

### HW#4 Sample Solution

In this assignment we assume the node size is 4 kilobytes. We will assume that tables are stored with 100% occupancy, while index nodes are 80% full.

**1.(a)** Each tree node contains  $n-1$  keys and  $n$  pointers. With 32 bytes per key, 8 bytes per pointer and a node of total size  $4 * 2^{10} = 4096$  bytes, we can find how many keys and pointers can fit in each node:

$$(n-1) * 32 + n * 8 \leq 4096 \Rightarrow n \leq 103.1$$

So we choose  $n = 100$  (we can have 99 keys and 100 pointers per node). With 80% occupancy per node, each node will contain  $(100-1) * 0.80 \sim 80$  keys. There are 10,000,000 index entries and thus the tree height would be

$$H = \lceil \log_{80}(10,000,000) \rceil = 4.$$

Here, the tree height is defined as the number of levels of nodes of the tree, i.e., the number of disk accesses.

**(b)** An unclustered index is necessarily dense and has one pointer per record in the leaf-level nodes. It differs from the dense clustered index in that the file is not sorted by the attribute(s) being indexed. This is irrelevant when looking up a single key. Therefore, the cost to perform a key-based lookup is the number of levels of the tree (traversing from the root to the leaf-level), plus one access for following the leaf-level pointer to the block containing the record:  $4+1=5$  block reads. The time to perform one block access is 4ms seek time for each block, plus  $(60*1000)/(2*15000) = 2ms$  average rotational latency, for a total of 6ms per nonsequential access, plus  $4KB/50MB/s=0.08ms$  to transfer each block to main memory. Thus, the total time for a lookup of an element is  $5 * (4ms + 2ms + 0.08ms) = 30.4ms$ .

**(c)**  $(n-1) * 12 + n * 8 \leq 4096 \Rightarrow n \leq 205.4$ . We choose  $n = 200$ . With 80% occupancy per node, each node will contain  $199 * 0.8 \sim 160$  keys. The tree height would be

$$H = \lceil \log_{160}(1,000,000,000) \rceil = 5$$

An unclustered index is not sorted by the key attribute. However, the leaf-level of the index is sorted. Therefore, we traverse the tree from the root to find the first pointer for a matching record (in  $5 * 6.08ms$ ). From that point, we fetch the leaf nodes of the index (using the sibling pointers) until we reach the upper bound of the range query, and then fetch for each encountered index entry the corresponding tuple using a random access. With 160 elements per leaf and 1,000,000 tuples (0.1% of all tuples) satisfying the condition, we must look up a total of 6250 leaf-level nodes (or blocks) of the index. Therefore the total cost is  $5 * 6.08ms + 6ms + 6250 * 6.08ms + 1,000,000 * 6.08ms \sim 6116$  seconds..

**(d)** Sparse clustered index: First, we need to check again the tree height, which now is

$$H = \lceil \log_{160}(10,000,000) \rceil = 4,$$

since we have only one index entry for each block of 100 tuples of *Rented*. The result set size will consist of  $1,000,000,000 * 0.1\% = 1$  million tuples, which are now organized in 10,000 consecutive blocks, since the file is sorted by the key attribute. So we perform the index lookup for the lower range bound in  $4 * 6.08ms$ , and then scan the 10,000 blocks, for a total cost of  $4 * 6.08ms + 6ms + 10000 * 0.08ms = 0.830$  seconds.

**2.(a)** Size of *Customer*: 1GB. Size of *Rented*: 40GB. 100MB of memory.

Let us assume that we partition *Customer* into 11 chunks of size 91MB each, leaving another 9MB of main memory as

buffer when scanning *Rented*. Thus, we basically read *Customer* once and *Rented* 11 times. Since the buffer for *Rented* is sufficiently large, we can assume that all accesses are sequential, so the total cost is  $(1 \text{ GB} + 11 * 40 \text{ GB}) / (50 \text{ MB/s}) \sim 8,820$  seconds.

