Database Technology

Topic 8: Data Structures for Databases

Olaf Hartig

olaf.hartig@liu.se







Storage Hierarchy



Traditional Storage Hierarchy





Magnetic Disk





Properties of Using Magnetic Disks

- Formatting divides the hard-coded sector into equal-sized blocks
 - Block is the unit of data transfer between disk and main memory
 - Typical block sizes: 512 8192 bytes
- Read/write from/to disk is a major bottleneck!

R/W time = seek time + rotational delay + block transfer (search track) (search block) time

12–60 ms

- CPU instruction: ca. 1 ns (10⁻⁹ secs)
- Main memory access: ca. 10 ns (10^{-8} secs)
- Disk access: ca. 1 ms (1M ns, 10^{-3} secs)



Files and Records



Terminology

- Data stored in files
- File is a sequence of records
- Records are allocated to file blocks
- Record is a set of field values
- For instance,
 - File = relation
 - Record = row
 - Field = attribute value





Blocking Factor

- Blocking factor (*bfr*) is the number of records per block
- Assume
 - -r is the number of records in a file,
 - -R is the size of a record, and
 - -B is the block size in bytes,

then: $bfr = \left|\frac{B}{R}\right|$



- Blocks needed to store the file: $b = \left[\frac{r}{bfr}\right]$
- Space wasted per block = B bfr * R



B



... avoid wasting space





Allocating File Blocks on Disk

• **Contiguous allocation:** file blocks allocated consecutively (one after another)

- Fast sequential access, but expanding is difficult

- Linked allocation: each file block contains a pointer to the next one
 - Expanding the file is easy, but reading is slower
- Linked clusters allocation: hybrid of the two above
 i.e., linked clusters of consecutive blocks
- Indexed allocation: index blocks contain pointers to the actual file blocks



File Organization

(Organizing Records in Files)



Heap Files

- Records are added to the end of the file
- Adding a record is cheap
- Retrieving, removing, and updating a record is expensive because it implies *linear search*
 - Average case: $\left[\frac{b}{2}\right]$ block accesses
 - Worst case: *b* block accesses

(recall, *b* is the number of blocks of the file)

- Record removal also implies waste of space
 - Periodic reorganization



Sorted Files

- Records ordered according to some field
- Ordered record retrieval is cheap (i.e., on the ordering field, otherwise expensive)
 - All the records: access the blocks sequentially
 - Next record: probably in the same block
 - Random record: *binary search*; hence, $\lfloor \log_2 b \rfloor$ block accesses in the worst case
- Adding a record is expensive, but removing is less expensive (deletion markers and periodic reorganization)



Binary Search





Internal Hashing

- Choose a field of the records to be the *hash field*
- Applying hash function h to the value x of the hash field returns the position of the record in the file
 e.g., h(x) = x mod r

(recall, *r* is the number of records in the file)

- Collision: different field values hash to same position
- Solutions to deal with collisions (collision resolution):
 - Check subsequent positions until one is empty
 - Use a second hash function
 - Put the record in an overflow area and link it



External Hashing

- Hashing for disk files
- Applying hash function to the value of the hash field returns a bucket number (instead of a position)
 - Bucket: one or several contiguous disk blocks
 - Table converts bucket number into address of block
- Collisions are typically resolved via overflow area
- Cheapest random retrieval (when searching for equality)
- Ordered record retrieval is expensive



Indexes

(Secondary Access Methods)



Overview

- Seen so far: file organization
 - Analogous to organization of books into chapters, sections, etc.
 - Determines primary method to access data in a file
 - e.g., sequential search, binary search
- Now: index structures
 - Allow for secondary access methods
 - Analogous to the index of a book
 - Goal: speed up access under specific conditions
 - Outline:
 - 1) Single-level ordered indexes (primary,
 - secondary, and clustering indexes)
 - 2) Multilevel indexes
 - 3) Dynamic multilevel indexes (B+-trees)



Single-Level Ordered Indexes





- Primary index: an additional sorted file whose records contain two fields:
 - V one of the values of F
 - P pointer to a disk block of the data file
- One index record (*V*,*P*) per data block such that the first data record in the data block pointed to by *P* has *V* as the value of the ordering key *F*





- Why is it faster to access a random record via a binary search in the index than in the data file?
 - Number of index records << number of data records
 - Index records smaller than data records (i.e., higher blocking factor for the index file than for the data file)
- What is the cost of maintaining a primary index? (if the order of the data records changes)



Clustering Index

- Assumptions:
 - Data file is sorted
 - Ordering field F is
 not a key (hence, we cannot assume distinct values)



- Clustering index: additional *sorted file* whose records contain two fields:
 - V one of the values of F
 - P pointer to a disk block of the data file
- One index record (V,P) for each distinct value V of the ordering field F such that P points to the first data block in which V appears



Clustering Index

- Efficiency gain?
- Maintenance cost?





