CS245 Midterm Solutions – Summer 1999

Part I. (16 points) Indicate whether each of the following statements is TRUE or FALSE. If you chose FALSE, please correct it or briefly explain why.

1. TRUE.

2. FALSE.
   Given fixed-size data blocks, a B-tree node contains less keys than a B+ tree node. So, B-tree is in fact deeper and less compact.

3. FALSE.
   Correction: ... The expected number of tuples in $R \bowtie S$ is less than or equal to **2000**.

4. TRUE.

5. TRUE.

6. FALSE.
   Merge join outputs sorted join result and can benefit other join operations up in the operator tree.

7. FALSE.
   We may need to leave some space in each block to reduce the I/O cost for insertion.

8. TRUE.

Part II. (12 points) Select one correct answer from the listed multiple choices. Points = 4 × number right.

1. (B).
   You can push the selection operation $\sigma_{B<10}$ up and down in E1 to transform it to E2. So, E1 and E2 are equivalent. In E3, the B attribute is projected out from R before joining with S. Then the join between R and S is no longer on the B attribute. Therefore, E3 is not equivalent to the other two expressions.

2. (C).
   The general join strategy that should be used in this problem is to read data blocks one by one from one relation, and use the index on the other relation to find the matching tuples. The choice (E) should be discarded first, because it does not help the join operation at all.

We then compare dense indexes with sparse indexes, and find that dense indexes are better than sparse indexes in this case given that they all fit in memory. For example, with a sparse index on $R.B$, we first fetch S blocks. For each S tuple, when looking for R tuples with a matching B value, we need to fetch a block of R even though the $S.B$ value is not in the $R.B$ index, because **not all key values appear in a sparse index**. However, if we use a dense index, as soon as we find out that the $S.B$ value is not in the $R.B$ index, we know that there is no tuples in R with that the matching B value, and we do not need to fetch R's data block any more. So, using a sparse index costs more I/Os than using a dense index in this case. Therefore, (A) is better than (B), and (C) is better than (D).

Finally, we compare choices (A) and (C). With (A), we have a dense index on $R.B$. We need to first read all blocks of S, and for each S tuple, we find matching tuples in R using the $R.B$ index. The total I/O cost is about $B(S) + N(S) \cdot N(R)/V(B,R)$ (given that R is not sorted on the B attribute). With (C), we first read R blocks, then find matching tuples in S. The I/O cost for this join scheme is about $B(R) + N(R) \cdot B(S)/V(B,S)$ (given that S is sorted on the B attribute). With a reasonable assumption that $V(B,R) \leq V(B,S)$, we can easily draw the conclusion that (C) achieves lower query cost than (A).

3. (D).

First, we can easily see that the queries in (A), (B), and (E) can benefit significantly from the given multi-key index. For the query in (B), we can also use the second level index (on the B attribute) in the multi-key index to find all the pointers to the records with $B = 20$. However, for the query in (D), we cannot use the multi-key index at all, because we basically need to fetch all the data records in R and check their A and C attributes.

Part III. (47 points) Write your final answer for each question.

1. (D) and 30 ms.

average seek time = 15 ms. rotational latency = 10/2 = 5 ms. transfer time = 60/6000 = 0.01 s = 10 ms. access time = seek time + rotational latency + transfer time = 15+5+10 = 30 ms.

2. (C) and $15 + 5 + (1 - v) \cdot 10/13$ ms.

# of blocks per track = $2^{30}/1000/10/2^{13} = 13$. transfer time = $(1 - v) \cdot 10/13$ ms. access time = $15 + 5 + (1 - v) \cdot 10/13$ ms.

3. $\lceil log_n \frac{r}{n-1} \rceil + 1$.

We first compute the maximum number of records that can be indexed using a l level B+ tree. Each inner node has a maximum fan-out of n. Each leaf node contains at most $n - 1$ record pointers. So, given an l-level B+ tree, we can index at most $n^{l-1} \cdot (n - 1)$ nodes. In other words, $n^{l-1} \cdot (n - 1) \geq r$. Therefore we get $l \geq \lceil log_n \frac{r}{n-1} \rceil + 1$.

4. $29/33 < p \leq 1$.

The size of a fixed-format record is $20 \cdot (10+30) = 800$ bytes. The average size of a variable-format record is $22 \cdot (10 + 30p) = 220 + 660p$ bytes. Using fixed-format records is better than using variable-format records (according to the storage requirement) if $800 < 200 + 660p$; that is, $29/33 < p \leq 1$.

