

# Data storage

# Tree indexes

Rasmus Pagh

February 7 lecture



# Access paths

- For many database queries and updates, only a small fraction of the data needs to be accessed.
- Extreme examples are looking or updating the single tuple with a given key value.
- General question: *"How do we access the relevant data, and not (much) more?"*
- Term.: Need efficient **access paths**.

# Choices influencing access paths

- How are the tuples in the relations arranged? (*this lecture*)
- What kinds of indexes exist? (*this lecture and later in course*)
- Have we pre-computed (parts of) the information needed? (*later in course*)

# Data storage in sequential files

- Storing meta-data: See textbook.
- A *relation* may be stored as a string, concatenating the tuples:  
(3,"Kurt",27),(11,"Jim",27),(89,"Elvis",42)
- The tuples may be sorted according to some attribute(s), or unordered.
- *Reading* data is very efficient.
- Problem:
  - How to accommodate changes to data: Insertions, deletions, modifications.

# From your toolbox: Linked lists

- Linked (or doubly linked) lists give great flexibility:
  - Insert a new tuple
  - Delete a tuple
  - Change the size of the data in a tuple
- Problem:
  - One “random” memory access per item when reading the list.

# Memory access cost

- Recall from last week:
  - The speed of the CPU is 2-3 orders of magnitude larger than the speed of a RAM access (non-sequential).
  - The speed of RAM access is 5-6 orders of magnitude faster than disk access (non-sequential).
- RAM vs disk analogy: Go to Australia to borrow a cup of sugar if your neighbor is not home!

# Analysis and comparison

- *I/O model*, i.e., we count the number of block reads/writes.
- For now, assume that tuples have fixed length. Let  $B$  denote the number of tuples per block, and  $N$  total #tuples.

<i>Storage method</i>	<i>Tablescan I/O</i>	<i>Update I/O</i>
Sequential file	$N/B$	$2N/B$
Linked list	$N$	$O(1)^*$

(sorted)

\* If update location known

## Aside: Main memory DBMSs

- It is increasingly common to store all (or all the most frequently accessed parts) in main memory.
- Still, non-volatile memory (hard drive, flash,...) is needed to provide durability in case of system crashes.
- Time-space trade-off persists: Slower memories are cheaper, i.e., can be used to store much more data.
- Also recall that even main memory, like disks, is *blocked*.

## Problem session

- Think of a way of storing *sorted* relations on disk such that:
  - Reading all data in the relation is efficient (close to the best possible efficiency).
  - Inserting and deleting data is efficient (assume the location of update is known).
- Do the analysis in the *I/O model*.
- As before, assume that tuples have fixed length, and let  $B$  denote the number of tuples per block.

# Amortized analysis

- Instead of analyzing the worst-case cost of a single operation, *amortized analysis* looks at the average cost of a sequence of operations.
- Example: An *unordered* sequential file
  - must use 1 I/O for some insertions, but
  - can do B insertions in 1 I/O using an internal memory buffer
- Conclusion: The amortized cost of an insertion is  $1/B$  I/Os. Tiny!
- Same thing for **your** proposal?



# Exercise from hand-out

- *Representation of relations*
- Questions a), b), and c).

# Searching sorted relations

- Suppose a sequential file of  $N$  tuples is sorted by an attribute,  $A$ .
- It can be searched for an  $A$ -value using binary search, in  $\log_2(N/B)$  I/Os.
- A (sorted) linked list may require a full traversal to locate an  $A$ -value.

<i>Storage method</i>	<i>Tablescan</i>	<i>Update</i>	<i>Search</i>
Sequential file	$N/B$	$2N/B$	$\log_2(N/B)$
Linked list (blocked)	$O(N/B)$	$O(1)$	$O(N/B)$

(sorted)

# Adding a directory

- A sorted linked list may be searched more quickly if we have a *directory*:
  - A sorted list with a *representative key* from each block (e.g., smallest key).
  - A pointer to the block of each key.
- But how do we search the directory?
  - Not so important if it fits in RAM.
  - Otherwise, it seems that this problem is *the same* as the original problem...
  - Is there any progress in this case?  
(Discussion on board.)



