File Organizations and Indexing

15-415, Spring 2003, Lecture 5

R&G end of Chapter 9 & Chapter 8

> "If you don't find it in the index, look very carefully through the entire catalogue."
>
> -- Sears, Roebuck, and Co., Consumer's Guide, 1897

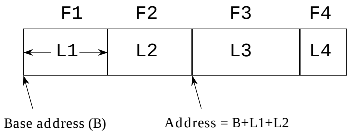
Review: Memory, Disks

- Storage Hierarchy: cache, RAM, disk, tape, ...
  - Can't fit everything in RAM (usually).
- "Page" or "Frame" - unit of buffer management in RAM.
- "Page" or "Block" unit of interaction with disk.
- Importance of "locality" and sequential access for good disk performance.
- Buffer pool management
  - Slots in RAM to hold Pages
  - Policy to move Pages between RAM & disk

Today: File Storage

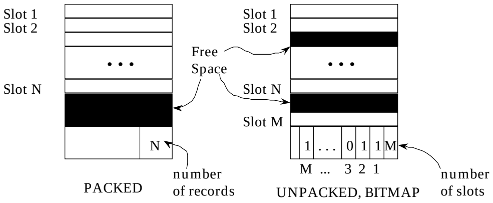
- Page or block is OK when doing I/O, but higher levels of DBMS operate on *records*, and *files of records*.
- Next topics:
  - How to organize records within pages.
  - How to keep pages of records on disk.
  - How to efficiently support operations on files of records.

Record Formats: Fixed Length



- Information about field types same for all records in a file; stored in *system catalogs.*
- Finding *i'th* field done via arithmetic.

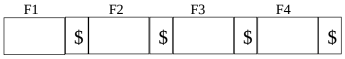
Page Formats: Fixed Length Records



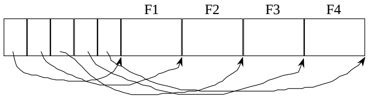
- *Record id = <page id, slot #>. In first alternative, moving records for free space management changes rid; may not be acceptable.*

Variable Length is more complicated

- Two alternative formats (# fields is fixed):



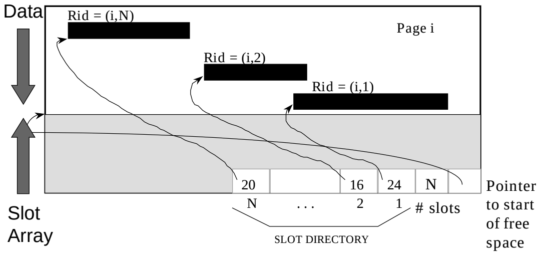
Fields Delimited by Special Symbols



Array of Field Offsets

✉ Offset approach: **pros** - direct access to i'th field and efficient storage of *nulls*; **cons** - small directory overhead and indirection on lookup.

“Slotted Page” for Variable Length Records



- *Record id = <page id, slot #>*
- Can move records on page without changing rid; so, attractive for fixed-length records too.
- Page is full when data space and slot array meet.

System Catalogs

- For each relation:
  - name, file name, file structure (e.g., Heap file)
  - attribute name and type, for each attribute
  - index name, for each index
  - integrity constraints
- For each index:
  - structure (e.g., B+ tree) and search key fields
- For each view:
  - view name and definition
- Plus stats, authorization, buffer pool size, etc.

☒ *Catalogs are themselves stored as relations*!

Attr_Cat(attr_name, rel_name, type, position)

attr_name	rel_name	type	position
attr_name	Attribute_Cat	string	1
rel_name	Attribute_Cat	string	2
type	Attribute_Cat	string	3
position	Attribute_Cat	integer	4
sid	Students	string	1
name	Students	string	2
login	Students	string	3
age	Students	integer	4
gpa	Students	real	5
fid	Faculty	string	1
fname	Faculty	string	2
sal	Faculty	real	3

Files

- FILE: A collection of pages, each containing a collection of records.
- Must support:
  - insert/delete/modify record
  - read a particular record (specified using *record id*)
  - scan all records (possibly with some conditions on the records to be retrieved)

Indexes

- Sometimes, we want to retrieve records by specifying the *values in one or more fields*, e.g.,
  - Find all students in the “CS” department
  - Find all students with a gpa > 3
- An *index* on a file speeds up selections on the *search key fields* for the index.
  - 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* (e.g., doesn’t have to be unique).
- An index contains a collection of *data entries*, and supports efficient retrieval of all records with a given search key value **k**.