**(b)** Size of *Rented*: 40GB. Size of *Copy*: 400MB. Size of *Movie*: 10MB. 100MB of memory.

In this case, relationship *Movie* fits into main memory entirely, so the first join between *Movie* and *Copy* consists of a scan of *Movie* and then a scan of *Copy*, for a cost of  $(400 \text{ MB} + 10 \text{ MB}) / (50 \text{ MB/s}) \sim 8.2$  seconds. We now have a result of size 410MB, since each tuple in *Copy* will be joined with one tuple of *Movie*. This result is temporarily written out at another cost of 8.2 seconds. Afterwards, we split this temporary result into 5 chunks of about 82MB size that each fit in main memory, leaving enough buffer space (18MB) to efficiently read *Rented* in the next join. Thus, we now read the temporary result once and *Rented* 5 times, for a cost of  $(410 \text{ MB} + 5 * 40 \text{ GB}) / (50 \text{ MB/s}) \sim 4000$  seconds for the second join. As always, we ignore the cost of the final writing of the result to disk.

**(c)** We assume only 1 pass here when sorting the *Customer* relation. We need to merge  $d = (100 * 10,000,000) / 100 \text{ M} = 10$  files, and thus we need 11 buffers of size about 9MB.

Initial sort pass: reading and writing 10 chunks of 100MB costs:

$$2 * 10 * (6 \text{ ms} + 100 \text{ MB} / (50 \text{ MB/s})) = 40120 \text{ ms}$$

Time for merge pass:

$$\text{Reading/Writing a buffer takes: } 6 \text{ ms} + (100 / 11 \text{ MB}) / (50 \text{ MB/s}) = 187.8 \text{ ms}$$

$$\text{Reading all 1000M takes: } 11 * 10 * 187.8 \text{ ms} = 20.658 \text{ seconds.}$$

$$\text{Total time for merge(read+write)} = 41.316 \text{ seconds.}$$

Total time for this sort is 81.436 seconds.

Sorting *Rented* relation: Here, we have to merge 400 files of 100 MB each, so it is not clear if it is better to use 2 passes of 20-way merges or one pass where we use 401 small buffers of size only about 250KB. We'll go for one pass here. Thus, in the merge phase we have 401 buffers of equal size.

Initial sort pass: reading and writing in chunks of 100MB

$$2 * 400 * (4 \text{ ms} + 2 \text{ ms} + 100 \text{ MB} / (50 \text{ MB/s})) = 1,604,800 \text{ ms.}$$

Time for merge pass:

$$\text{Reading/Writing a buffer takes: } 4 \text{ ms} + 2 \text{ ms} + (100 / 401 \text{ MB}) / (50 \text{ MB/s}) = 10.99 \text{ ms}$$

$$\text{Reading all data then takes: } 401 * 400 * 10.99 \text{ ms} = 1,762,796 \text{ ms}$$

$$\text{Total time for merge(read+write)} = 3,525,592 \text{ ms}$$

Total time = 1,604,800ms + 3,525,592ms = 5,130 seconds.

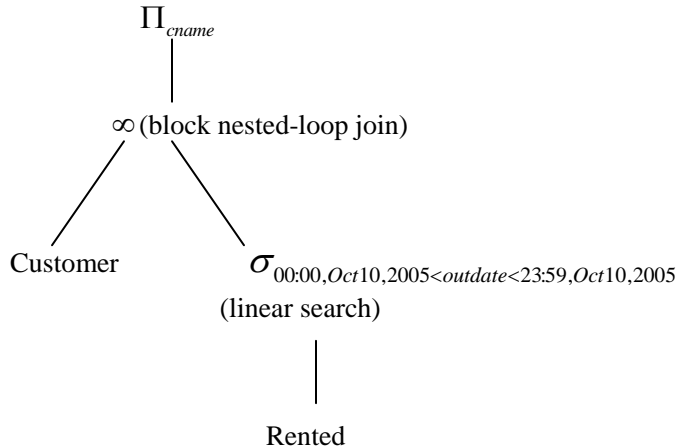
Since the relations are in sorted order now, tuples with the same value on the join attribute are in consecutive order. Thereby, each tuple in the sorted order needs to be read only once, and, as a result, each block is also read only once. So the cost for merging these two sorted relations is the cost of scanning both sorted relations, which adds another  $(40 \text{ GB} + 1 \text{ GB}) / (50 \text{ MB/s}) = 820$  seconds, for total of 5,950 seconds. So this is slightly better than the blocked nested-loop join in (a)

**3 (a)** Query 1: List the name(s) of customer(s) who rented out a movie on Oct,10,2005.

Query 2: List the names of all movies that have copies at the "MetroTech" branch.

