B561 Assignment 6
Physical database organization, Indexing, Join algorithms & Query processing
Due Date: November 30th

For this assignment, you will need the material covered in

- Conventional Indexes, B^+ Trees, Hashing
- Join algorithms
- Query processing
- Physical Database Organization

In the file `experiments.sql` supplied with this assignment, we have included several PostgreSQL functions that should be useful for running experiments. Of course, you may wish to write your own functions and/or adopt the functions in this .sql to suite the required experiments for the various problems in this assignment. Turn in all answers in `assignment6.pdf`

1 Data Generation and Helper Functions

For all the experiments you need to perform in this assignment, there are predefined helper functions present in `experiments.sql`.
Note that the outcomes of these experiments vary based on a lot of factors like your machine hardware and the memory allocated to PostgreSQL.

For some of the problems below, you will need to 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:

1. 100 records
2. 1000 records
3. 10000 records

Generate data using the `insertNewRecords(recSize int)` function present in `experiments.sql`. This function will populate `Person` and `worksFor` relations with `recSize` randomized records. For the `Company` relation, there is an `INSERT` script present in this file.

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 (ms)	Exec. time with index (ms)
100		
1000		
10000		

`Appropriate Index:  create index <your 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`

2 Indexes

Discussion PostgreSQL permits the creation of a variety of indexes on tables. 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 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;
```

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

1. Consider the relation `Company` populated with thousands of records. 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. `[5 pts]`

2. Consider a relation `Transaction(tid:int, timestamp:date, amount:float)` in a banking application. Assume that this relation undergoes thousands of modifications (`INSERTS|DELETES|UPDATES`) every second.
Discuss the advantages and disadvantages of creating an index on this table. In your discussion, specify what attribute(s) the index is being created on to support your claim. `[5 pts]`
(Note that `tid` is not a primary key here.)

3. 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. ***Illustrate this speedup*** by finding the execution times for this query for various sizes of the `worksFor` relation. `[7 pts]`

4. 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 `[7.5 pts]` .

5. (a) Assume that we have a primary index on the primary key of a table of N records which is maintained as a B^+-tree of order n. `[5 pts]`
   i. Argue why the search time to a record is $O(\log_n(N))$.
   ii. Argue why the insert time of a record is $O(\log_n(N))$.
   iii. Argue why the delete time of a record is $O(\log_n(N))$.

   (b) Consider the following parameters:

block size	=	4096 bytes
block-address size	=	9 bytes
block access time	=	10 ms (micro seconds)
record size	=	200 bytes
record key size	=	12 bytes

Assume that there is a B^+-tree, adhering to these parameters, that indexes $100 = 10^8$ million records on their primary key values.
Show all the intermediate computations leading to your answers. [8 pts]

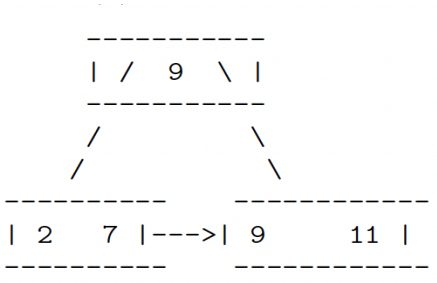
i. Specify (in ms) the minimum time to retrieve a record with key k in the B^+-tree provided that there is a record with this key.

ii. Specify (in ms) the maximum time to retrieve a record with key k in the B^+-tree.

iii. How many records would there need to be indexed to increase the maximum time to retrieve a record with key k in the B^+-tree by at least 20 ms?

iv. How would your answer to question 1(b)ii change if the block size is 8192 bytes.

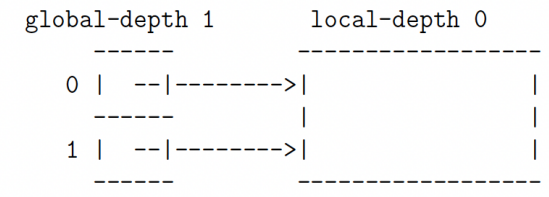
(c) Consider the following B^+-tree of order 2 that holds records with keys 2, 7, 9, and 11. (Observe that (a) an internal node of a B^+-tree of order 2 can have either 1 or 2 keys values, and 2 or 3 sub-trees, and (b) a leaf node can have either 1 or 2 key values.) [5 pts]



i. Show the contents of your B^+-tree after inserting records with keys 6, 10, 14 and 4, in that order.

