6.830/6.814 — Notes* for Lecture 6: Access Methods

Carlo A. Curino

September 29, 2010

1 Announcements

- Problem Set 2 is out today... due in two weeks... be aware that is going to overlap to LAB 2.
- Projects ideas and rules are posted online.

2 Readings

For this class the suggested readings are:

- In Database Management Systems, read:
  - Pages 273-289. If you are using another book, this is the introduction to the Section on Storage and Indexing which discusses different access methods and their relative performance.
  - Pages 344-358. This is an in-depth discussion of the B+Tree data structure and its implementation. Most database books, as well as any algorithms text (such as CLR or Knuth) will provide an equivalent discussion of B+Trees.
  - (was not assigned, but is an important reading) Pages 370-378 on Static Hashing and Extensible Hashing
- "The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles." Beckmann et al, in The Red Book.

*These notes are only meant to be a guide of the topics we touch in class. Future notes are likely to be more terse and schematic, and you are required to read/study the papers and book chapters we mention in class, do homeworks and Labs, etc.. etc..

3 Recap

We presented a broad overview of the DBMS internals.
We point out how important coming up with a good plan is.
I claimed disk I/O can be very important.

Remember I mentioned this last time:

CPU cost (# of instructions)	1Ghz == 1 billions instrs/sec	1 nsec / instr
RAM access	50ns	
I/Ocost(#ofpagesread,#ofseeks)	100MB/sec	10nsec/byte
(RandomI/O=pageread+seek)	10msec/seek	100seeks/sec

1 seek = 10M instructions!!!

Moreover there is a big difference from sequential and random IO. Example:

- Read 10KB from a random location on disk: 10ms + 100μs = 10.1ms;
- Read 10KB+10KB from a random location on disk (two blocks are next to each other): 10ms + 200μs = 10.2ms;
- Read 10KB+10KB from two random locations on disk: 10ms + 100μs + 10ms +100μs= 20.2ms;

WOW! So saving disk I/O, and in particular random ones is VERY important!! DB are usually well designed to: 1) avoid excessive disk I/O and try to be sequential, and 2) have a LOT of drives available (TPC-H competing machines: 144 cores AND 1296 disks!!)

Today we study how to do a good job at reducing Disk I/O by organize stuff intelligently on disk, next lecture we study what we should keep in RAM.

4 Access Methods

Today we get into more details on *Access Methods*, i.e., on the portion of the DBMS in charge of managing the data on disk. We will show a bunch of organization of data, and their performance in supporting typical accesses we need to support queries. Next lecture we will study the Buffer Manager, which tries to reduce the access to disk.

What are the functionalities we need to support:

- scan
- search (equality)

- search (range)
- insert
- delete

Various access methods:

- heap file: unordered, typically implemented as a linked list of pages
- sorted file: ordered records, expensive to maintain
- index file: data + extra structures around to quickly access data
  - might contain data (primary index)
  - or point at the data, often stored in a heapfile or other index (secondary index)
  - if the data are sorted in the same order of the field is index, we say is a *clustered* index (we will see this is good for scans since disk accesses are sequential)

Type of indexes:

- hash
- B+trees
- R*trees

4.1 Data organization within file

file organization:

- pages
- records (record ids: page_id, slot_id)

page layout:

- fixed length records page of slots,
- free bit map "slotted page" structure for var length records or slot directory (slot offset, len)

What about big records? Hard to place, and might overflow on another page.

tuple layout (similar story):

- fixed length (structure know by the system catalog, can predict where fields start)

- variable length, field slots, two options: delimiters or directory with pointers/offsets

What happen when the field size changes? Need to move stuff... so if we have a mix of fixed/variable, the fixed fields are best to be stored first. Null values? Good trick is using the pointers/offset.. if two have same value, it means the field in between is null... this makes storing nulls 0 space overhead.

