CPS216 Advanced Database Systems - Fall 2008
Assignment 2A

- Due date: Thursday, Oct. 9, 2008, in class (2.50 PM). Late submissions will not be accepted.
- Submission: In class, or email solution in pdf or plain text to shivnath@cs.duke.edu.
- Do not forget to indicate your name on your submission.
- State all assumptions. For questions where descriptive solutions are required, you will be graded both on the correctness and clarity of your reasoning.
- Email questions to shivnath@cs.duke.edu.

Question 1 **Points 10**

Suppose we want to sort table R(A) using a machine with 1 Gigabyte of memory. Let MAX(b) denote the maximum value of T(R) (i.e., number of tuples in R) that we can sort using the two-phase sort algorithm when the disk block (and the memory block) size is b bytes. Which is larger: MAX(4096) or MAX(8192)? Give brief explanation.

Question 2 **Points 15**

When we discussed the block nested-loop join algorithm in class, we defined its cost as the total number of disk blocks accessed. For example, we saw that a block nested-loop join of tables R and S, with R as the outer, has a cost = B(R) + B(R)B(S)/(M-1). We also designed the minimum cost block nested-loop join algorithm for this cost model. In this simple cost model, we did not distinguish between random accesses and sequential accesses. In this question, we will extend our cost model to account for random accesses and sequential accesses.

The following information is available:

- Table R is clustered and the blocks of R are laid out contiguously on disk. B(R) = 1000 and T(R) = 10,000. (Note: we follow the notation used in class.)
- Table S is clustered and the blocks of S are laid out contiguously on disk. B(S) = 500 and T(S) = 5000.
- M = 101 blocks.
- For simplicity, we will assume that a random access can be done on average in time $t_r = 20$ ms, and a sequential access can be done on average in time $t_s = 1$ ms. For example, scanning five contiguous blocks on disk, assuming the first access is random, incurs a cost $t_r + 4t_s$.

1. [**7 Points**] Design an algorithm for the block nested-loop join of tables R and S which has minimum cost when we distinguish between random and sequential disk accesses. Compute this minimum cost using the parameter values specified above.

2. [**8 Points**] How does your answer to (1) change if blocks of R are not laid out contiguously on disk? All other assumptions and parameters remain the same as specified above. Compute the minimum cost possible for block nested-loop join in this case.

Question 3 **Points 10**

Consider the following SQL query Q over tables R(B,C) and S(A,B).

```
Q: SELECT *
   FROM   S
   WHERE  S.A > (SELECT SUM(R.C)
                 FROM   R
                 WHERE  R.B = S.B)
```

The following information is available.

- R is clustered and the blocks of R are laid out contiguously on disk. B(R) = 1000 and T(R) = 10,000.
- S is clustered and the blocks of S are laid out contiguously on disk. B(S) = 500 and T(S) = 5000.
- M = 101 blocks

We can process Q using a simple extension to the block nested-loop join algorithm we studied in class. The algorithm would work as follows:

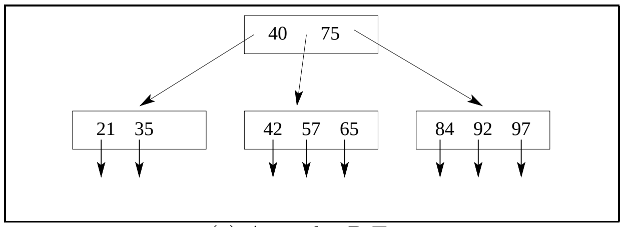
```
LOOP:
      /** S is the outer table and R is the inner table */
1.    Read the next M-1 blocks of S into memory;
2.    We store an extra field "sum" with each tuple of S in memory. (We assume that
         there is extra space in memory to store a constant amount of data per tuple.)
         This field, denoted s.sum for tuple s in S, keeps track of the
            sum of R.C values for R tuples that have R.B = s.B.
         s.sum is initialized to 0;
3.    Read blocks of R one by one using the remaining one block of memory;
4.    For every tuple s of S in the current M-1 blocks of S in memory
5.       For every tuple r of R in the current block of R in memory
6.          If (r.B = s.B) Then s.sum = s.sum + r.C;
7.    After we finish one full scan through R
8.       For every tuple s of S in the current M-1 blocks of S in memory
9.          If (s.A > s.sum) Then output s;
CONTINUE with LOOP till we finish one full scan through S;
```

In the above algorithm, we stored an extra field per tuple of S in memory. You can assume that a constant amount of extra space is available per tuple in memory. The cost model that we will use in this question is the same as the one we have been using in class, namely, the total number of blocks read or written, excluding the writes for the final output.