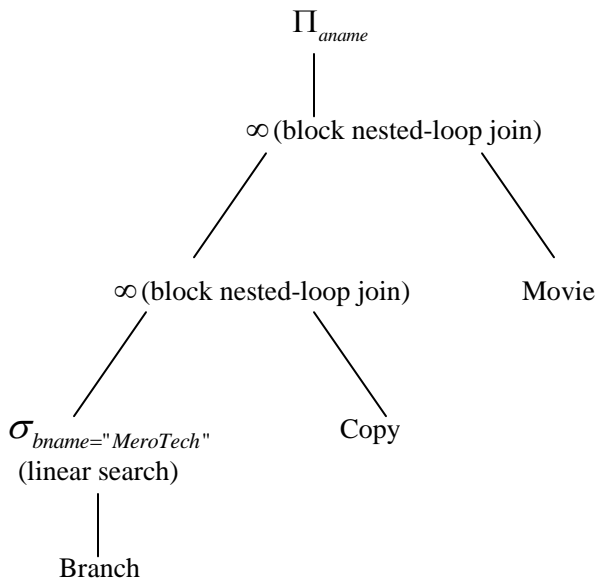
**(b)** Since all the relations are unclustered (i.e. not sorted in any way), we have to use the linear search algorithm. That is, scan the complete table and test all the records to see whether they satisfy the selection condition. Considering that block nested-loop join always beats the basic nested-loop join, we choose the block nested-loop join with the optimization on page 545.

**Query 1:** Cost for selection on *Rented* table is 10,000,000 block accesses. After the first selection, the number of blocks of relation *Rented* is  $b_{Rented} = 0.1\% * 10,000,000 = 10,000$  blocks, while the number of blocks of relation *Customer* is (of course) still  $b_{Customer} = 250,000$  blocks. Note that *Rented* fits into main memory after the first selection. The query evaluation plan is as follows:



The cost for the first selection would be  $6\text{ms} + 40\text{GB}/(50\text{ MB/s}) \sim 800$  seconds. Afterwards, the cost for the join would be the cost of reading *Customer*,  $1\text{GB}/(50\text{ MB/s}) = 20$  seconds, since the surviving tuples of *Rented* are already in memory. Thus, the total cost is about 820 seconds.

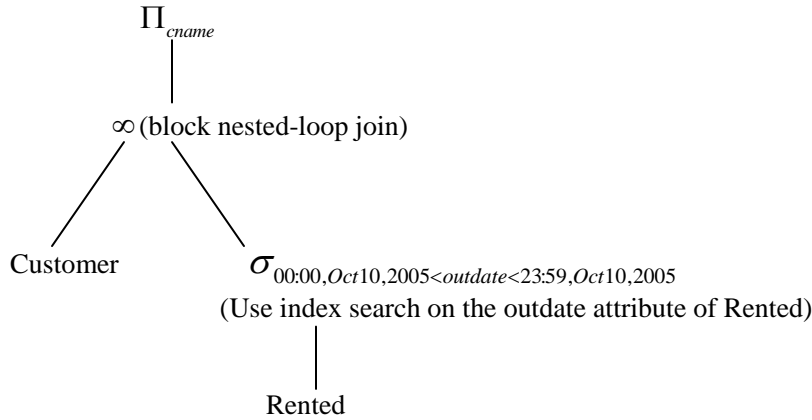
**Query 2:** The time cost for the selection of “bname=MetroTech” is the cost of scanning the small *Branch* table,  $6\text{ms} + 250 * 0.08\text{ms} = 26\text{ms}$ . After this selection there would be only 1 tuple left. Thus, the first join between *Branch* and *Copy* would simply consist of a scan of *Copy*, at a cost of  $(400\text{MB})/(50\text{ MB/s}) = 8$  seconds, and the result would consist of 1000 tuples (the average number of copies in a single branch). Thus, the result again fits into memory, and the next join with *Movies* also consists of a simple scan of the *Movies* table, at a cost of  $10\text{MB}/(50\text{ MB/s}) = 0.2$  seconds, for a total cost of about 8.226 seconds. (**Note:** Of course we could reduce the size of each tuple in the intermediate result by getting rid of some attributes after the first join through projection; this does not significantly change the cost.) Here is the query evaluation plan:



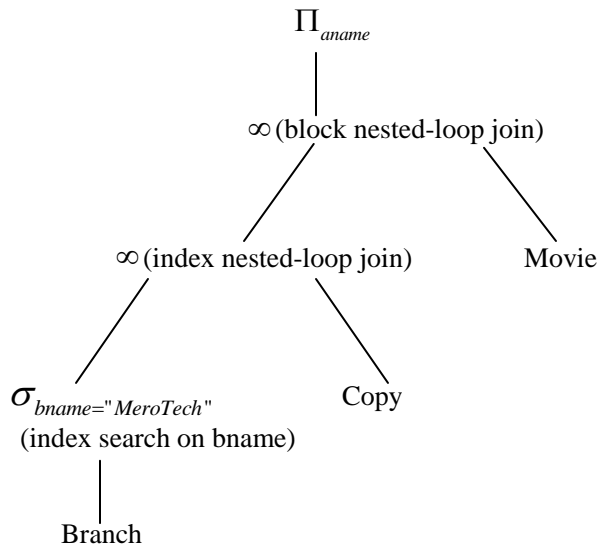
**(c) With index structures as described.**

**Query 1:** Since we have a sparse clustered index on the *outdate* attribute of *Rented*, we can use the index to retrieve all the tuples in *Rented* which have an *outdate* of Oct,10,2005. The result set size will be  $1,000,000,000 * 0.1\% = 1,000,000$  tuples. These 1 million tuples take up 10,000 blocks or 40MB. Let’s assume attribute *outdate* is at most 12 bytes. Then the height of the sparse clustered B+tree on *outdate* would be about 5 (approximate is enough). The time used to do the initial selection is then  $5 * 6.08 + 40\text{MB}/(50\text{ MB/s}) = 830.32$  ms. Since we have an unclustered index on *cid*, we could

try an indexed nested-loop join between *Customer* and *Rented*; however, this would result in 1,000,000 lookups into this index at a cost of about 30ms each if this index is height 4, for a total of about 30,000 seconds. It is much better to again use a nested-loop join, at the same cost of about 20 seconds as in 3(b). Thus, the total is about 20.83 seconds, compared to 820 seconds without indexes. Here is the evaluation plan:

