PostgreSQL Different Indexes

Fanavaran Anisa
Iran Linux House

Linux & Open Source Training Center

www.anisa.co.ir

Index ?

At its core, an index is a data structure that enhances the speed of data retrieval operations on a database table.

Indexes can have a **significant impact on memory usage, CPU utilization, and overall query performance.**

However, not all queries utilize indexes, and it’s crucial to understand when and how to use them effectively.

Types of Indices

- B-tree index
- GIN (Generalized Inverted Index)
- HASH indexes
- BRIN(Block Range INdex) indexes
- GiST/SP-GiST (Generalized Search Tree/Space-Partioned Generalized Search Tree
- Bloom (Extension)

```
CREATE INDEX (name_of_index) on (name_of_table) USING (index_type)
(name_of_column);
```

B-tree index

- The B-tree index is the default index type in PostgreSQL
- It's a balanced tree structure that supports filtering, sorting, and grouping operations efficiently. B-tree indexes are particularly useful for **equality** and **range queries**.
- PostgreSQL B-tree index query planner considers using b-tree index if it exist when the query involves

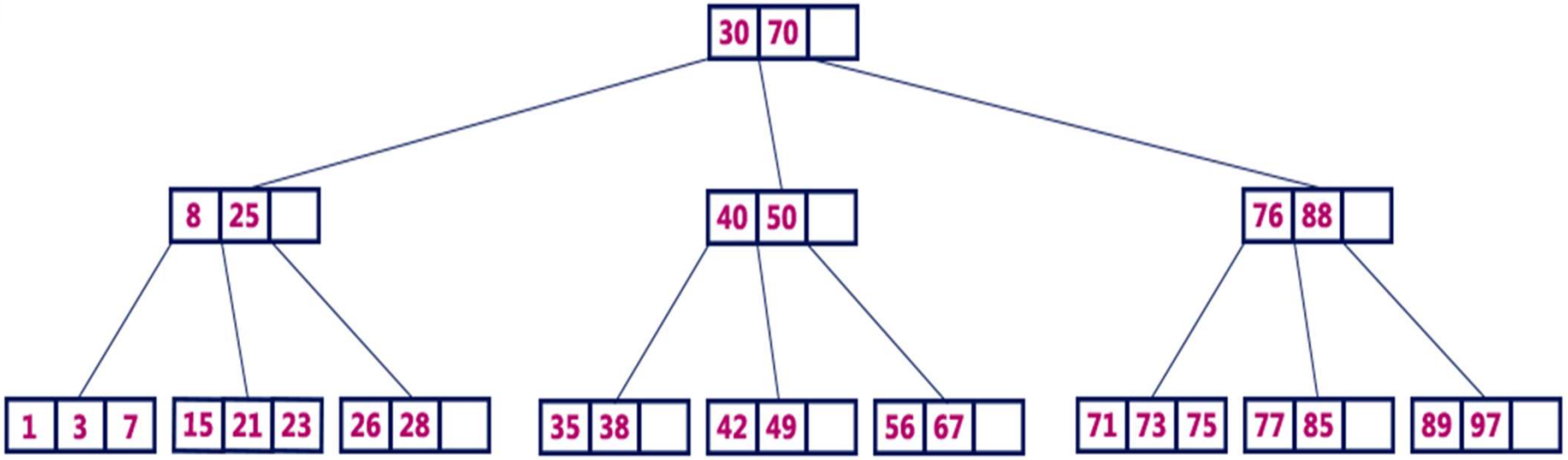
<=, =, <, >=
IN, Between
IS NOT NULL, IS NULL

```
CREATE INDEX btree_idx on test_idx USING BTREE (id);
```

The optimizer can also use a B-tree index for queries involving the pattern matching operators `LIKE` and `~` *if* the pattern is a constant and is anchored to the beginning of the string — for example, `col LIKE 'foo%'` or `col ~ '^foo'`, but not `col LIKE '%bar'`

B-tree index

- B-trees attempt to remain balanced, with the amount of data in each branch of the tree being roughly the same



B-tree index & Sorting

B-Tree index entries are sorted in ascending order by default. In some cases, it makes sense to supply a different sort order for an index.
Take the case when you're showing a paginated list of articles, sorted by most recent published first. We can have a published_at column on our articles table. For unpublished articles, the published_at value is NULL.

