Lecture 8. Indexes

What is an Index?

> An index is a data structure that allows the database server to **find and retrieve specific rows much faster** than scanning the whole table .
> Used to **enhance performance**, especially for queries with search conditions.

But indexes also add overhead:

- They take extra **disk space**
- They must be **maintained** on every `INSERT`, `UPDATE`, `DELETE`

Creating and dropping an index

```
CREATE INDEX test1_id_index
ON test1(id);
```

```
DROP INDEX test1_id_index;
```

How PostgreSQL uses indexes

- Once an index is created:
  - PostgreSQL updates it automatically whenever the table changes
  - The query planner will choose the index when it estimate it's cheaper than a sequential scan
- To help the planner, you may need to run

But you might have to run the ANALYZE command to update statistic:

```
ANALYZE table_name;
```

Indexes and UPDATE/DELETE/JOIN

Indexes can speed up:

- `UPDATE` and `DELETE` with search conditions (less data to touch).
- `JOIN` queries, when join conditions involve indexed columns

Cost of indexes

- After an index is created, PostgreSQL must keep it **in sync** with the table.
- This adds **overhead** to data manipulations.
- Indexes that are **rarely or never used** should be **removed**.

Index Types in PostgreSQL

PostgreSQL supports several index types:

- **B-tree (default)**
- **Hash**
- **GiST**
- **SP-GiST**
- **GIN**
- **BRIN**

Each type has a different algorithm and is suited to different query patterns. By default, `CREATE INDEX` uses **B-tree**, which covers most normal use cases.

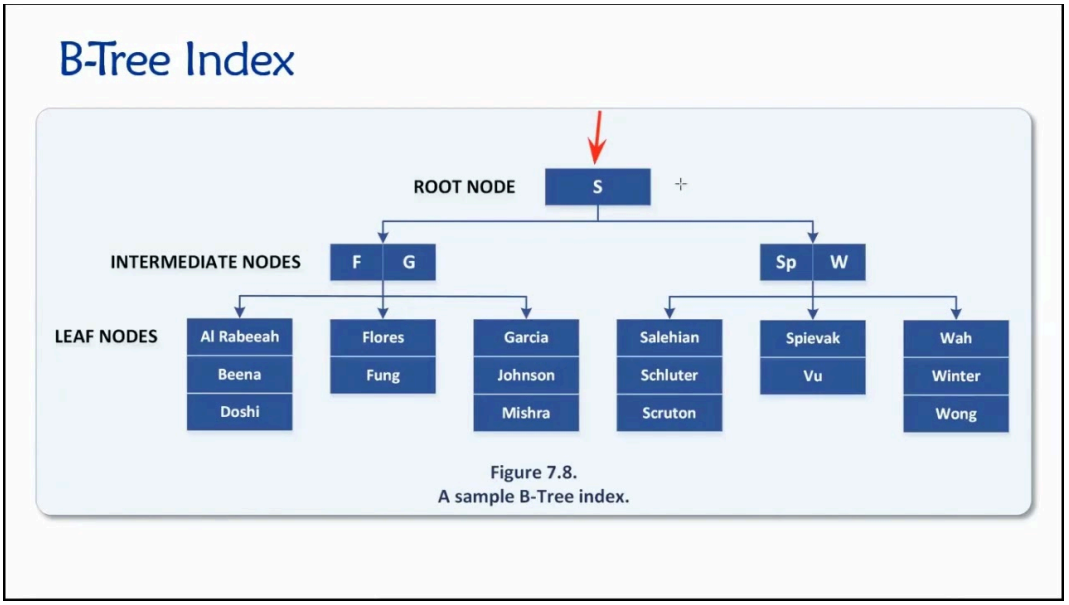
B-tree index

> **Default** index type. Supports **equality and range** queries on data that has a defined ordering.

PostgreSQL will consider using a B-tree index when an indexed column is compared with:

`<`, `<=`, `=`, `>=`, `>`

B-tree indexes can be used to **retrieve rows in sorted order** (useful with `ORDER BY`)