5 Cost model (enhanced one, from the book)

- Heap Files: Equality selection on key; exactly one match.
- Sorted Files: Files compacted after deletions.
- Indexes:
  - Alt (2), (3): data entry size = 10% size of record
  - Hash: No overflow buckets. 80%pageoccupancy→Filesize=1.25datasize
  - Tree: 67% occupancy (this is typical). → File size = 1.5 data size

We use:

- B: number of data pages
- R: number of record per page
- D: average time to read/write from disk
- C: average time to process a record (e.g., equality check)

	Scan	Equality	Range	Insert	Delete
Heap	BD	0.5BD	BD	2D	Search + D
Sorted	BD	$D\log_2 B$	$D\log_2 B$ + # matches	Search + BD	Search + BD
Clustered	1.5BD	$D\log_F 1.5B$	$D\log_F 1.5B$ + # matches	Search + D	Search + D
Unclustered tree index	BD (R+0.15)	D (1 + $\log_F 0.15B$)	$D\log_F 0.15B$ + # matches	D (3 + $\log_F 0.15B$)	Search + 2D
Unclustered hash index	BD (R+0.125)	2D	BD	4D	Serach + 2D

6 Extensible hashing

Good read on the book: pages... 370-378

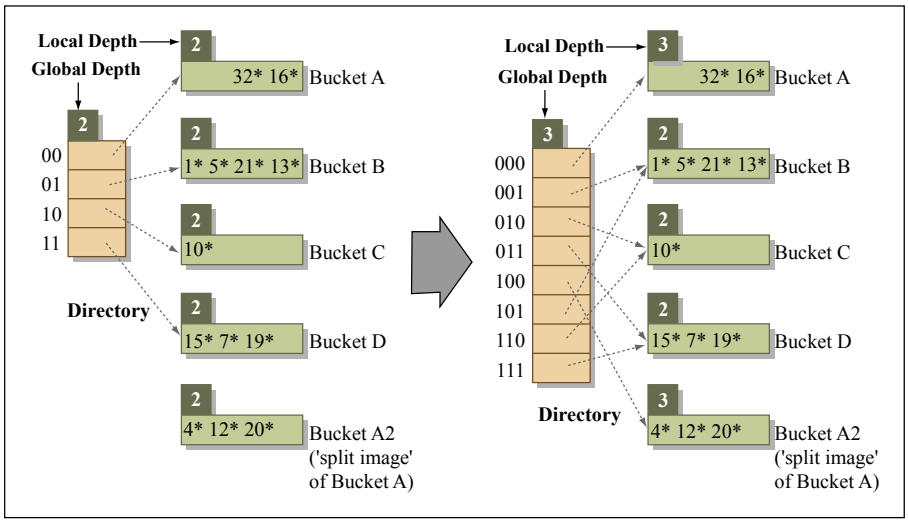

Image by MIT OpenCourseWare.

7 B+trees

Hierarchical indices are the most common type used—e.g., B+-Trees indices typically point from key values to records in the heap file. Shall we always have indexes? What are the pros and cons? (keep the index up-to-date, extra space required)

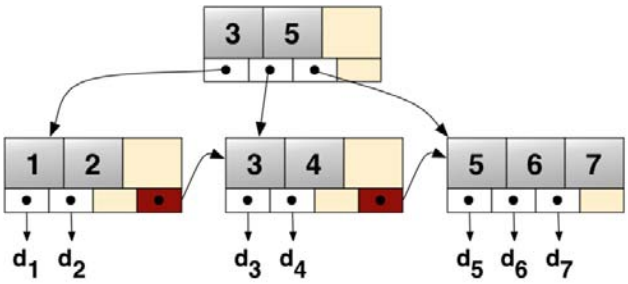

Figure 1: B+Tree graphical representation.

Courtesy of Grundprinzip on Wikipedia.

Special case, is a *"clustered"* index, i.e., when the order of the tuples on disk correspond to the order in which they are stored in the index. What is it good for? (range selections, scans).

6.830 / 6.814 Database Systems

Fall 2010