# Indexes

## Definition:

An index in PostgreSQL is a data structure that improves the speed of data retrieval operations on a database table at the cost of additional space and maintenance overhead. Indexes are built on one or more columns in a table and enable the database to quickly locate rows without scanning the entire table.

## Purpose of Indexes:

- **Improved Query Performance**: Indexes allow the database to avoid full table scans, significantly reducing query execution time.
- **Efficient Search**: When searching or filtering by specific columns, indexes speed up the search process, especially on large tables.
- **Sorting**: Indexes can be used to optimize sorting operations, particularly when combined with ORDER BY.
- **Uniqueness Enforcement**: Unique indexes ensure that no two rows have the same value for a specified column or set of columns.
- **Faster Joins**: Indexes on join keys speed up join operations between tables.

## Types of Indexes:

1. **B-tree Index:**
    - Default Index Type: The most commonly used index type in PostgreSQL, particularly suited for equality (=) and range queries (<, >, BETWEEN).
    - Use Case: Searching for specific values, ranges, and sorting.
    - Structure: A balanced tree structure that organizes the data for efficient lookup.
    - Syntax: CREATE INDEX idx_example ON table_name (column_name);

2. **Hash Index:**
    - Use Case: Suitable for equality comparisons, especially when searching for exact matches (=).
    - Limitations: Hash indexes only support equality searches and cannot be used for range queries.
    - Structure: Hashes the indexed column and stores pointers to the actual data.
    - Syntax: CREATE INDEX idx_example ON table_name USING hash (column_name);

3. **GiST (Generalized Search Tree) Index:**
    - Use Case: Supports complex data types such as geometric data, full-text search, and ranges.
    - Structure: A balanced tree structure that is flexible and can be used for various search types.
    - Example: Full-text search indexing.
    - Syntax: CREATE INDEX idx_gist ON table_name USING gist (column_name);

4. **GIN (Generalized Inverted Index) Index**
    - Use Case: Optimized for indexing composite values like arrays, JSONB, and full-text search.
    - Structure: Inverted index that allows quick searching of individual elements within a multi-element value.
    - Example: Indexing a JSONB column for quick retrieval of specific keys.
    - Syntax: CREATE INDEX idx_gin ON table_name USING gin (column_name);

5. **SP-GiST (Space-partitioned Generalized Search Tree) Index**
    - Use Case: Efficient for indexing multidimensional data such as geometric shapes, IP addresses, or any space-partitioned data.
    - Structure: Divides the space into partitions and uses a tree structure to index data.
    - Syntax: CREATE INDEX idx_spgist ON table_name USING spgist (column_name);

6. **BRIN (Block Range INdex) Index**
    - Use Case: Suitable for very large tables with naturally ordered data (e.g., time series data).
    - Structure: Uses blocks of data and stores a summary of each block to enable quick lookup.
    - Example: Indexing a column with ordered data (like timestamps).
    - Syntax: CREATE INDEX idx_brin ON table_name USING brin (column_name);

7. **Bloom Index**
    - Use Case: Used for indexing large sets of data with a low cardinality, especially for searching multiple columns.
    - Structure: A probabilistic data structure that allows for efficient membership queries.
    - Syntax: CREATE INDEX idx_bloom ON table_name USING bloom (column_name);

8. **Functional Index**
    - Use Case: Creates an index on the result of a function applied to one or more columns.
    - Example: Indexing the lowercased version of a string column for case-insensitive searching.
    - Syntax: CREATE INDEX idx_lower_name ON table_name (LOWER(column_name));

9. **Composite Index**
    - Use Case: Indexes on multiple columns to speed up queries that filter on more than one column.
    - Example: Indexing columns first_name and last_name together.
    - Syntax: CREATE INDEX idx_composite ON table_name (first_name, last_name);

## Basic Syntax of Creating an Index:

CREATE INDEX index_name ON table_name USING index_type (column_name);

**Parameters:**

- index_name: The name of the index. It should be unique within the table.
- table_name: The name of the table on which the index will be created.
- index_type: Optional. The type of index (e.g., btree, hash, gin, etc.). The default is btree.
- column_name: The column(s) to be indexed. This can be one or more columns. If you are creating a composite index (indexing multiple columns), you list them separated by commas.

## Drawbacks of Indexes:

1. **Write Performance**: Indexes can slow down insert, update, and delete operations because PostgreSQL has to update the index data whenever the underlying table is modified.
2. **Storage Consumption**: Indexes take up additional disk space. If you have many indexes on large tables, this can increase storage requirements significantly.
3. **Overhead for Small Tables**: For small tables, indexes might not provide much benefit and could add unnecessary overhead.
4. **Complexity in Query Plans**: Too many indexes on a table can confuse the PostgreSQL query planner, potentially resulting in suboptimal query plans.
5. **Index Corruption**: Although rare, indexes can become corrupted, especially in cases of hardware failures or improper shutdowns.

