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

1. Let us assume that all the records of the relation Company are uniformly distributed across the pages in the disk (secondary memory). Whenever a query is executed on a large relation, the records that match the conditions in the query are fetched from the disk in pages (the number of pages that are fetched depends on the size of the main memory) into the main memory. In this case, since the attribute headquarter has relatively fewer duplicates, it is much "closer to being a key" than cname. Therefore, there are fewer matching records for the query `SELECT * FROM Company WHERE headquarter = ...` than there would be for `SELECT * FROM Company WHERE cname = ...` Since a fewer bag of records implies lesser number of pages to be fetched from the disk, the attribute headquarter is better suited to be indexed and would lead to quicker lookups due to I/O operations being more cost efficient. (To get better insight into this problem - refer Example 8.11 in section 8.4.2 of your book.)

2. Whenever a relation is indexed, there is a compromise on space to achieve faster lookups and quicker query execution times. This is because we need an additional index file that has to be stored on the disk (or main memory) along with the actual data file. The size of the index file is proportional to the data file, and for a relation with trillions of records, we can assume that this index file is large enough that it has to be persisted on a disk. Using that intuition, it is evident that indexing a relation like Transaction comes with a set of disadvantages. Since it is undergoing constant modifications, an index file has to be constantly updated to account for the data being modified. If we assume that this index file is persisted on a disk, there is an additional cost to involved to perform reads and writes the to the index file, along with the actual data. Indexing the Transaction relation may have some advantages in cases where modifications to supplemental relations involve reading records from Transaction. For example, queries like `INSERT INTO SomeRelation SELECT * FROM Transaction WHERE timestamp = ...` there would be some bene-

fit indexing the attribute timestamp since it is:

- Being accessed frequently.
- Very likely to be unique, and could emulate the behavior of primary keys.

An interesting argument that can be made here is that this relation would benefit more from hash indexes rather than btree indexes, the reason being the complexity of performing updates on a hash indexed attribute is $O(1)$ instead of $O(\log_n N)$ for btree indexed attributes.

3. `CREATE INDEX salIdx ON worksFor(salary);`

4. `CREATE INDEX pnIdx ON person(pname);`
   `CREATE INDEX cnIdx ON worksFor(cname);`

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 $O(\log_n(N))$.

   Solution: For the solutions, see the last several slides of the lecture notes on B+-trees.

   (b) Consider the following parameters:

block size	=	4096 bytes
block-address size	=	9 bytes
block access time	=	10 ms (milliseconds)
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]`

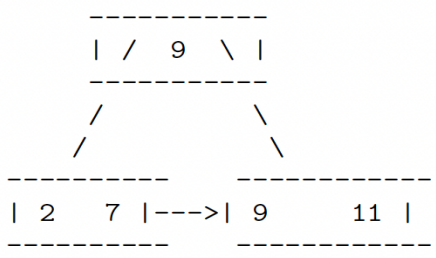
   i. Specify (in ms) the minimum time to retrieve a record with key k in the B^+-tree provided that there is a record with this key.
   Solution: This would happen if the height of the tree increases by 2. So we would then need $98^2 \times 10^8$ records. We first need to determine the order n of the B^+-tree. This can be done by finding the largest n such that $9(n+1)+12n \leq 4096$. That is, $n \leq \frac{4087}{21}$. Thus $n = 194$. The minimum time will be $(\lceil \log_{195}(10^8) \rceil + 1) \times 10$ ms $= 50$ ms. The $+1$ occurs because we need one more block access to the record in the data file. Observe that the minimum time is determined by the largest possible fanout of the nodes in the B^+-tree. This fanout is maximally $n + 1 = 194 + 1 = 195$.

ii. Specify (in ms) the maximum time to retrieve a record with key k in the B^+-tree.
Solution: The maximum time results when we have the minimum branching factor, i.e., $\lceil \frac{194}{2} \rceil + 1 = 98$. Since the minimum branching factor at the root is 2, we must determine the height of a tree that has half of the nodes 2, i.e., $10^8/2 = 5 \times 10^7$ nodes. The height of such a tree will be $\lceil \log_{98}(5 \times 10^7) \rceil = 4$. We then also need one more block access to get the record, so the maximum time is $(4 + 1 + 1) \times 10$ ms $= 60$ ms. So we note that the minimum and maximum times are off by just 1 block access.

