COS 597A, Fall 2011
Solutions to Problem Set 5

Problem 1

Every path from root to leave of a height-3 B+ tree is of length 3 and contains 4 nodes, including the root and leaf. Reading these nodes to reach the leaf requires **reading 4 pages**, assuming none of the nodes is already in memory buffer. We assume all these pages remain in the buffer.

The Algorithm in Figure 10.15 only checks one sibling. (it doesn't say which one to check if there are two, but this does not affect our accounting.) The sibling is reached from the parent , and this requires **1 additional page read**. Assume the leaf merges with its left sibling. (Merging with the right sibling is analogous.) Usie the page containing the left sibling for the merged node, leaving the leaf pointers to/from the merged node and the next leaf to the left correct. Writing the merged node takes **1 page write.** The pointers to/from the leaf on the right must be updated. Access this leaf using the leaf pointer and update the pointer to the merged node; this takes **1 page read and 1 page write**.

The parent and ancestors up the tree will be merged. Each merge requires getting a sibling of the interior node (**1 page read**) and writing the new merged node (**1 page write**). This is r**epeated for 2 interior nodes**. For the root node, the node is simply eliminated and no write is necessary.

Total: **8 page reads + 4 page writes = 12 pages disk I/O**

This solution assumes the leaves are doubly-linked as presented in class. The text book uses only singly linked lists - left to right. In this case, when merging two leaves, using the page containing the left leaf for the merged node means the pointer from the preceding leaf in the linked list is correct, and no extra update is necessary.

Problem 2
Part a:

For each page of R
    For each record in the page of R
        Using the hash index, look up the record in S whose S.A value agrees with the corresponding attribute of the record of R.
        If there is no such record in S or the record in S is not equal to the record in R, write the record in R to the output buffer.

Since S.A is a candidate key, at most one record of S will be found on each look-up. As written, each page of R will be read once and each record of R will cause one index look-up of a record in S. The hash index will give a pointer to the actual record in S. The hash index look-up cost varies depending on whether the resulting bucket has overflow pages. We will use an average cost. The actual record of S must be retrieved because all attribute values must agree for the record in S to eliminate the record from the result. The cost is then the cost to read all pages of R and the cost to look up a record of S for each record of R:

$M + n_R * (c + 1)$

c is the average cost to obtain the record pointer from the hash index; 1 is the cost to use the record pointer to retrieve the page containing the actual data record.

R can be read page by page using one buffer page. One buffer page should suffice for the hash index look-up for a record: read the first page of the bucket into the buffer, overwrite this buffer page with each consecutive overflow page of the bucket as necessary, and finally overwrite the bucket page in bufer with the page of S containing the actual record. One buffer page suffices for the output. We conclude 3 buffer pages suffice.

This cost assumes nothing about whether attribute A is a candidate key for R. If it is *not* a candidate key, records of R can be sorted in the buffer by their values of R.A, *and* only one index look-up per unique value is needed. This is a saving over n_R look-ups and can be increased by reading in several pages of R at a time.

Part b: ceiling($\sqrt{20000}$ + 1) = 143 and there are 150 buffers, so both sorts can be done in 4 reads/writes of the file. Therefore, 5(M+N)= 105,000 disk page I/Os are used by the algorithm that sorts R and S and scans in parallel .

For Part a: A typical value of c is 1.2, giving $M + n_R*(2.2)$ = 23,000 disk page I/Os. In this case, the algorithm of Part a is more efficient. Even if c=9, unrealistically large, the cost would be 101,000 disk page I/Os, still more efficient than the sort and scan algorithm.

If S is sufficiently smaller, the sort and scan method will win. For c=1.2, N<3600 suffices.

Problem 3

Let |R| denote the number of pages to store R, |S| denote the number of pages to store S, and |T| denote the number of pages to store T

Part a: For block nested loop, we want to minimize the number of times we re-read the inner-loop relation by bringing large chunks of the outer relation into the buffer at one time. For this reason, it is desirable to have T as the outer relation when it is one of the join arguments. We also need to consider how large the intermediate relation (result of the first join) is. Assuming pages of R, S, and T on disk are full:

R requires 10,000 pages,
S requires 10,000 pages and
T requires 40 pages.

The only information we have to help us estimate the size of any intermediate result is that A is the primary key of S. Therefore the join of S with R or T on A will have no more tuples than R or T has, respectively (at most one tuple of S will match each tuple of R or T on attribute A). Of course the tuples of the intermediate result will have more attributes. Therefore, the size of the records for these tuples is larger, and less records fit in a page. Our only estimate for the size of R ◊◊T is based on the product of the number of tuples in each. We have no information about the number of values of A as search key for the hash index on R.A or as search key for the hash index on T.A to help us reduce this coarse estimate.

Let us prune our search for a good plan by eliminating R ◊◊ T as a possible first join due to the large estimated size of the intermediate result (50 million tuples with less than 5 tuples per page).