# Coping with updates

- Easy solutions:
  - Insertions: If room in the relevant block, ok. Otherwise insert in an *overflow chain*. (The *ISAM* solution.)
  - Deletions: Just mark deleted, don't try to reuse space.
- More robust solution based on the lesson from linked lists:

*Introduce some "slack" to allow efficient updates.*

## B<sup>+</sup>-tree invariants on nodes

- Suppose a node (stored in a block) has space for  $B-1$  keys and  $B$  pointers.
- Don't want blocks to be too empty: Should be at least half full.
- Let's see how this works out! (Board.)
- Only possible with an exception: The root may have as little as 1 key and 2 non-null pointers.

# B<sup>+</sup>-tree properties

- Search and update cost  $O(\text{depth})$  I/Os.
- A B<sup>+</sup>-tree of depth  $d$  holds at least  $2 (B/2)^{d-1}$  keys
- Rewriting, this means that  $d < \log_{B/2}(N/2) = O(\log_B N)$
- Often the top level(s) will reside in internal memory. Then the operation time becomes  $O(\log_B(N/M))$ .



# Problem session

- Argue that B<sup>+</sup>-trees are optimal in terms of search time among pointer-based indexes:
  - Suppose we want to search among N keys, that internal memory can hold M pointers, and that a disk block can hold B pointers and B keys.
  - Further, suppose that the only way of accessing disk blocks is by following pointers.
  - Show that a search takes at least
$$\log_B(N/M) = \log(N/M) / \log B$$
I/Os in the worst case.
- **Hint:** Consider the size of the set of blocks that can be accessed in at most t=1,2,... I/Os.



## Aside: Sorting using B<sup>+</sup>-trees

- In internal memory, sorting can be done in  $O(N \log N)$  time by inserting the  $N$  keys into a balanced search tree.
- The number of I/Os for sorting by inserting into a B-tree is  $O(N \log_B N)$ .
- This is more than a factor  $B$  slower (!) than multiway mergesort (Feb 28 lecture).
- Moral: What works well on internal memory may fail miserably on disk.

## From your toolbox: Search trees

- You may have seen: AVL-trees, red-black trees, or (2,4)-trees.
- All have maximum degree 2 or 4, and depth  $O(\log N)$ .
- How do they compare to  $B^+$ -trees on external memory?
- Rough analysis:  $\log(N) = \log_B(N)\log(B)$
- Factor around 5-10 depending mainly on the outdegree of the  $B^+$ -tree.

# Buffering in B-trees

- Relatively new technique to speed up updates in external search trees.
  - Implicit in [Arge 1996]
  - Explicit in [Brodal and Fagerberg 2003]
  - Newer study by [Graefe 2007]
- We will consider an implementation that is based on constant degree search trees.
- Main idea: Use buffers to do many things in one I/O. (Board.)

# Buffering summary

- Trees with outdegree  $B$  and  $O(1)$  give:

<i>Index</i>	<i>Search I/O</i>	<i>Update I/O</i>
B+-tree	$O(\log_B N)$	$O(\log_B N)$
Buffered B-tree	$O(\log N)$	$O(\log(N)/B)$

- In general, the outdegree is a parameter that can be chosen to trade off search time and update time. (See [BF03] for such a trade-off.)

## More on B-trees

- Claim in textbook: “Rebalancing on higher levels of a B-tree occurs very rarely.”
- This is true (in particular at the top levels), but a little hard to see.
- Easier seen for *weight-balanced* B-trees. Details in [Pagh03].
- Just one of many B-tree variants...

## Yet more on B-trees

- In practice, variable length (possibly long) keys must be handled.
- Later in course: The *String B-tree*, an elegant solution that is efficient even for long strings.
- Space-saving trick: Key compression. See RG for details.
- Building a B-tree: Repeated insertion is much slower than “bulk-loading” the sorted list of keys. (Feb. 28)

## Two types of indexes

- In a primary index, records are stored in an order determined by the search key.
- A relation can have at most one primary index. (Often on the primary key.)
- A secondary index cannot take advantage of any specific order. Must contain all keys and corresponding references to tuples (“dense”).
- Sometimes, a secondary index provides a faster access path than a primary index! (More about that next week.)

# Summary

- We saw the basics of storing and indexing data on external memory.
- Many trade-offs (soft or hard):
  - Space usage vs update efficiency.
  - Update vs access efficiency.
  - Optimizing for one kind of access may slow down another type of access.
- We have started thinking and analyzing in terms of I/Os.

# Exercise from hand-out

- *B<sup>+</sup>-trees*
- Questions a), b), and c).