CS 186 Discussion Section
Week 5

Peter Alvaro and Kuang Chen

February 23, 2009

File Organization

Data entry alternatives

The three alternatives for data entries with search key k are:

1. data entry $k*$ is an actual data record matching the key
2. data entry is a $< k, rid >$ pair where rid is the record id of a data record matching the key
3. data entry is a $< k, rid - list >$ pair, where $rid - list$ is a list of record ideas matching the key

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.

Discussion question: 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) The use of a sorted file vs a tree based index
(d) 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) 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.

(d) 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 repeatedly 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.

Indexes

What are the main differences between ISAM and B+ tree indexes?

The main difference between ISAM and B+ tree indexes is that ISAM is static while B+ tree is dynamic. Another difference between the two indexes is that ISAMs leaf pages are allocated in sequence.

How many nodes must be examined for equality search in a B+ tree? How many for a range selection? Compare this with ISAM.

For equality search in a B+ tree, h nodes must be examined, where h = height of the tree. For range selection, number of nodes examined = h + m 1, where m = number of nodes that contains elements in the range selection. For ISAM, the number of nodes examined is the same as B+ tree plus any overflow pages that exist.

B+-tree Insert algorithm

Remember that: 50% of a node must be full

1. Starting at the root, search to a leaf node.
2. If the key value is in the leaf node you have a duplicate, stop.
3. If there is room in the node, put the new key value there, stop.
4. If there is not room, allocate a new leaf node.
5. Redistribute the keys and pointers such that half the keys go into the old node and half into the new node. If there is an odd number of keys put the extra key in the old node. Order the keys so that the smaller ones go into the old node and the larger ones go into the new node. All the keys in the old node will be smaller than any key in the new node.
6. Duplicate the smallest key in the NEW node into the parent non-leaf node.
7. If there is no parent node create a new root node, adjust the pointers and stop. If there is room in the parent node, adjust the pointers, stop.
8. If the parent node is full, allocate a new non-leaf node.

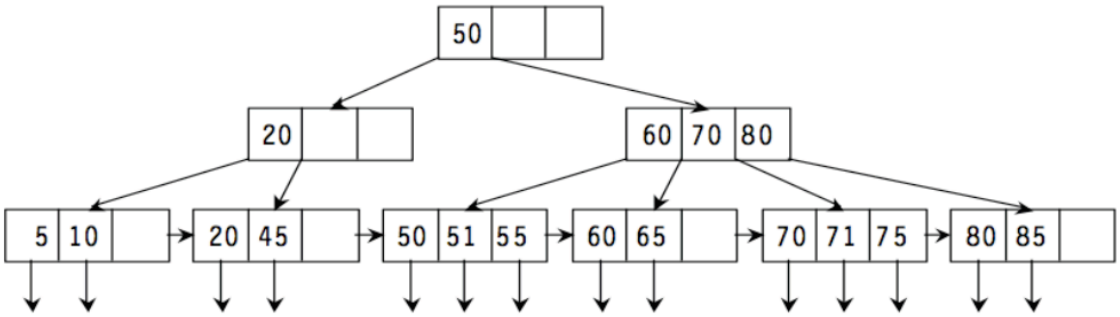
9. Redistribute the keys that were in the old non-leaf node, plus the new key, into the two non-leaf nodes, except the middle-valued key. Move the middle-valued key up the tree into the parent node. If there is an odd number of keys to be redistributed into the two nodes, put the extra one in the old(left) non-leaf node.

10. go to step 7.

11. Check that the tree is still balanced.

B+-tree example

For each of the following modifications, show the result B+-tree obtained by applying the modification to the B+-tree shown below. Suppose that the maximum fan-out is 4. (Always start with the B+-tree shown below; do not apply the modifications to the result of previous modifications.)



1. Insert 21.

2. Insert 79.

3. Delete 5 then 10 then 20.

One more example (full tree)

Insert 19 into the following tree.