Assignment 3

Index Tuning

Database Tuning

A4
Balint Peter, 12213073
Günter Lukas, 12125639
Ottino David, 51841010

April 28, 2025

Database system and version: *PostgreSQL 17.4-1*

1 Index Data Structures

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

1.1 B-Tree

Postgres uses B-Trees as it's standard data structure for storing indexes. Whenever the `CREATE INDEX` command is used without additional parameters, a B-Tree is created.

Although the documentation refers to these as B-Trees, they are in fact closer to B^+-trees. B-trees are classified as trees which include data inside the internal nodes, while a B^+-tree only includes data and pointers to the next node in the individual leaf nodes. PostgreSQL implements the latter scheme, where "data" actually just denotes a pointer to the heap.

On the other hand, since the actual data is not actually embedded in the index either way, the distinction may not really apply in the first place.

1.2 Hash

Hash indexes in Postgres store a 32-bit hash code of the indexed column which is then used to find and access data. Due to its nature hash indexes are considered when the indexed column is involved in a comparison using the "="-Operator.

1.3 GiST

GiST stands for Generalized Search Tree. It doesn't describe a specific index type but is more of a framework which allows Postgres-Users to implement their own arbitrary indexing schemes, for example B^+ trees. However, it comes with some built-in standard functionality.

1.4 SP-GiST

The "SP" in this index type stands for space-partitioned GiST, which means that the search space is repeatedly divided into different partitions of potentially different sizes. Searches that are well matched to the partitioning rule can be very efficient. Similar to GiST, SP-GiST allows users to develop their own custom data types and access methods.

1.5 GIN

GIN stands for "Generalized Inverted Index" and is used for complex data values that consist of different components, such as arrays or texts, and the expected queries search for different elements within this data. GIN stores each of these components, so called keys, within a separate index. Each key references to the tupels it is contained in. It is also possible to develop custom data types and access methods with this index type.

1.6 BRIN

BRIN, which is shorthand for "Block Range Indexes", is mainly used for very large tables in which attributes can naturally be ordered in some way and this order is resembled by their location in physical memory. BRIN stores a summary of all the data present in one or multiple pages that are besides each other in memory. This means that the index itself is very small and therefore fast to traverse. However, BRIN is a lossy type of index, which means that after traversing the index, the returned tupels will still have to be checked for the actual selection criteria.

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]

PostgreSQL does not directly provide clustered indexes. However, the command `CLUSTER` clusters a given table based on an already existing index, but this doesn't work on a hash index.

```
CREATE INDEX idx_employee_ssnum ON Employee(ssnum);
CLUSTER Employee USING idx_employee_ssnum;
```

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

Yes, it is possible to create clustering indexes on all attributes using the aforementioned method.

```
CREATE INDEX IDX_EMPLOYEE_NAME ON EMPLOYEE (NAME);
CLUSTER EMPLOYEE USING IDX_EMPLOYEE_NAME;
```

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

c) Is the clustering index dense or sparse?

The PostgreSQL-documentation does not outright state if the created indexes are dense or sparse. However, it is possible to view the number of entries of a given index which allows us to conclude if the index is dense or sparse.

```
SELECT COUNT(DISTINCT MANAGER)FROM EMPLOYEE;
CREATE INDEX IDX_EMPLOYEE_MANAGER ON EMPLOYEE (MANAGER);
SELECT reltuples FROM PG_CLASS
WHERE RELNAME = 'idx_employee_manager';
```

We know from the first query that there are 49.825 managers. But the total number of tuples in the index is 100.000, which is the total amount of rows. Therefore we can conclude that the index is dense.

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

Since the clustering index in PostgreSQL is not really a clustered index in the traditional sense, overflows are not really an issue since new data is just inserted whereever there is an open space and a repeated clustering based on the index leads to the whole data structure being reorganized. The fill factor can be set by the user for each table or index individually with the following statements:

```
ALTER TABLE EMPLOYEE SET (fillfactor = 80);
CREATE INDEX IDX_EMPLOYEE_MANAGER ON EMPLOYEE (MANAGER) WITH (fillfactor = 80);
```

The specified fill factor is stored in `pg_class` in the `reloptions`-attribute.

e) If new data is inserted into the table, the additions won't be clustered automatically and a new `CLUSTER`-operation has to be executed. Clustering took about 1,5 seconds in our case.

```
CLUSTER Employee;
```

The `CLUSTER`-operation without any arguemnts will cluster according to the last index used when clustering.

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]

As PostgreSQL does not use clustering indexes, it is enough to just use CREATE INDEX.

```
CREATE INDEX idx_dept_salary
ON employee (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)`?

[Your answer goes here ...]

Index-only scans are a relatively new feature in PostgreSQL, first introduced in version 9.2. The reason for this late introduction is that PostgreSQL indexes by default do not include information about tuple visibility to the MVCC (Multi-Version Concurrency Control) snapshot of the given transaction, so they would by default require accesses to the relevant tables anyway.

With the introduction of index-only scans, PostgreSQL now uses a visibility map, where a bit is set on all pages on which all tuples are visible for each snapshot available to active transactions. If the bit is set, no additional access to the table is required, and queries can be answered using only the index, provided it already includes all required data.

Of course, this still does not mean that no more table accesses are required; if the visibility bit is not set for a page (for example, because a concurrent transaction has unset it), the tuple must still be read from the table itself. Therefore the term "index-only scan" is somewhat misleading; "index-mostly scan" is more accurate.

PostgreSQL only considers index-only scans if the majority of these visibility bits are set. Therefore this feature is mainly useful for tables which are rarely (or never) modified.

4 Covered Query (Covering Index)

Following is an example use of a covering index. The goal is to compute the average salary of all employees that work in the 'TechdeptA' department.

For this we first create an index on the columns (dept, salary) of the employee table. Then we perform an SQL query on it.

