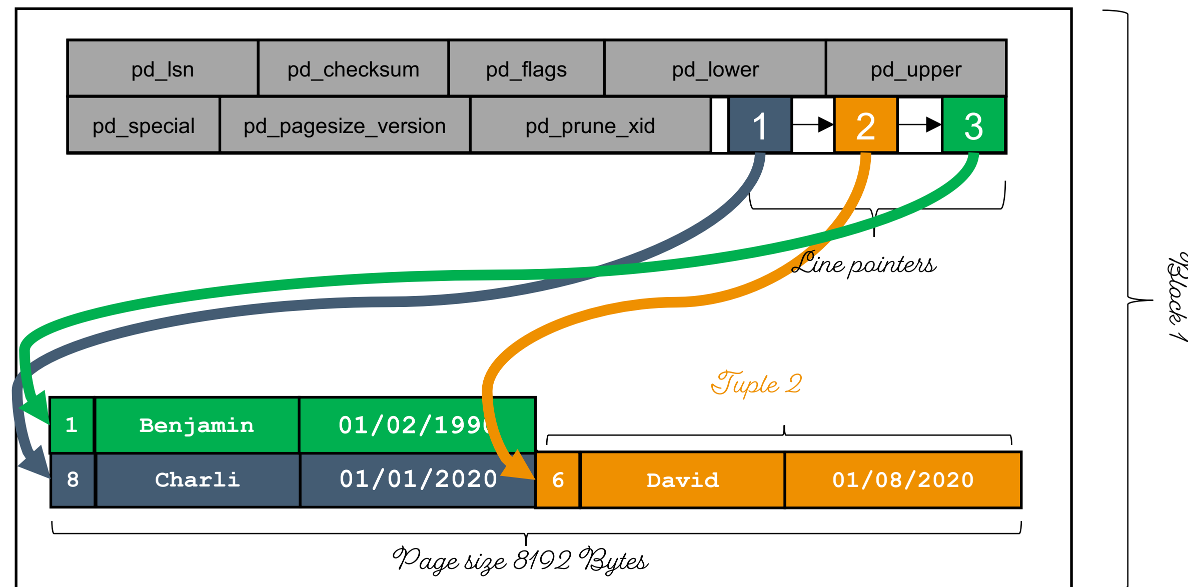


Hash Index

```
SELECT id, name FROM admin WHERE name
LIKE 'Alex Johns';

 id   | name
-----+-------
  16  | Alex Johns
(1 rows)
```







HASH Index

- What is a Hash index?
- Hash indexes only handles equality operators
- Hash function is used to locate the tuples

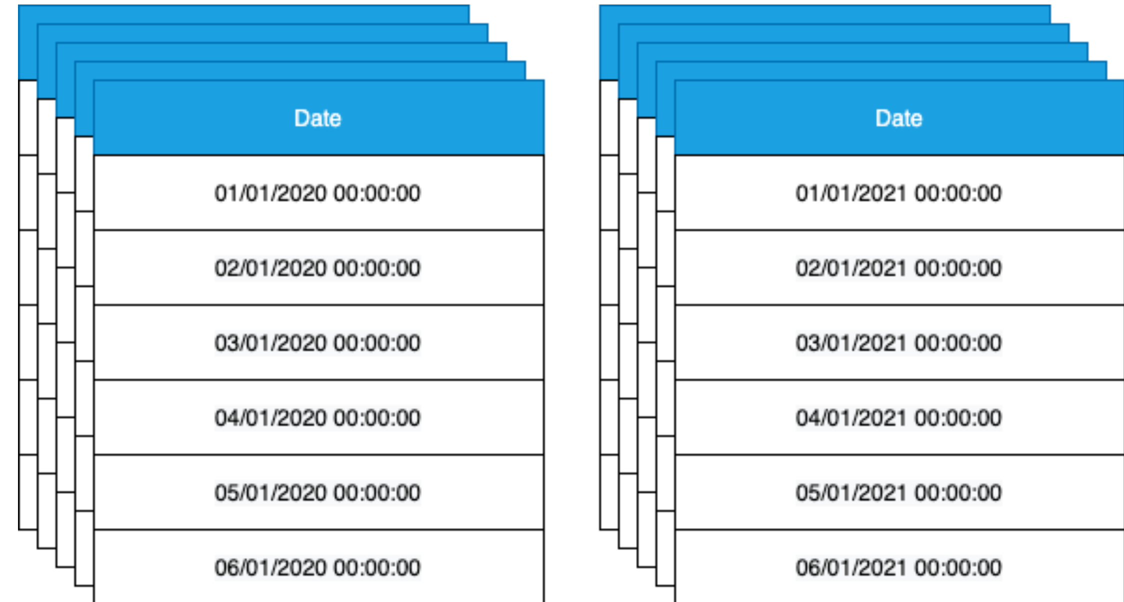
```
postgres=# \d bar
                  Table "public.bar"
 Column |       Type        | Collation | Nullable | Default
--------+-------------------+-----------+----------+---------
 id     | integer           |           |          |
 name   | character varying |           |          |
 dt     | date              |           |          |
Indexes:
    "idx_btree" btree (name)
    "idx_hash" btree (name)
```

```
CREATE INDEX idx_hash ON bar USING HASH (name);
```

```
EXPLAIN ANALYZE SELECT * FROM bar WHERE name = 'text%';
                                  QUERY PLAN
Index Scan using idx_hash on bar  (cost=0.43..8.45 rows=1 width=19) (actual time=0.023..0.023
rows=0 loops=1)
   Index Cond: ((name)::text = 'text%'::text)
 Planning Time: 0.080 ms
 Execution Time: 0.041 ms
(4 rows)
```