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?
Solution: This would happen if the height of the tree increases by 2. So we would then need $98^2 \times 10^8$ records.

iv. How would your answer to question 1(b)ii change if the block size is 8192 bytes.
Solution: In this case, $n \leq \lceil \frac{8183}{21} \rceil = 390$. Doing the same calculation as in question 1(b)ii, we would get $\lceil \log_{391}(5 \times 10^7) \rceil = 4$. So the answer $(4 + 1 + 1) \times 10$ ms $= 60$ ms. This suggests that doubling the size of a block does not necessarily decrease the search time.

(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.
Solution: Refer to B-Tree Solutions File.

6. **Solution**: Refer to B-Tree Solutions File.

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]`
   Solution: The complexity is $B(S) + \frac{B(R)B(S)}{100}$.

$$B(R) = \frac{1500000}{30} = 50000$$

$$B(S) = \frac{5000}{10} = 500$$

   So, $B(S) + \frac{B(R)B(S)}{100} = 500 + \frac{50000 \times 500}{100} = 250500$

   (b) How many block IO's are necessary to perform $R \bowtie S$ using the sort-merge join algorithm? Show your analysis. `[2 pts]`
   Solution: In this case, we first need to determine how much time is involved in sorting R and S. External sorting R takes $2B(R)\log_{100}(B(R))$. This is $2 \times 50000 \times \lceil \log_{100}(5 \times 10^4) \rceil = 300000$. Sorting S takes $2B(S)\log_{100}(B(S))$. This is $2 \times 500 \times \lceil \log_{100}(500) \rceil = 2000$. The merge phase of these sorted files is $B(R)+B(S)$ which is 50000+500 = 50500. So the total is $300000 + 2000 + 50500 = 352500$.

   (c) Repeat question 7b under the following assumptions. Assume that there are p different B-values and that these are uniformly distributed in R and S. `[2 pts]`
   Solution: The time to sort R and S remains the same. So this accounts for $300000 + 2000 = 302000$.

   - Consider the case $p = 1$. Clearly, neither R nor S fit in the buffer. So, we do a block nested-loop join. Since S is the smaller relation and since $p = 1$, this requires $B(S) + \frac{B(R)B(S)}{100}$ block accesses. So, in total, an additional 250500 block accesses.
   - Consider the case $p = 2$. Since now we have 2 B-values, we need 2 block nested-loop joins between a file of size $\frac{50000}{2}$ and a file of size $\frac{500}{2}$. Since S is the smaller relation, we can accomplish each of these in time $250 + \frac{250 \times 25000}{100} = 62750$. Together we have $2 \times 62750 = 125500$ block accesses.
   - Consider the case $p = 3$. Since now we have 3 B-values, we need 3 block nested-loop joins between a file of size $\frac{50000}{3}$ and a file of size $\frac{500}{3}$, so approximately between a file of size 17000 and one of size 170. Since S is the smaller relation, we can accomplish each of these in time $170 + \frac{170 \times 17000}{100} = 29070$. Thus, we need $3 \times 29070 = 87210$ block accesses.
   - The case for $p = 4$ is left as an exercise.

- Consider the case $p = 5$. Since now we have 5 B-values, we need 5 block nested-loop joins between a file of size $\frac{50000}{5} = 10000$ and a file of size $\frac{500}{5} = 100$. Since now each such chunk of S fits in memory, we get, for each such chunk, $100 + 10000 = 10100$ block accesses. Since $p = 5$, we get an extra 50500 block accesses.

The cases for $p > 5$ are similar and are left as an exercise.

(d) How many block IO's are necessary to perform $R \bowtie S$ using the hash-join algorithm? Show your analysis. `[2 pts]`
Solution: For questions 7b and 7d you can assume that the buffer is sufficiently large to hold all records from R and S for any given B-value.

