CSEN603
Assignment II

Omar Sherif
Amir George

Question 1

Hash Joins

We discuss three variations of the Hash Join.

- Simple Hash Join
- Grace/Partition Hash Join
- Hybrid Hash Join

We use example relations S with m tuples and R with n tuples where S is the smaller relation, and M memory buffers where each buffer can hold a relation block and A,B are S and R's join attributes respectively.

Simple Hash Join

Partition Phase Relation S is partitioned using a hash function h on the join attribute into $M-1$ buckets by loading S's blocks one by one into the remaining buffer emptying and saving each bucket to secondary memory as it is filled.

Join/Probe Phase Having S's $M-1$ buffers in memory R's blocks are loaded one by one in the remaining buffer and each tuple r in the block is applied to h on it's join attribute A,for each tuple s in bucket $h(r.A)$ if $s.B = r.A$ where B is the join attribute tuple $r \circ s$ is written to the output buffer.

Hashing S

```
foreach tuple s ∈ S do
    put s in bucket h(s.A)
end
```

Probing

```
foreach tuple r ∈ R do
    foreach tuple s in bucket h(r.A) do
        if r.A=s.B then
            output r ∘ s
        end
    end
end
```

Two relations Guest(Name,Roomno),Room(Roomno,telephone),we assume 4 buffers of memory are available and $h(x) = x \, mod \, 3$,where a block holds two tuples

Name	Roomno
Osama	2
Ahmed	9
Yehia	55
Bigby	47
Wander	33

Roomno	telephone
22	21849
2	21239
55	2354
11	211222
15	23521
9	143556

The result of Hashing *Guest*

0	1	2	3
Ahmed 9 Wander 33	Yehia 55	Bigby 47 Osama 2	

after loading the first block of Room

0	1	2	3
Ahmed 9 Wander 33	Yehia 55	Bigby 47	22 21849 2 21239

22 is hashed to bucket 1 but since the only tuple in that bucket doesn't have the same value of the join attribute no joining is done,but 2 is found in bucket 2 and joined and tuple **Osama 2 21239** is outputed,then the next block of room is loaded and the process continues.

Grace/Partition Hash Join

Partition Phase Relation S is partitioned using a hash function h on the join attribute into $M - 1$ buckets by loading S's blocks one by one into the remaining buffer emptying and saving each bucket to memory as it is filled,Relation R is partitioned in the same way,the hash function h ensures that tuples with the same value for the join attribute end up in the i_{th} bucket in the two relations.

Join/Probe Phase Going from 1 to $M - 1$,$min(R_i, S_i)$ according to number of blocks where X_i is the i_{th} partition of X is loaded to memory and block by block of S_i is loaded to memory where every tuple s of S_i is compared with every tuple r of R_i and $r \circ s$ is outputted whenever $s.A = r.B$ where A and B are S and R's join attribute respectively.

Hashing S **and** R

```
foreach tuple s ∈ S do
    put s in bucket h(s.A)
end
write buckets to secondary memory
foreach tuple r ∈ R do
    put r in bucket h(r.B)
end
write buckets to secondary memory
```

Probing

```
for k = 1 to M − 1 do
    X ← min(S_k, R_k)
    Y ← max(S_k, R_k)
    foreach tuple x ∈ X do
        foreach tuple y ∈ Y do
            if x.A=y.B then
                output x ∘ y
            end
        end
    end
end
```

Nested Loop Joins

The simplest join method where S is loaded into memory $M-1$ blocks at a time in $\lceil \frac{B(S)}{M-1} \rceil$ iterations where $B(S)$ is the number of blocks of S and R is loaded block at a time in the remaining buffer so that each tuple of S is compared to each tuple of R and the tuples where the join attribute is equal are outputed.

Nested Loop Join Algorithm

```
foreach M − 1 blocks of S do
    foreach block b of R do
        foreach tuple t in b do
            foreach tuple s in the M-1 blocks loaded do
                if t.B=s.A then
                    output r ∘ s
                end
            end
        end
    end
end
```

Merge Joins

The Merge Join takes two relations and sorts them if not already sorted it loads one block of each relation and takes the minimum first element according to the join attribute with value m,then checks wether m is present at the top if the second relation,if not tuples with m are dropped and the process repeats,if m is indeed found in the second relation,all tuples having m in the second relation are joined with the initial tuple loading as many blocks as necessary,the process repeats until one relation is exhausted.

Merge Join Algorithm

```
set i ← 1, j ← 1
while (i ≤ n) and (j ≤ m) do
    if R[i].A > S[i].B then
        set j ← j + 1
    end
    else if R[i].A < S[i].B then
        set i ← i + 1
    end
    else
        output R[i] ∘ S[i]
        set z ← j + 1
        while (z ≤ m) and (R[i].A = S[z].B) do
            output R[i] ∘ S[z]
            set z ← z + 1
        end
        set k ← i + 1
        while (k ≤ n) and (R[k].A = S[j].B) do
            output R[i] ∘ S[k]
            set k ← k + 1
        end
        set i ← k, j ← z
    end
end
```

Question 2

Bitmap Indices

A Bitmap index is constructed for a single value v of a single column in a relation R where R's tuples are numbered from 0 to n ,The index for v is a vector of n bits where if the n_{th} tuple of R has value v then the n_{th} bit of the vector is a 1 and 0 otherwise,as an example consider the relation Student.

Name	Age	Hobby
Osama	21	swimming
Ahmed	29	hiking
Ayman	20	reading
Bassem	22	swimming
Yasser	22	reading
Mostafa	21	reading

indices for Age :
20 : 001000
21 : 100001
22 : 000110

indices for Hobby :
swimming : 100100
reading : 001011
hiking : 010000

to add efficieny to Bitmaps run length encoding and other compression techniques can be used to minimize the number of bits as much as possible,for deletion and insertion properties the number associated with tuples is considred to be constant so that when inserting a new tuple in a relation simply a new bit is added to all it's bitmap indices and when a tuple is deleted a tombstone is inserted in it's position in all it's bitmap indices,another way of handling indices is by using *existence bitmap* [**?**] which is a bitmap similar to the others but it's n_{th} bit is 0 when the n_{th} row is deleted.

Bitmap indices are best used when

1. Insertion/Deletion rate is low.

2. When number of distinct values of a column are low "If the number of distinct values of a column is less than 1% of the number of rows in the table, or if the values in a column are repeated more than 100 times, then the column is a candidate for a bitmap index."