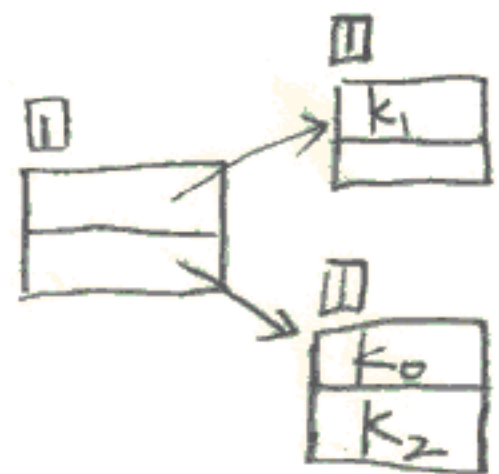


**Query 2:** Since we have an unclustered index on the *bname* attribute of *Branch*, we could do an index search to find the tuple associated with “MetroTech” (only one branch has “MetroTech” as the *bname*). Since *Branch* is the smallest table, the index would have height of about 2, which means that the selection would cost only  $3 * 6.08ms = 18.24ms$ . Now consider the first join, which is between the resulting tuple and *Copy*. Since *Copy* has an unclustered index on *bid*, we use this in an index nested-loop join. The height of this index is 3, so a lookup through this index takes about 18 ms to find the right index entries in the leaf level. There are approximately 1000 entries in *Copy* with the same bid as “MetroTech”, so afterwards we need to retrieve 1000 tuples from *Copy* at a cost of  $1000 * 6.08ms \sim 6.08$  seconds. After this join, we join the results with *Movie*. This can only be implemented by using block nested-loop, since no index exists for attribute *mid*. However, the result from the previous join consists of only 1000 tuples that clearly fit in main memory, so we only have to scan the small *Movie* relation at a cost of  $(10MB)/(50 MB/s) = 0.2$  seconds. Thus, the total cost is now only about 6.3 seconds, compared to 8 seconds without indexes. The evaluation plan is as follows:

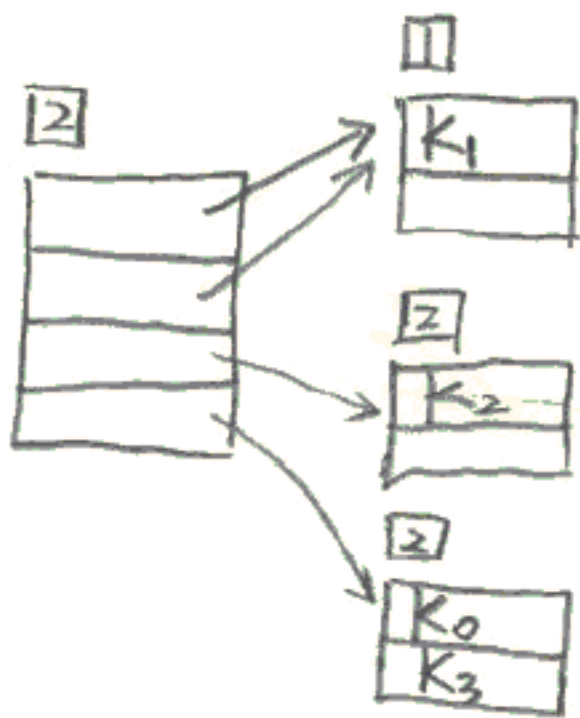


4. See extra page.

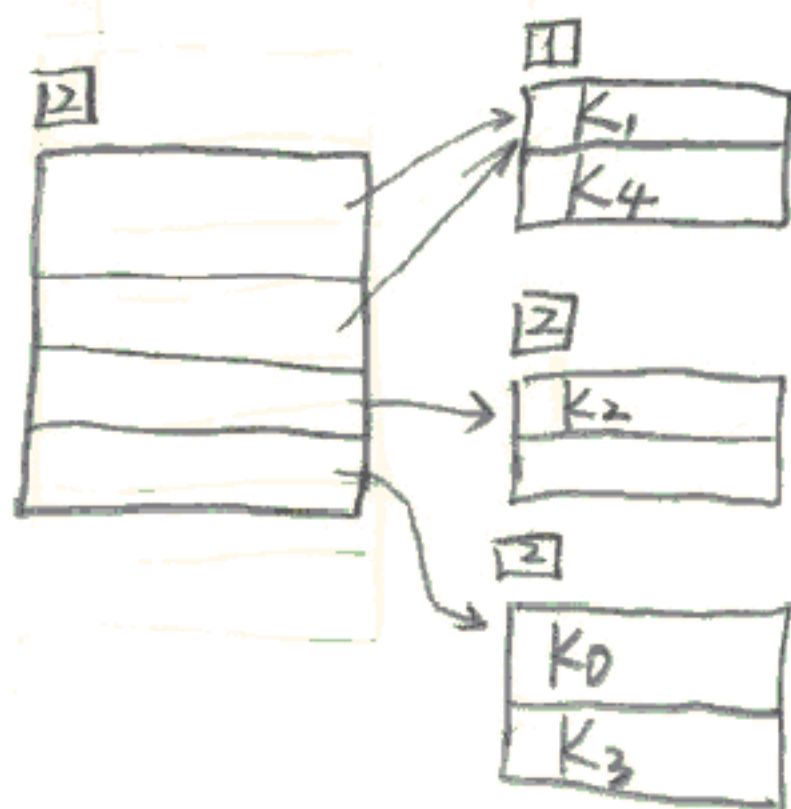
4.



