Index Structures

Anne Denton

Department of Computer Science
North Dakota State University

Outline

1. Concepts of index structures
   - Properties of indexes
   - Types of indexes
2. Multi-level indexes
   - Objectives for multi-level indexes
   - Definition of B^+ Trees
   - Insertion into a B^+ Tree

Table of Contents

1. Concepts of index structures
   - Properties of indexes
   - Types of indexes
2. Multi-level indexes
   - Objectives for multi-level indexes
   - Definition of B^+ Trees
   - Insertion into a B^+ Tree

- Are considered “access methods"
- Allow fast access to records based on “indexing field"
- Any field can be used as indexing field
- Not limited to primary key, but primary key is typically indexed
    - Important for constraint checking
    - Used for some join queries (especially when few records are returned)
- Multiple indexes possible for one table (file)

Single-level vs. multi-level indexes

- Single level index
  - Not common in practice
  - Similar to index of a book
  - Index is an ordered file that maps indexing field value to address
  - First field same type as indexing field
  - Second field: block or record pointer
- Multi-level index
  - Typical implementation
  - Different tree structures conceivable, but B^+ trees used in overwhelming majority of implementations
  - Binary trees not normally used due to large number of levels and inefficient use of data read from disk

Fraction of indexed records

- Dense index
  - One index entry corresponding to each record in the file
  - Necessary when the remaining records cannot be inferred
  - Example uses
    - Heap files
    - Attributes other than the search field in sorted or hash files
- Sparse index
  - Only some records in the file have corresponding index entries
  - Used when the position of some records can be inferred from others
  - Example uses
    - Sorted files
    - Attribute used for searching is not a key

Table of Contents

1 Concepts of index structures
- Properties of indexes
- Types of indexes

2 Multi-level indexes
- Objectives for multi-level indexes
- Definition of B^+ Trees
- Insertion into a B^+ Tree

Primary Index

- Index used for an physically ordered file
- File ordered according to the value of a key
  - Key: Field that uniquely identifies record, i.e., one record per key value
- Indexing field has same data type as ordering field of file
- Can be sparse: Only pointer to first record in block necessary (anchor record)
- Typical use
  - By default the primary key of a relation will be given a primary index
  - A file may also be sorted based on an alternate key
  - An index on an alternate key that is used for sorting would still be a primary index

Question 1

Consider the following sorted data that extends over multiple blocks

3	
5	
7	

12	
14	

How many index entries are needed?

1. Two, one for each block anchor, 3 and 12
2. Five, one for each of the search key values, 3, 5, 7, 12, and 14

Question 2

Consider the following sorted data that extends over multiple blocks

7	
14	
5	

3	
12	

How many index entries are needed?

1. Two, one for each block anchor, 7 and 3
2. Five, one for each of the search key values, 3, 5, 7, 12, and 14

Clustered Index

- Index used for a physically ordered file that is not a key
  - One entry in the index file for each distinct value of the indexing field (not for each record)
  - Sparse index
- Typical uses
  - Attribute that is normally searched for but is not unique
  - Not the default choice but may come out of database tuning

Secondary Index

- Index used for a non-ordering field
  - File could be a heap file
  - File could be sorted according to a different ordering field
- Multiple secondary indexes possible
- Must be dense (each record must be represented)
- Search time is longer than for primary index
- Can be viewed as logical sorting of the file
- Representing field that is not a key field
  - Option 1: One entry for each record
  - Option 2: List of pointers to all records with the given indexing field value
  - Option 3: Index entry represents a block of record pointers or linked list of blocks
- Compare to index in a book

Question 3

How many primary indexes can there be on a file

1. One
2. As many as there are attributes

Question 4

How many clustered indexes can there be on a file

1. One
2. As many as there are attributes

Question 5

How many secondary indexes can there be on a file

1. One
2. As many as there are attributes

Question 6 (Multiple answers may be correct)

Which of the following are good reasons for having a primary index on a primary key

1. Fast checking if key constraint is satisfied upon insertion into the table in question
2. Fast checking if referential integrity constraint is satisfied up upon insertion into a table that defines a foreign key that references the table in question
3. Fast join queries that return only a few records
4. Fast join queries that involve all or most records in the table
5. Fast “ORDER BY" queries by last name

Table of Contents

1. Concepts of index structures
   - Properties of indexes
   - Types of indexes
2. Multi-level indexes
   - Objectives for multi-level indexes
   - Definition of B^+ Trees
   - Insertion into a B^+ Tree

Multilevel Indexes

- Single-level indexes not practical, since binary search not efficient on disk
- Binary trees not practical either
  - Number of levels of the order of $\log_2(N)$ where N is number of levels
  - Each block read from disk only used for very small node
- Most practical solution: Use nodes that fill disk blocks

Question 7 (Multiple answers may be correct)

Which of the following make binary trees unsuitable for indexing records on disk?