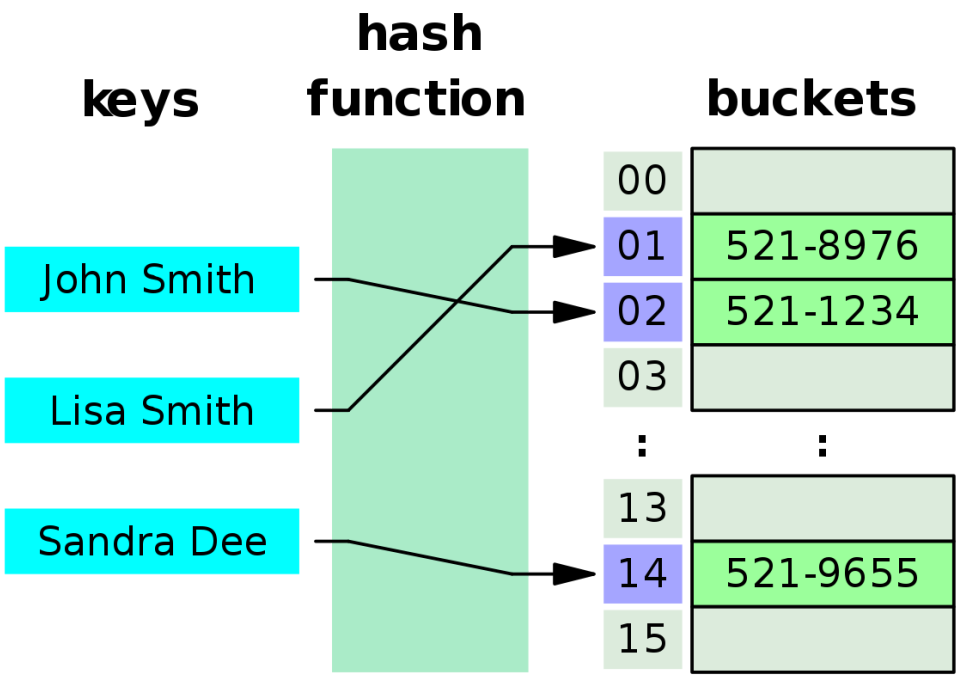


Hash index

- Can **only handle equality comparisons**
- Planner considers a hash index when the condition uses the `=` operator on the indexed column.

Create example:

```
CREATE INDEX name
ON table
USING HASH (column);
```



GiST index

> GiST indexes are not a single kind of index, but rather an **infrastructure** within can be implemented

Typical operators include:

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

Other index types

- **SP-GiST** - space-partitioned; good for partitioned search spaces (e.g. tries, quadtrees).
- **GIN** – Generalized Inverted Index; good for arrays, full-text search, many keys per row.
- **BRIN** – Block Range Index; good for very large tables where values are **physically ordered** (e.g. time-series).

Multicolumn Indexes

> An index can br defined on **more than one column** of a table

Example structure:

```
CREATE TABLE test2 (
    major int,
    minor int,
    name varchar
);
```

Frequent query:

```
SELECT *
FROM test2
WHERE major = constant AND minor = constant;
```

Then it is appropriate to define a multicolumn index:

```
CREATE INDEX test2_mm_idx
ON test2(major, minor);
```

PostgreSQL allows up to **32 columns** in one index definition (this limit can be changed at build time)

Indexes and `ORDER BY`

Indexes can sometimes **avoid a seperate sort step**

- **Only B-tree** indexes can produce **sorted output** directly.
- By default, B-tree stores entries in:
- **Ascending order**, with **NULLS LAST**

You can customize ordering when creating the index:

```
CREATE INDEX test2_mm_idx
ON test2 (major ASC, minor DESC NULLS FIRST);
```

Planner may use such an index to satisfy quries like:

```
SELECT *
FROM test2
ORDER BY major ASC, minor DESC NULLS FIRST
```

Unique Indexes

- Indexes can enforce **uniqueness** of:
  - a single colmn, or
  - a **combination** of columns
- Currently, only **B-tree** indexes can be declared **UNIQUE**

Behavior

- When an index is **declared UNIQUE:**
  - Multiple rows with the same indexed values are **not allowed**.
- `NULL` values are **not considered equal**:
  - A unique index *allows multiple NULLs*
- For a multicolumn unique index:
  - It rejects rows only when **all indexed columns** are equal to an existing row

Relations to consraint

PostgreSQL automtically creates a **unique index** when you define:

- a **PRIMARY KEY** constrint, or
- a **UNIQUE** constraint on a table

The index is the mechanism that actually **enforces the constraint**.

Indexes on Expressions

An index column **does not have to be a simple table column**:

- It can be a **function** or scalar **expression** of one or more columns

Use-case: you want fast access based on **computed values**.

Case-intensitive search example

Common pattern:

```
SELECT *
FROM mytable
WHERE lower(col1) = 'abc';
```

If you create a normal index on `col1`, PostgreSQL **may not** be able to use it efficiently for `lower(col1)`.

Instead, define an index on the **expression**:

```
CREATE INDEX mytable_lower_col1_idx
ON mytable (lower(col1));
```

Now the query:

```
SELECT *
FROM mytable
WHERE lower(col1) = 'abc';
```

can use this **expression index** to search quickly.