COS 597A:
Principles of
Database and Information Systems

Indexing files

1

Last time

- File = a collection of (blocks of) records
- File organizations: two issues
  - how records assigned blocks
  - how blocks put on disk
  - Heap: linked list (or directory) of blocks
    - records anywhere on any block - no order
    - blocks anywhere on disk
  - Sorted sequential blocks
    - Records sequential in each block by designated sort attribute
    - can binary search: get i^{th} block in one disk read
  - Hashing:
    - Designated hash attribute hashing records to buckets
    - Bucket => (primary) block for hash function value
      - blocks can be anywhere if hash table gives location

2

Focus on key elements of cost

Improvements only for attribute of sort or hash
Improve access using other attributes? => index

Avg. time	Heap	Sorted	Hashed
Search = (unique)	.5BD	$D\log_2 B$	D
Search range	BD	**$D(\log_2 B$ + # extra matching blocks)**	1.25 BD
Insert	2D	**Search + D + BD**	2D
Delete (have record location)	2D	2D+BD	2D

B data blocks in file **D** avg time to R/W block **R** records per block 3

Index

- Auxillary information on location of a record or block to facilitate retrieval
- Search key: attribute (i.e. field, column) used as look-up value for index
  - not confuse with {primary, candidate, super} key
  - alternate term "index field"
    - "index key" if attribute is a candidate key
  - Could actually be combination of attributes
    - e.g. LastName, FirstName
- Basic index is a file containing mappings:

  Seach key value ⟶ pointer(s) to block(s) containing records with given search key value

4

Index Types

1. Index works *with* file organization
   - Index and file work off same attribute
   - Example: Hashing file organization
     - Use index to get pointer to block serving as primary bucket for given hash value
   - called clustered index
   - some refer to as primary index
     - not necessarily on primary key of relation

5

Index Types cont.

2. Index works independent of file organization
   - File not organized on search key of index
   - Index must provide

     search key value ⟶ list of pointers to *all* file blocks that contain records with that value
   - Example hash index:
     - bucket contains list of block pointers
     - blocks may be scattered throughout the file
     - overflow if too many pointers for one bucket
   - called nonclustering index
   - come refer to as secondary index

6

A Sorted Index

- Consider sorted, sequential file but without consecutive blocks stored adjacently on disk
  - Each block sorted
  - Each block linked to next block in sorted order
  - ***Cannot*** binary search
- Index:

sorted attribute value	pointer to first block containing

Sorted order

- One entry per attribute value in data file => dense index
- Can binary search index entries if can keep in memory or in sequential disk blocks

Indexing sorted files - notes

- If index on sorted file using same attribute, index need not be dense (so sparse)
- Insert/delete for sorted file with sorted index costs to maintain sorted order in both
- Index may be sorted on different attributes(s) than file, but clustered as file is
  - Example: file sorted on (last_name, first_name) index sorted on last_name

Alternative sparse index for sorted file

again:
index search key same as sort attribute for file

file block number	block location	first value of search key on block

Sorted order

One entry *per file block*
Again, binary search if keep in memory or sequentially on disk

Compare costs:

dense sorted index **versus**
sparse sorted index with one value per data file block

- Use our crude estimates with
  - **B** data blocks in file
  - **D** avg time to R/W block
  - **R** records per block

- Suppose index record 1/10 size of data record
- Suppose search key (= sort attribute) is candidate key
- Cost search for unique value using dense index?

- Cost search for unique value using sparse index?

Cost example dense sorted index

- Use our crude estimates with
  - **B** data blocks in file
  - **D** avg time to R/W block
  - **R** records per block

- Suppose index record 1/10 size of data record
- Suppose search key (= sort attribute) is candidate key

- Cost search for unique value using dense index:
  - B/10 blocks in index file (file block size is fixed for all files)
  - Binary search cost = $D\log_2(B/10)$

  Total cost = $D\log_2(B/10) + D$ includes data block access

Cost example sparse sorted index

- Use our crude estimates with
  - **B** data blocks in file
  - **D** avg time to R/W block
  - **R** records per block

- Suppose index record 1/10 size of data record
- Suppose search key (= sort attribute) is candidate key

- Cost search for unique value using sparse index:
  - B blocks in data file => B entries in index file
  - 10R index records per file block => B/(10R) index blocks
  - Binary search cost = $D\log_2(B/(10R))$

  Total cost = $D\log_2(B/(10R)) + D$ includes data block access

Compare costs:

- Use our crude estimates with

  B data blocks in file   **D** avg time to R/W block

  R records per block

- Suppose index record 1/10 size of data record
- Suppose search key (= sort attribute) is candidate key

- Cost search for unique value using dense index?

  $D\log_2(B/10) + D$

- Cost search for unique value using sparse index?

