Flexible Indexing with Postgres

BRUCE MOMJIAN

Postgres offers a wide variety of indexing structures, and many index lookup methods with specialized capabilities.This talk explores the many Postgres indexing options. *Includes concepts from Teodor Sigaev, Alexander Korotkov, Oleg Bartunov, Jonathan Katz*

https://momjian.us/presentations

Last updated: January, 2021

Outline

1. Traditional indexing
2. Expression indexes
3. Partial indexes
4. Benefits of bitmap index scans
5. Non-B-tree index types
6. Data type support for index types
7. Index usage summary

1. Traditional Indexing

https://www.flickr.com/photos/ogimogi/

B-Tree

- Ideal for looking up unique values and maintaining unique indexes
- High concurrency implementation
- Index is key/row-pointer, key/row-pointer
- Supply ordered data for queries
  - ORDER BY clauses (and LIMIT)
  - Merge joins
  - Nested loop with index scans

But I Want More!

- Index expressions/functions
- Row control
- Small, light-weight indexes
- Index non-linear data
- Closest-match searches
- Index data with many duplicates
- Index multi-valued fields

2. Expression Indexes

```
SELECT * FROM customer WHERE lower(name) = 'andy';

CREATE INDEX i_customer_name ON customer (name); ×

CREATE INDEX i_customer_lower ON customer (lower(name));
```

Let’s Test It

```
CREATE TABLE customer (name) AS
SELECT 'cust' || i
FROM generate_series(1, 1000) AS g(i);

CREATE INDEX i_customer_name ON customer (name);

EXPLAIN SELECT * FROM customer WHERE name = 'cust999';
                                     QUERY PLAN
------------------------------------------------------
 Index Only Scan using i_customer_name on customer ...
   Index Cond: (name = 'cust999'::text)

EXPLAIN SELECT * FROM customer WHERE lower(name) = 'cust999';
                       QUERY PLAN
---------------------------------------------------------
 Seq Scan on customer  (cost=0.00..20.00 rows=5 width=7)
   Filter: (lower(name) = 'cust999'::text)
```

Create an Expression Index

```
CREATE INDEX i_customer_lower ON customer (lower(name));

EXPLAIN SELECT * FROM customer WHERE lower(name) = 'cust999';
                                  QUERY PLAN
----------------------------------------------------------------
 Bitmap Heap Scan on customer  (cost=4.32..9.66 rows=5 width=7)
   Recheck Cond: (lower(name) = 'cust999'::text)
   ->  Bitmap Index Scan on i_customer_lower ...
         Index Cond: (lower(name) = 'cust999'::text)
```

Other Expression Index Options

- User-defined functions
- Concatenation of columns
- Math expressions
- Only IMMUTABLE functions can be used
- Consider casting when matching WHERE clause expressions to the indexed expression

3. Partial Indexes: Index Row Control

- Why index every row if you are only going to look up some of them?
- Smaller index on disk and in memory
- More shallow index
- Less INSERT/UPDATE index overhead
- Sequential scan still possible

Partial Index Creation

```
ALTER TABLE customer ADD COLUMN state CHAR(2);

UPDATE customer SET state = 'AZ'
WHERE name LIKE 'cust9__';

CREATE INDEX i_customer_state_az ON customer (state) WHERE state = 'AZ';
```

Test the Partial Index

```
EXPLAIN SELECT * FROM customer WHERE state = 'PA';
QUERY PLAN
-----------------------------------------------------------
Seq Scan on customer (cost=0.00..17.50 rows=5 width=19)
Filter: (state = 'PA'::bpchar)

EXPLAIN SELECT * FROM customer WHERE state = 'AZ';
QUERY PLAN
-----------------------------------------------------------------------------
Bitmap Heap Scan on customer (cost=4.18..9.51 rows=5 width=19)
Recheck Cond: (state = 'AZ'::bpchar)
-> Bitmap Index Scan on i_customer_state_az ...
Index Cond: (state = 'AZ'::bpchar)
```

Partial Index With Different Indexed Column

```
DROP INDEX i_customer_name;

CREATE INDEX i_customer_name_az ON customer (name) WHERE state = 'AZ';

EXPLAIN SELECT * FROM customer WHERE name = 'cust975';
                         QUERY PLAN
-----------------------------------------------------------
 Seq Scan on customer  (cost=0.00..17.50 rows=1 width=19)
   Filter: (name = 'cust975'::text)
   Index Cond: (state = 'AZ'::bpchar)
```

