Overview of Storage and Indexing

Linda Wu

(CMPT 354 • 2004-2)

Topics

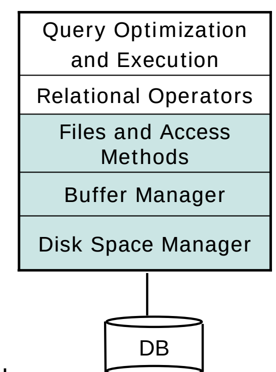
- Data on external storage
- File organizations
- Indexes
- File organizations comparison
- Choice of indexes
- Composite search keys
- Index-only plan
- Index creation in SQL

Data on External Storage

- External storage
  - Disks
    - Retrieve random page at a fixed cost
    - Reading several consecutive pages is much cheaper than reading them in random order
  - Tapes
    - Read pages in sequence
    - Cheaper than disks; used for archival storage
- Record id (rid)
  - A unique identifier for each record in a file
  - rid can be used to identify the disk address of the page containing the record

Data on External Storage (Cont.)

- Architecture
  - File layer makes call to buffer manager before processing a page, specifying the page's rid
  - Buffer manager stages pages from external storage to main memory, and writes them back
  - Disk space manager allocates, keeps track of, and recycles space on disk



File Organizations

- File of records
  - A relation is typically stored as a file of records, and each file consists of one or more pages
  - The file of records is implemented by the files and access methods layer
    - The file layer keeps track of pages allocated to each file, and available space within allocated pages
- The cost of page I/O should be minimized

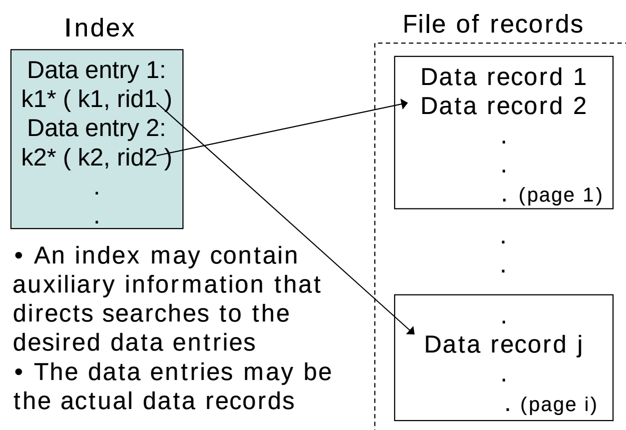
File Organizations (Cont.)

- File organization
  - A method of arranging a file of records on external storage
  - Many alternatives exist, each ideal for some situations, but not good for others
- Alternative file organizations
  - Heap (random order) files: suitable when typical access is a file scan retrieving all records, retrieving a record by its rid
  - Sorted files: best if records must be retrieved in some order, or only a range of records is needed; a file can sort on only one order
  - Indexes: data structures that organize records to optimize certain operations

Indexes

- Indexes
  - Data structures that allow us to find the records with given values in the search key fields
  - * Any subset of the fields of a relation can be the search key for an index on the relation
  - * Search key is not the same as key
- Data entries: records in an index file
  - A data entry with search key value k is denoted as $k*$
  - $k*$ supports efficient retrieval of data records with the given search key value k

Indexes (Cont.)



• An index may contain auxiliary information that directs searches to the desired data entries
• The data entries may be the actual data records

Indexes (Cont.)

- In a data entry, k^*, we can store:
  1. Data record with search key value k, or,
  2. <k, rid> pair (rid is the record id of a data record with search key value k), or,
  3. <k, rid-$list$> pair (rid-$list$ is a list of rids of data records with search key value k)
- Choice of alternatives for data entries is independent of indexing technique used to locate data entries with search key value k
  - Examples of indexing techniques: B+ trees, hash-based structures

Indexes (Cont.)

- Alternative 1 for data entries
  - If this is used, index structure is a file organization for data records (instead of a heap file or sorted file)
  - At most one index on a given collection of data records can use Alternative 1 (otherwise, data records are duplicated, leading to redundant storage and potential inconsistency)
  - If data records are very large, # of pages containing data entries is high. Implies size of auxiliary information in the index is also large, typically

Indexes (Cont.)

