Solution to Database Systems Homework #5

1. (8 points) Use an example to explain when and why a primary index on an attribute A of relation R can result in better performance (i.e., lower I/O cost) than a secondary index on the same attribute.

   Answer: When multiple tuples satisfy a query condition (either an equality condition or a single range condition) on an attribute (say A), having a primary index on A can result in better performance than a secondary index on A. For example, suppose the query condition is A = v (v is a value) and three tuples in R satisfy this condition. If we have a primary index on A, then the three tuples will be stored consecutively on disk, possibly in one page (or two in the worst case scenario). On the other hand, if we have a secondary index on A instead, then the three tuples will likely be stored in three different pages. As a result, having a primary index on A will yield fewer page I/Os than having a secondary index on A.

2. Suppose relation R has 5,000,000 tuples such that each tuple occupies 200 bytes. The tables are stored in consecutive storage spaces on disk. We want to create a B+ tree index based on attribute A of R. Suppose the type of A is char(6), each address/pointer occupies 4 bytes, the size of each page is 2KB (2,048 bytes), and each node in the B+ tree has an initial fill-factor of 80% (i.e., only 80% of each node/page in the B+ tree can be used initially, making the usable size of each page to be 2048 * 0.80 = 1638). **Note that the 80% fill-factor is only used for nodes (both leaf nodes and internal nodes) in the B+ tree, and pages used to store the tuples are all filled up to the fullest capacity possible.**

   (a) (8 points) How many levels this B+ tree will have? Compute the number of nodes at each level.

   Answer: 5 million tuples yield 5 million cells/pairs. Each cell is 6+4=10 bytes. 5 million cells have 50 million bytes. Each page is 2,048 bytes. Due to the 80% fill-factor, only 1,638 bytes can be used for each node in the B+tree initially. Each node/page can hold 1638/10 ≈ 163 cells. This means that 5000000/163 ≈ 30675 nodes/pages are needed to hold the 5 million cells, i.e., there are 30,675 leaf nodes in the B+tree.

   Each node in the bottom level contributes one cell to the next upper level. Thus, there will be 30,675 cells at the next upper level. 30,675 cells need to be held in 30675/163 ≈ 189 pages, i.e., 189 nodes/pages will be needed at the next upper level.

   The 189 nodes will produce 189 cells at the next upper level and these cells will be held in two pages/nodes.

   Finally the two nodes will contribute two cells to the next upper level which can be held in one page – the root page.

   Therefore, the B+ tree has 4 levels with 30,675 nodes at the 4th level, 189 nodes at the 3rd level, 2 nodes at the 2nd level and one node at the 1st (root) level.

   (b) (6 points) Suppose your answer to question (a) is *k* (i.e., the B+ tree has *k* levels), what would be the maximum possible number of tuples this *k*-level B+ tree can accommodate? For this question, the 80% fill factor still applies. (hint: make the root as full (as close to 1638 bytes) as possible)

   Answer: The root can have a maximum 163 child nodes. With the 163 fanout for each node, a 4-level B+ tree can have a maximum 163*163*163 = 4330747 nodes at the bottom level. The 4,330,747 nodes can hold a maximum of 4330747*163 = 705,911,761 cells. Thus, the 4-level B+ tree can accommodate a maximum of 705,911,761 tuples.

   (c) (8 points) Consider a selection condition "A between a1 and a2", where a1 and a2 are constants. Suppose the number of distinct values under attribute A between a1 and a2 (including a1 and a2) is 10. Suppose 100 tuples in R satisfy this condition. What would be the **maximum possible** number of pages

that need to be brought into the memory in order to find these 100 qualified tuples if the B+ tree is a primary index? Please justify your answer. (It is assumed that initially both the B+ tree and the table are stored on disk. We also assume that indirection pages are used to handle repeating values and for simplicity, we assume ten indirection pages are used for these 100 tuples.)

Answer: Since the index is a primary index, all the 100 qualified tuples will be stored next to each other. Since each tuple has 200 bytes and the page size has 2,048 bytes, each page can hold 10 tuples. In the worst case (for example, the first of these 100 tuples is stored in a different page), the 100 qualified tuples may be stored in up to 11 pages. The B+tree has 4 levels. This means that at least 4 pages of the B+tree will be read when a1 is used to perform the search. There are 10 cells in the leaf nodes of the B+ tree for the 100 tuples (because they have 10 distinct values under A) and these 10 cells appear in two pages in the leaf nodes of the B+tree in the worst case. Therefore, up to 5 pages (4 + 2 – 1) in the B+tree need to be brought into the memory. In addition, we need to read in the 10 indirection pages. In conclusion, in the worst case scenario, the total number of I/O pages to find the 100 tuples is 11 + 5 + 10 = 26.