  $D\log_2(B/(10R)) + D$

13

Compare costs: insertion

- Use our crude estimates with

  B data blocks in file   **D** avg time to R/W block

  R records per block

- Suppose index record 1/10 size of data record
- Suppose search key (= sort attribute) is candidate key

- Cost to insert = cost to insert in data file
  + cost to insert in index file

  = Search cost

  + D + D*B   write data file block and move records

  + D   write index entry

  + { D*B/10   move records for dense index

    D*B/(10R)   move records for sparse index }

14

BUT WAIT Compare costs: insertion

- Use our crude estimates with

  B data blocks in file   **D** avg time to R/W block

  R records per block

- Suppose index record 1/10 size of data record
- Suppose search key (= sort attribute) is candidate key
- Recall data file blocks not nec. stored consecutively on disk
  - so can use overflow blocks

- Cost to insert = cost to insert in data file
  + cost to insert in index file

  = Search cost

  + D + ~4D   write data file block and move ~1/2 records of block if overflow

  + D   write index entry

  + { D*B/10   move records for dense index

    D*B/(10R)   move records for sparse index }

15

Index independent of file organization

But look again,
if search key is a *candidate key*,
this index works for *any* file organization :

search key	pointer to *unique* block containing
↑ Sorted order ↓	

One entry per search key value - dense

Can binary search index as before if keep in memory or sequentially on disk

16

Sorted index for general case

- One value of search key found in many records
- Need list of pointers to blocks containing these records
- Dense index still works
- Most common arrangement:
  - indirection

Seach key	pointer to block containing list
↑ One entry per attribute value. Sorted order ↓	

17

Addressing costs

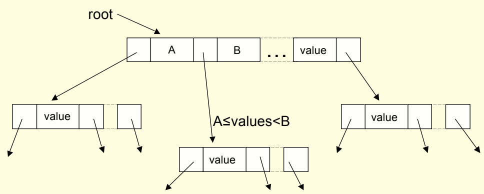
- Large sorted index costly in space and in time to insert/delete
  - When sorted index clustered, can use sparse index to avoid space
  - For general case, *must* have dense index
- Ideal: index to fit on one file block.
  - Keep in main memory
- Rarely achieve, so next best:
  - Index need not be stored sequentially on disk
  - Access cost is no worse than $O(\log_2 B)$

  => **Search Tree!**

18

Tree index

•Each node of tree fits in one block
•Each node of tree contains search key values
and pointers to subtrees for ranges of values
•A leaf is
-For clustered index: a block of data file
-For general index: a block of pointers to records with given index values



Static Trees

- Build for file of records as balanced tree
- Not gracefully accommodate insert/delete
- ISAM: Indexed Sequential Access Method

- We focus on dynamic search trees

Dynamic Trees

- Tree will change to keep balance as file grows/shrinks
- Tree height: longest path root to leaf
- N data entries
  Data entry is block of data file if clustered index
  Data entry is block of (value, record pointer) pairs otherwise
- Want tree height proportional to logN always

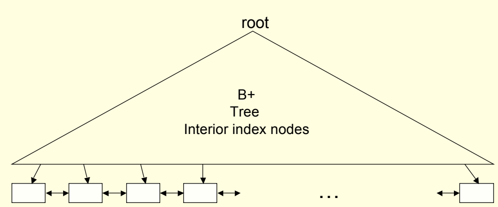
B+ Trees

- Most widely used dynamic tree as index
- Most widely used index

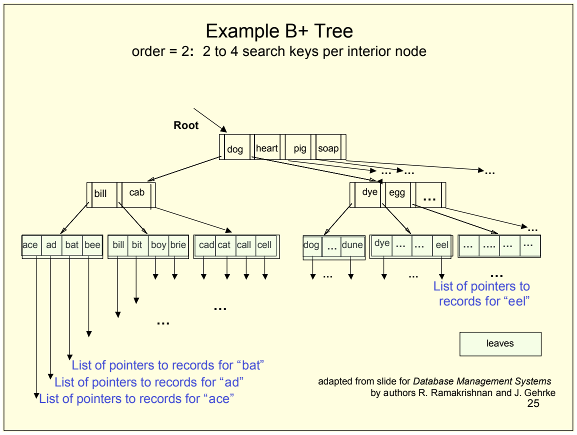
- Properties
  - Data entries only in leaves
    - Compare B-trees
  - One block per tree node, including leaves
  - All leaves same distance from root => balanced
  - Leaves doubly linked
    - Gives sorted data entries
  - Call search key of tree “B+ key”

B+ trees continued

- To achieve equal distance all leaves to root cannot have fixed fanout
- To keep height low, need fanout high
  - Want interior nodes full
- Parameter d - order of the B+ tree
- Each interior node except root has m keys for $d \le m \le 2d$
  - m+1 children
- The root has m keys for $1 \le m \le 2d$
  - Tree height grows/shrinks by adding/removing root
- d chosen so each interior node fits in one block



Leaves will be 1/2 full to full as well


Example B+ Tree
order = 2: 2 to 4 search keys per interior node
Root
dog
heart
pig
soap
bill
cab
dye
egg
ace
ad
bat
bee
bill
bit
boy
brie
cad
cat
call
cell
dog
dune
dye
eel
List of pointers to records for "eel"
leaves
List of pointers to records for "bat"
List of pointers to records for "ad"
List of pointers to records for "ace"
adapted from slide for *Database Management Systems*
by authors R. Ramakrishnan and J. Gehrke
25