University of California, Los Angeles
Department of Computer Science

Computer Science 143 **Prof. Ryan Rosario**

Homework 4

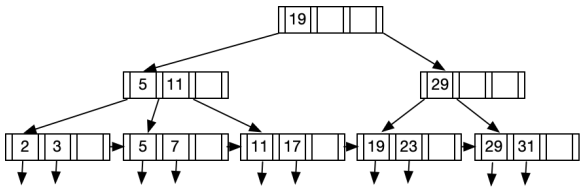
Solutions.

Part 1: Indices and B+ Trees

For any exercises that involve drawing, you can draw on paper and scan, take a photo etc., use a diagram program, etc. and submit the file in your homework writeup.

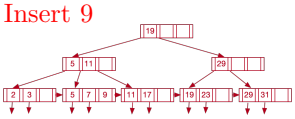
Exercises.

1. Using the B+ Tree below, built on top of a primary index, perform the following inserts. To match our solution, when you split, you should keep the first $\lceil \frac{n}{2} \rceil$ keys of the original node in the original node and move the rest over to the newly allocated node. (Remember that other resources may do this differently)

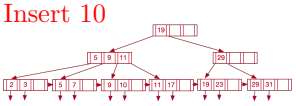


   (a) Insert 9, 10, and 8, in that order and show the resulting tree.

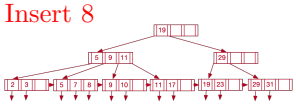
Insert 9



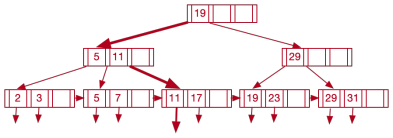
Insert 10



Insert 8



   (b) Using the original tree above, draw the path through the tree we would need to follow to find the **record** associated with key-value 11. Be careful! Remember where records are stored. Assume we are searching on a key.

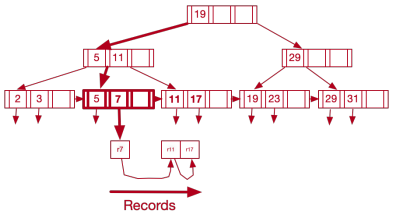


Note that most sources point to the node in general. The textbook is more precise by pointing to the first key in a node. Either is fine.

(c) Using the original tree above, draw the path through the tree we would need to follow to find the **records** associated with keys 7 to 17 inclusive. Be careful! Remember where records are stored. Again, assume we are searching on a key.

How we traverse the tree depends on if the underlying ordered index is a clustering or non-clustering index. In this case it is clustering (primary).

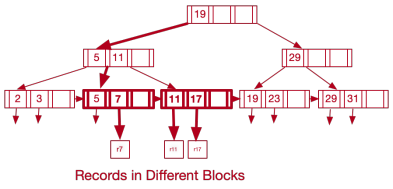
First, traverse to the leaf node contain the lowest key such that $k \geq 7$. In this case, $a = 7$ is in the leaf node so we can start there. We collect the pointer to the record with a key of 7. Since this is a primary/clustering index, we then follow **record** pointers (not the leaf nodes) from record to record and block to block until we find a key that is $\geq b = 17$. In this particular tree, $b = 17$ is in a leaf node.



Note that most sources point to the node in general. The textbook is more precise by pointing to the first key in a node. Either is fine. We are looking for you to point down into record space, and then iterate record by record. There are many ways to draw this. My drawing, for simplicity, assumes that key blocks are divided the same way as record blocks.

(d) Repeat the previous part assuming the underlying index is a secondary/non-clustering index and we are searching on a key.

Searching on a key is a bit of a red herring here. The only difference is that if we are searching on a non-key we may have to traverse a linked list of the duplicate key-values, each node pointing to, at worst, an additional block(s) to find all copies of the same key-value. Otherwise, in a secondary index, we must collect a pointer from each key in the range to each record.



2. Explain why it is better to use a clustering B+ Tree to perform a range query instead of a hash index. You will probably need to use some cost calculations to prove your point.