(d) (5 points) The same question as in (c) except that this time the B+ tree is a secondary index.

Answer: Each page can hold 10 tuples and the table occupies 5000000/10 = 500000 pages. In the worst case scenario, the 100 qualified tuples may be stored in 100 different pages. Again, up to 5 pages in the B+tree need to be read. In addition, 10 indirection pages need to be to read. Thus, the total number of I/O pages to find the 100 tuples in the worst case is thus 100 + 5 + 10 = 115.

3. Consider the following SQL query:

```
select B#, gpa
from students
where status = 'senior' and deptname = 'ME';
```

Suppose each student tuple occupies 200 bytes; all pages are of size 4KB; there are 20,000 students, among which 5,000 are seniors and 120 are ME major. Discuss how should the query be evaluated for the following cases (Hint: If there are different options, consider the number of page I/Os; don't forget to differentiate sequential I/Os from random I/Os. To simplify we assume that a random I/O is equivalent to 10 sequential I/Os in terms of I/O time. We further assume that the height of the B+tree is 3.):

(a) (5 points) Case 1: There is a secondary index on deptname but no index on status (no need to use the colors-of-balls formula, just assume that each qualified tuple will be stored in a different page).

Answer: Using the index on deptname to find all students who satisfy “deptname = 'ME' ” would incur about 120+4 random page I/Os (the 4 is for nodes in the B+tree that need to be brought into the memory). Sequentially scanning the entire table would incur 20000/20 = 1000 sequential page I/Os, which is equivalent to 100 random page I/Os based on our assumption. Thus, in this case, sequentially scanning the data table is more efficient in evaluating this query.

(b) (6 points) Case 4: There is a secondary index on status and a secondary index on deptname.

Answer: Since the number of ME students is much smaller than the number of seniors, if an index should be used, then it is better to use the index on deptname to find all students who satisfy “deptname = 'ME' ” and for each such student, check if he/she satisfies the condition on status. From the answer for (a) above, using the index on deptname is not as good as sequentially scanning the table to evaluation the query. Therefore, the best strategy is not to use any of these indexes and simply scan the table.

(c) (8 points) Case 5: There is a primary index on status and a secondary index on deptname.

Answer: Each page can hold 20 students. Since there is a primary index on status, the 5,000 senior student tuples will be stored together in 5000/20 = 250 consecutive pages (or 251 pages in the worst case scenario). Since the index on deptment is a secondary index, the 100 students majoring ME are likely stored in 100 random pages. Since reading in 250 (or 251) consecutive pages (sequential I/Os) will cost much less than reading in 100 random pages (random I/Os), a better evaluation strategy is to use the primary index to find all senior students and for each such student, check if he/she satisfies the condition on deptname.

4. Consider the join $R \infty_{R.A = S.B} S$, where R and S are two relations. Three join methods, i.e., nested loop, sort merge and hash join, are discussed in the class. Nested loop and sort merge may benefit from the existence of indexes and/or whether or not the tables are sorted based on the joining attributes. Identify three different scenarios (i.e., with given sizes of R and S, the index status on R.A and/or S.B, and whether the tables are sorted based on R.A and/or S.B) such that each of the following claims is true for one of the scenarios.

   (a) (10 points) Nested loop outperforms sort merge and hash join.
   (b) (10 points) Sort merge outperforms nested loop and hash join.
   (c) (10 points) Hash join outperforms nested look and sort merge.

   Method 1 is said to "outperform" Method 2 if (1) Method 1 incurs a smaller number of I/O pages than Method 2, or (2) Method 1 and Method 2 incur the same number of I/O pages but Method 1 incurs less CPU cost (e.g., needs fewer comparisons) than Method 2. Justify your answer.

   Answer: We assume that the memory buffer for the join is not large enough to hold the entire R. Finally, we also assume that each tuple in one table may match very few tuples in the other table.

   (a) Nested loop outperforms sort merge and hash join when S is very small and R has a (primary) index on the joining attribute (i.e., R.A). Consider the case when S has only 10 tuples that are all stored in one page and R occupies 1,000 pages. Suppose the B+ tree on R.A has three levels. In this case, the nested loop join that utilizes the index has a cost no more than 10*(3+1) = 40 I/O pages (here the "1" indicating that the matching tuples can fit into one page). For sort merge and hash join, the cost is at usually 1 + 1000 = 1001 I/O pages (even when no sort is needed for R).

