Fall 2007 CS186 Discussion Section: Week 4, 09/17 - 09/21

Your Friendly TAs

September 27, 2007

1 File Organizations

1. What is a clustered index? For which of the 3 data entry alternatives can we have a clustered index?

   A clustered index is one that is organized so that ordering of the data records within the corresponding file are the same as – or close to – the ordering of data entries in the index. Alternative 1 by definition is clustered. An index that uses Alternative 2 or 3 *can* be a clustered only if the data records are sorted on the search key field. Usually though, this is not the case.

2. You are about to create an index on a relation. Discuss some considerations that guide your choices of the following.
   (a) The choice of primary index
   (b) Clustered vs unclustered indexes
   (c) Hash vs tree indexes
   (d) The use of a sorted file vs a tree based index
   (e) Choice of search key for the index.

   (a) The choice of the primary key is made based on the semantics of the data. If we need to retrieve records based on the value of the primary key, as is likely, we should build an index using this as the search key. If we need to retrieve records based on the values of fields that do not constitute the primary key, we build (by definition) a secondary index using (the combination of) these fields as the search key.
   (b) A clustered index offers much better range query performance, but essentially the same equality search performance (modulo duplicates) as an unclustered index. Further, a clustered index is typically more expensive to maintain than an unclustered index. Therefore, we should make an index be clustered only if range queries are important on its search key. At most one of the indexes on a relation can be clustered, and if range queries are anticipated on more than one combination of fields, we have to choose the combination that is most important and make that be the search key of the clustered index.
   (c) If it is likely that ranged queries are going to be performed often, then we should use a B+ tree on the index for the relation since hash indexes cannot perform range queries. If it is more likely that we are only going to perform equality queries, for example the case of social security numbers, than hash indexes are the best choice since they allow for the faster retrieval than B+ trees by 2-3 I/Os per request.
   (d) First of all, both sorted files and tree-based indexes offer fast searches. Insertions and deletions, though, are much faster for tree-based indexes than sorted files. On the other hand scans and range searches with many matches are much faster for sorted files than tree-based indexes. Therefore, if we have read-only data that is not going to be modified often, it is better to go with a sorted file, whereas if we have data that we intend to modify often, then we should go with a tree-based index.
   (e) A composite search key is a key that contains several fields. A composite search key can support a broader range as well as increase the possibility for an index-only plan, but are more costly to maintain and store. An index-only plan is query evaluation plan where we only need to access the indexes for the data records, and not the data records themselves, in order to answer the query.

Obviously, index-only plans are much faster than regular plans since it does not require reading of the data records. If it is likely that we are going to performing certain operations repeatly that only require accessing one field, for example the average value of a field, it would be an advantage to create a search key on this field since we could then accomplish it with an index-only plan.

3. Choose one of the basic file organizations (heap, sorted, or hash) that is best for a large file, for each of the following scenerios:
   (a) Search for records based on a range of field values. (Sorted)
   (b) Perform inserts and scans where the order of records does not matter. (Heap)
   (c) Search for a record based on a particular field value. (Hash)

4. Fill in the I/O costs for the operations listed in the table. Assume that the relation R takes up $p(R)$ blocks of disk space and that it contains $t(R)$ tuples, the equality and range searches are performed on column $R.A$ which contains $v(R.A)$ unique values, and that the tree indices, again on column $R.A$ have height h and l leaf blocks. Calculate the costs in terms of I/Os – not time, as it is done in your books.

File Type	Scan	$R.A = c$	$R.A > c$	Insert	Delete
Heap	$p(R)$	$1/2p(R)$	p(R)	1	$1/2p(R)+1$
Sorted	$p(R)$	$\log_2 p(R) + \frac{p(R)}{v(R.A)}$	$\log_2 p(R) + \frac{max-c}{max-min}p(R)$	$\log_2 p(R) + p(R)$	$\log_2 p(R) + p(R)$
Clust. Tree	$h + p(R)$	$h + \frac{p(R)}{v(R.A)}$	$h + \frac{max-c}{max-min}p(R)$	$2h$	$2h$
Unclust. Tree	$h + l + t(R)$	$h + \frac{l}{v(R.A)} + \frac{t(R)}{v(R.A)}$	$\frac{max-c}{max-min}(l + t(R))$	$2h+1$	$2h+1$

As you can see by the assumptions, we're taking a different twist while filling this table, which we will revisit when we'll talk about Cost Estimation for Query Optimization (Chapters 14 & 15). In order to understand some of the formulas, we need to introduce the notion of *reduction factor* of an operator as *the percentage (in terms of tuples or scanned blocks) returned from the execution of the operator, over the total number of tuples (or blocks) of the initial relation.* In other words, $rf = \frac{output\ size}{input\ size} \in [0, 1]$. We will also assume that the distribution of unique values ($v(R.A)$) of attribute $R.A$ over the blocks of R is uniform. Finally, for the clustered tree case, we will regard that the index follows alternative one (p. 276 of your books), meaning that the data entries in the leaf nodes are the actual data records (in essence, the leaves are the blocks of the heap file), while for the unclustered tree case we follow alternative two (the data entry is a $< key, rid >$ pair, with the *rid* pointing to the corresponding record in the (separate) heap file).

