Indexing in Relational Databases

Sebastian Ernst, PhD

Course: Databases II, EAIiIBISIS.Ii8K.5dfa09851a120.22

Database indexing

Database indexing

- An index is a structure which accelerates data lookup in a table.
- An index is defined for specific attributes, called *index keys* or *search keys*.
- A *dense* index contains all values; a *sparse* one contains only some.
- As most things, choosing indexes is a matter of compromise and judgement: they may accelerate data lookup (reads), but require rebuilding and balancing on every insert, update or delete.

Indexing in PostgreSQL

Index types supported by PostgreSQL:

- B-trees (default)
- hash
- GiST/SP-GiST
- GIN

To create a simple index:

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

Observing indexes in action

- The `EXPLAIN` command shows the execution plan of a statement.
- The `ANALYZE` command collects statistics about the contents of tables and stores the results in the `pg_statistic` system catalog.
- `EXPLAIN ANALYZE` runs the command and provides actual execution statistics. Note that the command will actually be executed in this case.

Index types

B-trees

- Can be used for domains which have an ordering function…
- …e.g., when search keys can be compared using: $<, <=, =, >=, >$
- Also applicable: BETWEEN, IN, IS (NOT) NULL
- Can retrieve sorted data (more on that later).

Hash-based indexes

- Only support the equality comparison operator $=$, i.e. can be only used to find records with exact value of an attribute.
- Create by adding USING hash to the CREATE INDEX statement:

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

GiST

- GiST (Generalised Search Trees) is an infrastructure that allows implementation of various indexing strategies for various types of data.
- Out of the box, GiST supports indexing of multidimensional data and use of geometric operators (via SP-GiST).
- Example – retrieve 10 closest locations:

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

GIN

- GIN provides “inverted indexes” for indexing of non-atomic data.
- GIN also supports user-defined indexing strategies.
- For example, out of the box PostgreSQL supports strategies for one-dimensional arrays and their operators.

Advanced index usage

B-tree indexes vs. patterns

- B-tree indexes can be used for pattern matching (LIKE, ~) iff pattern is a constant anchored to the beginning of the string:
  - `col LIKE 'foo%'`
  - `col ~ '^foo'`
  - ~~`col LIKE '%bar'`~~
- Also applicable to case-insensitive operators (`ILIKE`, `~*`), but only if pattern starts with characters not affected by upper/lowercase conversion.

Multi-column indexes

Sometimes, we often need queries with WHERE clauses concerning more than one attribute:

```
SELECT * FROM test2
  WHERE major = 23
  AND minor = 42;
```

In such case, a multi-column index can be useful:

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

Such indexes can also be used for queries concerning a subset of search key attributes.

Combining indexes

- A multi-column index can be used if constraints on column values use the `AND` logical operator.
- PostgreSQL can also use multiple indexes in one query, or use one index for multiple vaues.
- Individual indexes are scanned and bitmaps of matching records are created. Later, these bitmaps are combined using operators from the `WHERE` clause, e.g. `AND` or `OR`.

Ordered indexes

- A B-tree index can provide results in a given order.
- By default, search key values are ordered ascending, with nulls at the end, but this can be changed:

```
CREATE INDEX test2_info_nulls_low ON test2 (info NULLS FIRST);
CREATE INDEX test3_desc_index ON test3 (id DESC NULLS LAST);
```

- An index can be used for queries with ORDER BY clauses identical with that used for its creation, or one that’s completely reversed.

Unique indexes

- An index can enforce that values in a column are unique (or that combinations of values are – for multi-column indexes).
- `NULL` values are never considered identical.
- Unique indexes are created using the `CREATE UNIQUE INDEX` command.

Indexes on expressions

Search keys do not need to include only columns – they can also include expressions.

If an application often performs a query like:

```
SELECT * FROM test1 WHERE lower(col1) ='value';
```

it may be advisable to create an index on that expression:

```
CREATE INDEX test1_lower_col1_idx ON test1 (lower(col1));
```

Partial indexes

An index can cover only some rows in a table.

For instance, if we want to log network requests, but only index those from outside of our local subnet, an appropriate index may be used:

```
CREATE INDEX access_log_client_ip_ix
  ON access_log (client_ip)
  WHERE NOT
    (client_ip > inet '192.168.100.0' AND
     client_ip < inet '192.168.100.255');
```

Indexes and collations

- An index only supports a single collation for a given column.
- Constants are automatically cast onto the appropriate collation.
- If values of different collations are to be compared, an index can be built for that collation:

```
CREATE INDEX test1c_content_y_index
  ON test1c (content COLLATE "y");
```