Consider two possibilities:

1. *Compute (T ◊◊ S) = I first*, with T the outer relation. The cost is |T| + |S| = **10,040 disk page I/Os** because T fits in the buffer. The intermediate result is estimated to be 1000 tuples with about 7 tuples per page (0.04 pages per tuple of T + 0.1 pages per tuple of S = 0.14 pages per tuple of the intermediate result, giving 7.14 tuples per page). So the intermediate result requires 143 pages, assuming records cannot be split across pages. *Then compute I ◊◊ R*. I uses the smaller number of pages and we would normally put it as the outer relation. If we materialize I, paying a write of 143 pages, we have the full buffer to do the second join and block nested loop join can be done in cost ($|I| + |I| + \lceil |I|/43 \rceil * |R|$) = **40,286 disk page I/Os**. If we don't materialize, computing (T ◊◊ S) is taking 42 buffer pages, one of which is the output page collecting I. We have three additional buffer pages but need only 2 for R and the final output. Therefore, we can use two to collect I. Then, the block nested loop is

executed in cost ⌈|I|/2⌉ * |R| = 720,000 disk page I/Os, with I again as the outer relation. Clearly, materializing is better.
TOTAL COST = 50,326 disk page I/Os

Note that we could give up fitting all of T in the buffer when computing T◊◊S and use more of the buffer pages for pipelined output to the second join calculation. However note that any reduction below 40 in the buffer pages allocated to T increases the cost of T◊◊S by at least |S|=10,000 because T will be read in pieces. Materializing only costs a write and read of I, which is 286.

2. *Compute (R ◊◊ S) = I first*. Since they are both the same size in pages, arbitrarily choose R as the outer relation. Use 43 buffer pages for R. The cost is |R|+ ⌈|R|/43⌉ ∗|S|) = 2,340,000. We need go no further – it is already clear that our first possibility is substantially cheaper.

Conclusion: compute T◊◊S= I first, with T the outer relation; then compute I◊◊R with I as the outer relation. I is materialized.
TOTAL COST = 50,326 disk page I/Os

Part b:

i. The point of hash join is to use the *same* hash function on both relations, hashing the shared attribute(s). Then records that match on join attributes will be in corresponding buckets and our search cost is reduced. The B+ tree index on S is not useful *for hash join*. The hash index on R could be useful *if* it is the refinement of a hash function that hashes S into F-1 buckets. Then we could proceed with the joining phase of the hash join algorithm using each bucket of S (assuming it fits in the buffer) and the corresponding buckets of R obtained from the hash index. Since permanent hash indexes such as the primary index on R are designed to get individual records quickly, it is unlikely to have the property we need for hash join. So we will assume we are using a different hash function for the join algorithm. If the hash index of R does lend itself, then we save at least one reading of R – more if we would have recursively partitioned some buckets of R using a new hash function.

ii. If we materialize the results I of the hash join, then T can be read into the buffer all at once, and a simple block nested loop algorithm suffices. Intermediate result I is at most 50,000 tuples and each tuple needs a record of size at most 0.3 pages, giving 3 records per page and 16,667 pages. Then writing I costs 16,667 disk page I/Os and the block nested loop costs |T| + |I| = 16,707 disk page I/Os

Beyond the basic answer: The result I of R ◊◊ S is produced grouped by hash value. So, if we pipeline I, it might be advantageous to join I and T using another hash join with the same family of hash functions. To do this effectively, we should partition all three of R, S, and T into hash buckets before doing the joining phase of the calculation of R◊◊S so that we have the full buffer for each partitioning phase. The joining phase of R◊◊S and the joining phase of I◊◊T must share the buffer**.** This may require buckets of R or S to be smaller if they are to fit in the buffer space allocated to R◊◊S. This may cause extra recursive hash partitioning of R and S, increasing the cost of the first hash join. Since T is small relative to R and S, it should partition into small enough buckets using only the first hash function of the family. However, this only works if we can insure that, when computing the join phase for R◊◊S, all sub-buckets of a single 1st-level bucket (using 1st hash function) are considered consecutively. Then we can read each bucket for T once to join with the pipelined I. The cost is approximately 3|T| = 120 disk page I/Os above the cost to compute R◊◊S using hash join. Note that doing the partitioning of all 3 base relations before starting the join phase of the first hash join is not the standard order of computation.

iii. We assume in the estimates below that all hash functions give a uniform distribution of records into buckets.

Using the block nested loop algorithm to compute $I \lozenge\lozenge T$:
The hash join of R and S required a second (recursive) partitioning of each of R and S – partitioning 10,000 pages into 44 buckets gives buckets too large to fit in the buffer. The 2nd-level partitioning gives $(44)^2$= 1936 buckets of 5 pages each – more than small enough. Each partitioning requires reading and writing |R| pages and |S| pages; the joining phase reads each bucket once, for a cost of |R|+|S|. The total cost of computing $R \lozenge\lozenge S = I$ is therefore 5(|R|+|S|) = 100,000 disk page I/Os. From (ii) above, materializing and computing $T \lozenge\lozenge I$ costs 33,374 disk page I/Os. Therefore the **total cost** of the two joins by this method is **133,374 disk page I/Os.**

Using the hash join algorithm to compute $I \lozenge\lozenge T$:
Note that the 2-level partitioning of R and S yields buckets of 5 pages, which should be plenty small enough even when sharing buffer with the second join calculation. The cost to partition R and S is as above. The first partitioning of T will give buckets of 1 page. Note that we have plenty of buffer pages for a bucket of S, a bucket of T, a page of a bucket of R, a page of I, and a page of final output. The **total cost** is 5(|R|+|S|) = 100,000 disk page I/Os for $R \lozenge\lozenge S$ plus 3|T| = 120 disk page I/Os for $T \lozenge\lozenge I$, equaling **100,120 disk page I/Os.**

Part c: Pipelining has been considered in the description of each plan in Parts a and b.