



Lecture 25 of 42

XML Structure and Document Schemas Discussion: Indexing

Monday, 23 October 2006

William H. Hsu
Department of Computing and Information Sciences, KSU

KSOL course page: <http://snipurl.com/va60>

Course web site: <http://www.kddresearch.org/Courses/Fall-2006/CIS560>

Instructor home page: <http://www.cis.ksu.edu/~bhsu>

Reading for Next Class:

First half of Chapter 12, Silberschatz *et al.*, 5th edition



Web Services

- The Simple Object Access Protocol (SOAP) standard:
 - * Invocation of procedures across applications with distinct databases
 - * XML used to represent procedure input and output
- A *Web service* is a site providing a collection of SOAP procedures
 - * Described using the Web Services Description Language (WSDL)
 - * Directories of Web services are described using the Universal Description, Discovery, and Integration (UDDI) standard





Chapter 12: Indexing and Hashing

- Basic Concepts
- Ordered Indices
- B+-Tree Index Files
- B-Tree Index Files
- Static Hashing
- Dynamic Hashing
- Comparison of Ordered Indexing and Hashing
- Index Definition in SQL
- Multiple-Key Access



Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
 - * E.g., author catalog in library
- **Search Key** - attribute to set of attributes used to look up records in a file.
- An **index file** consists of records (called **index entries**) of the form

search-key	pointer
------------	---------
- Index files are typically much smaller than the original file
- Two basic kinds of indices:
 - * **Ordered indices:** search keys are stored in sorted order
 - * **Hash indices:** search keys are distributed uniformly across "buckets" using a "hash function".





Index Evaluation Metrics

- Access types supported efficiently. E.g.,
 - * records with a specified value in the attribute
 - * or records with an attribute value falling in a specified range of values.
- Access time
- Insertion time
- Deletion time
- Space overhead



Ordered Indices

Indexing techniques evaluated on basis of:

- In an **ordered index**, index entries are stored sorted on the search key value. E.g., author catalog in library.
- **Primary index**: in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
 - * Also called **clustering index**
 - * The search key of a primary index is usually but not necessarily the primary key.
- **Secondary index**: an index whose search key specifies an order different from the sequential order of the file. Also called non-clustering index.
- Index-sequential file: ordered sequential file with a primary index.





Dense Index Files

- Dense index — Index record appears for every search-key value in the file.

Brighton		A-217	Brighton	750	
Downtown		A-101	Downtown	500	
Mianus		A-110	Downtown	600	
Perryridge		A-215	Mianus	700	
Redwood		A-102	Perryridge	400	
Round Hill		A-201	Perryridge	900	
		A-218	Perryridge	700	
		A-222	Redwood	700	
		A-305	Round Hill	350	



Sparse Index Files

- Sparse Index: contains index records for only some search-key values.
 - * Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value K we:
 - * Find index record with largest search-key value $< K$
 - * Search file sequentially starting at the record to which the index record points
- Less space and less maintenance overhead for insertions and deletions.
- Generally slower than dense index for locating records.
- Good tradeoff: sparse index with an index entry for every block in file, corresponding to least search-key value in the block.



Example of Sparse Index Files

Brighton	→	A-217	Brighton	750	→
Mianus	→	A-101	Downtown	500	→
Redwood	→	A-110	Downtown	600	→
		A-215	Mianus	700	→
		A-102	Perryridge	400	→
		A-201	Perryridge	900	→
		A-218	Perryridge	700	→
		A-222	Redwood	700	→
		A-305	Round Hill	350	→

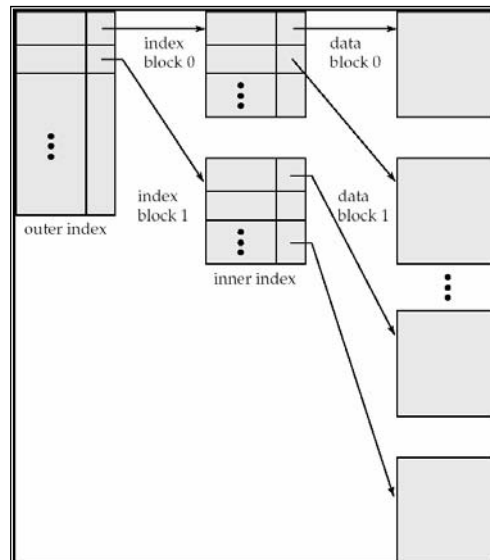


Multilevel Index

- If primary index does not fit in memory, access becomes expensive.
- To reduce number of disk accesses to index records, treat primary index kept on disk as a sequential file and construct a sparse index on it.
 - * outer index – a sparse index of primary index
 - * inner index – the primary index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion from the file.



