B561 Advanced Database Concepts Assignment 6
Fall 2023

Submitted by : Atharv J(ajangam)

Q1)
Answer)
When a query is run following the creation of an index, the process doesn't comb through the whole dataset. Instead, the index acts like a map, guiding the system straight to the specific data location, which it then retrieves into memory. When the indexed attribute has numerous duplicates, the system must retrieve and sift through all matching records, which can be time-consuming. Conversely, if the attribute is more unique with fewer duplicates, the index leads to fewer records, making the data retrieval process more efficient. Therefore, choosing an attribute with unique values for indexing enhances overall efficiency.

Q2)
Answer)
Indexing a database involves a trade-off between storage space and faster data retrieval. The process requires an additional index file, stored alongside the actual data file, which increases the overall storage requirement. The size of the index file grows with the size of the data file. For databases with trillions of records, this index file can be substantial enough to necessitate storage on a disk.

Particularly for a database like Transaction, which is frequently updated, indexing introduces specific challenges. Each modification in the data necessitates an update in the index file, adding overhead for disk read and write operations. However, in certain scenarios, such as when transactions are frequently queried based on specific attributes like timestamps, indexing can offer significant performance benefits. Timestamps are often unique and heavily queried, making them ideal candidates for indexing.

Moreover, the type of indexing can greatly influence efficiency in this context. Hash indexes are more suitable for this scenario than B-tree indexes. This is because updating a hash index typically has a constant time complexity (O(1)), whereas updating a B-tree index involves a logarithmic time complexity (O(log N)). This makes hash indexes particularly advantageous for databases with frequent updates, like the Transaction database, where the rapid update capability of hash indexes can significantly improve performance.

Q3)
Answer)

We should use Btree index rather than hash index as we need to get values between two and hash index does not support range between two values.

Creating index:

```
create index sal_Index on worksFor(salary);
```

Record size	**Exec time w/o index**	**Exec time with index**
100	0.203	0.104
1000	0.452	0.212
10000	1.101	0.689

Q4)
Answer)

We should use Hash indexes for this query because they are optimized for retrieving specific values, unlike the previous query which required searching within a range.

Creating Index:

```
create index cn_Index on worksFor using hash(cname);

create index p_Index on Person using hash(pid);
```

Record size	**Exec time w/o index (ms)**	**Exec time with index (ms)**
100	0.305	0.156
1000	0.401	0.311
10000	1.983	0.945

Q5)

a)

1) We know that in B+ tree Search Time is O(log_n(N)): In a B+-tree, a search operation involves traversing from the root to the leaf node. The height of a B+-tree is determined by the order of the tree and the number of records it contains. For a tree of order n, each node (except the root) can have a minimum of n/2 and a maximum of n children.

   The height of the B+-tree, h, is roughly log_n(N), where N is the number of records. This is because each level of the tree can hold up to n times more elements than the level above it.

   During a search, at most one node per level of the tree is accessed. Therefore, the search time is proportional to the height of the tree, leading to a time complexity of O(log_n(N)).

2) Insert Time is O(log_n(N)):Insertion in a B+-tree involves two steps: finding the correct position to insert and then inserting the record.

   The initial search to find the correct leaf node where the new record should be inserted takes O(log_n(N)) time, as discussed above.

   After finding the correct position, the record is inserted. In the worst case, this might require splitting the node if the node is already full. However, even with potential splitting and rebalancing, the operation affects only a logarithmic number of nodes (along the path from the leaf to the root), keeping the time complexity O(log_n(N)).

3) Delete Time is O(log_n(N)):Deletion in a B+-tree starts with searching for the record to be deleted, taking O(log_n(N)) time.

   Once the record is found, it is deleted. If the deletion causes a node to have fewer entries than the minimum required, additional steps like merging or redistributing entries with a sibling node are performed.