   (b) Sort merge outperforms nested loop and hash join in the following situation: both tables R and S are large such that none can be entirely held in the memory buffer, there is no index on either joining attribute, both tables are already sorted on the respective joining attributes and at least one of the joining attributes does not have repeating values. In this case, sort merge can be performed with one scan of each table and the cost is N+M I/O pages. For both nested loop and hash join, the cost will be higher than N+M, because when the smaller table (or the hash table for the smaller table) cannot be held in the memory, the other table must be scanned at least twice.

   (c) Consider the following situation: The hash table of the smaller table can be held in the memory and at least one memory page remains for the larger table, the larger table cannot be held in the memory and it is not sorted on the joining attribute, and no index exists on either joining attribute. In this case, the cost of hash join is N+M I/O pages, which is better than the sort merge method because the cost of the sort merge is more than N+M I/O pages due to the need to sort R. If the memory is large enough to hold the hash table of the smaller table, it is also large enough to hold the smaller table (because the hash table is larger than the table). This means that the cost of the nested loop method will also be N+M I/O pages. However, the number of comparisons for the nested loop method is n*m while the number of comparisons for the hash join method is m+n*b, where b is the average number of tuples in the bucket of the hash table and it is usually a small constant. Since n*m > m+n*b for large tables (when n and m are large), the hash join method is better than the nested loop method.

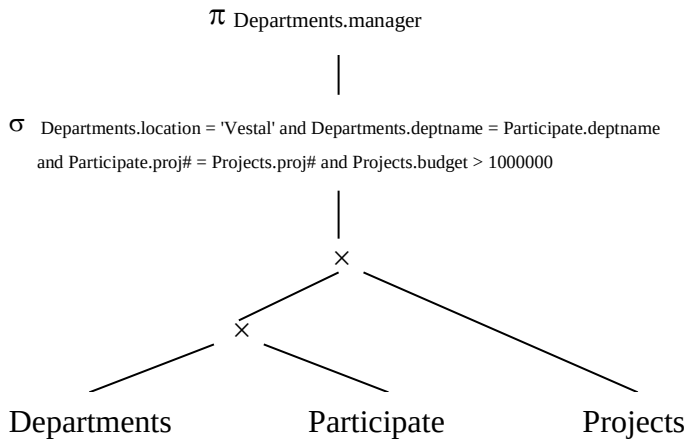
5. (16 points) Suppose there are three relations Departments(<u>deptname</u>, manager, location, telephone#), Projects(<u>proj#</u>, projname, budget, status) and Participate(<u>deptname, proj#</u>, start_date), where the primary key

of each relation is underlined. Apply algebraic based optimization method to find an execution plan for the following query (you may assume that most projects have budget higher than $1 million and very few projects are located in Vestal):

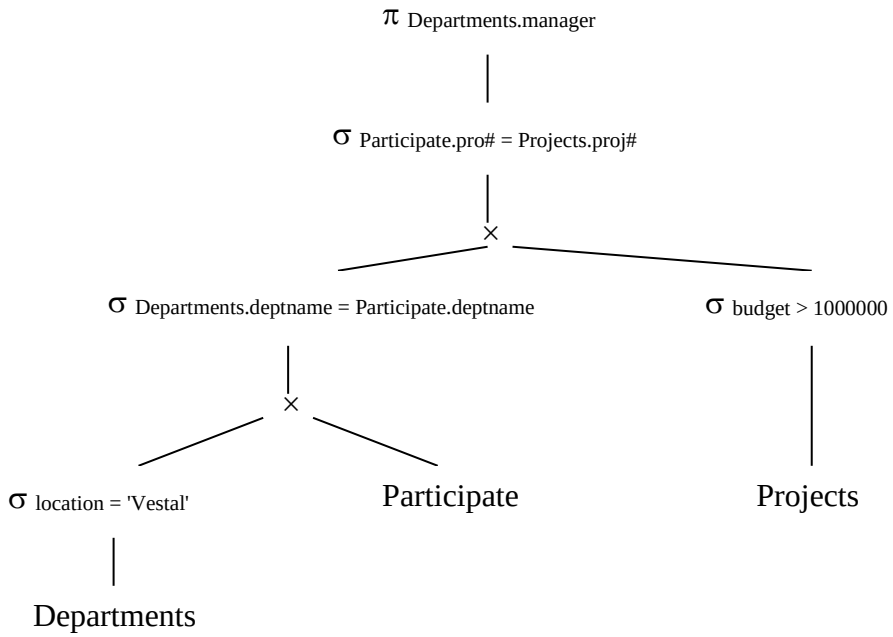
```
select d.manager
from Departments d, Participate p1, Projects p2
where d.location = 'Vestal' and d.deptname = p1.deptname
    and p1.proj# = p2.proj# and p2.budget >= 1000000
```