In this case, we can create an index like so:

```
CREATE INDEX articles_published_at_index
ON articles(published_at DESC NULLS LAST);
```

Hash index

- Hash index in PostgreSQL will handle only simple equality comparison, i.e. (=).
- Hash indexes store a 32-bit hash code derived from the value of the indexed column.
- he query planner will consider using a hash index whenever an indexed column is involved in a comparison using the equal operator:=
- it is suitable for Categorical attributes(city/country/…)

```
CREATE INDEX hash_idx on test_idx USING HASH (stud_id);
\d+ test_idx;
```

GiST

- GiST index is also known as the **Generalized Search Tree** index. The PostgreSQL GIST index will make it possible to construct the overall tree structure.

- when the data to be indexed is more complex than to do a simple equate or ranged comparison like finding nearest-neighbor and pattern matching.

- GiST index is useful for geometric data type and full complex search in PostgreSQL.
- GiST index consists of multiple node values. The node of the GiST index will be organized in a tree-structured way.

```
CREATE INDEX gist_idx_test ON GIST_IDX USING gist(circle_dim);
 \d+ GIST_IDX;
```

GiST

- GiST indexes are not a single kind of index, but rather an infrastructure within which many different indexing strategies can be implemented
- GiST index is also known as the **Generalized Search Tree** index. The PostgreSQL GIST index will make it possible to construct the overall tree structure.
- when the data to be indexed is more complex than to do a simple equate or ranged comparison like finding nearest-neighbor and pattern matching.
- GiST index is useful for geometric data type and full complex search in PostgreSQL.

GiST

- The particular operators with which a GiST index can be used vary depending on the indexing strategy (the operator class).
- As an example, the standard distribution of PostgreSQL includes GiST operator classes for several two-dimensional geometric data types, which support indexed queries using these operators:

```
<<  &<  &>  >>  <<|  &<|  |&>  |>>  @>  <@  ~=  &&
```

```
CREATE INDEX gist_idx_test ON GIST_IDX USING gist(circle_dim);
 \d+ GIST_IDX;
```

- GiST indexes are also capable of optimizing “nearest-neighbor” searches, such as

```
SELECT * FROM places ORDER BY location <-> point
'(101,456)' LIMIT 10;
```

SP-GiST

- SP-Gist or **Space partitioned Gist** indexes are useful when the data can be grouped into non-overlapping groupings.
- Like Gist index, it also provides an infrastructure for implementing different indexing strategies.
- SP-GiST permits implementation of a wide range of different non-balanced disk-based data structures, such as quadtrees, k-d trees, and radix trees (tries).
- As an example, the standard distribution of PostgreSQL includes SP-GiST operator classes for two-dimensional points, which support indexed queries using these operators:

<< >> ~= <@ <<| |>>

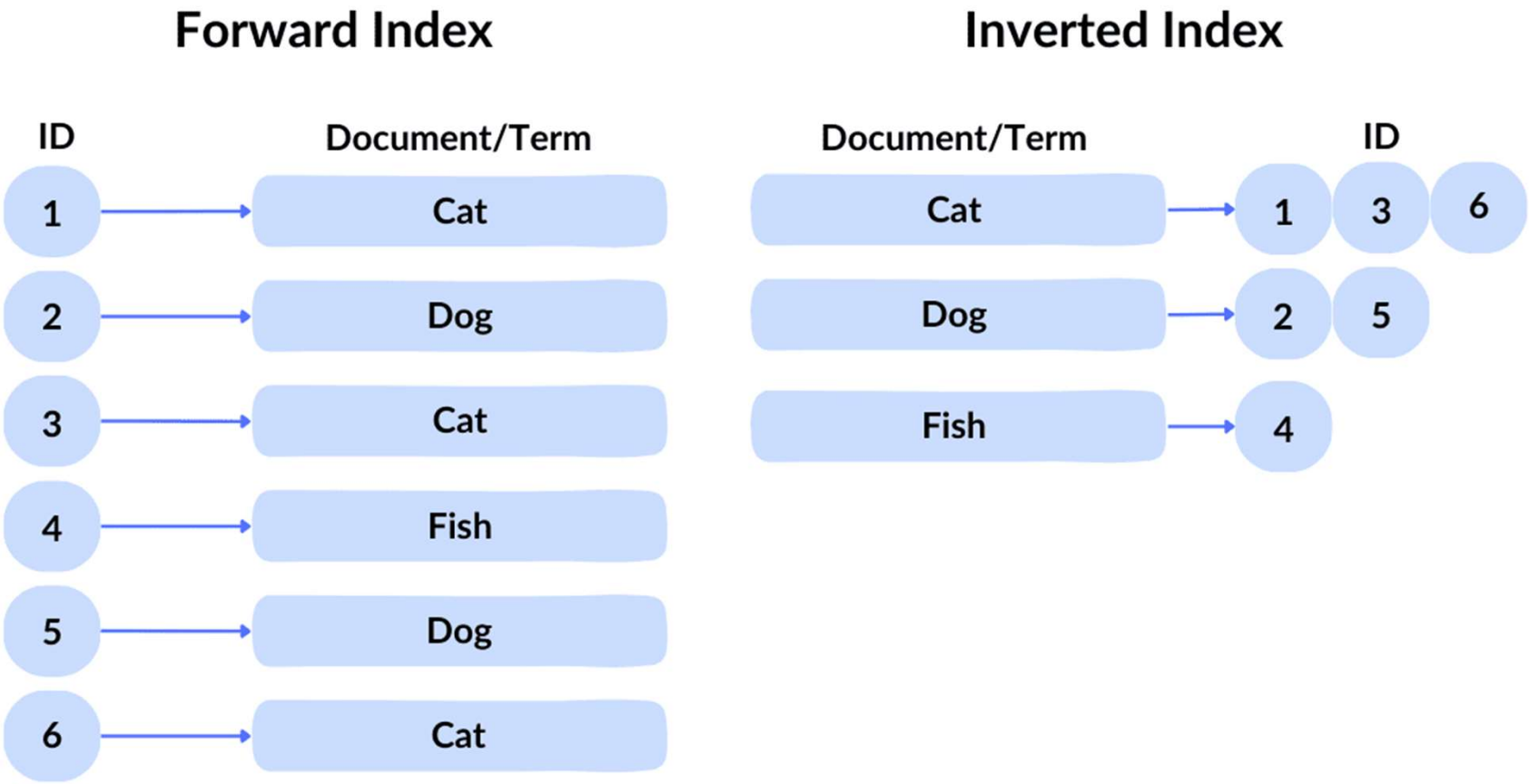
GIN

GIN indexes are “inverted indexes” **which are appropriate for data values that contain multiple component values, such as arrays.**

- An inverted index contains a separate entry for each component value, and can efficiently handle queries that test for the presence of specific component values.

- Like GiST and SP-GiST, GIN can support many different user-defined indexing strategies, and the particular operators with which a GIN index can be used vary depending on the indexing strategy.
- As an example, the standard distribution of PostgreSQL includes a GIN operator class for arrays, which supports indexed queries using these operators: **<@ @> = &&**

GIN



BRIN

- **Block Range Indexes** are useful for large size tables that have columns with some natural sort order. The BRIN index divides the table into block ranges and keeps a summary of those blocks. This summary includes min, max values of the range
- BRIN index is also called the **block range indexes**. It is smaller and less costly to maintain the comparison with the Btree index.
- Using a BRIN index **on a large table is a more practical approach** than using a Btree index without horizontal partitioning.

```
CREATE INDEX brin_idx ON test_idx USING BRIN(phone);
 \d+ test_idx;
```

Multicolumn Indexes

- PostgreSQL does allow creation of an index on multiple columns. However you can only have a maximum of 32 columns and such indexes only work with Btree, Gist, Gin and Brin.

```
CREATE TABLE test (x int, y int);
CREATE INDEX multi_idx ON test (x, y);
postgres=# EXPLAIN (costs off) SELECT * from test WHERE x = 10 AND y = 100;
QUERY PLAN
---------------------------------------
Index Only Scan using multi_idx on test
Index Cond: ((x = 10) AND (y = 100))
(2 rows)
```

Unique Indexes

A unique index enforces the uniqueness of the values in the column. In PostgreSQL a unique index can be created on one or multiple columns

```
CREATE TABLE test (x int, y int);
CREATE UNIQUE INDEX unique_idx ON test (x);
postgres=# \d test
Table "public.test"
Column | Type | Collation | Nullable | Default
--------+---------+-----------+----------+---------
x | integer | | |
y | integer | | |
Indexes:
"unique_idx" UNIQUE, btree (x)
```

NULL values are not considered equal, hence they are considered unique values. Adding multiple NULLs in a unique column won't result in index violation.