Given the above, let us attempt to explain the equality search operation ($R.A = c$) for a sorted file. We need to access $\log_2 p(R)$ pages on average for our binary search, to locate the block with the first qualifying tuple. We then need to retrieve all the blocks containing tuples with the same value. The number of such blocks corresponds to the *reduction factor* of the equality selection operator, times the number of blocks of R, $\frac{1}{v(R.A)}p(R)$. The same reasoning applies for the range selection predicate, only now, the percentage of blocks containing qualifying tuples will be $rf' = \frac{max-c}{max-min}$, where max and min are the maximum and minimum values of the column $R.A$.

For the clustered tree case, besides the cost we have to pay to retrieve all blocks with qualifying tuples, we have to descend the tree. The cost for the latter is h, the height of the tree. The worst case for insertion / deletion is that we have to split/merge every node in the path from the root to the leaves: h to reach the leaves, and h going back up performing the splits.

Finally, for the unclustered tree case, each data entry contained in a leaf page can point to a *different* block in the file (remember that in an unclustered index, the ordering between the data entries and the tuples stored in the heap file of the relation does not match). Thus, for the equality search case, we have to descend the tree (h), visit $\frac{1}{v(R.A)}l$, the leaf pages containing data entries with key value equal to c, and then follow the pointer of each qualifying data entry $\frac{1}{v(R.A)}t(R)$, that will take us to possibly different heap file blocks.

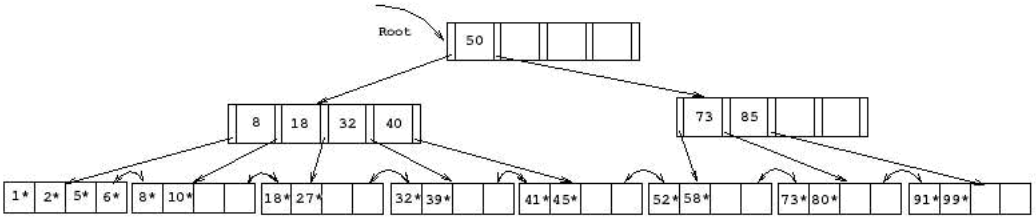

Figure 1: B+ tree

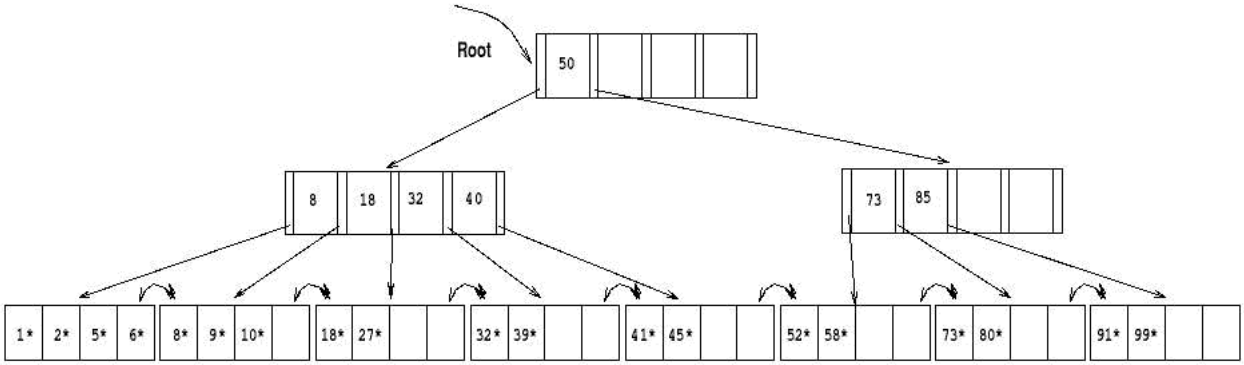

Figure 2: B+ tree, after the insertion of data entry 9.

2 Tree Indices

1. Consider the B+ tree index of order d = 2 shown in figure below:

   (a) Show the tree that would result from inserting a data entry with key 9 into this tree.

   (b) Show the B+ tree that would result from inserting a data entry with key 3 into the original tree. How many page reads and page writes does the insertion require?

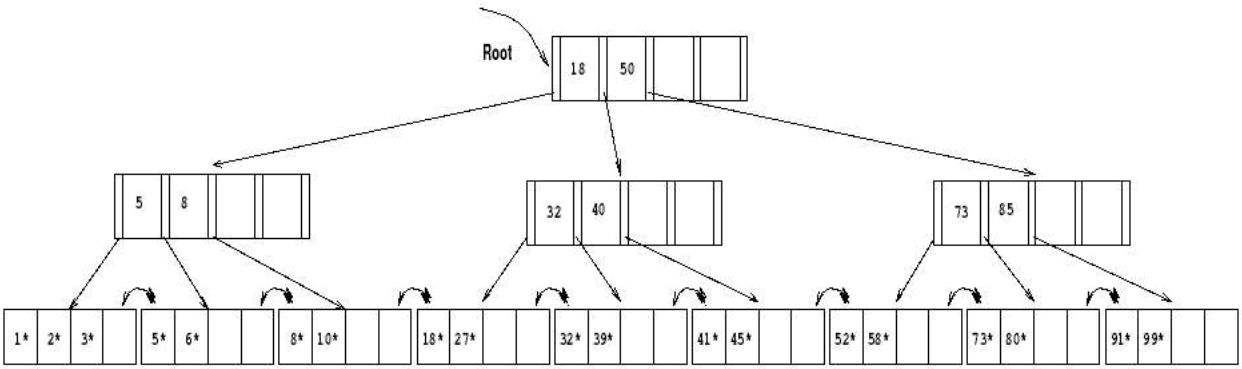

Figure 3: B+ tree, after the insertion of data entry 3.