Database Systems, WS 08/09
Information Systems Group
Saarland University

Prof. Dr. Jens Dittrich
Teaching Assistant: Josiane Xavier Parreira
Assignment 3

1 File Organizations

If you were about to create an index on a relation, what considerations would guide your choice? Discuss:

1. Direct vs. indirect index.
2. Clustered vs. unclustered indexes.
3. Hash vs. tree indexes. In particular, discuss how equality and range searches work.
4. The use of a sorted file rather than a tree-based index.

Solution Sketch

1. In direct indexes the leaf nodes contains the actual data records, whereas in indirect indexes leaf nodes are just pointers to the data pages containing the records. In indirect indexes, insertions and deletions are easier to handle, however a good performance for range queries can not be guaranteed, since sequential pointers in the leaves might point to pages that are not sequentially stored in disk.

2. 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.

3. 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.

   Hash indexes are especially good at equality searches because they allow a record look up very quickly with an average cost of 1.2 I/Os. B+-tree indexes, on the other hand, have a cost of 3-4 I/Os per individual record lookup. Assume we have the employee relation with primary key eid and 10,000 records total. Looking up all the records individually would cost 12,000 I/Os for Hash indexes, but 30,000-40,000 I/Os for B+- tree indexes. For range queries, hash indexes perform terribly since they could conceivably read as many pages as there are records since the data is not sorted in any clear grouping or set. On the other hand, B+-tree indexes have a cost of 3-4 I/Os plus the number of qualifying pages or tuples, for clustered or unclustered B+-trees respectively. Assume we have the employees example again with 10,000 records and 10 records per page. Also assume that there is an index on sal and query of age ¿ 20,000, such that there are 5,000 qualifying tuples. The hash index could cost as much as 100,000 I/Os since every page could be read for every record. It is not clear with a hash index how we even go about searching for every possible number greater than 20,000 since decimals could be used. An unclustered B+-tree index would have a cost of 5,004 I/Os, while a clustered B+-tree index would have a cost of 504 I/Os. It helps to have the index clustered whenever possible.

4. 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.

2 B+ tree Operations

Consider the B+ tree index of order d= 2 shown in Figure 1.

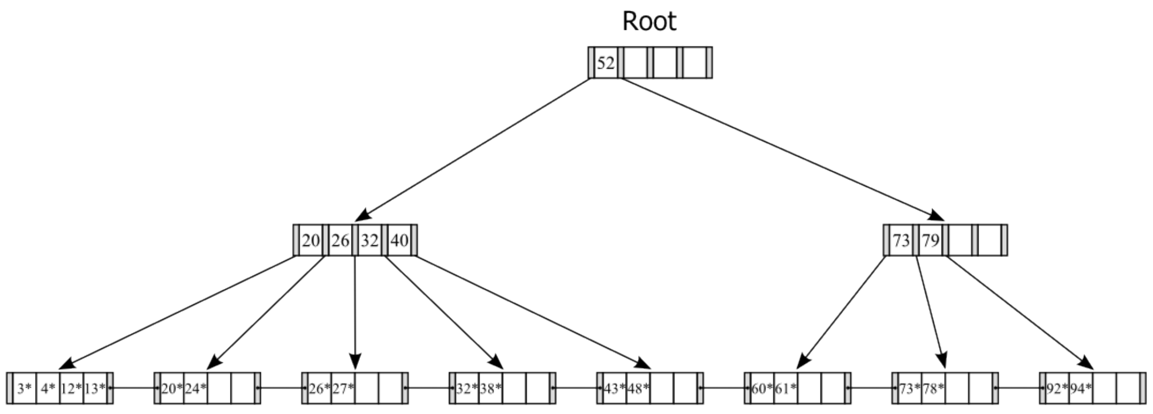

Figure 1: Example of a B+ tree.

1. Show the tree that would result from inserting a data entry with key 23 into this tree.
2. Show the B+ tree that would result from inserting a data entry with key 6 into the original tree. How many page reads and page writes will the insertion require?
3. Show the B+ tree that would result from deleting the data entry with key 20 from the original tree, assuming that the left sibling is checked for possible redistribution.
4. Show the B+ tree that would result from deleting the data entry with key 20 from the original tree, assuming that the right sibling is checked for possible redistribution.
5. Show the B+ tree that would result from starting with the original tree, inserting a data entry with key 50 and then deleting the data entry with key 60.
6. Show the B+ tree that would result from deleting the data entry with key 92 from the original tree.
7. Show the B+ tree that would result from starting with the original tree, inserting a data entry with key 63, and then deleting the data entry with key 92.
8. Show the B+ tree that would result from successively deleting the data entries with keys 32, 38, 43, 48, and 73 from the original tree.