**Example 1: Creating a Simple Index**

This example creates a basic index on the name column of the employees table.

```
CREATE INDEX idx_employee_name ON employees (name);

-- Usage
SELECT * FROM employees WHERE name = 'John Doe';  -- The index will speed up this query
```

Explanation:

- CREATE INDEX idx_employee_name creates an index named idx_employee_name.

- ON employees (name) specifies that the index will be created on the name column of the employees table.
- This index will speed up queries that filter or search by the name column.

When the query SELECT * FROM employees WHERE name = 'John Doe'; is executed, PostgreSQL will use the index to quickly find the rows that match 'John Doe' instead of scanning the entire table.

**Example 2: Creating a Unique Index**
This example creates a unique index on the email column of the employees table to ensure that each employee has a unique email.

```
CREATE UNIQUE INDEX idx_unique_email ON employees (email);

-- Usage
INSERT INTO employees (name, email) VALUES ('Alice', 'alice@example.com');  -- Works fine
INSERT INTO employees (name, email) VALUES ('Bob', 'alice@example.com');    -- Will throw an error due to the unique constraint
```

Explanation:
- The CREATE UNIQUE INDEX statement ensures that the values in the email column are unique.
- If you try to insert a second row with the same email ('alice@example.com'), PostgreSQL will throw an error due to the unique constraint enforced by the index.

**Example 3: Creating a Multi-Column Index**
This example creates an index on the combination of the department_id and salary columns, which can be useful for queries that filter by both columns.

```
CREATE INDEX idx_department_salary ON employees (department_id, salary);

-- Usage
SELECT * FROM employees WHERE department_id = 2 AND salary > 50000;  -- The index will speed up this query
```

Explanation:
- The CREATE INDEX idx_department_salary creates a composite index on the department_id and salary columns.
- When you query for employees in a specific department with a salary greater than a threshold (e.g., SELECT * FROM employees WHERE department_id = 2 AND salary > 50000;), PostgreSQL can efficiently use the composite index to speed up the search.

Note: The order of columns in a multi-column index matters. In this case, the index is more efficient for queries that filter by department_id first and then salary. If the query filters salary first, the index might not be used as effectively.

**Example 4: Creating a Partial Index**
This example creates a partial index on the employees table to index only employees whose salary is greater than 50,000. This is useful if most of your queries are filtering on high salary employees.

```
CREATE INDEX idx_high_salary ON employees (name, salary)
WHERE salary > 50000;

-- Usage
SELECT * FROM employees WHERE salary > 50000;  -- The partial index will be used for this query
```

Explanation:

- The WHERE salary > 50000 condition limits the index to only include employees with a salary greater than 50000.
- When you run a query that filters by salary > 50000, the partial index will be used to speed up the query, but for employees with lower salaries, the index won't be used.

Partial indexes are useful when there is a condition that restricts the set of rows that are queried often. For example, if you often query only high-salary employees, a partial index can save space and improve query performance.

**Example 5: Creating a GiST Index for Geometric Data**
This example creates a GiST index for spatial data in the locations table, which stores geographic coordinates.

```
CREATE TABLE locations (
   id SERIAL PRIMARY KEY,
   name TEXT,
   coordinates GEOMETRY(Point, 4326)
);

-- Create a GiST index on the 'coordinates' column
CREATE INDEX idx_gist_coordinates ON locations USING gist (coordinates);

-- Usage: Queries involving geometric operations can benefit from this index
SELECT * FROM locations
WHERE ST_DWithin(coordinates, ST_SetSRID(ST_MakePoint(-73.935242, 40.730610), 4326), 1000);
```

Explanation:

- This example assumes that the locations table stores geographical points as GEOMETRY data type.
- The CREATE INDEX statement creates a GiST index on the coordinates column, which can efficiently support geographic queries (such as ST_DWithin).
- The ST_DWithin function checks if two points are within a certain distance of each other, and the GiST index speeds up this type of spatial query.

**Example 6: Using B-tree Index for Range Queries**

This example demonstrates how a B-tree index works with range queries.

```
CREATE INDEX idx_salary_range ON employees (salary);

-- Usage
SELECT * FROM employees WHERE salary BETWEEN 40000 AND 60000;  -- The B-tree index will speed up this query
```

Explanation:

- A B-tree index is created on the salary column.
- Queries that filter the salary column using range conditions like BETWEEN or comparisons such as >, <, >=, or <= will benefit from the B-tree index, as it is optimized for such operations.

**Example 7: Dropping an Index**

This example drops the idx_employee_name index we created earlier.

```
DROP INDEX IF EXISTS idx_employee_name;
```

Explanation:

- The DROP INDEX statement removes the index from the database.
- The IF EXISTS clause ensures that no error is thrown if the index doesn't exist.