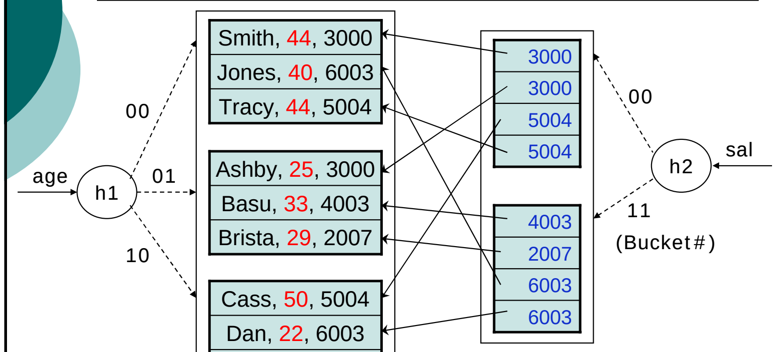
- Alternatives 2 & 3 for data entries
  - Data entries are typically much smaller than data records
  - Better than Alternative 1 with large data records, especially if search keys are small
    - Portion of index structure used to direct searches, which depends on the size of data entries, is much smaller than with Alternative 1
  - Alternative 3 is more compact than Alternative 2, but leads to variable sized data entries even if search keys are of fixed length

Index Data Structures

- Hash-based indexes
  - Good for equality searches
  - Index is a collection of buckets
    - Bucket = primary page plus zero or more overflow pages; contain data entries
  - Hashing function h

    h(r) = bucket # to which (data entry for) record r belongs
    - h looks at the search key fields of r
  - No need for "index entries"
  - Primary page in a given bucket can be retrieved in one or two disk I/Os

Index Data Structures (Cont.)

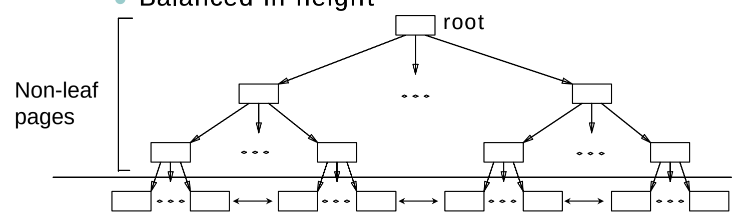


Employees file hashed on age (Alternative 1)

Index file of <sal,rid> pairs, hashed on sal (Alternative 2)

Index Data Structures (Cont.)

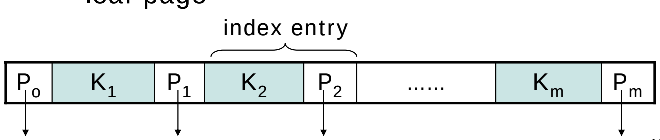
- B+ tree indexes
  - Non-leaf pages contain index entries; only used to direct searches
  - Leaf pages contain data entries, and are chained (via prev/next pointers)
  - Balanced in height



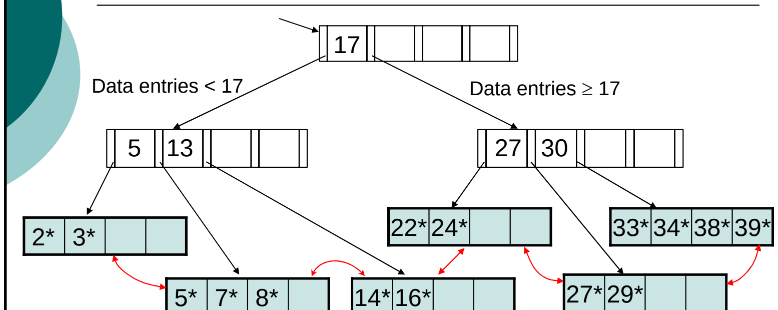
Leaf pages (sorted by search key)

Index Data Structures (Cont.)

- B+ tree index is good for range searches
  - Non-leaf page contains node pointers separated by search key values
    - The node pointer p to the left (right) of a key value k points to a subtree that contains only data entries $<$ ($\geq$) k
  - All searches begin from root, and the non-leaf pages direct searches to the correct leaf page



Index Data Structures (Cont.)



