CS 441: Midsemester Exam, Fall 2000

Write your ID on the top left of every page.
This is a closed-book exam.
Return the first two pages with the rest of your answer script.

I. Fill the blank (if any) and complete the following sentences in the space below. ...20

1. It is / is not possible in general to have two primary indices on the same relation for different search keys because

ANS: In a sequentially ordered file the, the index whose search key specifies the sequential order of the file is the *primary index*. As it is not possible to sequentially order a file on two different keys, you cannot have two primary indices on a file.

2. Latency time / data transfer rate is affected by the decision to keep related data together on a disk because

ANS: Two possible answers to this are

- *Latency time* is affected if data is clubbed together in the sectors of the same track.
- Over all *data transfer rate* is affected if related data occupy consecutive sector/cylinders.

3. I would / would-not recommend a hashed file organization for a relation where the hash function is computed on the primary key attribute of the relation because

ANS: A hash function computed on the primary key of an attribute will result in very little collisions and would be very desirable.

4. An example of a database application in which the pointer method of representing variable-length records is preferable to the reserved-space method is

because

ANS: the length of each record record will vary and reserving fixed space for the record will cause a large amount of space to be wasted as the average size of the records will be much smaller then the reserved size. This will be saved in the pointer method of storing the records at the cost of additional indirection.

5. When doing merge–join, the preferred buffer management algorithm is

ANS: **MRU**

6. If L_i and L_j are leaf nodes in a B+ tree index and $i > j$, then every search key value in L_i is $\underline{\quad > \quad}$ every search key value in L_j because

7. A significant advantage of a B+ tree index over an extendible hashing index for an attribute A of a relation R is

ANS: It can handle **range queries** efficiently.

8. A significant advantage of an extendible hashing index over a B+ tree index for an attribute A of a relation R is

ANS: **Point** queries can be handled efficiently.

9. The query processing cost of joining three relations is always/never/sometimes smaller than that for joining four relations

ANS: **Sometimes**

10. If I could recommend one "change" to this course, it would be

II. A large data warehouse is being designed to be continuously available, that is, without *any* down-time. You are asked to choose between (a) mirrored disks and (b) block interleaved distributed parity RAID. ...10

1. which would you choose and why?

ANS: *Mirrored* disks. In normal read operation both mirrored and RAID 6 give similar performance. But after a failure a read on the damaged disk in RAID 6 will require all the disk to be read, While in the case of mirror only the failed disk's mirror has to be read. Moreover when the RAID 6 disk is written to besides the data being written, the parity block has to be read and then rewritten. This extra read is not required in mirrored disk.

2. would your answer change if you are told, in addition, that the warehouse is updated very rarely.

ANS: If the warehouse is updated very rarely then RAID 6 is a better choice as its utilization of disks is better than that of mirrored disks.

III. Consider the following self-explanatory relational database schema: ...40

`Student (ID, name, dept-no, hostel-no, ...)` (ID is a key)
`Department (dept-no, building, ...)` (dept-no is a key)
`Hostel (hostel-no, warden-name, ...)` (hostel-no is a key)

1. Write a relational algebra expression that returns each student's ID along with the building for that student's department.

Ans: $\Pi_{\texttt{ID,building}}(\texttt{Student} \bowtie \texttt{Department})$

2. Write an SQL query that returns the same result.

Ans:
```
select S.ID ,D.building
from Student S, Department D
where S.dept-no=D.dept-no
```

3. Suppose there are 5000 students, 100 departments, and 10 hostels, and suppose that in the underlying storage manager for these relations, 10 student tuples can fit in one block, 25 department tuples can fit in one block, and 15 hostel tuples fit in one block. If a straightforward block nested loop join algorithm is used to compute the query from part (2), approximately how many disk block reads can be expected?

   You may assume: (i) the relations are not sorted and there are no indices. (ii) relation `student` is used as the "outer" relation in the nested loop join algorithm. (iii) a block is read into the buffer pool every time it is needed, and blocks are pinned (fixed) while they are in active use. (In other words, you needn't consider the size of the buffer pool or the replacement strategy in computing your answer.) Show your computation, and state any additional assumptions that you make.

Ans: $b_{\texttt{student}} = 500$
$b_{\texttt{department}} = 4$
Considering relation Student to be the outer relation of the block nested loop. No of block accesses will be.

$$\begin{aligned} &= b_{\texttt{student}} + b_{\texttt{department}} * b_{\texttt{student}} \\ &= 500 + 500 * 4 \\ &= \boxed{2500} \end{aligned}$$

4. Use the same scenario as part (3), but drop assumption (i) and assume in its place: (ia) Relation `Student` is sorted on attribute `Student.dept-no.` and (ib) Relation `Department` has a dense index on attribute `Department.dept-no.` If the nested loop join algorithm capitalizes on these new assumptions in the obvious way, approximately how many block reads can be expected? Show your computation, and state any additional assumptions that you make.

Ans: On an average there are 50 `student` tuples per `department` tuple.
Since the `student` is sorted on `dept-no` as a result of this the index into `department` would

have to be looked up only when `dept-no` of the student changes .There will be a total of 100 such changes (There are 100 different departments) Total block accesses.

500+1*100+1=601 (considering index block is read only once.)

500+2*100=700 (considering index block is read only every time.)

5. Use the same scenario as part (4), but assume that a sort-merge join algorithm is used to compute the query. Approximately how many block reads can be expected? Show your computation, and state any additional assumptions that you make.

ANS: The relation `student` is sorted but `department` is not. The two possibilities are

- Sort `department` and then perform the merge join. The total block reads will be block reads required for merging $(500 + 4)$ and the amount required for the sorting of `department`.