Indexes on Expressions

It is also possible in PostgreSQL database to create the indexed columns based on the result of a function or scalar expression computed from one or more columns of the table. Since the result of computed expression is stored on the index and it won't be computed at run time, the access to table data becomes much faster.

```
CREATE TABLE test (x int, y text);
postgres=# EXPLAIN (costs off) SELECT * from test
WHERE lower(y) = 'value';
QUERY PLAN
-----------------------------------
Seq Scan on test
Filter: (lower(y) = 'value'::text)
(2 rows)
```

Indexes on Expressions

```
postgres=# CREATE INDEX expr_idx ON test (lower(y));
CREATE INDEX
postgres=# EXPLAIN (costs off) SELECT * from test WHERE lower(y)
= 'value';
QUERY PLAN
---------------------------------------------
Bitmap Heap Scan on test
Recheck Cond: (lower(y) = 'value'::text)
-> Bitmap Index Scan on expr_idx
Index Cond: (lower(y) = 'value'::text)
(4 rows)
```

Partial Indexes

There are some situations where you don't want to index the whole table, **instead you will want to filter out some specific data based on some conditions**. This kind of index is called a partial index. The partial index only contains the data for those rows that fulfill that specific condition.

These indexes **contain a WHERE clause and result in much smaller index sizes** than the usual indexes (without conditions). Since they have a smaller footprint, they result in faster data access

```
CREATE TABLE test (x int, y text);
postgres=# EXPLAIN (costs off) SELECT * from test WHERE y IS NULL;
QUERY PLAN
----------------------
Seq Scan on test
Filter: (y IS NULL)
(2 rows)
```

Partial Indexes

```
postgres=# CREATE INDEX partial_idx ON test (y) WHERE y IS NULL;
CREATE INDEX
postgres=# EXPLAIN (costs off) SELECT * from test WHERE y IS NULL;
QUERY PLAN
-------------------------------------
Bitmap Heap Scan on test
Recheck Cond: (y IS NULL)
-> Bitmap Index Scan on partial_idx
```

Index-Only Scans

- Index only scans are the index where all the needed data by the query is **available directly from the index**.
- Normally when an index is created, it is stored separately from the table data and whenever a query is executed to fetch the data rows it is read from the index and table files.
- However, if the query only consists of indexed columns then there is no need to fetch the data from relation files, it can be directly returned from the index. Which can improve the query performance

```
CREATE TABLE test (x int, y int);
CREATE INDEX col_idx ON test (x);
postgres=# EXPLAIN (costs off) select x from test where x > 10;
QUERY PLAN
------------------------------------
Index Only Scan using col_idx on test
Index Cond: (x > 10)
```

Covering Indexes – A powerful feature

- The covering index is similar to Index-Only scans with a little difference. -
- What if you only want to create an index on a specific column but still want to be able to answer the query from the index without going to the main table. That's what the covering scans are.

- PostgreSQL has the option to allow this by using the 'INCLUDE' clause while creating the index.

Covering Indexes – A powerful feature

```
CREATE TABLE test (x int, y int);
CREATE INDEX col_idx ON test (x) include (y);
postgres=# \d test
Table "public.test"
Column | Type | Collation | Nullable | Default
--------+---------+-----------+----------+---------
x | integer | | |
y | integer | | |
Indexes:
"col_idx" btree (x) INCLUDE (y)
postgres=# EXPLAIN (costs off) select x, y from test where x > 10 ;
QUERY PLAN
-----------------------------------
Index Only Scan using col_idx on test
Index Cond: (x > 10)
(2 rows)
```

Monitoring and Analysis

• Regularly monitor and analyze the performance of your indexes using tools like **pg_stat_statements** and **pg_stat_user_indexes**.
• Identify slow queries and optimize them by considering appropriate indexes or making schema adjustments
•Reindex regularly after **Vacuuming** :

- Example: Rebuilding an index

VACUUM [full] table_name;
REINDEX INDEX index_name;

ALTER TABLE table_name SET autovacuum_vacuum_scale_factor = 0.2

https://use-the-index-luke.com/