- Find 28*? 29*? All > 15* and < 30*?
- Insert/delete: find data entry in leaf, then change it
  - Need to adjust parent sometimes
  - Change sometimes bubbles up the tree

Index Data Structures (Cont.)

- Disk I/Os with B+ tree index
  - One disk I/O for retrieving one page
  - # of I/O for a search =
    (the length of a path from root to a leaf) + (# of leaf pages with qualifying data entries)
  - Fan-out F: the average number of children for a non-leaf node
    - A tree with fan-out F and of height h has about F^h leaf pages

  * Faster than binary search in a sorted file!

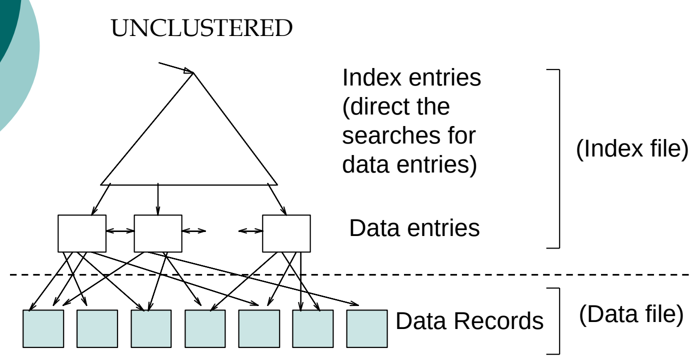
Index Classification

- Primary vs. secondary indexes
  - If the search key of an index contains primary key, it is called primary index; other indexes are called secondary indexes
    - A primary index contains no duplicates
    - Generally, a secondary index contain duplicates
  - Unique index: search key contains a candidate key
    - No duplicates exist in unique index

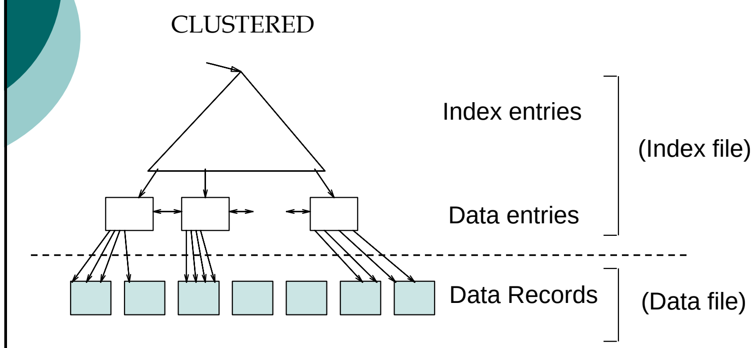
Index Classification (Cont.)

- Clustered vs. unclustered indexes
  - Clustered index: the order of data records is the same as, or close to, the order of data entries
  - Alternative 1 implies clustered; in practice, clustered also implies Alternative 1 since sorted files are rare (too expensive)
  - Alternatives 2 and 3 are unclustered
  - A file can be clustered on at most one search key
  - The cost of a range search query through index varies greatly based on whether index is clustered or not!

Index Classification (Cont.)



Index Classification (Cont.)



Index Classification (Cont.)

- Suppose that Alternative 2 is used for data entries, and that the data records are stored in a heap file
  - To build a clustered index, first sort the heap file (with some free space on each page for future inserts)
  - Overflow pages may be needed for inserts (thus, the order of data records is close to, but not identical to, the sort order)

File Organizations Comparison

- File organizations

  employees records: (name, age, sal)

  - Heap file
  - Sorted file, sorted on <*age, sal*>
  - Clustered B+ tree file, Alternative (1), search key <*age, sal*>
  - Heap file with unclustered B + tree index on search key <*age, sal*>
  - Heap file with unclustered hash index on search key <*age, sal*>

File Organizations Comparison (Cont.)

- Operations to compare
  - Scan (fetch all records in the file)
  - Equality search
  - Range search
  - Insert a record
  - Delete a record

File Organizations Comparison (Cont.)

- Cost model (execution time)
  - B: # of data pages when records are packed onto pages with no wasted space
  - R: # of records per page
  - F: fan-out of B+ tree
  - I/O cost → D: average time to read or write a page
  - CPU cost { C: average time to process a record
  - H: time to apply hash function to a record

* Using # of read or written pages as measure of I/O: ignore the gain of pre-fetching a sequence of pages

* Average case analysis based on simplistic assumptions

File Organizations Comparison (Cont.)

