B561 Assignment 7
Spring 22
Indexes & Indexing
Due: 21st April 2022, 11:59pm EST

This assignment covers the following topics -

- Indexes & Indexing
- B^+ Trees
- Hashing

In addition, you should read the following sections in Chapter 8, 14 and 15 in the textbook *Database Systems The Complete Book* by Garcia-Molina, Ullman, and Widom:

- Section 8.4.2: Some Useful Indexes
- Section 14.1: Index-structure Basics
- Section 14.2: B-Trees
- Section 14.3: Hashing and Extensible Hashing

Note that the section numbers may change depending on the version of the book you have.

Problems 4 and 7 - 20 points. All other problems - 10 points.

Submit all your answers in `assignment7.pdf`

1 Indexes & Indexing

Discussion PostgreSQL permits the creation of a variety of indexes on tables. We will review such `index creation` and examine their impact on data lookup and query processing. For more details, see the PostgreSQL manual:

https://www.postgresql.org/docs/13/indexes.html

Example 1 *The following SQL statements create indexes on columns or combinations of columns of the* `personSkill` *relation.*[1] *Notice that there are*

$$2^{arity(\texttt{personSkill})} - 1 = 2^2 - 1 = 3$$

such possible indexes.

```
create index pid_index on personSkill (pid);                    --
index on pid attribute
create index skill_index on personSkill (skill);                --
index on skill attribute
create index pid_skill_index on personSkill (pid,skill);        --
index (pid, skill)
```

Example 2 *It is possible to declare the type of index:* `btree` *or* `hash`. *When no index type is specified, the default is* `btree`. *If instead of a Btree, a* hash index *is desired, then it is necessary to specify a* `using hash` *qualifier:*

```
create index pid_hash on personSkill using hash (pid);
-- hash index on pid attribute
```

Example 3 *It is possible to create an index on a relation based on a* scalar expression *or a* function *defined over the attributes of that relation. Consider the following (immutable) function which computes the number of skills of a person:*

```
create or replace function numberOfSkills(p integer) returns integer as
$$
    select count(1)::int
    from   personSkill
    where  pid = p;
$$ language SQL immutable;
```

[1] Incidentally, when a primary key is declared when a relation is created, PostgreSQL will create a btree index on this key for the relation.

Then the following is an index defined on the `numberOfSkills` *values of persons:*

```
create index numberOfSkills_index on personSkill (
numberOfSkills(pid));
```

Such an index is useful for queries that use this function such as

```
select pid, skill from personSkill where
numberOfSkills(pid) > 2;
```

1. Consider the relation `Company` populated with thousands of records. For the purposes of this question, let us assume that the attribute `cname` is **not** a primary-key, and has many duplicate values. On the other hand the attribute `headquarter` has relatively fewer duplicates. Which attribute of `Company` is better suited to be indexed? Explain your answer in the context of loading data into memory once the index has been created and a query is executed on the relation.
2. Let us assume a relation `Transaction(tid: int, timestamp: date, amount: float)` in a banking application. Assume that this relation undergoes 10000 modifications(inserts/deletes/updates) every second. What advantages or disadvantages would creating an index on any attribute have on this table? (Note that `tid` is not a primary key here.)
3. Consider the following parameters:

block size	=	4096 bytes
block-address size	=	12 bytes
block access time (I/O operation)	=	15 μs
record size	=	150 bytes
record primary key size	=	10 bytes

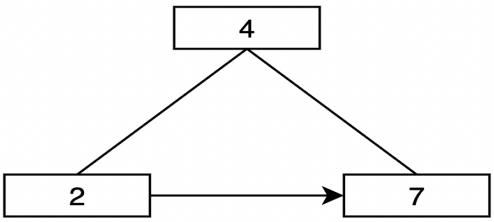
   Assume that there is a B^+-tree, adhering to these parameters, that indexes 2.5 billion records on their primary key values. Provide answers to the following problems and show the intermediate computations leading to your answers.

   (a) Specify (in milliseconds) the minimum time to determine if there is a record with key k in the B^+-tree. (In this problem, you can not assume that a key value that appears in an non-leaf node has a corresponding record with that key value.)

   (b) Specify (in milliseconds) the maximum time to insert a record with key k in the B^+-tree assuming that this record was not already in the data file.

   (c) How large must main memory be to hold the first two levels of the B^+-tree?

4. Consider the following B$^+$-tree of **order 2** that holds records with keys 4, 2, and 7.[2]
   Strategy for splitting leaf nodes: when a leaf node n needs to be split into two nodes n_1 and n_2 (where n_1 will point to n_2), then use the rule that an even number of keys in n is moved into n_1 and an odd number of keys is moved into n_2. So if n becomes conceptually $\boxed{k_1|k_2|k_3}$ then n_1 should be $\boxed{k_1|k_2}$ and n_2 should be $\boxed{k_3|}$ and $n_1 \rightarrow n_2$.