BRIN Index

- BRIN is a “Block Range Index”
- Used when columns have some correlation with their physical location in the table
- Space optimized because BRIN index contains only three items
  - Page/Block number
  - Min value of column
  - Max value of column



BRIN Index

Sequential Scan

```
postgres=# EXPLAIN ANALYZE SELECT *
           FROM bar
           WHERE dt > '2022-09-28'
           AND   dt < '2022-10-28';
                     QUERY PLAN
------------------------------------------------------
Seq Scan on bar (cost=0.00..2235285.00 rows=1
                  width=27)
                 (actual time=0.139..7397.090 rows=29
                 loops=1)
   Filter: ((dt > '2022-09-28 00:00:00)
       AND (dt < '2022-10-28 00:00:00))
   Rows Removed by Filter: 99999971
 Planning Time: 0.114 ms
 Execution Time: 7397.107 ms
(5 rows)
```

BRIN Index

```
postgres=# EXPLAIN ANALYZE SELECT *
           FROM bar
           WHERE dt > '2022-09-28'
           AND dt < '2022-10-28';
                     QUERY PLAN
------------------------------------------------------
Bitmap Heap Scan on bar (cost=92.03..61271.08 rows=1
  width=27) (actual time=1.720..4.186 rows=29 loops=1)
   Recheck Cond: ((dt > '2022-09-28 00:00:00')
            AND (dt < '2022-10-28 00:00:00'))
   Rows Removed by Index Recheck: 18716
   Heap Blocks: lossy=128
   -> Bitmap Index Scan on idx_brin
         (cost=0.00..92.03 rows=17406 width=0)
         (actual time=1.456..1.456 rows=1280 loops=1)
        Index Cond: ((dt > '2022-09-28 00:00:00')
            AND (dt < '2022-10-28 00:00:00'))
 Planning Time: 0.130 ms
 Execution Time: 4.233 ms
(8 rows)
```