Multilevel Index (Cont.)



Index Update: Deletion

- If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also.
- Single-level index deletion:
 - * Dense indices – deletion of search-key is similar to file record deletion.
 - * Sparse indices –
 - ⇒ if an entry for the search key exists in the index, it is deleted by replacing the entry in the index with the next search-key value in the file (in search-key order).
 - ⇒ If the next search-key value already has an index entry, the entry is deleted instead of being replaced.





Index Update: Insertion

- Single-level index insertion:
 - * Perform a lookup using the search-key value appearing in the record to be inserted.
 - * Dense indices – if the search-key value does not appear in the index, insert it.
 - * Sparse indices – if index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created.
 - ⇒ If a new block is created, the first search-key value appearing in the new block is inserted into the index.
- Multilevel insertion (as well as deletion) algorithms are simple extensions of the single-level algorithms



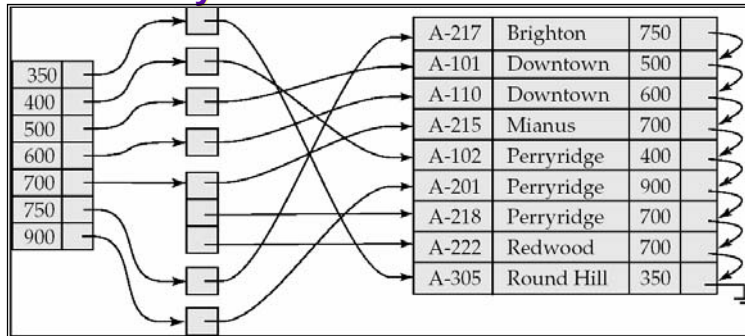
Secondary Indices

- Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index) satisfy some condition.
 - * Example 1: In the *account* relation stored sequentially by account number, we may want to find all accounts in a particular branch
 - * Example 2: as above, but where we want to find all accounts with a specified balance or range of balances
- We can have a secondary index with an index record for each search-key value
 - * index record points to a bucket that contains pointers to all the actual records with that particular search-key value.





Secondary Index on *balance* field of account



Primary and Secondary Indices

- Secondary indices have to be dense.
- Indices offer substantial benefits when searching for records.
- When a file is modified, every index on the file must be updated, Updating indices imposes overhead on database modification.
- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive
 - * each record access may fetch a new block from disk



B+-Tree Index Files

B+-tree indices are an alternative to indexed-sequential files.

- Disadvantage of indexed-sequential files: performance degrades as file grows, since many overflow blocks get created. Periodic reorganization of entire file is required.
- Advantage of B+-tree index files: automatically reorganizes itself with small, local, changes, in the face of insertions and deletions. Reorganization of entire file is not required to maintain performance.
- Disadvantage of B+-trees: extra insertion and deletion overhead, space overhead.
- Advantages of B+-trees outweigh disadvantages, and they are used extensively.



B+-Tree Index Files (Cont.)

A B+-tree is a rooted tree satisfying the following properties:

- All paths from root to leaf are of the same length
- Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children.
- A leaf node has between $\lceil (n-1)/2 \rceil$ and $n-1$ values
- Special cases:
 - * If the root is not a leaf, it has at least 2 children.
 - * If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and $(n-1)$ values.