```
//Create B-Tree index on employee(dept, salary).
CREATE INDEX s_idx ON employee(dept, salary);

//Execute the actual query.
EXPLAIN (analyze, buffers)
SELECT AVG (salary)
FROM employee
WHERE dept = 'TechdeptA';
```

For this, PostgreSQL returns the following execution plan:

```
Aggregate(cost=42.22..42.23 rows=1 width=4)
    (actual time=0.305..0.305 rows=1 loops=1)
Buffers: shared hit=7
-> Index Only Scan using s_idx on employee e (cost=0.42..39.49 rows=1090 width
    (actual time=0.108..0.236 rows=1002 loops=1)
    Index Cond: (dept = 'TechdeptA'::text)
    Heap Fetches: 0
    Buffers: shared hit=7
Total runtime: 0.348 ms
```

Here, the newly created index is used to answer the query solely using information included in the index. Since the table does not have to be read, the number of heap fetches remains zero.

5 Covered Query (but not prefix)

Now we wish to count the number of employees that have a salary less than 2000. The salary attribute is included in the index; however, this is sorted by dept.

```
SELECT COUNT(*)
FROM employee
WHERE salary < 2000;
```

```
Aggregate (cost=2207.90..2207.91 rows=1 width=0)
    -> Seq Scan on employee e (cost=0.00..2084.01 rows=49556 width=0)
        Filter: (salary < 2000)
```

Here the planner does not take the use of the index on (dept, salary) into consideration, but rather employs a sequential scan and filters on salary.

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

In addition to classic non-clustered indexes, PostgreSQL offers two additional functions that can further increase the efficiency of indexes: Partial Indexes and Indexes on Expressions.

6 Partial indexes

A partial index is only created for a subset of the table. Which tuples are included in the index is defined via a WHERE clause. This allows common values, whose inclusion in the index is of no use anyway, to be specifically excluded.

For values that appear very frequently in a table, the PostgreSQL query planner often decides against using an index anyway and prefers a sequential scan instead. The partial index means that only a specific part of the table is indexed, which offers several advantages:

- The index is smaller and requires less storage space.
- Queries that actually use the index can be executed more efficiently.
- The maintenance effort for the index is also reduced.

Nevertheless, the potential performance gain should not be overestimated - in many scenarios, the advantage of a partial index remains relatively small. Additionally, common values will commonly be deduplicated by the engine anyway (see example 4).

7 Indexes on expressions

With an index on expressions, the value of a function can be saved to column contents in the index. This is particularly useful when functions are used in queries:

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

Here, all data records from test1 are to be found in which col1 - irrespective of upper or lower case - contains the value 'value'.

Instead, we can create an index on the expression:

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

Now the index only saves the already transformed values. Thus PostgreSQL can answer the query significantly faster, as the function call is replaced by a direct comparison on the index.

8 Key Compression and Page Size

If your system supports B^+ trees, what kind of key compression (if any) is supported?

While PostgreSQL does not appear to use prefix compression for its B^+-trees, it does have a form of compression called B-tree deduplication since version 13. When all indexed values in a tuple match those of another in the same table, these form a so-called "posting list", where the key values only appear once followed by a list of all relevant tuple ids.

This may have a significant impact on the size of indexes with a large amount of common values. For example:

```
-- check initial index size
SELECT pg_indexes_size(relid) FROM pg_catalog.pg_statio_user_tables where relname
-- -> 18530304
-- now, create an index on dept with deduplication (default)
CREATE INDEX ON employee (dept);
SELECT pg_indexes_size(relid) FROM pg_catalog.pg_statio_user_tables WHERE relname
-- -> 19243008
DROP INDEX employee_dept_idx;
-- try again, this time without deduplication:
CREATE INDEX ON employee (dept) with (deduplicate_items = off);
SELECT pg_indexes_size(relid) FROM pg_catalog.pg_statio_user_tables WHERE relname
-- -> 20897792
```

In this contrived example, the optimization has saved us 1.57M. However, as noted in the previous example, indexes with such a large number of duplicate keys may also end up being ignored by the planner.

How large is the default disk page?

The default disk page can be queried using the following command, which on a typical PostgreSQL installation will return 8192 (bytes).

```
SELECT current_setting('block_size');
```

To change the block size, PostgreSQL must be recompiled, passing the *–with-blocksize* parameter to the configure script. Only powers of two between 1024 and 32768 are supported.

```
./configure --with-blocksize=<size in kB>
```

Time Spent on this Assignment

Time in hours per person: **XXX**

References

Important: Reference your information sources!

https://stackoverflow.com/questions/25004505/b-tree-or-b-tree

https://www.postgresql.org/docs/current/indexes-types.html

https://www.shiksha.com/online-courses/articles/difference-between-b-tree-and-b-plus-tree-blogId-155905

https://www.postgresql.org/docs/current/gin.html

https://www.postgresql.org/docs/8.1/gist.html

https://www.postgresql.org/docs/16/brin-intro.html

https://www.postgresql.org/docs/current/sql-cluster.html

https://www.cybertec-postgresql.com/en/what-is-fillfactor-and-how-does-it-affect-postgresql-performance

https://www.postgresql.org/docs/current/storage-vm.html

https://www.postgresql.org/docs/current/indexes-index-only-scans.html

https://www.postgresql.org/docs/current/mvcc-intro.html

https://stackoverflow.com/a/67252824

https://www.postgresql.org/docs/16/btree-implementation.html#BTREE-DEDUPLICATION

https://www.postgresql.org/docs/current/install-make.html#CONFIGURE-OPTIONS-MISC