   Like insertion, these additional steps might require adjustments up to the root of the tree. However, since they involve only a logarithmic number of nodes in the path from the leaf to the root, the time complexity remains O(log_n(N)).

b)

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

$$9(n + 1) + 12n \leq 4096$$
$$n \leq 4087/21 \leq 195$$

1) Min time = (log_n(N) + 1) * block access time
   = $([log_{195}(10^8)\,] + 1) * 10ms$
   = $5 * 10ms$
   = $50ms$

2) Branching factor = [194/2] +1
   = 98

   Height for the tree:
   = $[log_{98}(5 * 10^7)$
   = 4

   Max time :

   = $(Height + 2) * 10ms$
   = $(4 + 2) * 10ms$
   = $60ms$

3) To increase the max retrieval time by 20ms, we need to increase the height of the tree by 2

   New height = 4+2 = 6

   $$log_{98}(x) = 6$$
   $$x = 98^6$$

4) New block size = 8192

   $$9(n + 1) + 12n \leq 8192$$
   $$21n \leq 8183$$
   $$n \leq 8183/21 \approx 389$$

Branching factor :

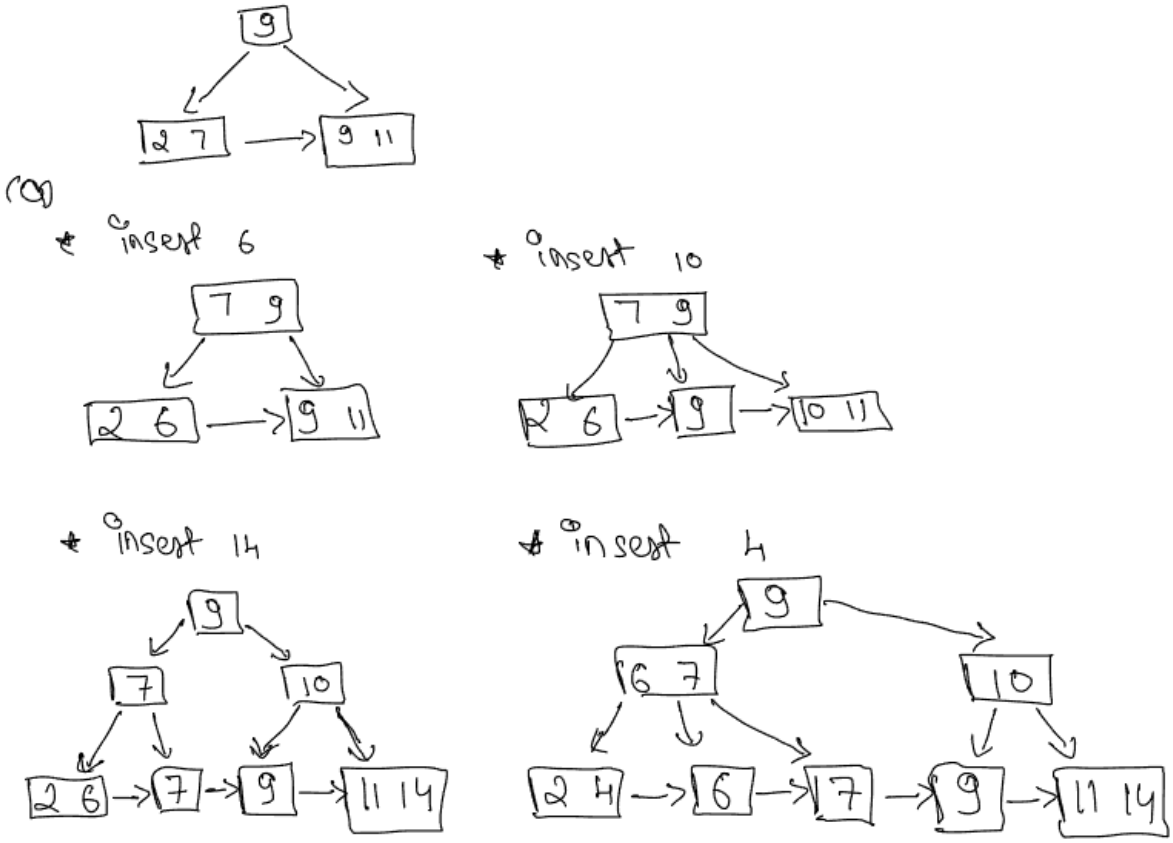
$$= [389/2] + 1$$
$$= 196$$

Height = 4

Max time = (4+2) * 10ms

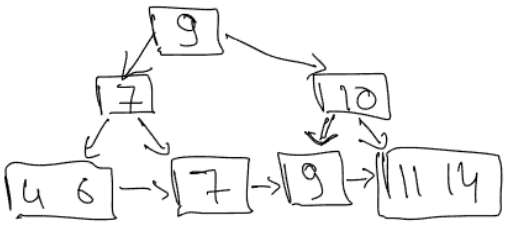
Therefore the max time does not change.