1. Since each node only has two child node pointers, the number of levels is large, which may mean many disk accesses
2. Listing elements in sequence requires a tree traversal that involves many disk accesses
3. Binary trees are generally inefficient and have been superseded by B^+ Trees
4. The question is poorly phrased: Binary trees are suitable for indexing records on disk

Overview over Objectives

- Make the most out of any disk access
  - Important because of slow disk access
  - Large block size ($\sim$ 4K)
- Allow fast retrieval in sequence of search key values
  - Important for returning ordered lists
  - Important for inequalities, e.g. "$<$" or "$>$")
- Minimize the number of tree levels
  - Number of disk accesses in retrieval given by number of tree levels that have to be read
  - Large fanout reduces number of tree levels
- Avoid major reorganizations
  - Limit number of block reorganization upon insertion

Optimizing use of each disk access

- Disk access, for HDDs, takes about 6 orders of magnitude more time than for RAM (i.e., a factor of about 1 million)
- The typical disk block size is 4KB
- Make the most out of any disk access
  - Choose node size to match block (page) size
  - The information in the node of a binary tree is in the 10s of bytes, not in the thousands of bytes
  - Information about $\sim$ 100 child node pointers fits in one block

Maximizing fanout

- The number of levels in a tree is approximately $\log_f N$, where f is the fanout, and N is the number of records that are being indexed
- E.g. for $N = 10^6$, the number of levels in a binary tree is greater than 20, but for a fanout of 100, only 3 levels would be needed
- Number of disk accesses in retrieval can be as large as number of levels, i.e., reducing number of levels is critical for fast access
- Store as little information as possible in internal nodes (e.g., no record pointers)

Question 8

A multi-level index could be constructed to have internal nodes that include

- Search key values, child-node pointers, and record pointers for each search key value
- Search key values, child-node pointers, but no record pointers, and instead have all record pointers at the leaf level

The first design, in comparison with the second, is bound to result in a tree with

1. More levels
2. Fewer levels

Avoiding sparsely filled nodes

- Only create nodes by splitting overflowing existing nodes
- Split from the bottom up, so that any split corresponds to an insertion into a higher-level node
- When parent node is full, split parent node
- Upon deletion, merge adjacent nodes that are less than half full
  - Alternative: Rebuild index occasionally

Question 9 (Multiple answers may be correct)

Which of the following insertion strategies may result in nodes that are less than half full for large-fanout trees?

1. Growing the tree by adding leaves, as is done for binary trees
2. Growing the tree splitting nodes from the leaf-level towards the root node

Fast retrieval in sequence of search key values

- Needed for many queries
  - “ORDER BY" statement requires sorted output
  - Selection criteria that involve inequalities, e.g. “$<$" or “$>$"
- Would require tree traversal in a binary tree
- Solution
  - B$^+$ Trees have all record pointers in leaf level, all leaf nodes at same level, and pointers between nodes

Question 10 (Multiple answers may be correct)

Which of the following types of trees allow retrieving data ordered by the search key value without requiring a tree traversal

1. Binary trees
2. B+ Trees

Avoid major reorganizations

- Insert new search key values by locating node into which they belong
- When that node is full, split it
- Do not reorganize any other nodes
- Never leads to nodes that are less than half full, at least for insertions only
- Combining leaves during deletions more complicated

Table of Contents

1 Concepts of index structures
- Properties of indexes
- Types of indexes

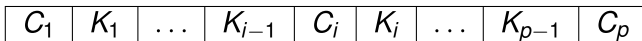
2 Multi-level indexes
- Objectives for multi-level indexes
- Definition of B+ Trees
- Insertion into a B+ Tree

B^+ Trees

- B^+ Trees are overwhelmingly used instead of binary trees when disk access is involved
- Even tables with hundreds of millions of records require no more than 4-5 disk accesses
  - Fanout in the hundreds of child-node pointers
- The number of blocks that require reorganization is never larger than the number of levels in the tree
  - When a leaf node is split, splitting may propagate up in the tree up to root node
  - Neighboring nodes not affected
- They offer a logical sorting of the data
- Note that while binary trees only have one type of node, B^+ Trees distinguish internal and leaf nodes
  - B Trees used the same nodes for both, but were not as efficient

Internal nodes

- Each internal node has the following structure:



- Where the C_i are child-node pointers and K_i are search key values
- Number of child-node pointers, p, determined based on block (page) size and storage requirements for child-node pointers and search key values
- Notice that there are no record pointers in internal nodes
- Requires searching to leaf level for each search key value
  - Overall beneficial because it reduces the number of levels
  - Also enables iterating through leaf level

Question 11

Consider a B^+ Tree in a database with

- Page/block size: 4KB
- Child-node pointer size: 8 bytes
- Search key value size: 8 bytes

What will the fanout be?