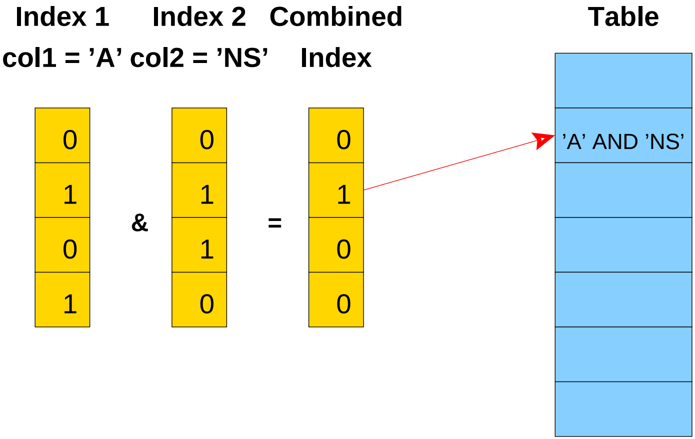
Partial Index With Different Indexed Column

```
EXPLAIN SELECT * FROM customer
WHERE name = 'cust975' AND state = 'AZ';
                                     QUERY PLAN
-----------------------------------------------------
 Index Scan using i_customer_name_az on customer ...
   Index Cond: (name = 'cust975'::text)

EXPLAIN SELECT * FROM customer
WHERE state = 'AZ';
                                   QUERY PLAN
----------------------------------------------------------------
 Bitmap Heap Scan on customer  (cost=4.17..9.50 rows=5 width=19)
   Recheck Cond: (state = 'AZ'::bpchar)
   ->  Bitmap Index Scan on i_customer_name_az ...
```

4. Benefits of Bitmap Index Scans

- Used when:
  - an index lookup might generate multiple hits on the same heap (data) page
  - using multiple indexes for a single query is useful
- Creates a bitmap of matching entries in memory
- Row or block-level granularity
- Bitmap allows heap pages to be visited only once for multiple matches
- Bitmap can merge the results from several indexes with AND/OR filtering
- Automatically enabled by the optimizer

Bitmap Index Scan



5. Non-B-Tree Index Types

https://www.flickr.com/photos/archeon/

Block-Range Index (BRIN)

- Tiny indexes designed for large tables
- Minimum/maximum values stored for a range of blocks (default 1MB, 128 8k pages)
- Allows skipping large sections of the table that cannot contain matching values
- Ideally for naturally-ordered tables, e.g., insert-only tables are chronologically ordered
- Index is 0.003% the size of the heap
- Indexes are inexpensive to update
- Index every column at little cost
- Slower lookups than B-tree

Generalized Inverted Index (GIN)

- Best for indexing values with many keys or values, e.g.,
  - text documents
  - JSON
  - multi-dimensional data, arrays
- Ideal for columns containing many duplicates
- Optimized for multi-row matches
- Key is stored only once
- Index is key/many-row-pointers
- Index updates are batched, though always checked for accuracy
- Compression of row pointer list
- Optimized multi-key filtering

Generalized Search Tree (GIST)

GIST is a general indexing framework designed to allow indexing of complex data types with minimal programming. Supported data types include:

- geometric types
- range types
- hstore (key/value pairs)
- intarray (integer arrays)
- pg_trgm (trigrams)

Supports optional “distance” for nearest-neighbors/closest matches. (GIN is also generalized.)

Space-Partitioned Generalized Search Tree (SP-GiST)

- Similar to GiST in that it is a generalized indexing framework
- Allows the key to be split apart (decomposed)
- Parts are indexed hierarchically into partitions
- Partitions are of different sizes
- Each child needs to store only the child-unique portion of the original value because each entry in the partition shares the same parent value.

Hash Indexes

- Equality, non-equality lookups; no range lookups
- Crash-safe starting in Postgres 10
- Replicated starting in Postgres 10

I Am Not Making This Up

```
SELECT amname, obj_description(oid, 'pg_am')
FROM pg_am ORDER BY 1;

 amname |             obj_description
--------+----------------------------------------
 brin   | block range index (BRIN) access method
 btree  | b-tree index access method
 gin    | GIN index access method
 gist   | GiST index access method
 hash   | hash index access method
 spgist | SP-GiST index access method
```

Index Type Summary

- B-tree is ideal for unique values
- BRIN is ideal for the indexing of many columns
- GIN is ideal for indexes with many duplicates
- SP-GIST is ideal for indexes whose keys have many duplicate prefixes
- GIST for everything else