c)

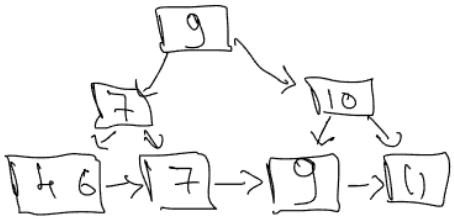


(b) Deleting the record (2, 14, 4, and 10)

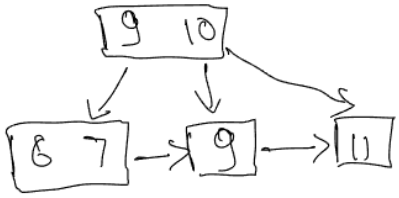
* Delete 2



* delete 14



* Delete 4



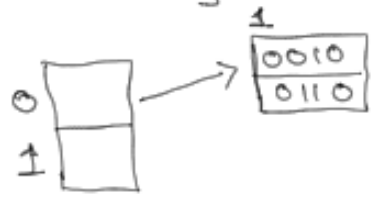
* delete 10

Q6)

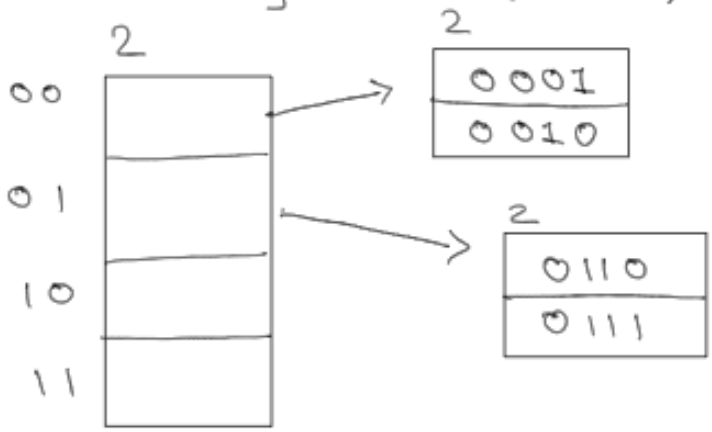
86

(a) Insert sequence.

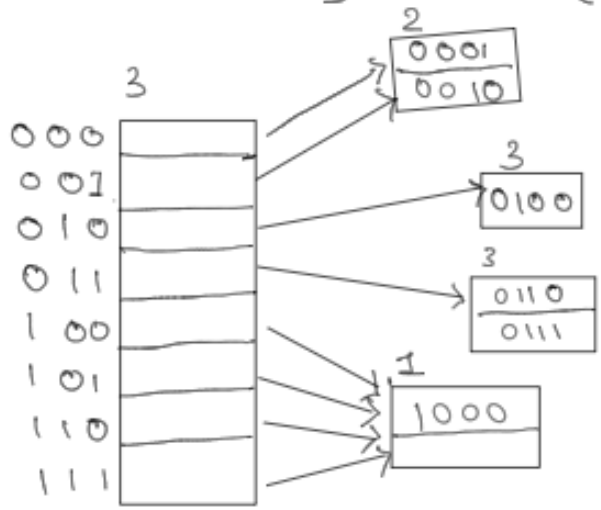
(i) record with key 2 and 6 (0010, 0110)



