Chapter 6: Index Structures for Files

index - access structure used to speed up retrieval of recoreds
external to the data

allows quick access to a record using a specified field as a search criterion
- hashing from Ch 4 only permits this kind of access to key attribs

index structure - usually defined on a single field - indexing field
- stores each value of the field along with a list of pointers to the blocks that contain records with that field value
- values in index structure are ordered - binary search is possible
  - smaller than entire file so binary search more efficient

types of indexes:
- primary index: specified on an ordering key field
  ex: TOY db - customer file - ordered on CUST_NUM - indexing field
  index structure contains pointer to the block containing the corresponding CUST_NUM value
- clutering index: if several records in the file can have the same value for the ordering field
- secondary index: specified for non-ordering fields

Primary Indexes:
- two fields in the structure
- first field is of the same data type as the key ordering field of the data file
- second field is a pointer to a disk block
- one index entry for each block in the data file - total number of entries in the index file is the number of disk blocks in the ordered data file
  - non-dense index: fewer index entries than data records
- each index entry contains:
  key value of the first record in the block pointed to by the pointer in the second field
  - first record in each block of the data file is the anchor record

(draw picture here - use TOY db)

- Problems with primary index: insertion / deletion of records
    - have to change some index entries since anchor records may change
    - possible solutions:
        - use an unordered overflow file
        - linked list of overflow records for each block
        - use deletion markers for deletion

Example:

Given: r = 30000 records; B=1024 bytes / block; R=100 bytes / record

Compute:

bfr = floor(B / R) = floor(1024 / 100) = 10 records / block

blocks needed for file = b = ceiling(r / bfr) = ceiling(30000 / 10) = 3000 blocks

binary search: #accesses approx = ceiling($\log_2 b$) = ceiling($\log_2 3000$) = 12 block accesses

size of ordering field V = 9 bytes; size of block pointer P = 6 bytes; size of each index entry R_i = 6 + 9 = 15 bytes

bfr for index = floor(1024 / 15) = 68 entries / block

#entries in index = 3000 (number of blocks in file)

blocks needed for index = b_i = ceiling(r_i / bfr_i) = ceiling(3000 / 68) = 45 blocks

binary search:#accesses approx = celing($\log_2 b_i$) = ceiling($\log_2 45$) = 6 block accesses

additional block access to search for record totals 7 block accesses (better than 12)

Clustering Indexes:

- clustering field - data file sorted on a non-key field - may not be unique
- use clustering index to speed up retrieval on such a file
- clustering index has one entry for each distinct value of the clustering field
    - index entry contains a pointer to the first block in the data file that has the corresponding field value in it

(draw picture - use TOY db - assume TOY file is sorted by manufacturer)

- insertion problem can be handled by reserving a whole block for each value of the clustering field - link together all blocks needed to store data with that value
- non-dense index because it has one entry for each unique value of the clustering field

Secondary Indexes:

- the first field of a secondary index is the same type as a non-ordering field of the data file
- second field is a block pointer or record pointer

- secondary index on a key field (having distinct values for each entry in data file)
    - called secondary key
    - one index entry for each record in the data file - dense index
    - index ordered on the key field - can do binary search

(draw picture)

- with record pointers - index points directly to the location of the field
- blocks pointers - index points to the block containing the field - do linear search once block is moved to main memory

- secondary index on non-key field - 3 possible solutions:
  1) include several index entries with same value - one for each record in data file - dense index

  2) variable length records for index entries - list of pointers for each index value

  3) sindle entry for each index field value - pointer field points to a block of record pointers indicating all records in the data file containing the index field value

- secondary index usually larger than primary - so longer search
  - BUT - gain is greater since without it a linear search of the whole data file would be necessary