6. Data Type Support for Index Types

https://www.flickr.com/photos/jonobass/

Finding Supported Data Types - B-Tree

```
SELECT opfname FROM pg_opfamily, pg_am
WHERE opfmethod = pg_am.oid AND amname = 'btree'
ORDER BY 1;
```

```
abstime_ops
array_ops
bit_ops
bool_ops
bpchar_ops
bpchar_pattern_ops
bytea_ops
char_ops
datetime_ops
enum_ops
float_ops
integer_ops
interval_ops
jsonb_ops
macaddr_ops
money_ops
name_ops
network_ops
numeric_ops
oid_ops
oidvector_ops
pg_lsn_ops
range_ops
record_image_ops
record_ops
reltime_ops
text_ops
text_pattern_ops
tid_ops
time_ops
timetz_ops
tinterval_ops
tsquery_ops
tsvector_ops
uuid_ops
varbit_ops
```

These data types are mostly single-value and easily ordered. B-tree support for multi-valued types like tsvector is only for complete-field equality comparisons.

Finding Supported Data Types - BRIN

```
SELECT opfname FROM pg_opfamily, pg_am
WHERE opfmethod = pg_am.oid AND amname = 'brin'
ORDER BY 1;
```

```
abstime_minmax_ops       interval_minmax_ops       reltime_minmax_ops
bit_minmax_ops           macaddr_minmax_ops        text_minmax_ops
box_inclusion_ops        name_minmax_ops           tid_minmax_ops
bpchar_minmax_ops        network_inclusion_ops     time_minmax_ops
bytea_minmax_ops         network_minmax_ops        timetz_minmax_ops
char_minmax_ops          numeric_minmax_ops        uuid_minmax_ops
datetime_minmax_ops      oid_minmax_ops            varbit_minmax_ops
float_minmax_ops         pg_lsn_minmax_ops
integer_minmax_ops       range_inclusion_ops
```

Finding Supported Data Types - GIN

```
SELECT opfname FROM pg_opfamily, pg_am
WHERE opfmethod = pg_am.oid AND amname = 'gin'
ORDER BY 1;

    opfname
----------------
 array_ops
 jsonb_ops
 jsonb_path_ops
 tsvector_ops
```

These date types are multi-value, where each value is independent.

Finding Supported Data Types - GIST

```
SELECT opfname FROM pg_opfamily, pg_am
WHERE opfmethod = pg_am.oid AND amname = 'gist'
ORDER BY 1;

   opfname
--------------
 box_ops
 circle_ops
 jsonb_ops
 network_ops
 point_ops
 poly_ops
 range_ops
 tsquery_ops
 tsvector_ops
```

These date types are multi-value — some have independent values (JSON, tsvector), others have dependent values (point, box).

Finding Supported Data Types - SP-GiST

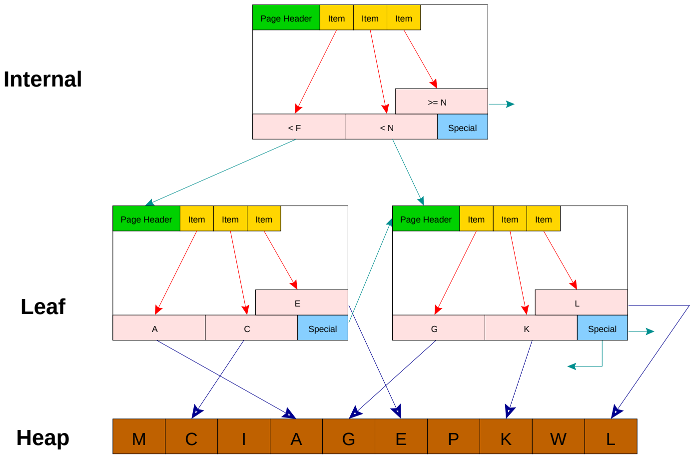
```
SELECT opfname FROM pg_opfamily, pg_am
WHERE opfmethod = pg_am.oid AND amname = 'spgist'
ORDER BY 1;

    opfname
----------------
 kd_point_ops
 quad_point_ops
 range_ops
 text_ops
```

For text, this is useful when the keys are long.