insert  $k_3$

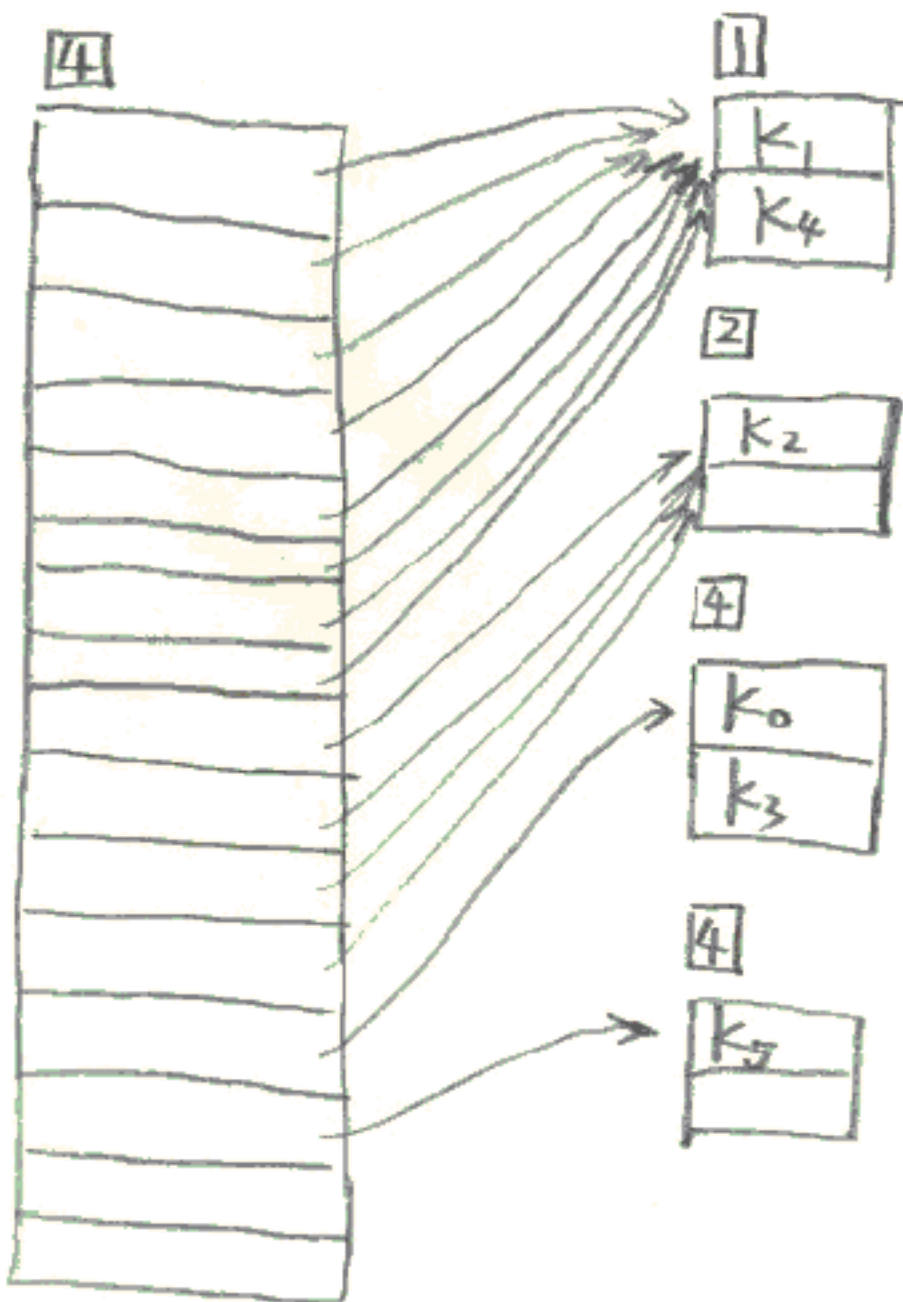


insert  $k_4$



After

insert  $k_5$



insert  $k_6$