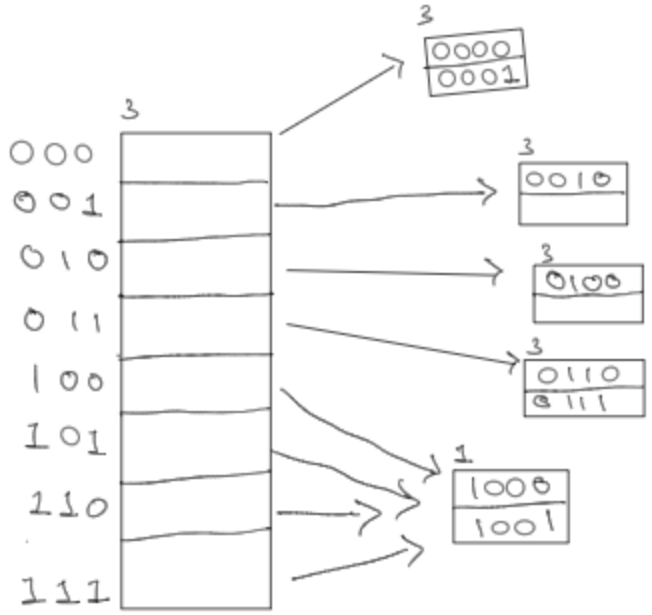
(ii) record with key 1 and 7 (0001, 0111)



(iii) record with key 4 and 8 (0100, 1000)

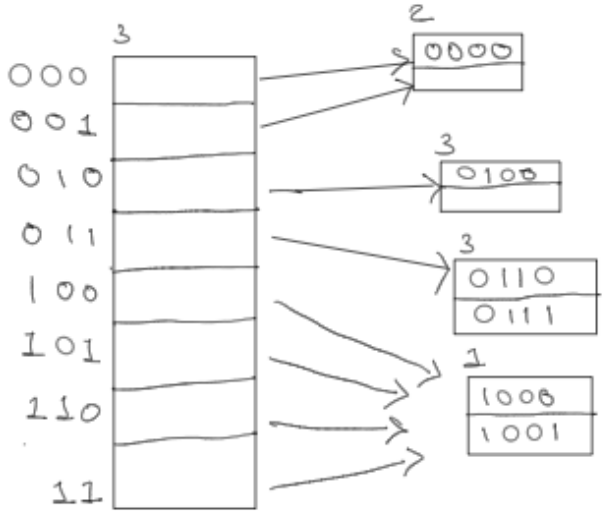


(iv) records with key 0 and 9 (0000, 1001)

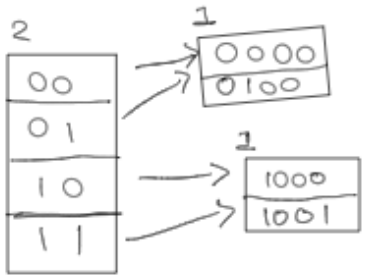


(b) Deleting

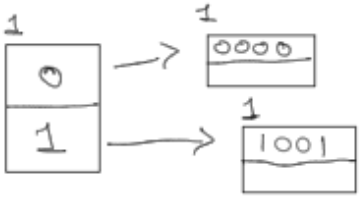
(i) records with key 1 and 2 (0001, 0010)



(ii) records with key 6 and 7 (0110, 0111)



(iii) record with key 0 and 9. (0000, 1001)



Q7)

Answer)

Given:

R = 1,500,000 records
S = 5000 records
M = 101
n(R) = 30
n(S) = 10
b(R) = 1,500,000/30 = 50000 blocks
b(S) = 5000/10 = 500 blocks

a) Block Nested Loops Join: b(R) +b(R)*b(S)/M

Total I/Os = 50000 + (50000*500)/101
= **297525.**

Hence Block Nested Loops Join takes around 297525 I/O Operations.

b) Sort-Merge Join: b(R) +b(S) + 2*b(R)*ceil(logM(b(R))) + 2*b(S)*ceil(logM(b(S)))

Total I/Os = 50000 + 500 + 2*50000*ceil(log101(50000)) + 2*500*ceil(log101(500))

= **352500**

Hence Sort-Merge Join takes around 352500 I/O Operations.

c) Optimized Block Nested Loops Join with p partitions:

Let's take p = 5,
R = 150000/5 = 300000 records
S = 5000/5 = 1000 records
M = 101

b(R) = 300000/30 = 100,000 blocks
b(s) = 1000/10 = 100 blocks

Number of IO operation = b(R) +b(S) + 2*b(R)*ceil(logM(b(R))) + 2*b(S)*ceil(logM(b(S)))