Index Type Examples

https://www.flickr.com/photos/samcatchesides/

B-Tree


Internal
Page Header
Item
Item
Item
>= N
< F
< N
Special
Leaf
Page Header
Item
Item
Item
E
A
C
Special
Page Header
Item
Item
Item
L
G
K
Special
Heap
M
C
I
A
G
E
P
K
W
L

BRIN Example

```
CREATE TABLE brin_example AS
SELECT generate_series(1,100000000) AS id;

CREATE INDEX btree_index ON brin_example(id);
CREATE INDEX brin_index ON brin_example USING brin(id);

SELECT relname, pg_size_pretty(pg_relation_size(oid))
FROM pg_class
WHERE relname LIKE 'brin_%' OR relname = 'btree_index'
ORDER BY relname;
   relname    | pg_size_pretty
--------------+----------------
 brin_example | 3457 MB
 btree_index  | 2142 MB
 brin_index   | 104 kB
```

GIN Example Using tsvector_ops

```
CREATE TABLE articles (doc TSVECTOR);

INSERT INTO articles VALUES ('The fox is sick');

INSERT INTO articles VALUES ('How sick is this');

SELECT ctid, * FROM articles ORDER BY 1;
 ctid  |              doc
-------+---------------------------
 (0,1) | 'The' 'fox' 'is' 'sick'
 (0,2) | 'How' 'is' 'sick' 'this'
```

GIN Example Using tsvector_ops

```
SELECT ctid, * FROM articles ORDER BY 1;
 ctid  |              doc
-------+---------------------------
 (0,1) | 'The' 'fox' 'is' 'sick'
 (0,2) | 'How' 'is' 'sick' 'this'

fox     (0,1)
is      (0,1), (0,2)
sick    (0,1), (0,2)
this    (0,2)
How     (0,2)
The     (0,1)
```

Integer arrays are indexed similarly.

GIN Example Using JSON

```
CREATE TABLE webapp (doc JSONB);

INSERT INTO webapp VALUES
('{"name": "Bill", "active": true}');

INSERT INTO webapp VALUES
('{"name": "Jack", "active": true}');

SELECT ctid, * FROM webapp ORDER BY 1;
 ctid  |                doc
-------+----------------------------------
 (0,1) | {"name": "Bill", "active": true}
 (0,2) | {"name": "Jack", "active": true}
```

GIN Example Using jsonb_ops (default)

```
(0,1) | {"name": "Bill", "active": true}
(0,2) | {"name": "Jack", "active": true}

CREATE INDEX i_webapp_yc ON webapp
USING gin (doc /* jsonb_ops */);

active (0,1), (0,2)
name   (0,1), (0,2)
true   (0,1), (0,2)
Bill   (0,1)
Jack   (0,2)
```

GIN Example Using jsonb_path_ops

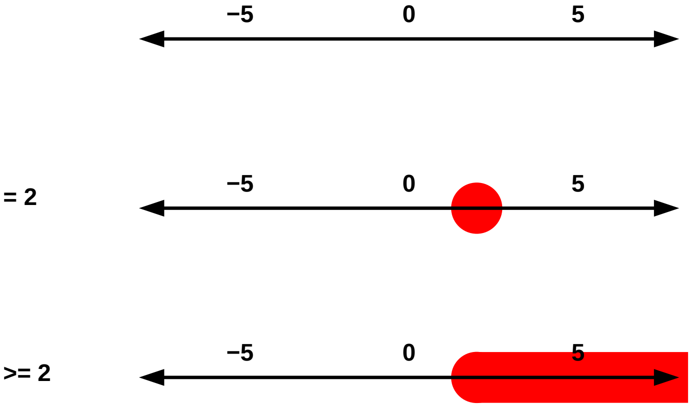
```
(0,1) | {"name": "Bill", "active": true}
(0,2) | {"name": "Jack", "active": true}

CREATE INDEX i_webapp_doc_path ON webapp
USING gin (doc jsonb_path_ops);

hash(active, true) (0,1), (0,2)
hash(name, Bill)   (0,1)
hash(name, Jack)   (0,2)
```

Nested keys have their parent keys (paths) prepended before hashing.

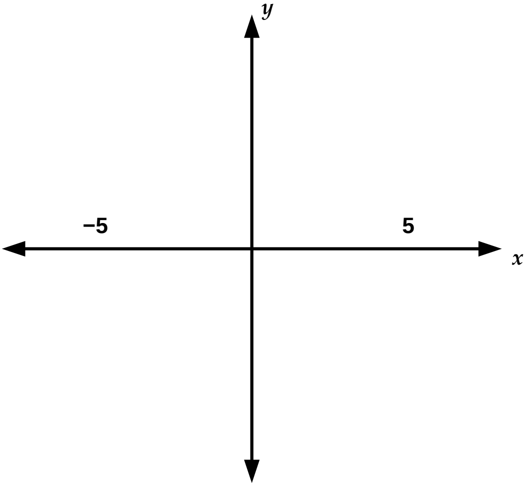
GiST

- Supports data types with loosely-coupled values, like tsvector, JSONB
- Uniquely supports data types with tightly-coupled values
  - multi-dimensional types (geographic)
  - range types
  - IP network data type