(e) Repeat question 7d under the following assumptions. Assume that there are p different B-values and that these are uniformly distributed in R and S. `[2 pts]`
Solution: The time to hash R and S is $2(B(R) + B(S)) = 101000$. The rest of the analysis for different p values is as in question 7c.

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.[1] 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]`.
Solution:

```
SELECT DISTINCT pr.a
FROM (SELECT DISTINCT p.a, r1.b FROM P p NATURAL JOIN R r1) pr JOIN
     (SELECT DISTINCT r2.a, r2.b FROM R r2 JOIN
         (SELECT DISTINCT r3.a FROM R r3 NATURAL JOIN S s) r3s ON
         (r2.b = r3s.a)) r2s ON (pr.b = r2s.a);
```

(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

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

the above table for Q_1 and Q_2 in Example 5. [7.5 pts]
Solution:
Query plan for Q3

```
QUERY PLAN
-----------------------------------------------
HashAggregate
Group Key: p.a
-> Hash Join
   Hash Cond: (r2.b = r3.a)
   -> Hash Join
      Hash Cond: (r1.b = r2.a)
      -> Hash Join
         Hash Cond: (p.a = r1.a)
         -> Seq Scan on p
         -> Hash
         -> Seq Scan on r r1
      -> Hash
      -> Seq Scan on r r2
   -> Hash
   -> Hash Join
      Hash Cond: (s.b = r3.b)
      -> Seq Scan on s
      -> Hash
      -> Seq Scan on r r3
(19 rows)
```

```
QUERY PLAN
-------------------------------------------------------------
HashAggregate
Group Key: p.a
-> Hash Join
   Hash Cond: (r1.b = r2s.a)
   -> HashAggregate
      Group Key: p.a, r1.b
      -> Hash Join
         Hash Cond: (p.a = r1.a)
         -> Seq Scan on p
         -> Hash
         -> Seq Scan on r r1
   -> Hash
   -> Subquery Scan on r2s
      -> HashAggregate
         Group Key: r2.a, r2.b
         -> Hash Join
            Hash Cond: (r2.b = r3.a)
```

```
                -> Seq Scan on r r2
                -> Hash
                -> HashAggregate
                   Group Key: r3.a
                   -> Hash Join
                      Hash Cond: (s.b = r3.b)
                      -> Seq Scan on s
                      -> Hash
                      -> Seq Scan on r r3
(26 rows)
```

Following are the differences between query plan for Q3 and Q4 that impacts the performance of the queries:

- One of the main differences between query plans of Q3 and Q4 is that there is duplicate elimination after every join, that results in reducing the size of joined relation. This in turn reduces the time required to compute other joins. Hence reducing the overall time required to run the query.
- Further, the order in which joins are done can greatly impact the performance of the query. We can observe in Q3 the joins occur in the order in which they are present in the from clause. Whereas, in the query plan for Q4 we can observe a different order of joins is forced due to manual optimizations.

Due to these reasons, we can see significant improvement in performance of the query.

Experiment on various sizes of relation R: Relation R was generated with values in the range $[0, 1000]$ for both attributes a and b. The size of relations P and S were kept constant and the query was run for varying sizes of relation R. Following table summarizes the observations where Q3 represents the pure sql query and Q4 represents the optimized RA SQL query.

size n of relation R	Q3	Q4
10^3	141.723 ms	12.411 ms
10^4	673.596 ms	24.108 ms
10^5	464933.732 ms	146.160 ms

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]
Solution:

```
SELECT q.val
FROM (
    SELECT r.a AS val, r.b
    FROM P p NATURAL JOIN R r
    EXCEPT
    SELECT r1.*
    FROM (SELECT DISTINCT r.* FROM P p NATURAL JOIN R r) r1 NATURAL JOIN S s
) q;
```

(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]
Solution:

```
EXPLAIN (COSTS OFF)
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));
```

QUERY PLAN for Q5 ———————————————

- A Hash Semi Join is performed on relations P and RS
- RS is a relation formed by running a Hash Anti Join on relations R and S
- Hence this query gets executed in Linear time of P, R, and S i.e., $O(P + R + S)$