   The big difference between these two is seek time. In a clustering B+ tree, we require $h_i + 1$ seeks, h_i to traverse the tree and one extra to seek to the first record (from key a). In a hash index, we would have perform a seek for each key lookup within the range, in the worst case. Remember that a hash index is *not* and ordered index. If we cannot iterate over the keys of the hash index, we would also have to construct the range of values from a to b first which is impossible if the key is something like a float. If we can iterate over the keys, we also have a seek for those blocks, and then block transfers for the key set. Remember that random access means more seeks (than ordered/sequential).

3. Suppose we have a secondary B+ Tree index, entirely on disk, for relation R and we want to retrieve records with keys in (a, b) exclusive. Suppose that there are keys in this tree that are $\leq a$ and that there are keys in this tree that are $\geq b$. Write an expression for the total worst case execution time spent on disk I/O. Let's say this B+ Tree is stored on a hard disk that has a seek time of 90 microseconds and block transfer time of 2 microseconds. Assume that the seek time already takes the rotational latency induced by a 7200rpm disk into account. What is your time estimate? You *may* need a variable b denoting the number of blocks read, or n the number of records processed. *See the annotated lecture slides for an example.* Assume that keys are unique.

   This is a range query. There is nothing special about using an exclusive range (a, b) over an inclusive range $[a, b]$. The statement " Suppose that there are keys in this tree that are $\leq a$ and that there are keys in this tree that are $\geq b$" tells us that the keys we are searching for are surrounded by other keys in the tree if they are not actually in the tree themselves.

   Note that rule A6 in the textbook is not applicable to this problem as A6 is only valid for comparisons $k \leq v, k < v, k > v, k \geq v$.

First, we scan down to a leaf node, where the keys live. This requires $h\,(t_S + t_T)$ disk I/O time. Now we are seeked to the proper key block (leaf node). In the memory buffer, we search for the smallest key greater than a and follow its pointer to the record on disk. This incurs a seek and a block transfer $t_S + t_T$. Since this is a non-clustering index, we must pull the records corresponding to each key. Each record may be in a different block, so the second component is $n_r\,(t_S + t_T)$. Finally, as we traverse, we will need to traverse to other leaf nodes potentially, which incurs another seek and block transfer for each leaf node traversed. Let b be the number of leaf node blocks we traverse. Suppose the upper bound is 50 and the largest key in a leaf node is 48. We will need to read in one additional leaf node to find this upper bound, even if it is not in the block. Regardless, let b be the number of leaf node blocks traversed.

$$h\,(t_S + t_T) + n_r\,(t_S + t_T) + b\,(t_S + t_T)$$

which is equivalent to

$$(h + n_r + b)\,(t_T + t_S)$$

If you count the extra block transfer to find the upper bound (you don't absorb it into b), you would get:

$$(h + n_r + b + 1)\,(t_T + t_S)$$

Substituting in $t_T = 2\mu s, t_S = 90\mu s$ we get

$$\boxed{(h + n_r + b)\,(2 + 90) = 92\,(h + n_r + b)}$$

or

$$\boxed{(h + n_r + b + 1)\,(2 + 90) = 92\,(h + n_r + b + 1)}$$

Part 2: Join Algorithms Exercises.

4. Let R and S be two relations and assume neither fits entirely in the memory buffer. R contains three attributes A, B, C and S contains three attributes C, D, E. R contains 20,000 tuples, S has 50,000 tuples. 50 tuples of R fit in a block, and 20 tuples of S fit in a block. Compute an estimate for the total number of block transfers and seeks using the **block nested-join loop** to compute $\bowtie_\theta$ assuming θ is a simple equijoin. Make the same assumption as in lecture that one block from R and one block from S fit into the buffer.

   (a) Consider the case where R is the outer relation.

   Since R contains 20,000 tuples and 50 tuples fit in a block, we have that $b_r = 400$. Since S contains 50,000 tuples and 20 tuples fit in a block, we have that $b_s = 2500$. In the worst case, only one block of R and one block of S fit in the buffer. So if R is the outer relation, the total number of disk block reads is

   $$b_r b_s + b_r = 400 \times 2500 + 400 = 1,000,400$$

   From the lecture notes, we see that the total number of seeks is $2b_r$ because we have not assumed that S is entirely in the buffer. So the total number of seeks is

   $$2 \times 400 = 800$$

   (b) Repeat the previous calculation with R as the inner relation.

