Assignment 3

Index Tuning

Database Tuning

New Group 8
Frauenschuh Florian, 12109584
Lindner Peter, 12101607
Weilert Alexander, 12119653

May 6, 2024

Database system and version: `Postgres 14.11` with driver `postgresql 42.7.3`

1 Index Data Structures

Which index data structures (e.g., B^+ tree index) are supported?

B-Tree [1]

Default when using `CREATE INDEX`

```
CREATE INDEX idx_name ON table_name (column_name);
```

Hash [1]

Hash index stores 32-bit hash derived from the indexed column. Only supports equality comparisons.

```
CREATE INDEX idx_name ON table_name USING HASH (column_name);
```

GiST [1] [2]

"Generalized Search Tree", an infrastructure for many index strategies and is a lossy index (may produce false matches).

```
CREATE INDEX idx_name ON table_name USING GIST(column_name);
```

SP-GiST [1]

"Space-partitioned GiST", focuses on non-balanced data structures.

```
CREATE INDEX idx_name ON table_name USING SPGIST(column_name);
```

GIN [1] [2]

"Generalized Inverted Index" suitable for values containing multiple components, e.g., arrays.

```
CREATE INDEX idx_name ON table_name USING GIN(column_name);
```

BRIN [1]

"Block Range INdex" indexes ranges of values.

```
CREATE INDEX idx_name ON table_name USING BRIN(column_name);
```

Extensions [3]

Postgres also supports the `bloom` extension. The bloom filter data structure is used to check if an element is a member of a set.

2 Clustering Indexes

Discuss how the system supports clustering indexes, in particular:

a) How do you create a clustering index on `ssnum`? Show the query.[1]

First, we assume the table `Employee` to be the same as in the previous assignment:

```
CREATE TABLE IF NOT EXISTS Employee (
    ssnum INTEGER PRIMARY KEY,
    name VARCHAR(64) UNIQUE NOT NULL,
    manager VARCHAR(64),
    dept VARCHAR(64),
    salary INTEGER,
    numfriends INTEGER);
```

We note that `ssnum` is the primary key of the table, hence Postgres automatically creates an B-Tree based, *unique index* for it [4].

Now to be able to create a clustering index, we need an index to begin with. As mentioned, such an index already exists for `ssnum`. Thus, we can *cluster* this index according to [5] by performing:

```
CLUSTER Employee USING idx_ssnum;
```

Here we assumed the index on `ssnum` to be named `idx_ssnum`.

If we had to create such an index by ourselves, one could accomplish this by performing:

```
CREATE UNIQUE INDEX idx_ssnum ON Employee [USING BTREE] (ssnum);
```

The additional command in parentheses `USING BTREE` is optional, since it already is the default for Postgres (and Postgres only supports unique indexes using B-Trees) [4].

Now in order to create a clustered hash index, we first need to create the index itself following [1]:

[1] Give the queries for creating a hash index *and* a B^+ tree index if both of them are supported.

```
CREATE INDEX idx_ssnum ON Employee USING HASH (ssnum);
```

We note that we now can no longer create a unique index, since it is not supported as mentioned before. Afterwards, we again can cluster this created index:

```
CLUSTER Employee USING idx_ssnum;
```

b) Are clustering indexes on non-key attributes supported, e.g., on `name`? Show the query.

Yes, clustering indexes are also supported on non-key attributes. As mentioned earlier, we first need an index on name to cluster it:

```
CREATE INDEX idx_name on Employee (name);
```

Followed by that, we can now cluster this created index by performing:

```
CLUSTER Employee USING idx_name;
```

c) Is the clustering index dense or sparse?

In general, Postgres only supports dense indexes [6]. Hence, clustering indexes are dense as well. The only exception from this is the GIN index type, which is a somewhat sparse index.

d) How does the system deal with overflows in clustering indexes? How is the fill factor controlled?

According to [7], Postgres fills the index pages up to the provided `fillfactor`. Once such an index page reached the configured `fillfactor`, it gets split up. This split leads to a decrease in efficiency for index lookups.

The `fillfactor` itself represents a percentage of how many entries are placed inside a page before a split happens. It can be set during the creation of an index, e.g.:

```
CREATE UNIQUE INDEX idx_name ON Employee (name) WITH (fillfactor = 70);
```

Per default, the `fillfactor` is set to 70% for B-Tree indexes.

e) Discuss any further characteristics of the system related to clustering indexes that are relevant to a database tuner.

For clustering indexes it is important to note, that `CLUSTER` itself is a one-time operation, where the sorting of the table does not get maintained by Postgres. This means that as soon as any updates to the table were made, Postgres does not ensure that the *clustering* property is still satisfied. Hence, when performing a `CLUSTER` command, only the rows that are currently inside the table are affected.

Additionally, during the clustering process the table is locked for any other read or write accesses [5].

