Indexes In Relational Databases

Mashenkov Timofei
mashfeii

March 21, 2025

1. Introduction

Index is a special object which is created for single or several columns. It is used to speed up the search of data in the table. It can be compared with book and its table of contents, where you can find the necessary information much faster than just reading the book from the beginning to the end.

1.1. Simple Example

```sql
CREATE TABLE users (
    id INT PRIMARY KEY,
    first_name VARCHAR(255),
    last_name VARCHAR(255),
    city VARCHAR(255),
);
```

Listing 1: Simple example of table creation

If we fill the following table with 1 million records and try to retrieve all users from Moscow city, the query will take a long time:

```
EXPLAIN ANALYZE SELECT * FROM users WHERE city = 'Moscow';
```

Such query can take up to 180ms of execution time, so we need to optimize it:

```
CREATE INDEX idx_users_city_hash ON users USING HASH (city);
```

Such adjustment will reduce execution time to 30ms in average, also the Scan type will be changed to Index Scan.

2. Types of Indexes

- **B-Tree** - the most common type of index. It is used for columns with low cardinality.
- **Hash** - used for columns with high cardinality.
- **GiST** - used for geometric data types.
- **GIN** - used for indexing array data types.
- **BRIN** - used for very large tables.

2.1. Partial Index

Index may be created only for the **subset of data**, not the whole table. It is useful when you need to index only a part of the table.

```sql
 CREATE INDEX idx_users_city_moscow ON users (email) WHERE city = 'Moscow';
```

Listing 2: Partial index example

2.2. B-Tree

- Universal index for most cases (`=`, `<`, `>`, `<=`, `>=`, `BETWEEN`, `ORDER BY`).
- Used in `PRIMARY KEY`, `UNIQUE`, `FOREIGN KEY` constraints and within filtering on equality (`WHERE column = 'value'`).
- Does not fit the full-text search (`LIKE '%text%'`).
- Can slow down inserting and updating.

2.3. Hash

- Fast and concise, but only for `=` operator.
- Not used for ranges (`<`, `>`, `BETWEEN`) and sorting (`ORDER BY`).
- In PostgreSQL works better on fields like `user_id`, `email`, `ip_address`.
- **If you are not sure, use `B-Tree`.**

2.4. GiST

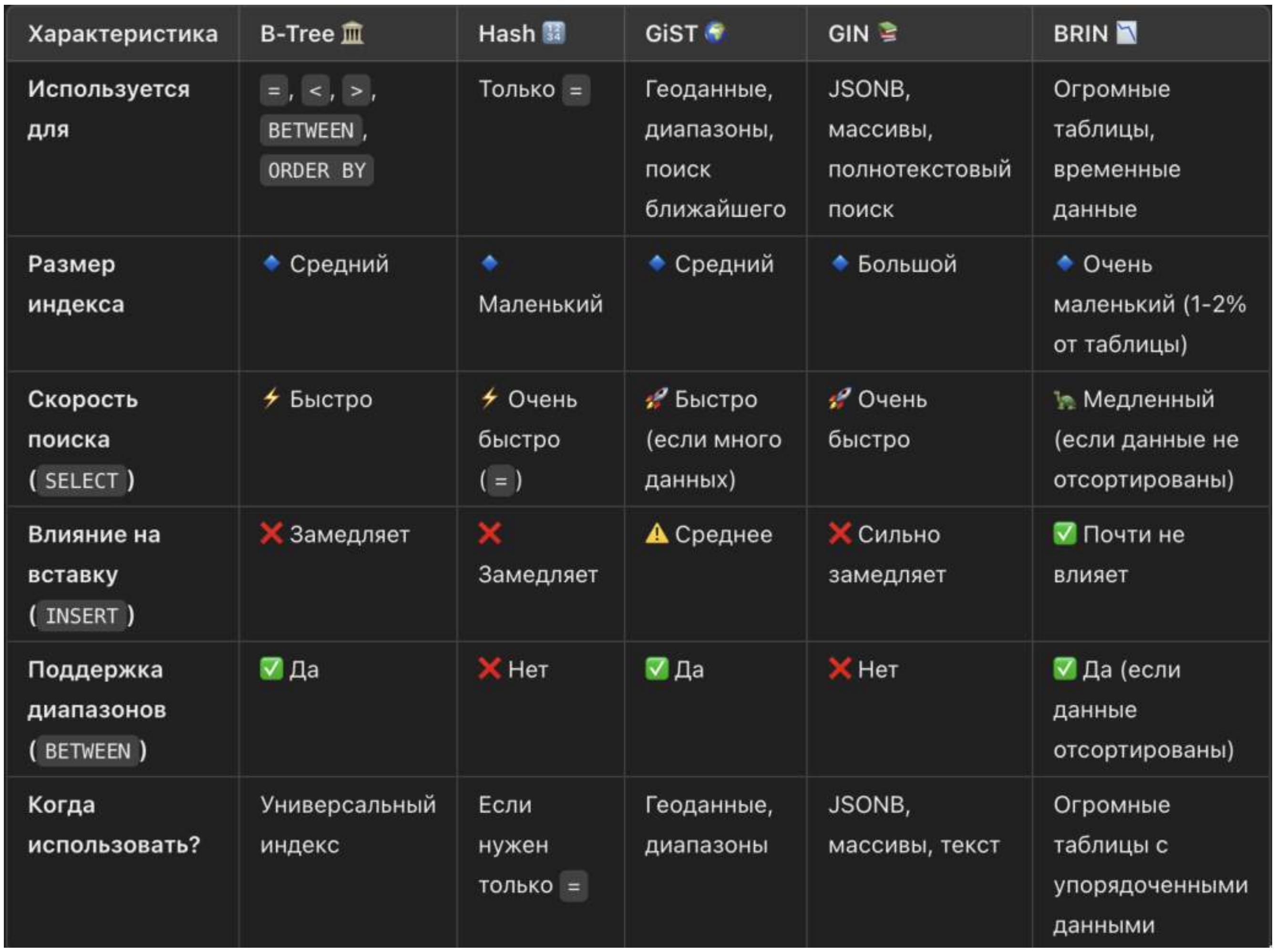
- Universal index for complex data types
- Used for geodata, ranges, arrays, `JSONB`, full-text search.
- Does not replace B-Tree for `=`, `BETWEEN`, `ORDER BY`.
- Alternatives: for text (`GIN`), for standard requests (`B-Tree`).