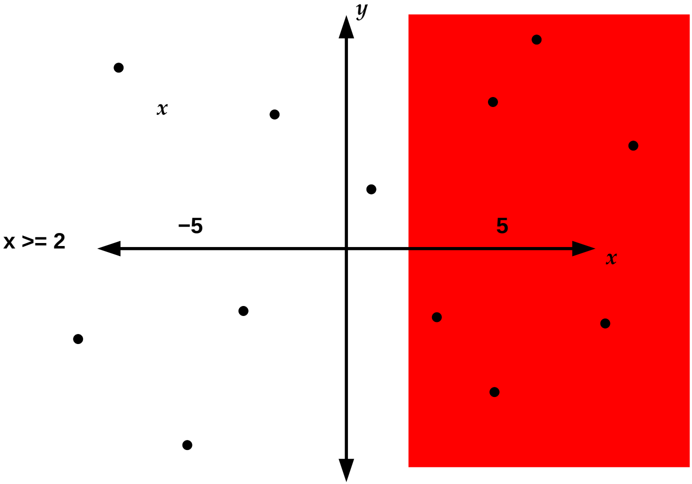
Linear Indexing



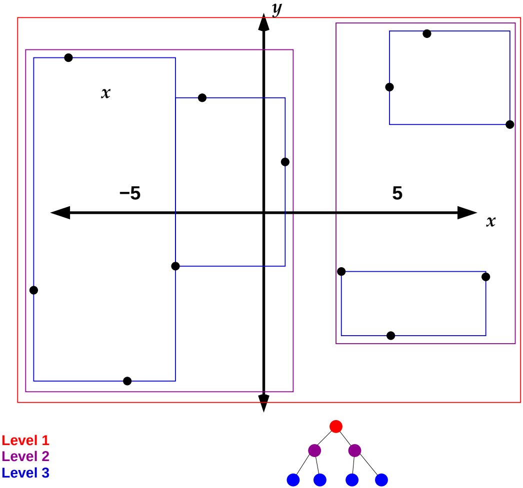
Multi-Dimensional



Linear Methods Are Inefficient



R-Tree Indexes Bounding Boxes



Geographic objects (lines, polygons) also can appear in R-tree indexes. based on their own bounding boxes.

GiST Two-Dimensional Ops

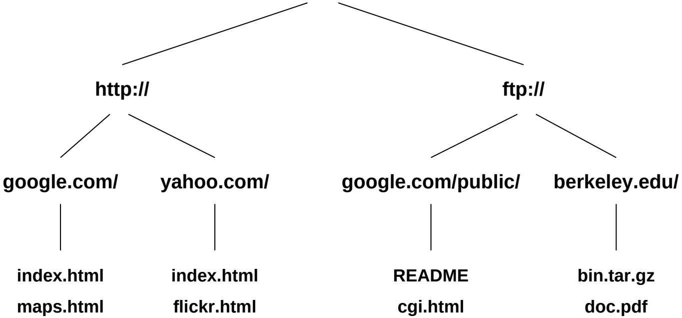
```
box_ops
circle_ops
point_ops
poly_ops
```

PostGIS also uses this indexing method.

Range Indexing With GiST

GiST range type indexing uses large ranges at the top level of the index, with ranges decreasing in size at lower levels, just like how R-tree bounding boxes are indexed.

SP-GiST TEXT_OPS Example (Suffix Tree)



Internally split by character. B-trees use range partitioning, e.g., A–C, rather than common prefix partitioning, so a B-tree key must store the full key value.

Other SP-GiST Index Examples

- quad_point_ops uses four corner points in square partitions of decreasing size
- kd_point_ops splits on only one dimension

Extension Index Support

- btree_gin (GIN)
- btree_gist (GiST)
- cube (GiST)
- hstore (GiST, GIN)
- intarray (GiST, GIN)
- ltree (GiST)
- pg_trgm (GiST, GIN)
- PostGIS
- seg

7. Index Usage Summary

https://www.flickr.com/photos/jubilo/

When To Create Indexes

- pg_stat_user_tables.seq_scan is high
- Check frequently-executed queries with EXPLAIN (find via pg_stat_statements or pgbadger)
- Squential scans are not always bad
- If pg_stat_user_indexes.idx_scan is low, the index might be unnecessary
- Unnecessary indexes use storage space and slow down INSERTs and some UPDATEs

Evaluating Index Types

- Index build time
- Index storage size
- INSERT/UPDATE overhead
- Access speed
- Operator lookup flexibility

Conclusion

https://momjian.us/presentations

https://www.flickr.com/photos/philipp_zurmoehle/