Secondary Indexes on Key Field

- Index on a *non*-ordering *key* field *F*
 - Data file may be sorted or not
- Secondary index: additional *sorted* file whose records contain two fields:
 - V one of the values of F
 - P pointer to a disk block of the data file
- One index record per data record



ID#

SSN

4945864

Dept. Salary

2000

12



Data File

Secondary Indexes on Non-Key

Index on a non-ordering non-key field





Summary of Single-Level Indexes

	Index f orderin	Index field used for ordering the data file		Index field <i>not</i> used for ordering the data file		
Index field is key	Primary ir	Primary index		Secondary index (key)		
Index field is not k	ey Clusterino	Clustering index		Secondary index (non-key)		
	Type of index	Numbe index en	r of tries			
	Primary	Number of blocks in data file				
	Clustering	Number of dist index field valu	tinct Jes			
	Secondary (key)	Number of rec in data file	ord			
	Secondary (non-key)	Number of record or number of d index field value	ords listinct Jes			



Multilevel Indexes



Multilevel Indexes

- Index on index (first level, second level, etc.)
- Works for primary, clustering, and secondary indexes as long as the first-level index has a distinct index value for every entry

20

Second Level

First Level

- How many levels?
 - Until the last level fits into a single disk block



Number of index levels + 1



Multilevel Indexes (cont'd)

- When using a (static) multilevel index, record insertion, deletion, and update may be expensive because all the index levels are *sorted* files
- Solutions:
 - Overflow area + periodic reorganization
 - Dynamic multilevel indexes that leave some space in index blocks for new entries (e.g., B-trees and B+-trees)



Dynamic Multilevel Indexes

(B-Trees and B+-Trees)



Search Trees

- Used to guide the search for a record
 Generalization of binary search
- Nodes of a search tree of order *p* look like:



- $-q \leq p$
- Every K_i is a key value
- Every P_i is a tree pointer to a subtree (or a null pointer)
- Within each node: $K_1 < K_2 < \ldots < K_{q-1}$
- For every value X in the subtree: $K_{i-1} < X < K_i$





- B-tree is a variant of a *balanced* search tree
 Balanced: all leaf nodes are at the same level (Why is this good?)
- Additional constraints:
 - In addition to a tree pointer P_i, each key value K_i is associated with a data pointer Pr_i to the record with value K_i
 - Each internal node must have at least $\left|\frac{p}{2}\right|$ tree pointers (i.e., is at least half full)



B+-Trees

- Variation of B-trees, most commonly used
- In contrast to a B-tree, in a B+tree the leaf nodes are different from the internal nodes; that is:
 - Internal nodes have key values and tree pointers only (no data pointers)
 - Leaves have key values and data pointers
 - Usually, each leaf node additionally has a pointer to the next leaf to allow for ordered access (much like a linked list)
- Every key value is present in one of the leaves
- Of course, B+-trees are balanced



Internal Nodes of a B+-Tree



- $q \le p$ (where *p* is the order of the B+-tree)
- Every K_i is a key value, every P_i is a tree pointer
- Within each node: $K_1 < K_2 < \ldots < K_{q-1}$
- For every value X in the subtree: $K_{i-1} < X \leq K_i$
- Every internal node (except the root) has at least tree pointers



<u>p</u> 2

Leaf Nodes of a B+-Tree

<i>K</i> ₁	Pr ₁		Ki	Pr _i		K_q	Pr _q	P _{next}
-----------------------	-----------------	--	----	-----------------	--	-------	-----------------	-------------------

- $q \le p$ (where *p* is the order for leaf nodes of the B+-tree)
- Every *K_i* is a key value
- Every Pr_i is a data pointer to the record with key value K_i
- *P* is a pointer to the next leaf node
- Within each node: $K_1 < K_2 < \ldots < K_q$
- Every leaf node has at least $\left[\frac{p}{2}\right]$ key values



Retrieval of Records in a B+-Tree

- Very fast retrieval of a random record, at worst: $\left| \log_{\left[\frac{p}{2}\right]} N \right|^{+1}$
 - p is the order of the internal nodes
 - *N* is the number of leaf nodes







Insert: 8





Insert: 5





Overflow - create a new level

Insert: 1





Insert: 7





Insert: 3





Overflow - Split Propagates a new level

Insert: 12





Insert: 9





Overflow - Split, propagates







Resulting B+-tree







Summary

- Storage hierarchy
 - Accessing disk is major bottleneck
- Organizing records in files
 - Heap files, sorted files, hash files
- Indexes
 - Additional sorted files that provide efficient secondary access methods
- Primary, secondary, and clustering indexes
- Multilevel indexes
 - Retrieval requires reading fewer blocks
- Dynamic multilevel indexes
 - Leave some space in index blocks for new entries
 - B-tree and B+-tree

