Deep Dive Into PostgreSQL Indexes

Ibrar Ahmed
Senior Database Architect - Percona LLC
May 2019
Table Characteristics

- Rows / Tuples stored in a table
- Every table in PostgreSQL has physical disk file(s)

postgres=# **CREATE TABLE** foo(id int, name text);
postgres=# **SELECT** relfilenode **FROM** pg_class **WHERE** relname **LIKE** 'foo';

```
relfilenode
-------------
   16384
```

- The physical files on disk can be seen in the PostgreSQL `$PGDATA` directory.

```
$ ls -lrt $PGDATA/base/13680/16384
-rw------- 1 vagrant vagrant 0 Apr 29 11:48 $PGDATA/base/13680/16384
```

- Tuple stored in a table does not have any order
Selecting Data 1/2

- Select whole table, must be a sequential scan.
- Select table’s rows where id is 5432, it should not be a sequential scan.
SELECT ctid, * FROM foo;

ctid | id | name
-------+-----+
(0,1) | 1  | Alex
(0,2) | 2  | Bob
(2 rows)

- How to select the data from the HEAP?
- Need to scan each and every page and look for the tuple in the page

Cost?
PostgreSQL Indexes

https://www.postgresql.org/docs/current/indexes.html
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

```
postgres=# EXPLAIN SELECT name FROM bar WHERE id = 5432;

QUERY PLAN
----------------------------------------------------------------------------
Seq Scan on bar (cost=0.00..159235.00 rows=38216 width=32)
Filter: (id = 5432)
```

```
postgres=# CREATE INDEX bar_idx ON bar(id);

postgres=# EXPLAIN SELECT name FROM bar WHERE id = 5432;

QUERY PLAN
----------------------------------------------------------------------------
Bitmap Heap Scan on bar (cost=939.93..64313.02 rows=50000 width=32)
Recheck Cond: (id = 5432)
  -> Bitmap Index Scan on bar_idx (cost=0.00..927.43 rows=50000 width=0)
    Index Cond: (id = 5432)
```
Index

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

```
postgres=# CREATE INDEX idx_btree ON bar(id);
```

```
postgres=# SELECT relfilenode FROM pg_class WHERE relname LIKE 'idx_btree';
```

```
relfilenode
-------------
  16425
```

- PostgreSQL index has its own file on disk.

The physical file on disk can be seen in the PostgreSQL $PGDATA directory.

```
$ ls -lrt $PGDATA/13680/16425
-rw-------1 vagrant vagrant 1073741824 Apr 29 13:05 $PGDATA/base/13680/16425
```
Creating Index 1/2

- Index based on single column of the table

```
postgres=# CREATE INDEX bar_idx ON bar(id);
```

```
postgres=# EXPLAIN SELECT name FROM bar WHERE id = 5432;
```

```
QUERY PLAN

---------
 Bitmap Heap Scan on bar (cost=939.93..64313.02 rows=50000 width=32)
   Recheck Cond: (id = 5432)
   -> Bitmap Index Scan on bar_idx (cost=0.00..927.43 rows=50000 width=0)
      Index Cond: (id = 5432)

```
Creating Index 2/2

PostgreSQL locks the table when creating index

```
CREATE INDEX idx_btree ON bar 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 bar USING BTREE(id);
CREATE INDEX
Time: 23025.372 ms (00:23.025)
```
CREATE INDEX idx_exp ON bar (lower(name));

EXPLAIN SELECT * FROM bar WHERE lower(name) LIKE 'Text1';

    QUERY PLAN

-------------------------------------------------------------------------------------------------------------------------------
Seq Scan on bar   (cost=0.00..213694.00  rows=50000 width=40)
  Filter: (lower((name)::text) ~~ 'Text1 '::text)

CREATE INDEX idx_exp ON bar (lower(name));

EXPLAIN SELECT * FROM bar WHERE lower(name) LIKE 'Text1';

    QUERY PLAN

-------------------------------------------------------------------------------------------------------------------------------
Bitmap Heap Scan on bar   (cost=1159.93..64658.02  rows=50000 width=40)
  Filter: (lower((name)::text) ~~ 'Text1 '::text)
  -> Bitmap Index Scan on idx_exp   (cost=0.00..1147.43  rows=50000 width=0)
    Index Cond: (lower((name)::text) = 'Text1 '::text)
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())

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
---------------------------------------------------------------
Seq Scan on bar (cost=0.00..238694.00 rows=3333333 width=40)
  Filter: ((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())
CREATE INDEX idx_full ON bar(id);

EXPLAIN SELECT * FROM bar
WHERE id < 1000
AND name LIKE 'text1000';

SELECT pg_size_pretty(pg_total_relation_size('idx_full'));

pg_size_pretty

-----------------
214 MB
(1 row)

Q: What will happen when we query where id >1000?
A: Answer is simple, this index won’t selected.