5. $n_A = 4$, $n_B = 6$.

Consider a partitioned hash table that uses a 10-bit hash key containing n_A bits of the hash value on A and n_B bits of the hash value on B. For the query "`select...where A = a`" we need to access 2^{10-n_A} buckets, and for the query "`select...where B = b`" we need to access 2^{10-n_B} buckets. The average number of buckets that must be accessed for the queries given n_A and n_B is $2^{10-n_A} \cdot 0.2 + 2^{10-n_B} \cdot 0.8 = 0.1 \cdot (2^{11-n_A} + 2^{n_A+3})$. (Note that $n_A + n_B = 10$.) To minimize the average query cost, we need $11 - n_A = n_A + 3$; that is, $n_A = 4$ and $n_B = 6$.

6. $\pi_{A,E}(\pi_{A,C}(\sigma_{A<10}(R)) \bowtie \pi_{C,E}(\sigma_{C+D<100}(S)))$.

Note that the selection operation $\sigma_{C+D<100}$ can be performed directly on S, because with the join condition $R.C = S.C$ we can rewrite the selection condition $R.C + S.D < 100$ to $S.C + S.D < 100$.

7. $q_1 \rightarrow i_6$, $q_2 \rightarrow i_2$, $q_3 \rightarrow i_3$, $q_4 \rightarrow \varnothing$ $q_5 \rightarrow i_1$, $q_6 \rightarrow i_5$.

This question is to test three knowledge points: (1) Hash tables are better for probe queries, while index sequential files, B+ trees, and grid indexes are better for range queries. (2) Multi-key indexes and partitioned hash tables are useful for queries on multiple key values. (3) File scanning generally does not need to use index.

8. min = 512, max = 4096.

The hash directory size is minimum when the first 12 bits in the 15-bit hash keys of the 4096 records are all distinct (ranging from $0 \cdots 0$ to $1 \cdots 1$). Then, we can use $12 - 3 = 9$ bits to hash all the records, and the directory size is 512.

The hash directory size is maximum when the first $15 - 12 = 3$ bits in the 15-bit hash keys of the 4096 records are all the same (e.g., 000). We then need 9 extra bits to hash the records. So, $i = 3 + 9 = 12$ and the size of the hash directory is 4096.

9. min = max = 683.

Using linear hashing, we add a new bucket only when the average number of records in a bucket is higher than the threshold 6. So, we will have exactly $4096/6 = 683$ buckets in the linear hash table (although we may have different number of overflow blocks given different hash key distributions).

Part IV. (15 points) Answer the following questions and **briefly** show your reasoning steps.

(1) The best plan is to perform an index join using the two B+ tree indexes on $R.C$ and $S.C$ (referred as RI and SI below). First, we find the left most leaf nodes in RI and SI. Using the sequential pointers in the leaf nodes, we can fetch all leaf nodes of the B+ trees in the order of C value. Therefore, we can merge the leaf nodes in RI and SI to find all the C values shared by R and S tuples. For each shared C value, we use the pointer in RI's leaf node to fetch the joined R tuple, and write its $R.A$ value to the output result. (Note that we do not need to read S tuples because the query is only asking for the $R.A$ attribute values in the join result.)

We use the four memory blocks as follows. One block holds a leaf node of RI; one block holds a leaf node of SI; one block is for reading an R data block; and one block is for the output.

(2) The joined result size is $\frac{n_R \cdot n_S}{max(V(C,R),V(C,S))} = \frac{20000 \cdot 2000}{10000} =$ **4000 tuples**.

Based on the number of records in R and S, we can compute the number of levels in the B+ tree indexes, and obtain that RI has 3 levels, and SI has 2 levels. Since each leaf node in an order 100 B+ tree contains at least 50 record pointers, we know that there are at most $20000/50 = 400$ leaf nodes in RI, and $2000/50 = 40$ leaf nodes in SI.

During the query, we need 2 I/Os (fetching the root and an inner node) to find the leftmost leaf node in RI, and we need 1 I/O to find the leftmost leaf node in SI. Then, we need at most 400 I/Os to read all the leaf nodes in RI, and at most 40 I/Os to read all the leaf nodes in SI. Finally, we need at most 4000 I/Os to read the R tuples. (Since R is not clustered on the C value, we may need one block I/O for each joined tuple.) Therefore, the total worst case query cost is $2 + 1 + 400 + 40 + 4000 =$ **4443 I/Os**.