   If R is now the inner relation, our computation of block reads becomes

   $$b_s b_r + b_s = 2500 \times 400 + 2500 = 1,002,500$$

   And the number of seeks is

   $$2b_s = 2 \times 2500 = 5000$$

Thus, if we executed the following query:

```
SELECT *
FROM R
JOIN S
ON R.C = S.C;
```

it would be converted to the relational algebra expression $R \bowtie_{R.C=S.C} S$. Since both relations have one column in common, C, the expression would be optimized to a natural join $R \bowtie S$. But, recall that

$$R \bowtie S = S \bowtie R$$

Which one does the optimizer pick? The optimizer picks the one with the lowest cost, that is $R \bowtie S$, assuming that $R \bowtie S$ means that R is the outer relation.

5. We can implement joins more complex than an inner equijoin using the algorithms discussed in class. Suppose we want to implement the *antijoin*. $R \triangleright S$ gives us all tuples in R that **do not** have a match in S. Which join algorithm would be best and why?

   Credit for this problem is going to depend on the explanation provided due to the book being very strict on their implementations of the algorithms. Most algorithms can be slightly modified to work with non-equijoin or anti-join. Based on a hint give in class and on Piazza, students were to just assume an anti-join is an equijoin.

   Anti-joins can be implemented in SQL using `NOT EXISTS` with a subquery, or the set `NOT IN` followed by a subquery that returns a set. (Semi-joins on the other hand can be implemented using subqueries with `EXISTS` or `IN`). `(NOT) EXISTS` is faster than `(NOT) IN`.

   Hash Join can be lightly modified to work with the antijoin. To compute the antijoin, we look for keys in R that do not exist in S. We treat S as the build side and load as many blocks of S as we can into a memory buffer and build a hash table. If the entire relation S fits in RAM, and thus the hash table contains all of S, then we do one full table scan of R by key and check whether or not the hash of the key in each row of R is in the hash table. If it is **not** then we output the row from R. The modification is that we check if the key is *not* in the hash table rather than if it *is* in the hash table.

   If S does not fit in RAM and we must build the hash table blockwise (as discussed in the textbook and class), we have a small problem. Just because a row's key is not in a particular hash table, it does not mean the key does not exist in the relation. So, we can add a temporary flag for each row in an array or linked list. As we scan R, if the key exists in the hash table, we set the flag to denote that we will **not** output the row to the user. If the key does not exist in the hash table, we do not set the flag. The flags maintain the same state across full table scans of R. Once we are finished scanning R once-and-for-all, we do one final table scan on R and output the rows that do not have a flag (by traversing the array or linked list in tandem). If the flags are temporarily stored in the relation (there are pros and cons), we don't need to traverse an extra data structure. Any row that does not have the flag set has a key that was never found in the hash tables and we output those rows to the user.

   Because there may be hash collisions, particularly for large tables, we may need to perform this process iteratively or do some post-processing. **This is similar to Postgres implementation.**

   A **merge join** is another good option, though we have to spend time and disk I/O first sorting the relations. We performed the interleaved linear scan on R (outer) and S (inner). Each time we move the outer pointer down, we set a bit to 0 to denote that we have not found a match for the outer key. As we move the inner pointer down, we flip the bit to 1 if we find a match. If we reach the case where $k_S > k_R$, that is, the inner key is greater than the outer key, we move the outer pointer down. But before we do that, we check the bit. If the bit is 1, we output nothing. If the bit is 0, we output the record with k_R into the result set. We then continue until neither pointer can move further.

   A **nested loop join** or **block nested loop join** would be the worst performing as we would need to still examine all blocks in R and S quadratically. However, it is the only algorithm that works out-of-the-box for **any** join type (including non-equi joins).

   An **index nested loop join** may improve this performance depending on what index we use. A hash index would be preferable to a B+ Tree. Using a B+ tree is wasteful and would not receive full credit.