# Database Systems Homework #5
**(Due: November 30, 2018, at the start of class)**

Please remember to include the following statement at the beginning of your submitted assignment and SIGN it. Your assignment won't be graded with the signed statement.
"I have done this assignment completely on my own. I have not copied it, nor have I given my solution to anyone else. I understand that if I am involved in plagiarism or cheating I will have to sign an official form that I have cheated and that this form will be stored in my official university record. I also understand that I will receive a grade of 0 for the involved assignment and my grade will be reduced by one level (e.g., from A to A- or from B+ to B) for my first offense, and that I will receive a grade of "F" for the course for any additional offense of any kind."

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.

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.

   (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)

   (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.)

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

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 first 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).
(b) (6 points) Case 4: There is a secondary index on status and a secondary index on deptname.
(c) (8 points) Case 5: There is a primary index on status and a secondary index 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.

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.