2.5. GIN

- The best fit for the `JSONB`, arrays, full-text search.
- Works faster than `B-Tree`, `GiST` for searching complex data types.
- Slow under the `INSERT`/`UPDATE`/`DELETE` operations, does not support `ORDER BY`.

2.6. BRIN

- The best fit for very large tables with ordered data.
- Occupies much less space and speed up range requests.
- Does not replace `B-Tree`, but perfectly works with time data.
- **If the data is not ordered, `BRIN` is useless.**



Характеристика	B-Tree 🏛	Hash 🔢	GiST 🌍	GIN 📚	BRIN 📉
Используется для	`=`, `<`, `>`, `BETWEEN`, `ORDER BY`	Только `=`	Геоданные, диапазоны, поиск ближайшего	JSONB, массивы, полнотекстовый поиск	Огромные таблицы, временные данные
Размер индекса	🔹 Средний	🔹 Маленький	🔹 Средний	🔹 Большой	🔹 Очень маленький (1-2% от таблицы)
Скорость поиска (`SELECT`)	⚡ Быстро	⚡ Очень быстро (`=`)	🚀 Быстро (если много данных)	🚀 Очень быстро	🐢 Медленный (если данные не отсортированы)
Влияние на вставку (`INSERT`)	❌ Замедляет	❌ Замедляет	⚠️ Среднее	❌ Сильно замедляет	✅ Почти не влияет
Поддержка диапазонов (`BETWEEN`)	✅ Да	❌ Нет	✅ Да	❌ Нет	✅ Да (если данные отсортированы)
Когда использовать?	Универсальный индекс	Если нужен только `=`	Геоданные, диапазоны	JSONB, массивы, текст	Огромные таблицы с упорядоченными данными

3. PostgreSQL Optimizations

Request within the time of execution also contains the time of planning. Planning stands for the time when Database Management System (DBMS) decides how to execute the query.

Type of Optimization	What is does?
Rewriting Rules	Simplifies `WHERE`, `IN`, `JOIN`, `CTE`
Indexes Using	`Index Scan`, `Index Only Scan`, `Bitmap Index Scan`
`JOIN` Optimization	`Nested Loop`, `Hash Join`, `Merge Join`
`ORDER BY` Optimization	Using indexes instead of sorting
Parallel Execution	Parallels `SELECT`, `JOIN`, `AGGREGATE`

Table 1: PostgreSQL Optimizations