- Assumptions
  - Heap File
    - Equality selection on candidate key; exactly one match
    - Insert at the end of the file
  - Sorted File
    - The pages in the file are stored sequentially
    - File is compacted after deletions
  - Equality / range selection
    - Selections match the search key <age, sal>, i.e., they are specified on at least the first field in the search key

File Organizations Comparison (Cont.)

- Indexes
  - Tree
    - Typically 67% page occupancy
    - Implies file size = 1.5 data size
  - Hash
    - Bucket contains only 1 page
    - 80% page occupancy => File size = 1.25 data size
  - Alternatives (2), (3)
    - Data entry size = 10% size of record
    - Both index data entries and actual file are scanned in case of unclustered indexes

File Organizations Comparison (Cont.)

- Cost of I/O operations

	Scan	Equality search	Range search	Insert	Delete
Heap	BD	0.5BD	BD	2D	Search + D
Sorted	BD	D $\log_2 B$	D ($\log_2 B$ + # matching pgs)	Search + BD	Search + BD
Clustered	1.5BD	D $\log_F$ 1.5B	D ($\log_F$ 1.5B + # matching pgs)	Search + D	Search + D
Unclustered tree index	BD (R + 0.15)	D(1 + $\log_F$ 0.15B)	D ($\log_F$ 0.15B + # matching recds)	Search + 2D	Search + 2D
Unclustered hash index	BD (R + 0.125)	2D	BD	Search + 2D	Search + 2D

Understanding the Workload

- For each query in the workload:
  - Which relations does it access?
  - Which attributes are retrieved?
  - Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?
- For each update in the workload:
  - Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?
  - The type of update (INSERT/DELETE/UPDATE), and the attributes that are affected

Choice of Indexes

- What indexes should be created?
  - Which relations should have indexes?
  - What field(s) should be the search key?
  - Should we build several indexes?
- What kind of index should it be?
  - Clustered?
  - Hash/tree?
- Trade-off
  - Indexes make queries faster, updates slower; require disk space, too

Choice of Indexes (Cont.)

- Index selection guidelines
  1. Try to choose indexes that benefit as many queries as possible
  2. Choose clustered index based on the important queries that would benefit the most from clustering
  3. Attributes in WHERE clause are candidates for index keys
     - Exact match condition suggests hash index
     - Range query suggests tree index
     - Clustering is especially useful for range queries; also help on equality queries if there are many duplicates

Choice of Indexes (Cont.)

4. Multi-attribute search keys should be considered when a WHERE clause contains several conditions
   - Order of attributes is important for range queries
   - Such indexes can sometimes enable index-only strategies
     - For index-only strategies, clustering is not important

Choice of Indexes (Cont.)

- Clustered indexes
  - A file organization for data records
  - At most one clustered index on a given collection of data
  - There can be several unclustered indexes on a data file
  - Clustered index is expensive to maintain, therefore, used only when there are frequent queries that benefit from it
  - Clustered index is typically built using tree, not hashing

Choice of Indexes (Cont.)

- Example 1: B+ tree index on *age* can be used to get qualifying tuples
  - How selective is the condition?
  - Is the index clustered?
- Example 2: equality queries and duplicates
  - If many employees collect stamps, clustering on *hobby* helps!

```
SELECT  E.dno
FROM    Emp E
WHERE   E.age > 40
                    (1)
```

```
SELECT  E.dno
FROM    Emp E
WHERE   E.hobby = 'Stamp'
                    (2)
```


Choice of Indexes (Cont.)