BRIN Index On Disk Size Comparison

- B-TREE Index

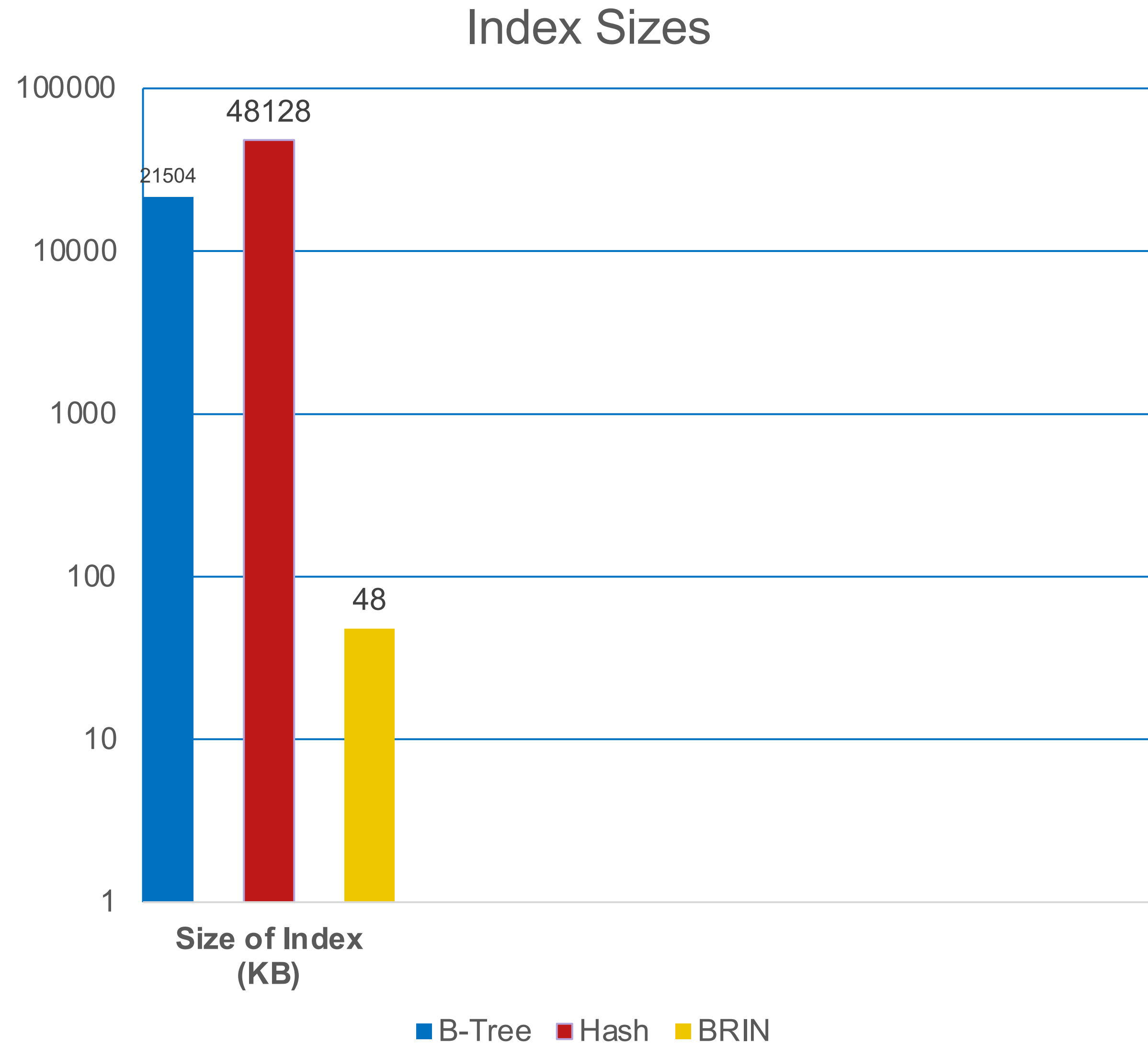
```
CREATE INDEX idx_btree ON bar USING BTREE (date);
```

- Hash Index

```
CREATE INDEX idx_hash ON bar USING HASH (date);
```

- BRIN Index

```
CREATE INDEX idx_brin ON bar USING BRIN (date);
```



GIN Index

- Generalized Inverted Index
- GIN is to handle where we need to index composite values
- Slow while creating the index because it needs to scan the document up front

```
postgres=# \d bar
                Table "public.bar"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 id     | integer |           |          |
 name   | jsonb   |           |          |
 dt     | date    |           |          |
```

```
postgres=# SELECT DISTINCT name, dt FROM bar LIMIT 5;
                                   name                                   |     dt
--------------------------------------------------------------------------+------------
 {"name": "Alex", "phone": ["333-333-333", "222-222-222", "111-111-111"]}  | 2019-05-13
 {"name": "Bob", "phone": ["333-333-444", "222-222-444", "111-111-444"]}   | 2019-05-14
 {"name": "John", "phone": ["333-3333", "777-7777", "555-5555"]}           | 2019-05-15
 {"name": "David", "phone": ["333-333-555", "222-222-555", "111-111-555"]} | 2019-05-16
(4 rows)
```