Solution Sketch

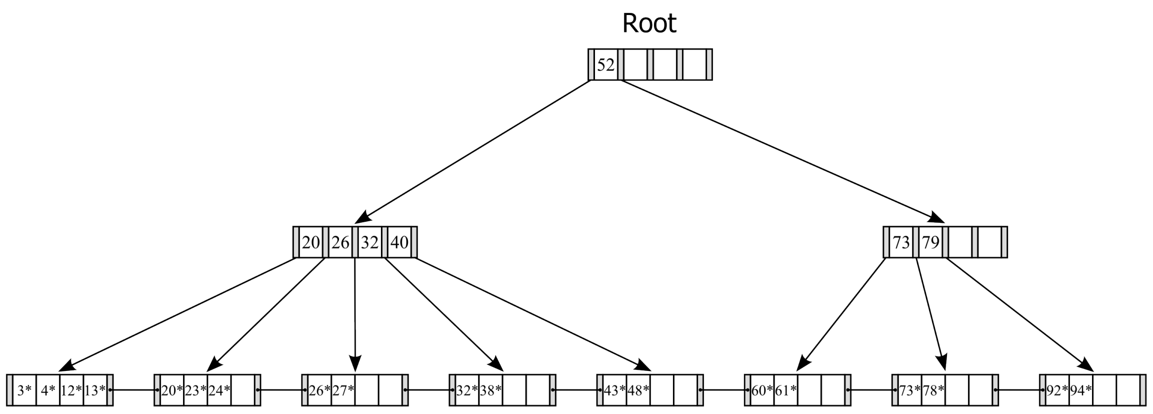

Figure 2: Solution 1

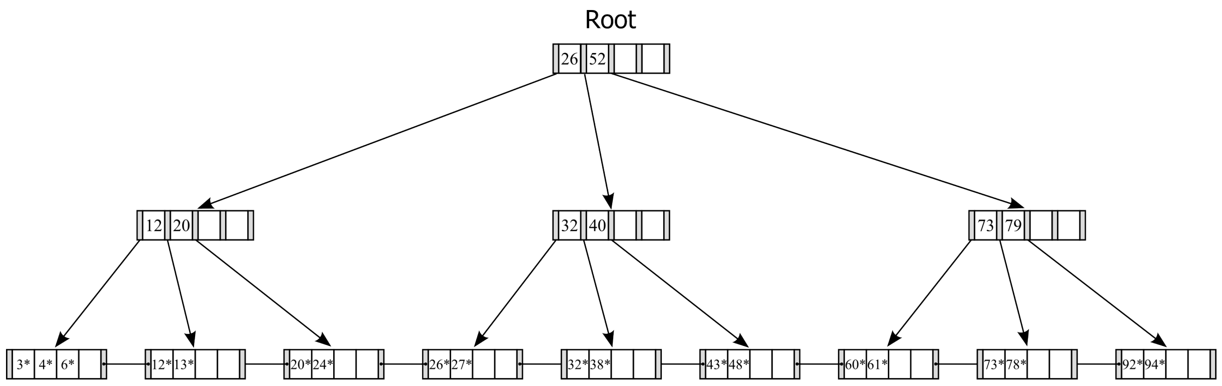

Figure 3: Solution 2

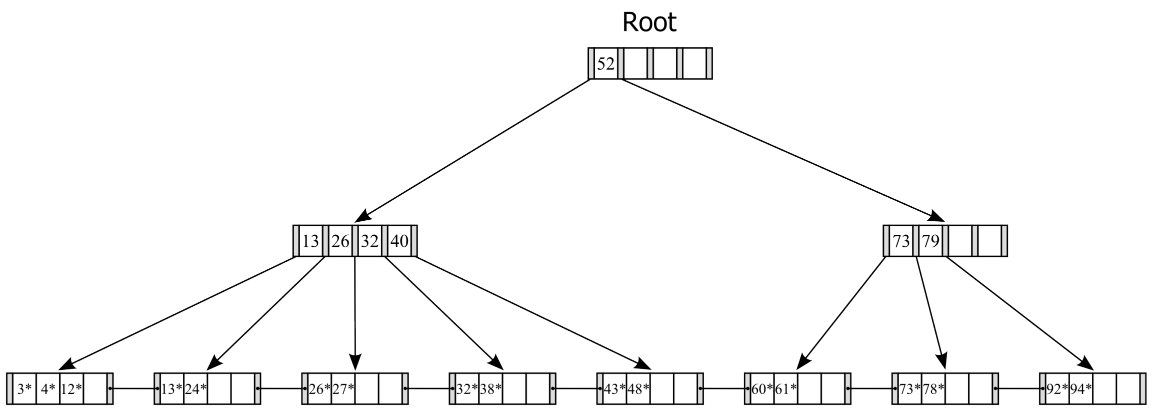

Figure 4: Solution 3

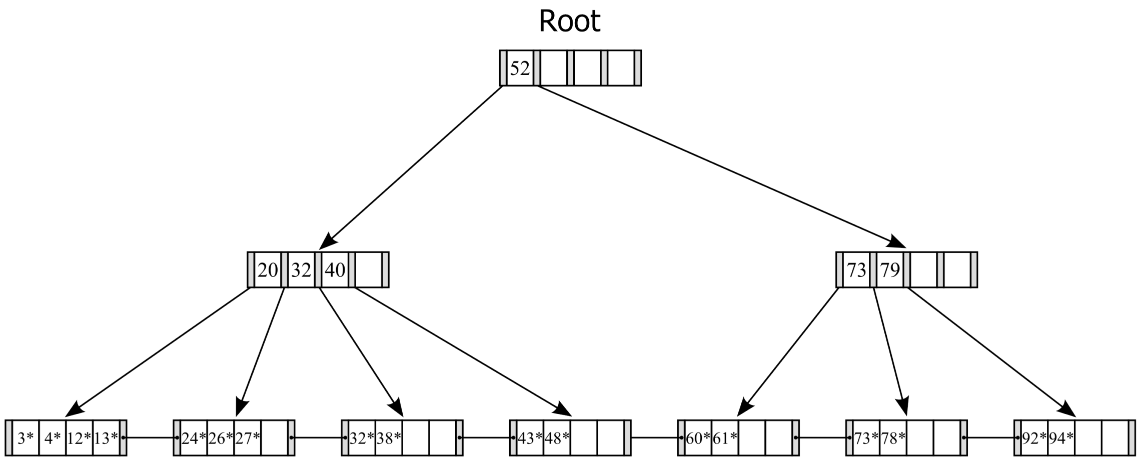

Figure 5: Solution 4

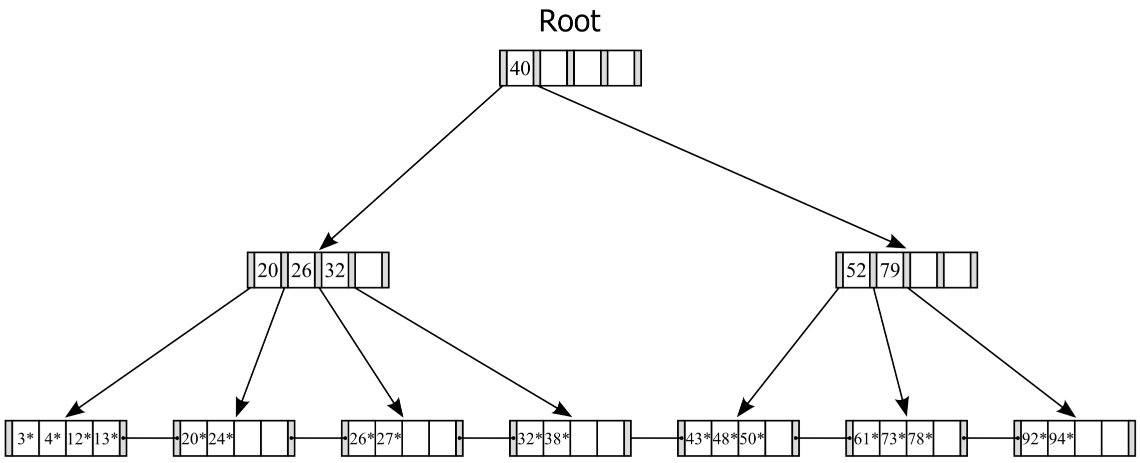

Figure 6: Solution 5

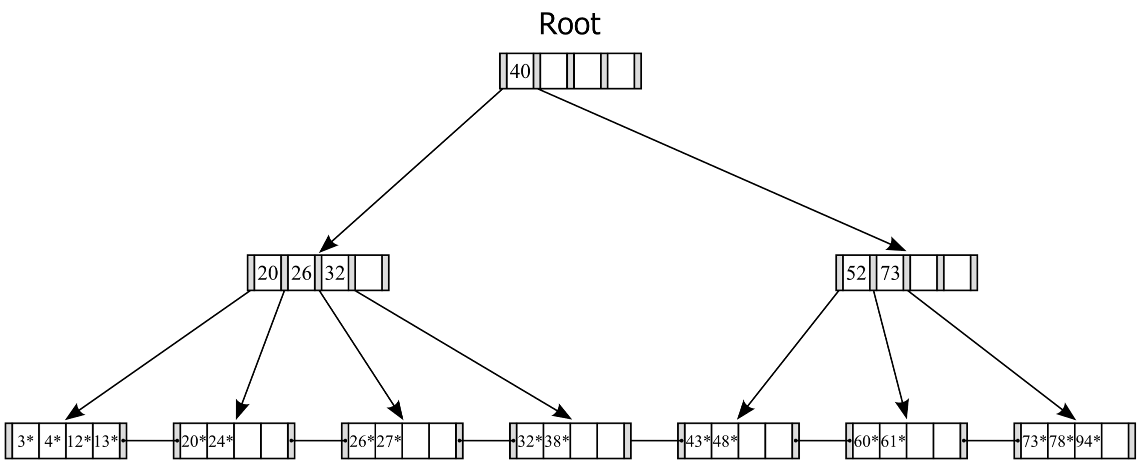