3 Non-Clustering Indexes

Discuss how the system supports non-clustering indexes, in particular:

a) How do you create a combined, non-clustering index on `(dept,salary)`? Show the query.[1]

Postgres uses non-clustering indexes per default [8]. Only B-tree, GiST, GIN, and BRIN indexes support combined indexes in Postgres [9].

```
CREATE INDEX idx_dept_salary ON Employee [USING BTREE] (dept, salary);
```

b) Can the system take advantage of covering indexes? What if the index covers the query, but the condition is not a prefix of the attribute sequence `(dept,salary)`?

Taking advantage of covering indexes with `index-only scans` is only possible under the following restrictions:

1. Index type must support index-only scans. `B-Tree` always supports it, while `GiST` and `GIN` only support it for some operator classes.
2. The query must reference only columns that are part of the index. Columns can be added to a covering index by using the `INCLUDE` clause.

When these criteria are met, then the system can take advantage of covering indexes.

Assume an index like the one created in 3a. When executing a covered, but non-prefix query like

```
SELECT dept
FROM Employee
WHERE salary < 48000;
```

then the DBMS reads the table sequentially and does not make use of the index, since the table is primarily sorted by the `dept` attribute.

c) Some additional relevant characteristics/features are:

- **Partial indexes** [10]:
  Additionally, one can create a partial index on attributes. This might be useful to keep the index small and still be able to cover recurring queries which select certain value ranges. For example:

  ```
  CREATE INDEX idx_salary ON Employee (salary)
  WHERE salary < 100000;
  ```

- **Order of the index** [11]:
  With Postgres it is possible to influence the order of the index by using `ORDER By`, which is especially useful for multi-column indexes. One can use the following keywords when creating an index:
  - ASC
  - DESC
  - NULLS FIRST
  - NULLS LAST

  For example:

  ```
  CREATE INDEX idx_id_desc ON test (id DESC NULLS LAST);
  ```

  It is important to note, that such an ordering can make an index unusable in certain scenarios, i.e. when executing a query with a different `ORDER BY`.

- **Indexes on Expressions** [12]: Postgres not only supports indexes on attributes alone, but also on functions or scalar expressions computed on attributes. For example, one could create an index

  ```
  CREATE INDEX idx_col1_lower ON test (lower(col1));
  ```

  to perform a fast case-insensitive comparison when executing queries like

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

4 Key Compression and Page Size

If your system supports B^+ trees, what kind of key compression (if any) is supported? How large is the default disk page? Can it be changed?

The documentation of Postgres lacks an explanation if key compression is supported in the version we are making use of. Even through additional research we were not able to find any hints that this feature is implemented in Postgres. Hence, one has to assume that Postgres does not support key compression to this point.

The default size of a disk page equals 8 kilobytes. According to [13] it does not seem as one can change this value easily by tools or commands that are directly supported from Postgres itself.

Time Spent on this Assignment

Time in hours per person:

- Florian Frauenschuh: **5.5**
- Peter Lindner: **6**
- Alexander Weilert: **5**

References

[1] PostgreSQL. *Index Types*. Accessed: 2024-04-30. 2024. URL: https://www.postgresql.org/docs/14/indexes-types.html.

[2] PostgreSQL. *GIN and GiST Index Types*. Accessed: 2024-04-30. 2024. URL: https://www.postgresql.org/docs/14/textsearch-indexes.html.

[3] PostgreSQL. *Bloom Index*. Accessed: 2024-04-30. 2024. URL: https://www.postgresql.org/docs/14/bloom.html.

[4] PostgreSQL. *Unique Indexes*. Accessed: 2024-04-30. 2024. URL: https://www.postgresql.org/docs/14/indexes-unique.html.

[5] PostgreSQL. *CLUSTER*. Accessed: 2024-04-30. 2024. URL: https://www.postgresql.org/docs/14/sql-cluster.html.

[6] user2993792. *PostgresSQL Indices - Are They Dense or Sparse?* Accessed: 2024-04-30. 2013. URL: https://stackoverflow.com/questions/19987786/postgresql-indices-are-they-dense-or-sparse.

[7] PostgreSQL. *CREATE INDEX*. Accessed: 2024-04-30. 2024. URL: https://www.postgresql.org/docs/14/sql-createindex.html.

[8] MAK. *Cluster and Non cluster index in PostgreSQL*. Accessed: 2024-04-30. 2015. URL: https://stackoverflow.com/questions/27978157/cluster-and-non-cluster-index-in-postgresql.

[9] PostgreSQL. *Multicolumn Indexes*. Accessed: 2024-04-30. 2024. URL: https://www.postgresql.org/docs/current/indexes-multicolumn.html.

[10] PostgreSQL. *Partial Indexes*. Accessed: 2024-05-06. 2024. URL: https://www.postgresql.org/docs/14/indexes-partial.html.

[11] PostgreSQL. *Indexes and ORDER BY*. Accessed: 2024-05-06. 2024. URL: https://www.postgresql.org/docs/14/indexes-ordering.html.

[12] PostgreSQL. *Indexes on Expressions*. Accessed: 2024-05-06. 2024. URL: https://www.postgresql.org/docs/14/indexes-expressional.html.

[13] PostgreSQL. *Determining Disk Usage*. Accessed: 2024-04-30. 2024. URL: https://www.postgresql.org/docs/14/disk-usage.html.