

Deep Dive Into PostgreSQL Indexes

Learn PostgreSQL index

Ibrar Ahmed

Senior Database Architect - Percona LLC

June 2019



Tables (Heap)

"Tuple" and "Rows" are synonym

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

"Relfilenode" is table file name of that table

"pg_class" is system table to contain table information

```
postgres=# CREATE TABLE foo(id int, name text);
postgres=# SELECT relfilenode FROM pg_class WHERE relname LIKE 'foo';
relfilenode
-----
16384
```

"16384" is table filename

\$PGDATA is Data Directory

- 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

Tables (Heap)

- Select whole table, must be a sequential scan.
- Select table's rows where id is 5432, it should not be a sequential scan.

```
EXPLAIN SELECT name FROM bar;
```

Make sense?

QUERY PLAN

```
Seq Scan on bar (cost=0.00..163693.05 rows=9999905 width=11)
```

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

QUERY PLAN

```
Gather (cost=1000.00..116776.94 rows=1 width=11)
```

```
Workers Planned: 2
```

why?

```
-> Parallel Seq Scan on bar (cost=0.00..115776.84 rows=1 width=11)
```

```
Filter: (id = 5432)
```

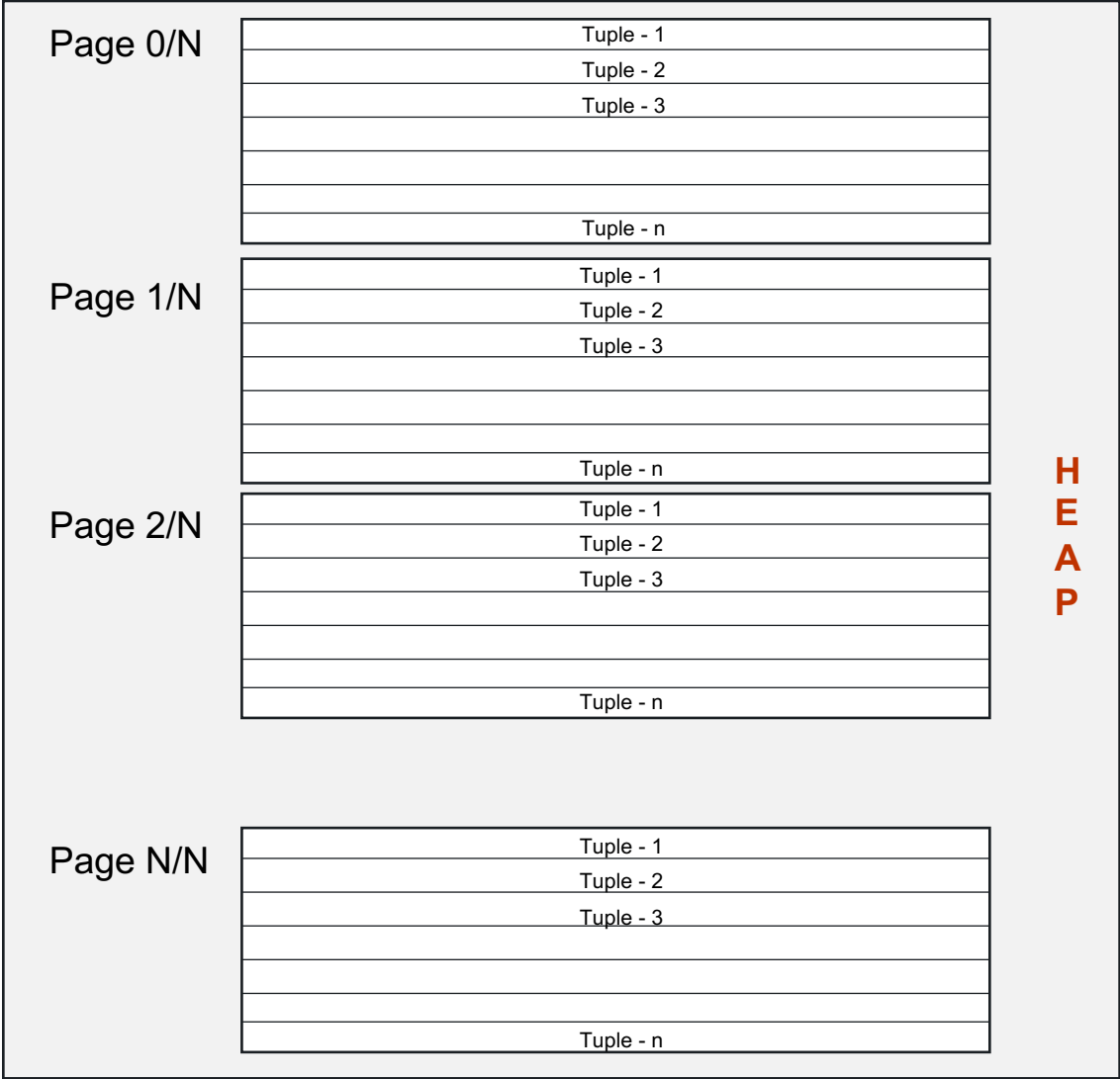
Selecting Data 2/2

```
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
(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;
               Cost of the query  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  64313/159235 * 100 = 40%
```

```
-----
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);
```

PostgreSQL's Catalog for relations/index

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

```
-----  
16425
```

Physical file name of the index

- 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

"bar" is a table and "id" is column

```
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)
```

Expression Index 1/2

```
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)
```

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..61568.60 rows=16667 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 < 10000;

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)
        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.