Multilevel Indexes:
- indexes of indexes - goal is to reduce the search space
- with single level indexes we have an ordered file on which we can perform a binary search
  - binary search reduces search space by a factor of 2 for each step
  - $\log_2 b$ accesses to find the desired entry (b = # blocks)
- multilevel indexes reduce the search space by a factor of bfr each time
  - $\log_{fo} b$ accesses - fo = fanout = blocking factor
- first level of a multilevel index must be an ordered file of distinct values
- create a primary index to the first level -becomes the second level
  - one entry for each block in the first level index file
- repeat this process for the second level creating a third level index
- continue until all entries of some index fit in a single block - top level
- a multilevel index with r entries in the first level will have t levels where $t = \text{ceiling}(\log_{fo} r)$
  - derive this: each level reduces the number of index entries by a factor of fo
  - since we want to reduce until there is only one block we use the formula: $1 <= (r / (fo)^t)$

(draw picture - use from book)

Example:

Convert primary index of previous example into a multilevel index

Given: r = 30000 records; B=1024 bytes / block; R=100 bytes / record

bfr_i = fo = 68 entries / block; #blocks in first level index = b_1 = 45 blocks

Compute:

b2 = ceiling(b1 / fo) = ceiling(45 / 68) = 1
second level is top level
#accesses = 1 block at each level + 1 data block access = 2 + 1 = 3
better than 7 accesses of single level index

- Problem: insertion / deletion again
- Solution - dynamic multilevel indexes - leave space at end of each block for inserting

Dynamic Multilevel Indexes using B-Trees and B+-Trees:

Search Tree - of order p

- each node contains <= p-1 search values and q <= p children
- at each node: < P1, K1, P2, K2, ..., Pq-1, Kq-1, Pq >
  - where: P1 ... Pq are pointers to subtrees
    K1 < K2 < ... < Kq-1 are key values
    subtree pointed to by Pi has key values between Ki-1 and Ki (assuming values are unique)

(draw picture example)

- Problem: insertion can make tree unbalanced (all leaf nodes not at same level) - deletion can cause wasted space

B-Tree - solves these problems

- search tree with additional constraints (that solve the above problems)
- of order p
- each node contains <= p-1 search values and q <= p children
- at each node:
  - < P1, <K1, Pr1>, P2, <K2, Pr2>, ..., Pq-1, <Kq-1, Prq-1>, Pq>
  - where: Pi is a pointer to a subtree
    Ki is a key value
    Pri is a data pointer to record containing search key value Ki
  - K1 < K2 < ... < Kq-1
  - for X search key value in subtree pointed to by Pi:
    - if i = 1, X < Ki
    - if i = q, X > Ki-1
    - else Ki-1 < X < Ki
  - At most p tree pointers
  - At least ceiling(p / 2) tree pointers
    - root has at least 2 unless only node in tree
  - q-1 search key field values (data pointers)
- all leaf nodes are at the same level - null tree pointers

- insertion and deletion algs are not detailed - B+-Tree algs are because more widely used

(draw picture)

B+-Tree - most commonly used

- data pointers stored only at leaf nodes - point to block where data is located if unique (otherwise to block of pointers)
- structure of leaf nodes different from internal nodes
- leaf nodes are linked to provide sequential access to records as well as access to individual records
- leaf nodes correspond to base level of multilevel indexes - internal nodes correspond to higher level indexes
- each internal node:

< P1, K1, P2, K2, ... , Pq-1, Kq-1, Pq>
q <= p
Pi is a tree pointer
K1 < K2 < ... < Kq-1
for X search key value in subtree pointed to by Pi:
if i = 1, X <= Ki
if i = q, X > Ki-1
else (1 < i < q) Ki-1 < X <= Ki
At most p tree pointers
At least ceiling(p / 2) tree pointers
- root has at least 2 if it is internal
q pointers, q <= p, q-1 search field values

- each leaf node:

<<K1, Pr1>, <K2, Pr2>, ... , <Kq-1, Prq-1>, Pnext>
q <= p
Pri is a data pointer
Pnext points to next leaf node
K1 < K2 < ... < Kq-1
At least floor(p / 2) values

- all leaf nodes are at the same level

- pointers in internal nodes are tree pointers to blocks that are tree nodes
- pointers in leaf nodes are data pointers to the dat file records or blocks
    - except for Pnext - pointer to next leaf node

- Advantage over B-tree:
    -internal nodes do not include data pointers, so can have more entries per node - greater fanout for index files
- Search / insertion / deletion algorithms in book - look if you want

(draw picture)