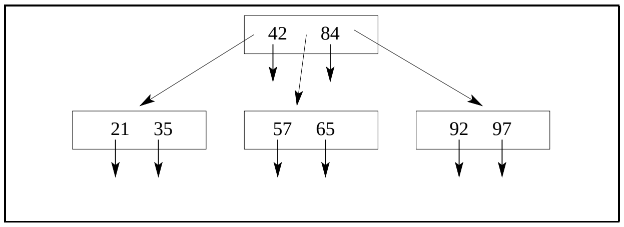
Suppose you have a non-clustering B-tree index on R.B. For this question, assume that both R.B and S.B values are distributed uniformly in [1,...,1000]. (That is, there are T(R)/V(R.B) = 10,000/1000 = 10 R tuples for each value of R.B in [1,...,1000]; and there are T(S)/V(S,B) = 5000/1000 = 5 S tuples for each value of S.B in [1,...,1000].) Also assume that the index fits completely in memory. For these parameters, what is the most efficient way to process Q using the non-clustering index on R.B? Compute the cost of this algorithm.

Question 4 **Points 35**

In this question, we will study a variant of a B-Tree that we will call B^--Tree[1]. In a regular B-Tree, all the keys indexed by the B-Tree occur only in the leaf nodes; the keys in the internal nodes are used only as roadmaps to locate the indexed keys in the leaf nodes. In a B^--Tree, the keys indexed by the tree occur in all types of nodes: leaf, internal, and root. In fact, any key that occurs in any node of a B^--Tree is one of the keys indexed by the tree. Figure 1 shows a B-Tree and a B^--Tree for the same set of indexed keys.



(a) A regular B-Tree



(b) A B^--Tree

Figure 1: A B-Tree and a B^--Tree indexing the same set of keys: $21, 35, 42, 57, 65, 84, 92, 97$

We have intentionally not specified most of the details of the B^--Tree in Figure 1. You are required to work out the details to make the B^--Tree function as an efficient disk-based index, and answer the following questions that ask for details of your B^--Tree structure.

1. (**3 Points**) Describe the general structure of an internal node in your B^--Tree? The general structure describes the keys and pointers in a node and their semantics (meaning).
2. (**3 Points**) Describe the general structure of a leaf node in the B^--Tree? Specifically, would your leaf node contain a *next* pointer like the leaf nodes in a regular B-Tree? Briefly explain why or why not?
3. (**4 Points**) Suppose each node of the B^--Tree has to be stored in exactly one disk block. Let n_l denote the maximum number of keys that you store in a leaf node, and n_i the maximum number of keys that you store in an internal node. Would you chose $n_l = n_i$ like in a regular B-Tree? Briefly elaborate.

For the remaining parts of this question, assume that your B^--Tree is used to index 100 million *distinct* keys and data record pointers. The size of a key is 8 bytes and the size of a pointer (record or block) is 8 bytes. Each disk block is 4096 bytes, and each node of the B^--Tree has to be stored in one disk block. Assume that you constructed your B^--Tree to ensure maximum space utilization (i.e., almost all the nodes in the B^--Tree are full).

4. (**4 Points**) How many disk blocks does the B^--Tree require?
5. (**4 Points**) What is the number of levels in the B^--Tree?
6. (**4 Points**) What is the expected number of nodes visited during an equality search when (I) the equality search involves a key that is indexed, and (II) the equality search involves a key that is not in the index. For (I) assume that each of the 100 million keys are equally likely to be searched.
7. (**4 Points**) Give a *rough* estimate of the number of nodes visited during a range search that returns $100,000$ keys.

[1]Historically, the B^--trees discussed in this problem are called B-Trees, and the B-Trees that we covered in the class are called B^+-trees.

8. (**9 Points**) Which index structure is better—the B$^-$-Tree that you designed, or the regular B-Tree—with respect to each of the following criteria?

   a. Space efficiency.

   b. Search efficiency.

   c. Complexity of designing update operations.

Question 5 **Points 15**

Consider a B-Tree with parameter n. For each node there is a limit on the minimum number of pointers that the node can have. For internal nodes, the limit given in class is $\lceil \frac{n+1}{2} \rceil$, for leaf nodes the limit is $\lfloor \frac{n+1}{2} \rfloor$ (pointers to data), and for the root node the limit is 2 (assuming the B-Tree has at least 2 indexed keys).

Indicate whether each of the statements (a)-(c) are true or false. Provide brief explanation.

(a) The limits for internal and leaf nodes can be reduced below $\lceil \frac{n+1}{2} \rceil$ and $\lfloor \frac{n+1}{2} \rfloor$, respectively.

(b) The limits for internal and leaf nodes can be increased beyond $\lceil \frac{n+1}{2} \rceil$ and $\lfloor \frac{n+1}{2} \rfloor$, respectively.

(c) For root node, the limit can be increased beyond 2.

Question 6 **Points 15**

Consider insertion and deletion operations over a B-Tree. Clearly, an insertion or a deletion operation changes the *state* of a B-Tree. By "state" we mean the exact set of nodes comprising the B-Tree, and the keys and pointers stored in these nodes. Assume there are no duplicate keys.

Indicate whether each of the statements (a)-(c) are true or false. Provide brief explanation.

(a) Inserting a key k and immediately deleting it can leave the B-Tree in a different state.

(b) Inserting key k_1 followed by key k_2 always leaves the B-Tree in the same state as inserting k_2 followed by k_1.

(c) Consider the insertion of key k_1 followed immediately by the deletion of key k_2 ($k_1 \neq k_2$). The height of the B-Tree can increase during the insertion of k_1 and decrease during the deletion of k_2.