B+-Tree Node Structure

- Type:

P_1	K_1	P_2	\dots	P_{n-1}	K_{n-1}	P_n
-------	-------	-------	---------	-----------	-----------	-------

 - * K_i are the search-key values
 - * P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered

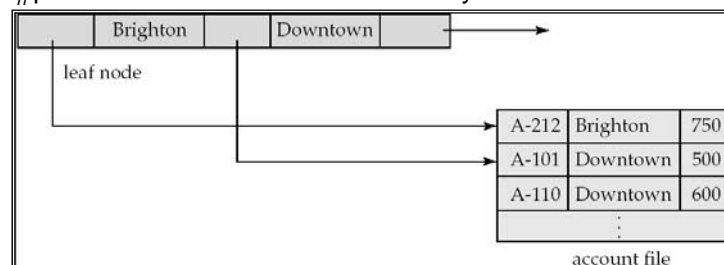
$$K_1 < K_2 < K_3 < \dots < K_{n-1}$$



Leaf Nodes in B+-Trees

Properties of a leaf node:

- For $i = 1, 2, \dots, n-1$, pointer P_i either points to a file record with search-key value K_i , or to a bucket of pointers to file records, each record having search-key value K_i . Only need bucket structure if search-key does not form a primary key.
- If L_i, L_j are leaf nodes and $i < j$, L_i 's search-key values are less than L_j 's search-key values
- P_n points to next leaf node in search-key order



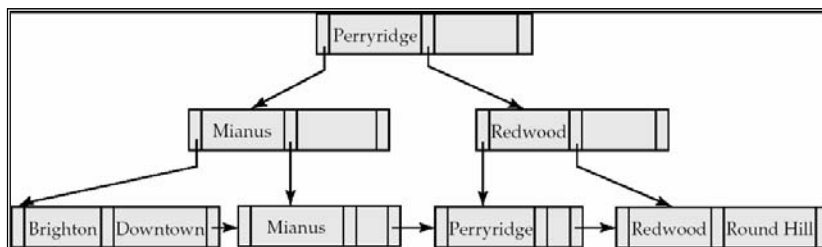


Non-Leaf Nodes in B⁺-Trees

- Non leaf nodes form a multi-level sparse index on the leaf nodes.
For a non-leaf node with m pointers:
 - * All the search-keys in the subtree to which P_1 points are less than K_1
 - * For $2 \leq i \leq m-1$, all the search-keys in the subtree to which P_i points have values greater than or equal to K_{i-1} and less than K_{i-1}



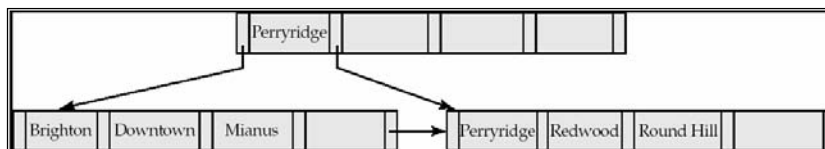
Example of a B⁺-tree



B⁺-tree for *account* file ($n = 3$)



Example of B⁺-tree



B⁺-tree for *account* file ($n = 5$)

- Leaf nodes must have between 2 and 4 values ($\lceil (n-1)/2 \rceil$ and $n-1$, with $n = 5$).
- Non-leaf nodes other than root must have between 3 and 5 children ($\lceil n/2 \rceil$ and n with $n=5$).
- Root must have at least 2 children.



Observations about B⁺-trees

- Since the inter-node connections are done by pointers, “logically” close blocks need not be “physically” close.
- The non-leaf levels of the B⁺-tree form a hierarchy of sparse indices.
- The B⁺-tree contains a relatively small number of levels (logarithmic in the size of the main file), thus searches can be conducted efficiently.
- Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time (as we shall see).





Queries on B+-Trees

- Find all records with a search-key value of k .
 1. Start with the root node
 1. Examine the node for the smallest search-key value $> k$.
 2. If such a value exists, assume it is K_j . Then follow P_j to the child node
 3. Otherwise $k \geq K_{m-1}$, where there are m pointers in the node. Then follow P_m to the child node.
 2. If the node reached by following the pointer above is not a leaf node, repeat step 1 on the node
 3. Else we have reached a leaf node.
 1. If for some i , key $K_i = k$ follow pointer P_i to the desired record or bucket.
 2. Else no record with search-key value k exists.