CREATE INDEX idx_part ON bar(id) WHERE id < 10000;

EXPLAIN SELECT * FROM bar
WHERE id < 1000
AND name LIKE 'text1000';

SELECT pg_size_pretty(pg_total_relation_size('idx_part'));

pg_size_pretty

-----------------
240 kB
(1 row)
Index Types

https://www.postgresql.org/docs/current/indexes-types.html
B-Tree Index 1/2

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


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 foo USING BTREE (name);

postgres=# EXPLAIN ANALYZE SELECT * FROM foo WHERE name = 'text%';

QUERY PLAN

----------------------------------------------------------------------------------------------------------------
| Index Scan using idx_btree on foo (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)
CREATE TABLE foo(id INTEGER, name TEXT);
INSERT INTO foo VALUES(1, 'Alex');
INSERT INTO foo VALUES(2, 'Bob');

SELECT ctid, * FROM foo;
ctid  | id  | name
-------+-----+-------
(0,1)  | 1   | Alex
(0,2)  | 2   | Bob

Index have the key and the location of the tuple.

ctid  | name
-------+-------
(0,1)  | Alex
(0,2)  | Bob
(2,2)  | Alex
HASH Index

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

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

```
postgres=# \d bar
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>Table &quot;public.bar&quot;</td>
<td></td>
</tr>
<tr>
<td>Column</td>
<td>Type</td>
<td>Collation</td>
</tr>
<tr>
<td>--------</td>
<td>---------------</td>
<td>-----------</td>
</tr>
<tr>
<td>id</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>character varying</td>
<td></td>
</tr>
<tr>
<td>dt</td>
<td>date</td>
<td></td>
</tr>
</tbody>
</table>

Indexes:
"idx_btree" btree (name)
"idx_hash" btree (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%'

Planning Time: 0.080 ms
Execution Time: 0.041 ms
(4 rows)
BRIN Index 1/2

- 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 number
  - Min value of column
  - Max value of column

```sql
CREATE INDEX idx_btree ON bar USING BTREE (date);
CREATE INDEX idx_hash ON bar USING HASH (date);
CREATE INDEX idx_brin ON bar USING BRIN (date);
```
**BRIN Index 2/2**

**Sequential Scan**

```sql
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
| Planning Time: 0.130 ms
| Execution Time: 7397.107 ms
```

**BRIN Index**

```sql
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 Scan**

```
-> 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)
```
GIN Index 1/2

- 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

```sql
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": "David", "phone": ["333-333-555", "222-222-555", "111-111-555"]} | 2019-05-16
(4 rows)
```
GIN Index 2/2

- 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

```sql
CREATE INDEX idx_gin ON bar USING GIN (name);
```

```
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
```

Even if you create a BTREE index, it won’t be considered. Because it does not know the individual element in value.
GiST Index

- Generalized Search Tree
- A GiST index is lossy
- Tree-structured access method
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
Index Only Scans

- Index is stored separately from the table’s main storage (PostgreSQL Heap)

- Query needs to scan both the index and the heap

- Index Only Scans only used when all the columns in the query part of the index

- In this case PostgreSQL fetches data from index only
Index Only Scans

CREATE INDEX idx_btree_ios ON bar (id,name);

EXPLAIN SELECT id, name, dt FROM bar WHERE id > 100000 AND id < 100010;

<table>
<thead>
<tr>
<th>QUERY PLAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index Scan using idx_btree_ios on bar (cost=0.56..99.20 rows=25 width=19)</td>
</tr>
<tr>
<td>Index Cond: ((id &gt; 100000) AND (id &lt; 100010))</td>
</tr>
<tr>
<td>(2 rows)</td>
</tr>
</tbody>
</table>

EXPLAIN SELECT id, name FROM bar WHERE id > 100000 AND id < 100010;

<table>
<thead>
<tr>
<th>QUERY PLAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index Only Scan using idx_btree_ios on bar (cost=0.56..99.20 rows=25 width=15)</td>
</tr>
<tr>
<td>Index Cond: ((id &gt; 100000) AND (id &lt; 100010))</td>
</tr>
<tr>
<td>(2 rows)</td>
</tr>
</tbody>
</table>
### Duplicate Indexes

**SELECT**

```sql
SELECT
    indrelid::regclass relname, 
    indexrelid::regclass indexname, indkey
FROM pg_index
GROUP BY relname, indexname, indkey;
```

<table>
<thead>
<tr>
<th>relname</th>
<th>indexname</th>
<th>indkey</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg_index</td>
<td>pg_index_indexrelid_index</td>
<td>1</td>
</tr>
<tr>
<td>pg_toast</td>
<td>pg_toast.pg_toast_2615_index</td>
<td>1 2</td>
</tr>
<tr>
<td>pg_constraint</td>
<td>pg_constraint_conparentid_index</td>
<td>1 11</td>
</tr>
</tbody>
</table>

**SELECT**

```sql
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;
```

<table>
<thead>
<tr>
<th>relname</th>
<th>indexname</th>
<th>indkey</th>
</tr>
</thead>
<tbody>
<tr>
<td>bar</td>
<td>2</td>
<td>btree</td>
</tr>
</tbody>
</table>

(1 row)
### Unused Indexes

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

<table>
<thead>
<tr>
<th>relname</th>
<th>indexrelname</th>
<th>idx_scan</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>idx_foo_date</td>
<td>0</td>
</tr>
<tr>
<td>bar</td>
<td>idx_btree</td>
<td>0</td>
</tr>
<tr>
<td>bar</td>
<td>idx_btree_id</td>
<td>0</td>
</tr>
<tr>
<td>bar</td>
<td>idx_btree_name</td>
<td>6</td>
</tr>
<tr>
<td>bar</td>
<td>idx_brin_brin</td>
<td>4</td>
</tr>
</tbody>
</table>

(7 rows)
“Poor leaders rarely ask questions of themselves or others. Good leaders, on the other hand, ask many questions. Great leaders ask the great questions.”

Michael Marquardt author of Leading with Questions
Thank You to Our Sponsors
Introducing gh-ost: triggerless, painless, trusted online schema migrations

11:20 → 12:10
Matterhorn 2