ii. Starting from your answer in question 1(C)i, show the contents of your B^+-tree after deleting records with keys 2, 14, 4, and 10, in that order.

6. 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** block which has local depth 0. So the hashing structure looks like this:



Assume that a block can hold at most two records.

(a) Show the state of the hash data structure after each of the following insert sequences:[2] [2.5 pts]

i. records with keys 2 and 6.

ii. records with keys 1 and 7.

iii. records with keys 4 and 8.

iv. records with keys 0 and 9.

(b) Starting from the answer you obtained for Question 2(c)i, show the state of the hash data structure after each of the following delete sequences: [2.5 pts]

i. records with keys 1 and 2.

ii. records with keys 6 and 7.

iii. records with keys 0 and 9.

[2] 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.

3 Join Algorithms

7. Let R(A,B) and S(B,C) be two relations and consider their natural join R $\bowtie$ S. Assume that R has 1,500,000 records and that S has 5,000 records. Furthermore, assume that 30 records of R can fit in a block and that 10 records of S can fit in a block. Assume that you have a main-memory buffer with 101 blocks.

   (a) How many block IO's are necessary to perform R $\bowtie$ S using the block nested-loops join algorithm? Show your analysis. `[2 pts]`

   (b) How many block IO's are necessary to perform R $\bowtie$ S using the sort-merge join algorithm? Show your analysis. `[2 pts]`

   (c) Repeat question 6b under the following assumptions. Assume that there are p different B-values and that these are uniformly distributed in R and S.
   Observe that to solve this problem, depending on p, it may be necessary to perform a block nested-loop join per occurrence of a B-value. `[2 pts]`

   (d) How many block IO's are necessary to perform R $\bowtie$ S using the hash-join algorithm? Show your analysis. `[2 pts]`

   (e) Repeat question 6d under the following assumptions. Assume that there are p different B-values and that these are uniformly distributed in R and S.
   Observe that to solve this problem, depending on p, it may be necessary to perform a block nested-loop join per different B- value. `[2 pts]`

4 Generating Binary Relations

Generating binary relations The idea behind generating a set can be generalized to that for the generation of a binary relation.[3] To generate a binary relation of n randomly selected pairs of integers (x, y) such $x \in [l_1, u_1]$ and $y \in [l_2, u_2]$, you can use the function `BinaryRelationOverIntegers`

Example 4 *To generate a binary relation with* 20 *randomly selected pairs with first components in the range* $[3, 8]$ *and second components in the range* $[2, 11]$*, do the following:*

```
select x, y from BinaryRelationOverIntegers(20,3,8,2,11);
```

Generating functions A relation generated by `BinaryRelationOverIntegers` is in general not a function since it is possible that the relation has pairs (x, y_1) and (x, y_2) with $y_1 \neq y_2$. To create a (partial) *function* $f : [l_1, u_1] \to [l_2, u_2]$ of n randomly selected function pairs, we can use the function `FunctionOverIntegers`

[3] Clearly, all of this can be generalized to higher-arity relations.

5 Experiments to Test the Effectiveness of Query Optimization

In the following problems, you will conduct experiments in PostgreSQL to gain insight into whether or not query optimization can be effective. In other words, can it be determined experimentally if optimizing an SQL or an RA expression improves the time (and space) complexity of query evaluation? Additionally, can it be determined if the PostgreSQL query optimizer attains the same (i.e., better or worse) optimization as optimization by hand. Recall that in SQL you can specify each RA expression as an RA SQL query. This implies that each of the optimization rules for RA can be applied directly to queries formulated in RA SQL.

In the following problems you will need to generate artificial data of increasing size and measure the time of evaluating non-optimized and optimized queries. The size of this data can be in the ten or hundreds of thousands of tuples. This is necessary because on very small data is it is not possible to gain sufficient insights into the quality (or lack of quality) of optimization. Additionally, you are advised to examine the query plans generated by PostgreSQL.

For the problems in this assignments, we will use three relations:[4]