Figure 7: Solution 6

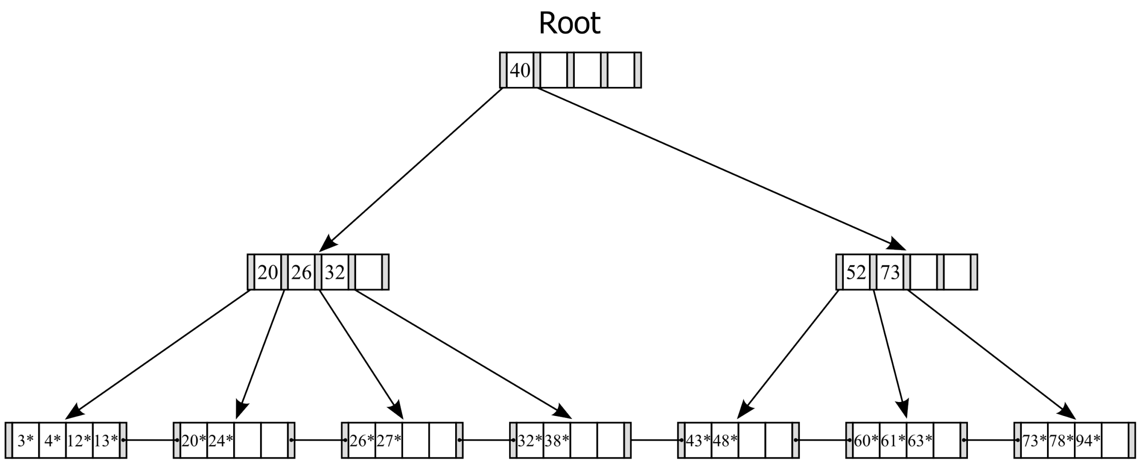

Figure 8: Solution 7

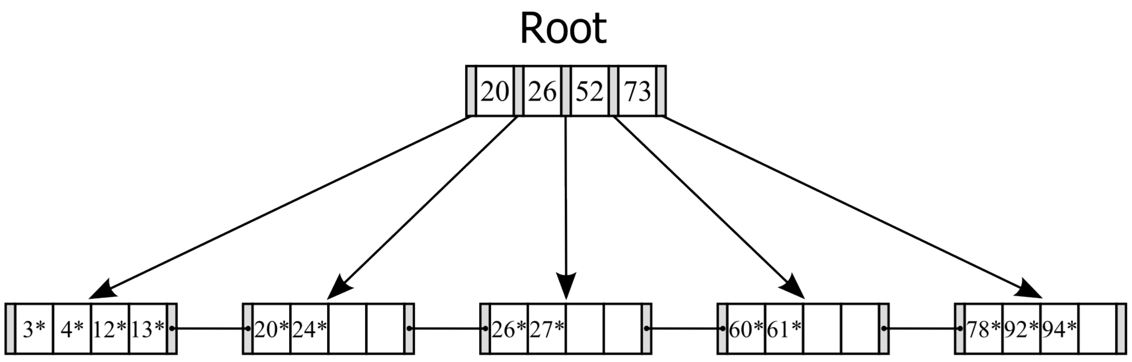

Figure 9: Solution 8

3 Shared scans vs. B+ trees

Concurrent queries with common sequences of disk accesses can make maximal use of a buffer pool by using *shared scans*. The idea of shared scans is the following: given a batch of k queries, it sequentially scans the table with the data records and, for each record, it evaluates all k queries and stores the intermediate results. If the end of the table is reached the scan resumes from the beginning of the table (one can see it as a "cyclic scan"). If the table is stored on disk, shared scans improves I/O since only sequential accesses are performed.