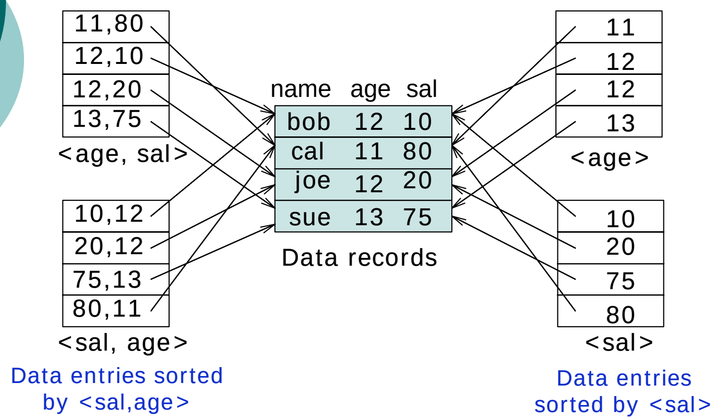
- Example 3: GROUP BY query
  - If every tuple has *age* > 10, using *age* index and sorting the retrieved tuples may be costly
  - Clustered *dno* index may be better!

```
SELECT    E.dno, COUNT (*)
FROM      Emp E
WHERE     E.age > 10
GROUP BY  E.dno
                    (3)
```


Composite Search Keys

- Composite search keys: search on a combination of fields
  - Equality query: every field value is equal to a constant value
    - e.g. <sal,age> index: age=20 and sal =75
  - Range query: some field value is not a constant
    - e.g. age =20; or, age=20 and sal > 10
- Data entries in index are sorted by search key to support range queries
- Composite indexes are larger, updated more often

Composite Search Keys (Cont.)



Composite Search Keys (Cont.)

To retrieve Employees records with conditions:

- *age*=30 AND *sal*=4000
  - An index on <*age*, *sal*> would be better than an index on *age* or an index on *sal*
- 20<*age*<30 AND 3000<*sal*<5000
  - Clustered tree index on <*age*, *sal*> or <*sal*, *age*> is best
- *age*=30 AND 3000<*sal*<5000
  - Clustered <*age*, *sal*> index is much better than <*sal*, *age*> index

Index-Only Plans

- A number of queries can be answered without retrieving any tuple from one or more of the relations involved if a suitable index is available

```
SELECT    E.dno, COUNT(*)
FROM      Emp E
GROUP BY  E.dno                (1)
```

<dno>

```
SELECT    AVG(E.sal)           (2)
FROM      Emp E
WHERE     E.age=25 AND
  E.sal BETWEEN 3000 AND 5000
```

Tree index on <age, sal> or <sal, age>

```
SELECT    E.dno, MIN(E.sal)
FROM      Emp E
GROUP BY  E.dno                (3)
```

Tree index on <dno, sal>

Index-Only Plans (Cont.)

- Index-only plan is possible for (4) if we have a tree index on <dno, age> or <age, dno>
  - Which one is better?
  - What if we consider query (5)?

```
SELECT    E.dno, COUNT (*)
FROM      Emp E
WHERE     E.age=30
GROUP BY  E.dno                (4)
```

```
SELECT    E.dno, COUNT (*)
FROM      Emp E
WHERE     E.age>30
GROUP BY  E.dno                (5)
```

Tree index on <dno, age> or <age, dno>

Index-Only Plans (Cont.)

- Index-only plans can also be found for queries involving more than one table

```
SELECT   D.mgr
FROM     Dept D, Emp E
WHERE    D.dno=E.dno
```

<E.dno>

```
SELECT   D.mgr,  E.eid
FROM     Dept D, Emp E
WHERE    D.dno=E.dno
```

<E.dno, E.eid>

Index Creation in SQL

- SQL-99 does not include any statement for creating or dropping indexes
- In practice, every commercial relational DBMS supports indexes

Syntax (MS SQL Server)

```
CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ]
INDEX index_name
   ON { table | view } ( column [ ASC | DESC ] [ ,...n ] )
[ WITH < index_option > [ ,...n] ]
[ ON filegroup ]
```


Summary

- Many alternative file organizations exist, each appropriate in some situation
- If selection queries are frequent, sorting the file or building an *index* is important
- Index is a collection of data entries plus a way to quickly find entries with given key values
- Data entries can be actual data records, <key, rid> pairs, or <key, rid-list> pairs
- There can be several indexes on a given file of data records, each with a different search key

Summary (Cont.)

- Indexes can be classified as clustered vs. unclustered, primary vs. secondary; the differences have important consequences for utility / performance
- Understanding the nature of the *workload* for the application, and the performance goals, is essential to developing a good design
- Indexes must be chosen to speed up important queries (and perhaps some updates!)