Show the query tree after each optimization rule is applied. Show also the relational algebra expression corresponding to the fully optimized query tree.

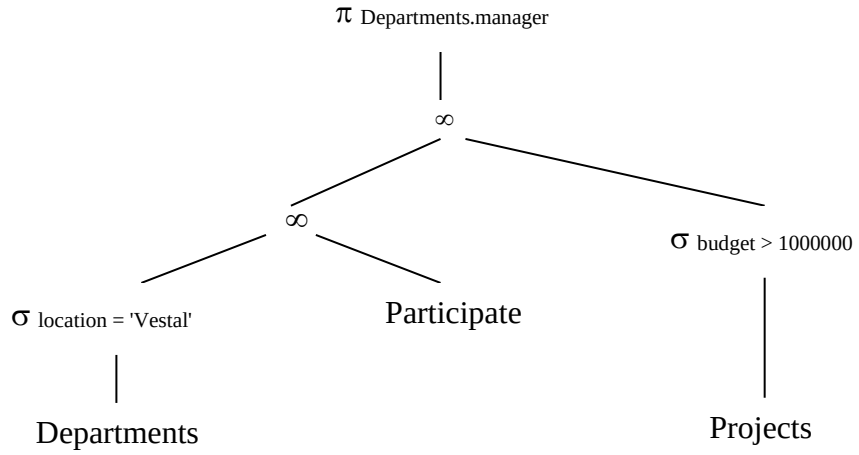
Answer: The initial query tree is:



(a) Perform selections as early as possible.

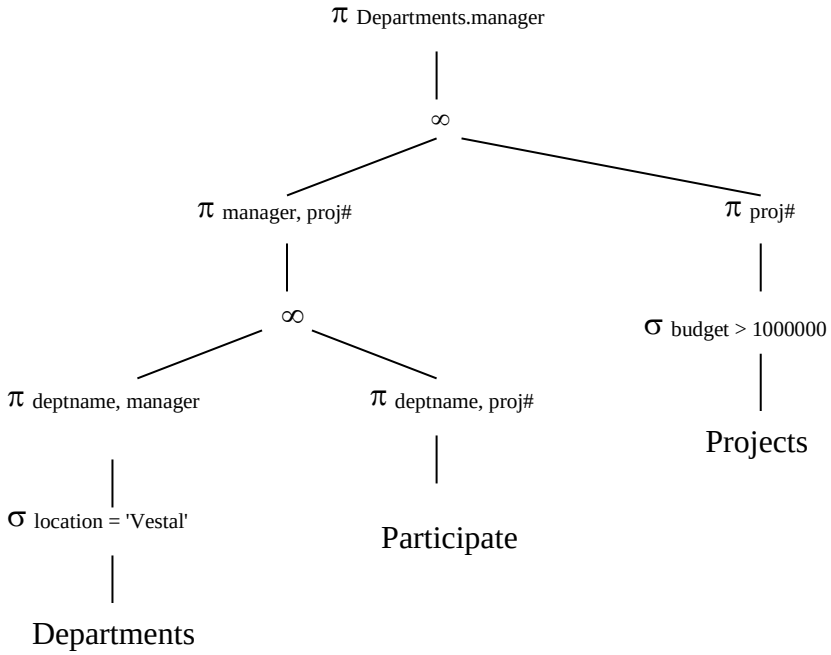


(b) Replace Cartesian products by joins whenever possible.



(c) Perform most restrictive joins first. No Change to the tree. (More projects will participate in the join than departments.)

(d) Project out useless attributes early.



The final optimized relational algebra expression is:

$\pi_{manager}(\pi_{manager, proj\#}(\pi_{deptname, manager}(\sigma_{location = 'Vestal'}(Departments)) \infty \pi_{deptname, proj\#}(Participate)) \infty \pi_{proj\#}(\sigma_{budget > 1000000}(Projects)))$