Databases Assignment 1

Query optimization
15 March 2021

1. Using B-Tree index: B-tree index basically creates a tree that will try to be balanced and even. When indexing is used first, the database is searched for a given key in correspondence to B-tree and gets the index in O(log(n)) time complexity. Then, it performs another search in B+tree by using the already found index in O(log(n)) time and gets the record. B-tree indexes are valuable on the most common data types such as text, numbers, and timestamps.

   Each of these nodes in B-tree and B+tree is stored inside the Pages. Pages are fixed in size. Pages have a unique number starting from one. A page can be a reference to another page by using a page number.

   Q.1 -

   i. Creating a B-tree-index "btree_index_amount" on the single column(amount) of the table "payment"
   The B-tree index is most useful and efficient when used on range queries and equalities. We have a comparison condition on the *amount* attribute which is a numeric(double with a range [0,11]) datatype therefore the B-tree index would be of great use here.

   Nothing is mentioned about common values in indexing creation, so we assume that after applying the B-tree index, it would traverse through the levels of the tree to find the suitable table it is searching for. This will probably take less time, but it doesn't show a significant improvement because of using an aggregation function inside the query (count).

   ii.
```
CREATE INDEX btree_index_amount ON public.payment USING btree
        (amount ASC NULLS LAST)
            INCLUDE(amount)
            TABLESPACE pg_default;
```

   iii. Cost Before indexing = 510.99..12062691.37 and actual time = 678.179..112457.984
   Cost After indexing = 510.99..12007381.21 and actual time = 501.179..84672.975

The difference between the two costs: 12062691.37 - 12007381.21 = 55310.16
The improvement percentage: 55310.16 / 12062691.37 = 0.4585% .

Therefore, we can see, cost and actual time decreased a bit.

Q2. -

i. Creating the B-tree-index "btree_index_last_update" on the single column(last_update) of the table "rental".

The B-tree index is most useful and efficient when used on range queries and equalities.
We have a comparison condition on the *last_update* attribute with the same attribute on the other side of the inequality. The *last_update* attribute is a *timestamp* datatype, thus B-tree indexes would be of great use here.

It will have a significant improvement as only one record in the whole table has a *last_update* value that's actually different from the others.

ii. CREATE INDEX btree_index_last_update ON public.rental USING btree (last_update)

```
    (last_update ASC NULLS LAST)
    INCLUDE(last_update)
    TABLESPACE pg_default;
```

iii. Cost before indexing = 534.78..2284549.49
Cost after indexing = 531.06..1557112.30
The difference between the two costs: 2284549.49 - 1557112.30 = 727,437.19
The improvement percentage: 727,437.19 / 2284549.49 = 31.841%

Q3. -

i. Creating the B-tree-index "btree_index_rental_duration" on the single column(rental_duration) of table "film".

The B-tree index is most useful and efficient when used on range queries and equalities. We have a comparison condition on the *rental_duration* attribute with the same attribute on the other side of the inequality. The *rental_duration* attribute is a numeric datatype, thus B-tree indexes would be of great use here.

ii. CREATE INDEX btree_index_rental_duration ON public.film USING btree (rental_duration)

```
    (rental_duration ASC NULLS LAST)
    INCLUDE(rental_duration)
```

TABLESPACE pg_default;

iii. Cost before indexing = 7747.19..7747.20
Cost after indexing = 5110.74..5410.27
The difference between the two costs: 7747.20 - 5410.27 = 2,336.93
The improvement percentage: 2,336.93 / 7747.20 = 30.164%

2. Using Hash index -

Hash index is used by hashing the values of the provided column. This resulting value is used as an index to retrieve some previous records. Therefore it is effective and is mostly used during inequality operator cases.

Hash indexes at times can provide faster lookups than B-Tree indexes. But the problem is that they're limited to only equality operators, so they only work while looking for exact matches.

Q1. -

The equality operator mentioned in this query doesn't have a constant on its right-hand side. That means, even if we *hashed rental_id*, we still need to loop on them all and use it for matching. This is utterly useless.

Also, creating a hash index on any combination of columns wouldn't improve the performance since we already have a unique index on the PK, the PostgreSQL is already using this index to optimize the query.

When we run the EXPLAIN ANALYZE on the query before creating any indexes, we can clearly see that the pg-query-optimizer is already using indexes and trying to optimize. Even if forced the pg to use other "indexes", it wouldn't and will still count on the unique index on the PK instead of the manually created one.

Q2. - For the same reasons for Q1, hash indexes won't be a benefit for this query either.

Q3. - For the same reasons for Q1 and Q2, hash indexes won't be a benefit for this query either.

Additionally in this query, there aren't any equality conditions, only range conditions, which cannot be optimized using a hash index.

We can analyze that using B-tree indexes on these 3 queries is far better than using hash indexes because most of the conditions in the queries are range conditions or inequalities.

3. Using SP-GiST index:

   SP-GiST indexes are mostly used when the data has a natural clustering element to it and is also not an equally balanced tree. Therefore, we can use SP-GiST for larger datasets with natural but uneven clustering.

   Also, SP-GiST supports partitioned search trees that facilitate the development of a wide range of different non-balanced data structures. It makes an unbalanced tree.

   The most common example it works on is the phone, address, etc.

   So basically there is nothing to apply in the three queries as it doesn’t support date and most of the data types used in the above queries.