Given a table with 1 Mio. records sequentially stored on disk, each record of the size of 100B. Assume the hard disk has a sequential read performance of 100MB/sec. k queries are evaluated at a time.

1. How long does it take until all queries are processed? Do not consider CPU time for now only consider I/O cost.
2. An alternative to shared scans is to use a B+ tree. Assume that a random access on disk can be done in 5ms and all intermediate index nodes of the tree are available from the DB-buffer. How long does it take to process k queries on the B+-tree?
3. Which solution is better in terms of I/O-time? How would you choose k for shared scans?
4. Now, in addition to the I/O cost, let's also consider CPU cost. Assume that it takes 100 nanoseconds for the CPU to process a record for each query during the scan. Similarly, assume that it takes 20 nanoseconds to perform a key comparison in main memory inside any tree node. For which values of k will the shared scan be better than the B+-tree?
5. What other gain can you imagine by using a scan instead of a tree-structure?

Solution Sketch

1. For processing the set of k using shared scans we just need to read the whole table once. Given that there is 100MB of data (1Mio. records x 100B) that can be read at a speed of 100MB/sec, it will take 1 second to perform a full scan.
2. With B+ trees we can execute only one query at the time. However, the processing of each query is very efficient since we need only one random access per query, therefore it takes only 5ms to process each query. Therefore, for k queries it would take (k x 5ms).
3. The best I/O performance depends on the value of k. With shared scan we can do k queries in 1 second, but with the same time we can process 200 queries using B+ trees!!! Therefore, unless k is bigger than 200, shared scan is not a good alternative.
4. In shared scans we need to compare each record against all k query selection conditions. In a table with 1Mio. records it will take

   $$1 \text{ Mio. x } k \text{ x } 100 \text{ nanoseconds} = k \text{ x } 100\text{ms}$$

   until we can return the results of all queries. Note that all queries have to wait till all of them were compared against a record before moving to the next one.

   The number of comparisons in a B+-tree is $\log_2(N)$, where N is the number of entries. Thus for 1 Mio. records, we need about 20 comparisons for each query, which takes 20 x 20ns = 0.4ms. When using B+-trees the CPU can serve at most 2,500 queries per second, whereas with shared scans the CPU can serve at most only 10 queries per second.

   In principle there is no good reason to use a shared scan. If we only model I/O, shared scans is the clear winner, however the linear complexity of CPU time for scanning will loose against the log-complexity of an index.

5. Scan may make sense if queries are not using a single attribute but may refer to a large number of attributes: a low selectivity may kill the index. Furthermore, when inserting and updating, maintaining a different index for each attribute may become very expensive. In general: updating while scanning may be better. Assume you update a UdS student DB by a value that appears very often like "all students that live in Saarbrücken". Then you will touch huge portions of the data and a scan will be more effective.

4 Project: Main-Memory Efficient Tree-index

This task must be performed in small teams from 2-4 people. Please make sure to register your group ASAP. This task is part of the project as defined in the first week of this lecture.

Task: Develop a cache-efficient main-memory resident tree-index. The index should be able to perform point queries, range queries, inserts, updates, and deletes. Assume a key range 0,..., 10 million. Values are any 4 byte integer. Please do not use any multi-threading for the moment. Persisting changed entries is not necessary.

1. Take the B+-tree framework available from the lecture Web-site as a starting point. Alternatively, you may develop your index from scratch (whatever suits you best).
2. Implement at least one of the techniques presented in the lecture including CSB+-trees, pB+-trees, fpB+-trees, or CSS-trees.
3. Ensure functional correctness of your index by comparing the result set produced by your index against a reference set.
4. Execute the workload available from the lecture web-site. Try to optimize performance (wall clock time) of your method for that workload.

Keep in mind that the goal of this exercise is to further use your index for the final task as it will be defined by the ACM programming contest.

Your code has to run on a linux server. Only Java, C, C++ may be used as programming languages. Your code should be handed-in on Thursday, Nov 27, midnight latest. This will allow us to run all code submissions on the same server. Note that the benchmark used by us for this evaluation will slightly differ from the published benchmark. However, the characteristics will be the same.

Solution Sketch

Programming exercise...