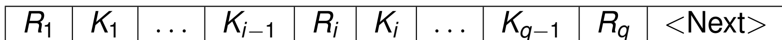
1. 4
2. 16
3. 258
4. 1024

Estimate of fanout of internal nodes

- Maximum fanout is calculated based on page size of the DBMS (which is typically chosen equal to the block size of the operating system)
- As order of magnitude estimate assume
  - Page/block size: 4KB
  - Child-node pointer size: 8 bytes
  - Search key value size: 8 bytes
- Fanout: 256
- This estimate is based on integer search keys
- When the indexed data type is VARCHAR, the fanout can be much smaller

Leaf nodes

- Each left node has the following structure:



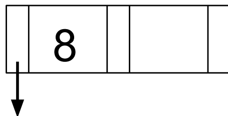
- Number of record pointers, q, may be different from the p of child-node pointers
- <Next> pointer allows iterating through leaf level
- There are formats that integrate entire records into leaf node

Table of Contents

1. Concepts of index structures
   - Properties of indexes
   - Types of indexes
2. Multi-level indexes
   - Objectives for multi-level indexes
   - Definition of B+ Trees
   - Insertion into a B+ Tree

Empty B+ Tree

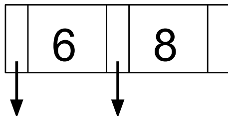
- We use a small example for demo purposes
- Initial empty node is a leaf node
- Assume that value 8 is inserted



- Arrow symbolizes record pointer

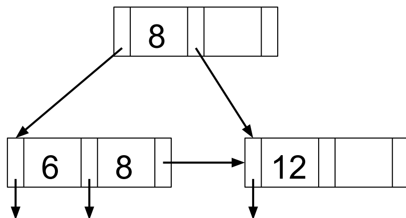
Initial insertions

- Initial insertions only result in internal reordering of node
- Internal reordering is fast since it is done in memory
- Assume that value 6 is inserted



Node splitting

- In this small example the next insertion triggers node splitting
- Requires adding an internal node as root node
- Search key value in root node has to be replicated in leaf level
  - Whether it is replicated in left or right node depends on definition
  - Whatever the definition is it has to be applied consistently
- Assume that value 12 is inserted



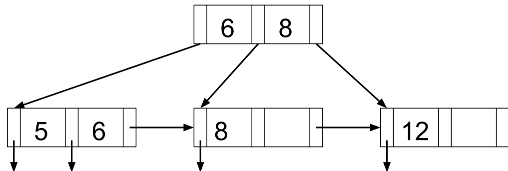
Question 12 (Multiple answers may be correct)

Assume that the next value to be added is 5. Which of the following statements are correct?

1. No node splitting is necessary
2. A leaf node will have to be split
3. The root node will have to be split

Further node splitting

- Insertions start by locating appropriate node
- Assume that the next inserted value is 5
- 5 is smaller or equal to 8 and is hence expected in left node
- In this small example, the node is full and has to be split
- The internal node can still accommodate an additional value and does not have to be split



- Notice <Next> pointer between leaf nodes

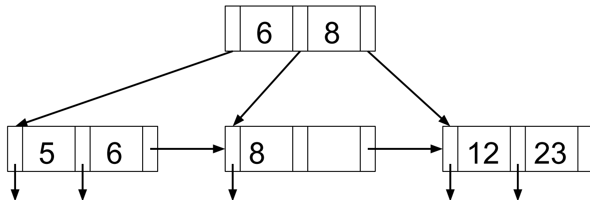
Question 13 (Multiple answers may be correct)

Assume that the next value to be added is 23. Which of the following statements are correct?

1. No node splitting is necessary
2. A leaf node will have to be split
3. The root node will have to be split

Insertions without node splitting

- In an actual B^+ Tree, most insertions can be done without node splitting
- Assume that the next inserted value is 23
- 23 is greater than 8 and is inserted in the rightmost node



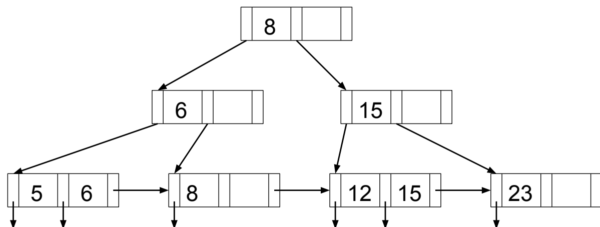
Question 14 (Multiple answers may be correct)

Assume that the next value to be added is 15. Which of the following statements are correct?

1. No node splitting is necessary
2. A leaf node will have to be split
3. The root node will have to be split

Splitting of root node

- Assume that the next inserted value is 15
- 15 is greater than 8 and belongs into the rightmost node
- Node has to be split and so does the root node
- Notice that 8 does not have to be replicated in an internal node



Question 15

Which of the following statements is correct about B^+ Trees?

1. Search key values are never replicated
2. The same search key value may occur in a leaf node and one internal node
3. The same search key value may occur in a leaf node and multiple internal nodes

Visualization

- The following visualization may help, but uses the convention of taking the right child node pointer upon equality `https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html`

B+ Tree limitations

- B+s are use almost exclusively for indexing a single attribute or a multiple attributes that form a hierarchy
- To index multiple attributes use composite indexes / keys
- Order matters for storage, since sorting starts with first attribute
- For true multi-dimensional indexing check spatial indexes
  - No hierarchy between latitude and longitude
  - R trees, quad trees and others address this problem
  - Some (in particular R-trees) borrow ideas from B+ Trees