= **176250**

d) Hash Join: 3 * (b(R) + b(S))

Total I/O = 3 * (50000 + 500)
= **151500**

e) Optimized Hash Join with p partitions:
b(R) = 750,000/30 = 25000
b(S) = 2500/10 = 250

Total I/O = 3* (25000+250)
= **75750**

Q8)

Answer)

Given, Q3 =

```
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;
```

Let us translate this query to RA SQL by first introducing joins

RA SQL :

```
select distinct p.a
from P p JOIN R r1 ON p.a = r1.a
JOIN R r2 ON r1.b = r2.a
JOIN R r3 ON r2.b = r3.a
JOIN S s ON r3.b = S.b;
```

Optimized RA SQL:

```
select distinct p.a
from P p NATURAL JOIN R r1
NATURAL JOIN (select distinct r2.a AS b from R r2) r2
NATURAL JOIN (select distinct r3.a AS b from R r3) r3
NATURAL JOIN S s;
```

b)
We have Relation R with values in the range [100,100000] for both the attribute a and b.

The query execution time is given below for both Q3 and Q4:

Size	Q3 time(in ms)	Q4 time(in ms)
100	0.458	0.197
1000	1.573	1.012
10000	215.659	9.832
100000	25788.502	116.332

We can see that query Q4 is much faster than Q3 as it uses hash joins and hash aggregate instead of merge join. Also, the difference in the execution plans of Q3 and Q4 is observed in how they handle duplicate removal after each join. This process essentially reduces the size of the resulting joined datasets, thereby lessening the time required for any following joins. This approach can play a significant role in diminishing the overall execution time of the query.

Question 9)

a)

Given 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));
```

Translated and optimized Q5, let's call is Q6:

```
Q6:
select q.ra
from (
select r.a as ra, 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)

Query Q7:

```
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));
```

Table for comparing Q5, Q6, and Q7:

Size	**Q5 time(ms)**	**Q6 time(ms)**	**Q7 time(ms)**
100	0.129	2.047	0.817
1000	0.449	1.788	1.287
10000	1.379	14.344	13.772
100000	12.612	258.623	158.247
1000000	248.530	3239.372	2635.301

Question 10)

a)

Size n of relation S	**Avg execution time to scan S (in ms)**	**Avg execution time to sort S (in ms)**
10	0.015	0.023
100	0.021	0.024
1000	0.061	0.100
10000	0.512	0.735

100000	4.982	7.213
1000000	52.869	108.750
10000000	485.552	1484.325
100000000	5021.787	25982.205

Scanning runs on a linear time scale, which means that the execution time grows in direct proportion to the amount of the dataset. Sorting, on the other hand, has a time complexity of O(nlog(n)), making it fundamentally slower than scanning.

Furthermore, as the dataset size increases, the time required for sorting increases dramatically. This is due to the requirement for more buffer pages, which increases the time required for reading and writing data.

b)

The time taken to sort is somewhat proportional to nlog(n). This is because the number of buffer pages are limited, as the dataset is small enough to fit within the available buffer pages, the sorting can be efficiently handled in memory. However, as the size of the dataset increases beyond the capacity of these buffer pages, the sorting operation becomes more complex. To ensure the fast execution we require more buffer pages than are available.

Question 11)

a)

select distinct from S;

Size n of relation S	**execution time (in ms)**
10	0.023
100	0.034
1000	0.213
10000	3.156
100000	29.587
1000000	555.262

10000000	6327.973
100000000	68432.039

b)

select a from S group by (a);

Size n of relation S	**execution time (in ms)**
10	0.025
100	0.041
1000	0.374
10000	5.778
100000	33.021
1000000	496.743
10000000	6167.257
100000000	66002.031

c)

We've observed that the query plans for distinct and group by queries, which aim to remove duplicates from a dataset, are somewhat similar. Consequently, the execution times for these queries are nearly identical. Upon examining the query plans, we've noticed that both queries involve placing the dataset form S into a hash table using a specific key, denoted as a This approach ensures that duplicate records are hashed into the same bucket within the hash table, allowing for efficient duplicate elimination.