  To sort `department` you require at least two blocks (for merging). Thus the 4 blocks of `department` can be sorted in 16 block accesses. Total block accesses/reads.

  $500 + 4 + 16 = 520$

- Or use a scan on the index to provide the merge operator with tuples from `department` in sorted order.

  A read of the `department` relation will be done each time the `dept-no` in student changes. So besides the 500 blocks for `student`, 101/200 blocks will be read from `department` depending on weather the index block is pinned or not.(Assuming index occupies 1 block)

  $500 + 101 = 601$ (index block is pinned)

  $500 + 200 = 700$ (index block is not pinned)

6. Use the same scenario in (3) but assume that you have a buffer pool with 4 blocks. How would your answer to (3) change?

ANS: Consider `student` relation to be in the outer loop, 1 buffer for `department`, 2 for `student` and 1 for output.

Now for every 2 blocks of outer loop 4 block of `department` would be read. Thus we get.

$$\frac{500}{2} * 4 + 500 = 1500$$

7. Suppose the most often executed query is "which students stay in hostel n_1 through n_2.

   Consider two different requirements: (a) minimize additional storage costs, and (b) minimize processing cost.

   For each of these requirements how would you store the `Student` relation (also indicate which indices you would maintain, if any)?

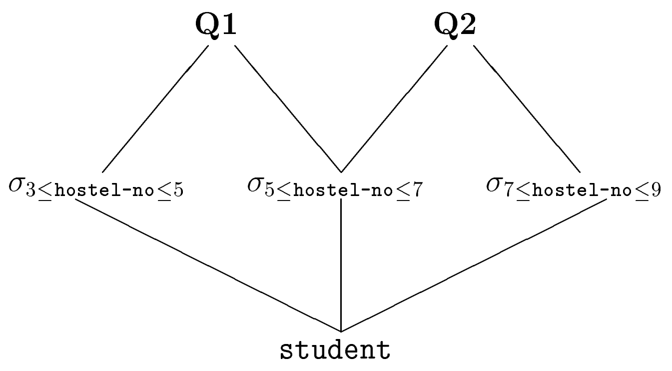
ANS: (a) Minimize additional storage The relation `student` should be sorted on the attribute `hostel-no`. This incurs no additional storage cost but ensures efficient evaluation of the given query.

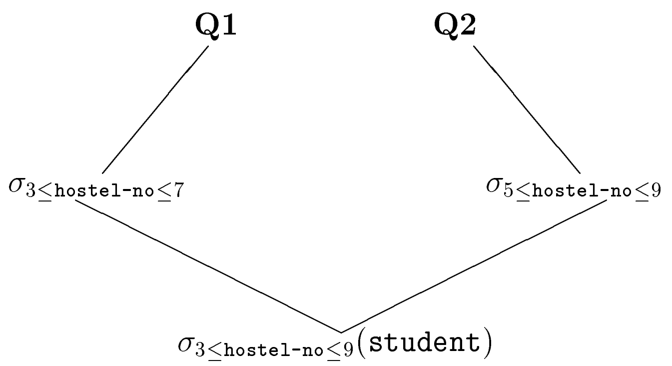
(b) Minimize processing cost If in addition to sorting the relation `student` this we add a sparse index `hostel-no`, processing cost will be minimized.

8. Two such queries are posed together. The first is interested in students staying in hostels 3 through 7 and the second in students staying in hostels 5 through 9.

   How would you optimize these two queries given that they are posed together? Show the query graph that will be used for the final query processing.

ANS: Two possible optimized query graphs are.





IV. Consider the following equivalence:

$$\pi_{L_1\cup L_2}(E_1 \bowtie_\theta E_2) = \pi_{L_1\cup L_2}((\pi_{L_1\cup L_3}(E_1)) \bowtie_\theta (\pi_{L_2\cup L_4}(E_2)))$$

Here L_1 and L_2 are sets of attributes from E_1 and E_2 respectively.
L_3 is the set of attributes of E_1 involved in θ but not in L_1 or L_2.
L_4 is the set of attributes of E_2 involved in θ but not in L_1 or L_2. ...15

1. Explain why this transformation is correctness-preserving.

2. From a performance perspective, when would you recommend using the transformed expression during heuristic query optimization.

ANS: The transformation will make a difference when the relations $\pi_{L_1\cup L_3}(E_1)$ and $\pi_{L_2\cup L_4}(E_2)$ are much smaller than E_1 and E_2. This would happen when E_1 or E_2 have many and/or large attributes of which few are used for the join or in the final result.

3. From a performance perspective, when would you *not* recommend using the transformed expression, that is, recommend using the expression as is.

ANS: The transformation will not be useful if any of the relations have an index on the join attribute, or when the difference in the size of E_1/E_2 before and after projection is insignificant.

V. $r \bowtie s$ is sorted on an attribute of r and is estimated to produce n records. But a user is really interested only in the top m records, where $m << n$. ...15
How would you evaluate the query

1. when the join is on a foreign key of r referencing s.

ANS:
- Obtain the top m elements of r ordered on the attribute of r used to sort $r \bowtie s$. This can be done with one pass of relation r if m tuples fit in memory (which is likely as $m << n$). If m tuples don't fit in memory a sort of r till the first m elements are obtained could be done.
- Now the obtained tuples can be joined with s, as join is the foreign key of r referencing s each of the m tuples will join with some tuple in s.

2. when the join is not on a foreign key.

ANS: When the join is not on a foreign key then an tuple in r might not join with any tuple of s hence a scheme as used in the above case will not work.

Perform $r \bowtie s$ and then obtain the top m elements, or sort the result of $r \bowtie s$ till the top m tuples are obtained.

Total Marks = 100