```
P(a int)
R(a int, b int)
S(b int)
```

To generate `P` or `S`, you should use the function `SetOfIntegers` which generate a set of up to n randomly selected integers in the range $[l, u]$:

To generate `R`, you should use the function `BinaryRelationOverIntegers` which generates up to n randomly selected pairs with first components in the range $[l_1, u_1]$ and second components in the range $[l_2, u_2]$:

Example 5 *Consider the query* Q_1

```
select distinct r1.a
from   R r1, R r2
where  r1.b = r2.a;
```

This query can be translated and optimized to the query Q_2

```
select distinct r1.a
from   R r1 natural join (select distinct r2.a as b from R r2) r2;
```

Imagine that you have generated a relation `R`*. Then when you execute*

[4] A typical case could be where `P` is `Person`, `R` is `Knows`, and `S` is the set of persons with the Databases skill. Another case could where `P` is the set of persons who work for Amazon, `R` is `personSkill` and `S` is the set of skills of persons who live in Bloomington. Etc.

```
explain analyze
select distinct r1.a
from   R r1, R r2
where  r1.b = r2.a;
```

the system will return its query plan as well as the execution time to evaluate Q_1 *measured in ms. And, when you execute*

```
explain analyze
select distinct r1.a
from   R r1 natural join (select distinct r2.a as b from R r2) r2;
```

the system will return its query plan as well as the execution time to evaluate Q_2 *measured in ms. This permits us to compare the non-optimized query* Q_1 *with the optimized query* Q_2 *for various differently-sized relations* R*. Here are some of these comparisons for various differently-sized random relations* `R`*. In this table,* `R` *was generated with lower and upper bounds* $l_1 = l_2 = 1000$ *and* $u_1 = u_2 = 1000$*.*

`R`	Q_1 `(in ms)`	Q_2 `(in ms)`
10^4	27.03	7.80
10^5	3176.53	58.36
10^6	69251.58	400.54

Notice the significant difference between the execution times of the non-optimized query Q_1 *and the optimized query* Q_2*. So clearly, optimization works on query* Q_1*.*

Incidentally, below are the query plans for Q_1 *and* Q_2*. Examining these query plans should reveal why* Q_1 *runs much slower than* Q_2*. (Why?)*

```
            QUERY PLAN for Q1
-----------------------------------
 HashAggregate
   Group Key: r1.a
   ->  Hash Join
         Hash Cond: (r1.b = r2.a)
         ->  Seq Scan on r r1
         ->  Hash
               ->  Seq Scan on r r2
```

```
              QUERY PLAN  for query Q2
-----------------------------------------
 HashAggregate
   Group Key: r1.a
   ->  Hash Join
         Hash Cond: (r1.b = r2.a)
         ->  Seq Scan on r r1
         ->  Hash
               ->  HashAggregate
                     Group Key: r2.a
```

```
->  Seq Scan on r r2
```

We now turn to the problems for this section.

8. Consider query Q_3

```
select distinct p.a
from   P p, R r1, R r2, R r3, S s
where  p.a = r1.a and r1.b = r2.a and r2.b = r3.a and r.b = S.b;
```

Intuitively, if we view R as a graph, and P and S as node types (properties), then Q_3 determines each P-node in the graph from which there emanates a path of length 3 that ends at a S-node.[5] I.e., a P-node n_0 is in the answer if there exists sequence of nodes (n_0, n_1, n_2, n_3) such that (n_0, n_1), (n_1, n_2), and (n_2, n_3) are edges in R and n_3 is a S-node.

   (a) Translate and optimize this query and call it Q_4. Then write Q_4 as an RA SQL query just as was done for query Q_2 in Example 5.
   [5 pts].

   (b) Compare queries Q_3 and Q_4 in a similar way as we did for Q_1 and Q_2 in Example 5 .
   You should experiment with different sizes for R. Incidentally, these relations do not need to use the same parameters as those shown in the above table for Q_1 and Q_2 in Example 5. [7.5 pts]

9. Consider query Q5