QUERY PLAN for Q6

```
EXPLAIN (COSTS OFF)
SELECT q.val
FROM (
    SELECT r.a AS val, r.b
```

```
    FROM P p NATURAL JOIN R r
    EXCEPT
    SELECT r1.*
    FROM (SELECT DISTINCT r.* FROM P p NATURAL JOIN R r) r1 NATURAL JOIN S s
) q;
```

- A HashSetOp Except is performed on relations RP and PRS
- RP is a relation formed by running a hash join on relations R and P; runs in $O(R+P)$
- PRS is a relation formed by hash join of S and PR; PR is formed by hash-join of P and R with duplicate elimination
- Overall time complexity: $O(|P{+}R|{+}|P{+}R{+}S|{+}$duplication elimination$)$

QUERY PLAN for Q7

```
EXPLAIN (COSTS OFF)
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));
```

- NestedR calculated using hash join between P and R and sort-based aggregation on p.a; runtime: $O(P+R+PR\log PR)$
- Ss is a sequential scan on S; runs in $O(S)$
- Overall complexity: $O(P+R+S+PR\log PR)$

10. Create successively larger sets of n randomly selected integers in the range $[1,n]$. You can do this using the `makeS` function.[2]

    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`;

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

- The '`explain analyze select x from S order by 1`' statement provides the query plan and execution time in milliseconds (ms) for sorting `S`.[3]

```
   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:[4]

size n of relation `S`	avg execution time to scan `S` (in ms)	avg execution time to sort `S` (in ms)
10^1	0.019	0.026
10^2	0.027	0.046
10^3	0.068	0.107
10^4	0.540	1.001
10^5	5.346	11.588
10^6	52.025	143.198
10^7	793.907	1869.766
10^8	10611.758	18183.824

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

- The sorting operation involves additional steps compared to the scan. Initially, for smaller sizes, the database likely performs an in-memory sort (e.g., quicksort), which is relatively fast.
- As the size of `S` increases, the database might switch to disk-based sorting algorithms (like external merge sort) due to memory constraints, which are considerably slower.
- The significant jump in execution time for larger sizes (especially from 10^6 onwards) suggests that the database is dealing with large amounts of data that exceed the available memory, causing it to rely on disk storage, which is slower than in-memory operations.

(b) What do you observe about the execution time to sort `S` as a function of n? `[5 pts]`
Solution: Execution Time for Sorting as a Function of n:

[3] Recall that 1ms is $\frac{1}{1000}$ second.
[4] 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.

- The execution time for sorting increases more rapidly than that for scanning. This is expected since sorting is a more complex operation with a higher computational complexity.
- For small n values (up to 10^4), the increase in execution time is moderate. However, beyond 10^5, there is a sharp increase in execution time. This indicates a non-linear relationship between the size of n and the sorting time, likely exponential or polynomial.
- The drastic increase in sorting times for larger values of n (10^7 and 10^8) can be attributed to the increased use of disk storage for sorting operations, as well as the inherently higher complexity of sorting compared to simple scanning.
- This pattern is typical in database operations where in-memory operations are significantly faster, but as the data size grows beyond what can be comfortably held in memory, disk I/O becomes a bottleneck, leading to much longer execution times.

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]`

    ```
    Select DISTINCT x from S;
    ```

    (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]`

    ```
    Select x from S Group By(x);
    ```

    (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]`

size n of relation `S`	scan S Distinct (in ms)	scan S Group By (in ms)
10^1	0.050	0.041
10^2	0.077	0.064
10^3	0.369	0.337
10^4	3.556	3.835
10^5	44.893	42.911
10^6	990.927	589.225
10^7	9404.149	6580.447
10^8	135681.713	96334.483

Solution: Compare and Contrast:

- The execution time for eliminating the duplicates using the Group By method usually takes less time than the Distinct method.

- For small values of n (up to 10^4) the execution time slightly differs because both queries use HashAggregation with a time complexity of $|n|$.
- As the size of S increases, the database might switch to disk based sorting algorithms (like external merge sort) due to memory constraints, which are considerably slower.
- We see a sharp increase in execution time for larger values of n (after 10^6). This is because the database runs out of memory and uses External Merge Sort which is much slower.