GIN Index

- Generalized Inverted Index
- GIN is to handle where we need to index composite values
- Slow while creating index because it needs to scan the document up front

```
CREATE INDEX idx_gin ON bar USING GIN (name1);
```

```
postgres=# EXPLAIN ANALYZE SELECT * FROM bar
            WHERE name @> '{"name": "Alex"}';
                      QUERY PLAN
-----------------------------------------------------
Seq Scan on bar  (cost=0.00..108309.34 rows=3499
width=96) (actual time=396.019..1050.143 rows=1000000
loops=1)
   Filter: (name @> '{"name": "Alex"}'::jsonb)
   Rows Removed by Filter: 3000000
 Planning Time: 0.107 ms
 Execution Time: 1079.861 ms
```

```
postgres=# EXPLAIN ANALYZE SELECT * FROM bar
           WHERE name @> '{"name": "Alex"}';
                     QUERY PLAN
-----------------------------------------------------
Bitmap Heap Scan on bar  (cost=679.00..13395.57
rows=4000 width=96) (actual time=91.110..445.112
rows=1000000 loops=1)
   Recheck Cond: (name @> '{"name": "Alex"}'::jsonb)
   Heap Blocks: exact=16394
   ->  Bitmap Index Scan on
idx_gin  (cost=0.00..678.00 rows=4000 width=0)
(actual time=89.033..89.033 rows=1000000 loops=1)
         Index Cond: (name @> '{"name":
"Alex"}'::jsonb)
 Planning Time: 0.168 ms
 Execution Time: 475.447 ms
```

GiST Index

- Generalized Search Tree
- It is Tree-structured access method
- It is a indexing framework used for indexing of complex data types.
  - Used to find the point within box
  - Used for full text search
  - Intarray

```
CREATE TABLE simple_points(p point);
INSERT INTO simple_points(p) values (point(2,2));
INSERT INTO simple_points(p) values (point(2,4));
INSERT INTO simple_points(p) values (point(4,2));
INSERT INTO simple_points(p) values (point(4,4));
INSERT INTO simple_points(p) values (point(5,5));
```

```
CREATE TABLE simple_box(b box);
INSERT INTO simple_box VALUES (box(point(2,2),
point(4,4)));
```

```
CREATE INDEX simple_points_idx on simple_points using gist(p);
```

```
EXPLAIN SELECT * FROM simple_points, simple_box where p <@ b;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Nested Loop  (cost=10000000000.13..10000000133.82 rows=7 width=48)
   Join Filter: (simple_points.p <@ simple_box.b)
   ->  Seq Scan on simple_box  (cost=10000000000.00..10000000023.60 rows=1360 width=32)
   ->  Materialize  (cost=0.13..8.23 rows=5 width=16)
         ->  Index Scan using simple_points_idx on simple_points  (cost=0.13..8.21 rows=5 width=16)
(5 rows)
```

Where and What?

- B-Tree: Use this index for most of the queries and different data types
- Hash: Used for equality operators
- BRIN: For really large sequentially lineup datasets
- GIN: Used for documents and arrays
- GiST: Used for full text search

Duplicate Indexes

```
SELECT indrelid::regclass relname,
       indexrelid::regclass indexname, indkey
FROM pg_index
GROUP BY relname,indexname,indkey;
         relname            |                    indexname                     | indkey
----------------------------+--------------------------------------------------+----------
 pg_index                   | pg_index_indexrelid_index                        | 1
 pg_toast.pg_toast_2615     | pg_toast.pg_toast_2615_index                     | 1 2
 pg_constraint              | pg_constraint_conparentid_index                  | 11
```

```
SELECT indrelid::regclass relname, indkey, amname
FROM pg_index i, pg_opclass o, pg_am a
WHERE o.oid = ALL (indclass)
AND a.oid = o.opcmethod
GROUP BY relname, indclass, amname, indkey
HAVING count(*) > 1;
 relname | indkey | amname
---------+--------+--------
 bar     |      2 | btree
(1 row)
```

Index Stats (pg_stat_user_indexes, pg_stat_statement)

```
postgres=# \d pg_stat_user_indexes;
         View "pg_catalog.pg_stat_user_indexes"
    Column     |  Type  | Collation | Nullable | Default
---------------+--------+-----------+----------+---------
 relid         | oid    |           |          |
 indexrelid    | oid    |           |          |
 schemaname    | name   |           |          |
 relname       | name   |           |          |
 indexrelname  | name   |           |          |
 idx_scan      | bigint |           |          |
 idx_tup_read  | bigint |           |          |
 idx_tup_fetch | bigint |           |          |
```

Unused Indexes

```
SELECT relname, indexrelname, idx_scan
FROM  pg_catalog.pg_stat_user_indexes;

 relname | indexrelname   | idx_scan
---------+----------------+----------
 foo     | idx_foo_date   |        0
 bar     | idx_btree      |        0
 bar     | idx_btree_id   |        0
 bar     | idx_btree_name|        6
 bar     | idx_brin_brin |        4
(7 rows)
```

THANK YOU

GET IN TOUCH

@ibrar_ahmad

https://www.facebook.com/ibrar.ahmed

https://www.linkedin.com/in/ibrarahmed74/

www.pgelephant.com