```
select p.a
from   P p
where  exists (select 1
               from   R r
               where  r.a = p.a and
                      not exists (select 1 from S s where r.b = s.b));
```

   (a) Translate and optimize this query and call it Q_6. Then write Q_6 as an RA SQL query just as was done for Q_2 in Example 5. [5 pts]

   (b) An alternative way to write a query equivalent with Q_5 is as the object-relational query

```
with nestedR as (select P.a, array_agg(R.b) as bs
                 from   P natural join R
                 group by (P.a)),
     Ss as (select array(select b from S) as bs)
select a
from   nestedR
where  not (bs <@ (select bs from Ss));
```

   Call this query Q_7.
   Compare queries Q_5, Q_6, and Q_7 in a similar way as we did in Example 5.
   [7.5 pts]

[5] Such a query is typical in Graph Databases.

6 Physical DB Organization

We have learned about *external sorting*. The problems in this section are designed to look into this sorting method as it implemented in PostgreSQL.

10. Create successively larger sets of n randomly selected integers in the range $[1, n]$. You can do this using the `makeS` function.[6]

    This function generates a bag S of size n, with randomly select integers in the range $[1, n]$. Now consider the following SQL statements:

    ```
    select makeS(10);
    explain analyze select x from S;
    explain analyze select x from S order by 1;
    ```

    - The '`select makeS(10)`' statement makes a bag `S` with 10 elements;
    - The '`explain analyze select x from S`' statement provides the query plan and execution time in milliseconds (ms) for a simple scan of `S`;
    - The '`explain analyze select x from S order by 1`' statement provides the query plan and execution time in milliseconds (ms) for sorting `S`.[7]

```
    QUERY PLAN
-----------------------------------------------------------------------------------------------------
 Sort  (cost=179.78..186.16 rows=2550 width=4) (actual time=0.025..0.026 rows=10 loops=1)
   Sort Key: x
   Sort Method: quicksort  Memory: 25kB
   ->  Seq Scan on s  (cost=0.00..35.50 rows=2550 width=4) (actual time=0.004..0.005 rows=10 loops=1)
 Planning Time: 0.069 ms
 Execution Time: 0.034 ms
```

Now construct the following timing table:[8]

size n of relation `S`	avg execution time to scan `S` (in ms)	avg execution time to sort `S` (in ms)
10^1		
10^2		
10^3		
10^4		
10^5		
10^6		
10^7		
10^8		

[6] You should make it a habit to use the PostgreSQL `vacuum` function to perform garbage collection between experiments.

[7] Recall that 1ms is $\frac{1}{1000}$ second.

[8] It is possible that you may not be able to run the experiments for the largest `S`. If that is the case, just report the results for the smaller sizes.

(a) What are your observations about the query plans for the scanning and sorting of such differently sized bags `S`?`[5 pts]`

(b) What do you observe about the execution time to sort `S` as a function of n?`[5 pts]`

11. Typically, the `makeS` function returns a bag instead of a set. In the problems in this section, you are to conduct experiments to measure the execution times to eliminate duplicates.

(a) Write a SQL query that uses the `DISTINCT` clause to eliminate duplicates in `S` and report you results in a table such as that in Problem 10a.`[2.5 pts]`

(b) Write a SQL query that uses the `GROUP BY` clause to eliminate duplicates in `S` and report you results in a table such as that in Problem 10a.`[2.5 pts]`

(c) Compare and contrast the results you have obtained in problems 11a and 11b. Again, consider using `EXPLAIN ANALYZE` to look at query plans.`[2.5 pts]`