   (a) Show the contents of your B^+-tree after inserting records with keys $8 \rightarrow 9 \rightarrow 3 \rightarrow 6 \rightarrow 12$.

   (b) Show the contents of your B^+-tree after deleting records with keys $9 \rightarrow 7 \rightarrow 3 \rightarrow 2$.

   Note: You need to show the structure of the B^+-tree after an insert/delete operation for **each** key in the order presented in the question. Credit will not be given for directly presenting the final structure of the tree.

[2]Recall that (a) an internal node of a B$^+$-tree of **order 2** can have either 1 or 2 keys values, and 2 or 3 sub-trees, and (b) a leaf node can have either 1 or 2 key values.

Generating data to test Indexes
For the problems below, create appropriate indexes for the `Person (pid, pname) and worksFor(pid, cname, salary)` relations. You need to test the queries below after populating the table with:

(a) 100 records

(b) 1000 records

(c) 10000 records

Generate data using the `insertNewRecords(recSize int)` function present in `dataGen.sql`. This function will populate `Person` and `worksFor` relations with `recSize` randomized records.

For example, after creating the tables `Company, Person and worksFor`, call `insertNewRecords(100)` to create a set of 100 randomized records in the `person and worksFor` relations.

Note: Your task is to illustrate the speedup in these queries. One suggestion is to create a table and list your `create index` query next to the table as follows :-

Record size	Exec. time (without index)	Exec. time (with index)
100		
1000		
10000		

`Appropriate Index: create index ...`

It is recommended that you use the `pgAdmin` GUI to perform these tests as it makes things easy.
Use the `vacuum` command to clear your cache after running an experiment and use the `explain analyze` command to get the execution time of the query.
For example, `explain analyze select * from Person`.
`https://www.postgresql.org/docs/current/sql-explain.html`
`https://www.postgresql.org/docs/current/sql-vacuum.html`

5. Create an appropriate index on the `worksFor` relation that speedups the range query

```
select pid, cname
from   worksFor
where  salary between s1 and s2;
```

Here s1 and s2 are two salaries with s1 < s2.

(a) ***Illustrate this speedup*** by finding the execution times for this query for various sizes of the `worksFor` relation.

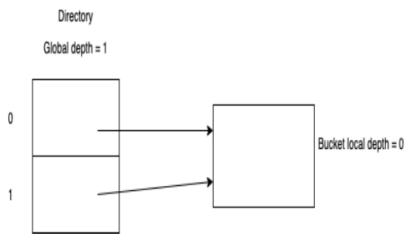
(b) Theoretically speaking, which index (`btree or hash`) is suited better for this query? Explain your reasoning. What happens to the query execution time across different range sizes across s1 and s2?

6. Create indexes on the `worksFor` and `Person` relation that speedup the multiple conditions query

```
select pid, pname
from   Person
where  pid in (select pid from worksFor where cname = c)
```

Here c is the cname of a company. ***Illustrate this speedup*** by finding the execution times for this query for various sizes of the `worksFor` relation.

7. Consider an extensible hashing data structure wherein (1) the initial global depth is set at 1 and (2) all directory pointers point to the same **empty** bucket which has local depth 0. So the hashing structure looks like this:



Assume that a bucket can hold at most two records.

(a) Show the state of the hash data structure after each of the following insert sequences:[3]

i. records with keys 1 and 3.
ii. records with keys 6 and 9.
iii. records with keys 4 and 5.
iv. records with keys 7 and 11.

(b) Starting from the answer you obtained for Question 7a, show the state of the hash data structure after each of the following delete sequences:

i. records with keys 4 and 11.
ii. records with keys 9 and 7.
iii. records with keys 1 and 5.

As in the text book, the bit strings are interpreted starting from their left-most bit and continuing to the right subsequent bits.

8. In practice, several hash functions don't operate as well as expected. Briefly explain the disadvantages of **linear hashing**.

Consider a **linear hash function** for integer keys `i`:

$$h(i) = i^2 mod(B)$$

where `B` is a positive integer that represents the number of buckets and `mod` is the modulo function.

(a) Evaluate `h(i)` for `B = 10`.
(b) Evaluate `h(i)` for `B = 16`.
(c) What would be the best value of B for h(i)?

In your answers, use the intuition you gained from explaining the disadvantages of linear hashing, and use diagrams (if necessary) to support your evaluation.

[3]For Problem 7 - you should interpret the key values as bit strings of length 4. So for example, key value 7 is represented as the bit string 0111 and key value 2 is represented as the bit string 0010.