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

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 >

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 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)

B-Tree Index 2/2

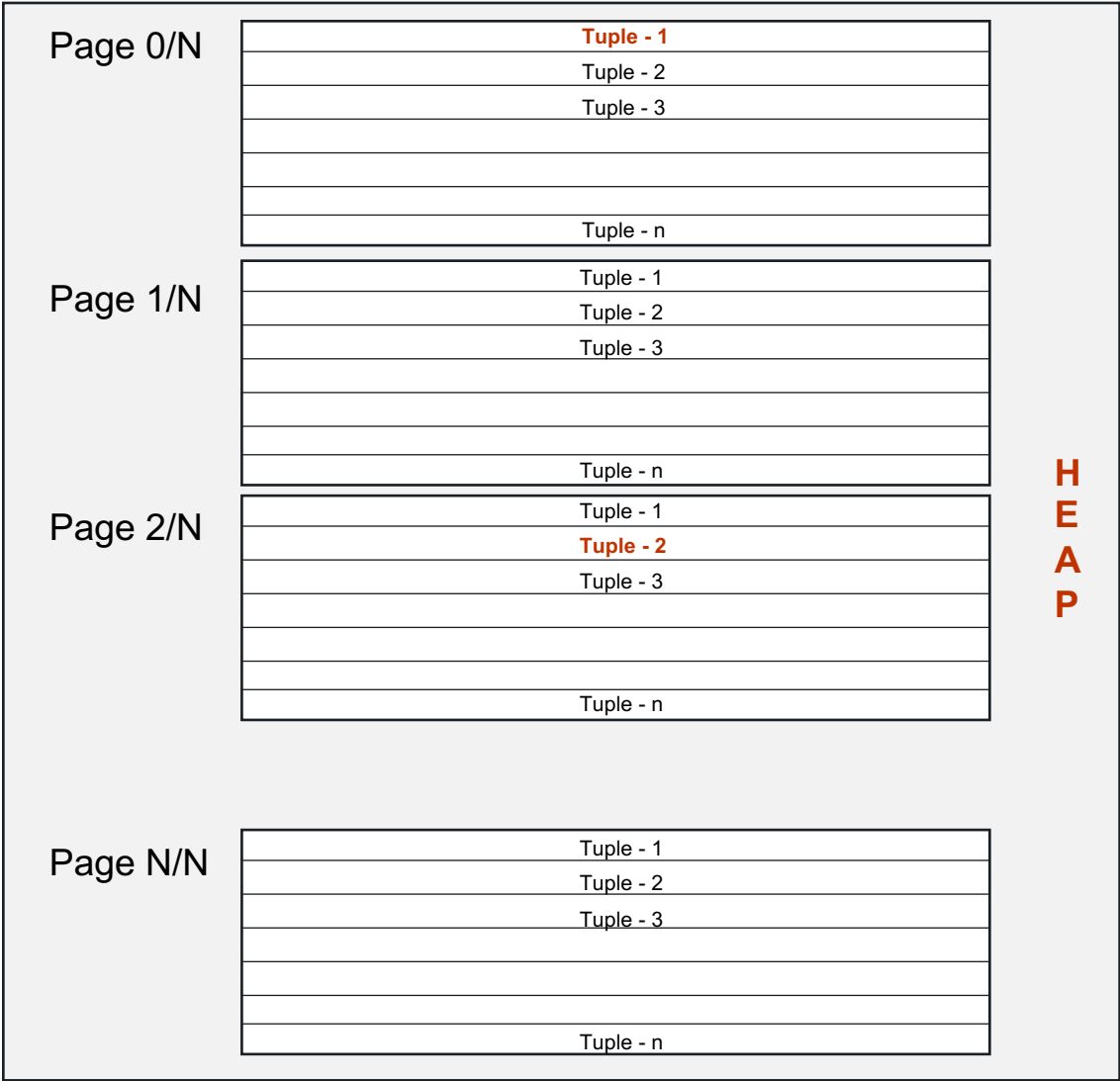
```
CREATE TABLE foo(id INTEGER, name TEXT);
INSERT INTO foo VALUES (1, 'Alex');
INSERT INTO foo VALUES (2, 'Bob');
```

```
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



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 "public.bar"				
Column	Type	Collation	Nullable	Default
id	integer			
name	character varying			
dt	date			

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% '::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 *Like date, date of city have logical group*
- Space optimized because BRIN index contains only three items
 - Page number
 - Min value of column
 - Max value of column

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

Size of Index = 21 MB

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

Size of Index = 47 MB

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

Size of Index = 48KB

BRIN Index 2/2

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)
```

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

```
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": ["333-33333", "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 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

```
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":
  Rows Removed by Filter: 30
Planning Time: 0.107 ms
Execution Time: 1079.861 ms
```

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

```
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
loops=1)
  Recheck Cond: (name @> '{"name": "Alex"}'::jsonb)
  Index Cond: (name @> '{"name":
  Planning Time: 0.168 ms
Execution Time: 475.447 ms
```

GiST Index

- **Generalized Search Tree**
- **Tree-structured access method**
- **Used to find the point within box**
- **Used for full text search**



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 In PostgreSQL term (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;
```

QUERY PLAN

```
Index Scan using idx_btree_ios on bar (cost=0.56..99.20 rows=25 width=19)
```

```
Index Cond: ((id > 100000) AND (id < 100010))
```

```
(2 rows)
```

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

QUERY PLAN

```
Index Only Scan using idx_btree_ios on bar (cost=0.56..99.20 rows=25 width=15)
```

```
Index Cond: ((id > 100000) AND (id < 100010))
```

```
(2 rows)
```

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)

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)

A large white question mark is positioned in the upper center of the image. Below it, on the left, is a small, light-colored elephant figurine. To the right of the elephant is a lamp with a black, ribbed shade and a white base. The background is dark and textured, resembling a stone wall. The overall scene is dimly lit, with the question mark and the elephant being the primary light sources.

?

“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**