Index Classification

- Representation of data entries in index
  - i.e., what is at the bottom of the index?
  - 3 alternatives here
- Clustered vs. Unclustered
- Primary vs. Secondary
- Dense vs. Sparse
- Single Key vs. Composite
- Tree-based, hash-based, other

Alternatives for Data Entry **k*** in Index

1. Actual data record (with key value **k**)
2. <**k**, rid of matching data record>
3. <**k**, list of rids of matching data records>

- Choice is orthogonal to the indexing technique.
  - Examples of indexing techniques: B+ trees, hash-based structures, R trees, …
  - Typically, index contains auxiliary info that directs searches to the desired data entries
- Can have multiple (different) indexes per file.
  - E.g. file sorted on *age*, with a hash index on *salary* and a B+tree index on *name*.

Alternatives for Data Entries (Contd.)

Alternative 1:

Actual data record (with key value **k**)

- If this is used, index structure is a file organization for data records (like Heap files or sorted files).
- At most one index on a given collection of data records can use Alternative 1.
- This alternative saves pointer lookups but can be expensive to maintain with insertions and deletions.

Alternatives for Data Entries (Contd.)

Alternative 2

<**k**, rid of matching data record>

and Alternative 3

<**k**, list of rids of matching data records>

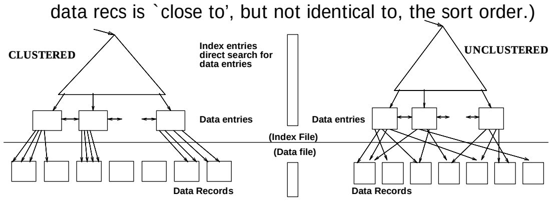
- Easier to maintain than Alternative 1.
- If more than one index is required on a given file, at most one index can use Alternative 1; rest must use Alternatives 2 or 3.
- Alternative 3 more compact than Alternative 2, but leads to *variable sized data* entries even if search keys are of fixed length.
- Even worse, for large rid lists the data entry would have to span multiple pages!

Index Classification - clustering

- *Clustered* vs. *unclustered*: If order of data records is the same as, or `close to', order of index data entries, then called *clustered index.*
  - A file can be clustered on at most one search key.
  - Cost of retrieving data records through index varies *greatly* based on whether index is clustered or not!
  - Note: Alternative 1 implies clustered, *but not vice-versa*.

Clustered vs. Unclustered Index

- Suppose that Alternative (2) is used for data entries, and that the data records are stored in a Heap file.
  - To build 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, order of data recs is `close to', but not identical to, the sort order.)



Clustered vs. Unclustered Index

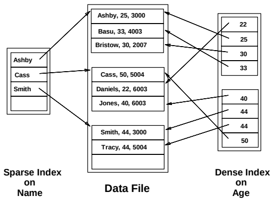
- What are the tradeoffs????
- Clustered Pros
  - Efficient for range searches
  - May be able to do some types of compression
  - Possible locality benefits (related data?)
  - ???
- Clustered Cons
  - Expensive to maintain (on the fly or sloppy with reorganization)

Primary vs. Secondary Index

- *Primary*: index key includes the file's primary key
- *Secondary*: any other index
  - – Sometimes confused with Alt. 1 vs. Alt. 2/3
  - – Primary index never contains duplicates
  - – Secondary index may contain duplicates
    - If index key contains a candidate key, no duplicates => unique index

Dense vs. Sparse Index

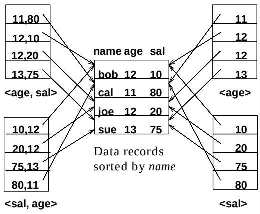
- *Dense:* at least one data entry per key value
- *Sparse:* an entry per data page in file
  - – Every sparse index is clustered!
  - – Sparse indexes are smaller; however, some useful optimizations are based on dense indexes.
  - – Alternative 1 always leads to dense index.



Composite Search Keys

- Search on *combination* of fields.
  - – Equality query: Every field is equal to a constant value. E.g. wrt <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 sorted by search key for range queries.
  - – Lexicographic or Spatial order.

Examples of composite key indexes using lexicographic order.



Tree vs. Hash-based index

- Hash-based index
  - – Good for equality selections.
    - File = a collection of <u>*buckets*</u>. Bucket = *primary* page plus 0 or more *overflow* pages.
    - *Hash function* **h**: **h**(*r*) = bucket in which record *r* belongs. **h** looks at only the fields in the *search key.*
- Tree-based index
  - – Good for range selections.
    - Hierarchical structure (Tree) directs searches
    - Leaves contain data entries sorted by search key value
    - B+ tree: all *root*->leaf paths have equal length *(height)*

Alternative File Organizations

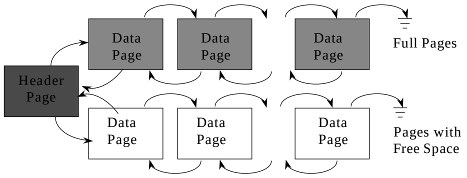
Many alternatives exist, *each good for some situations, and not so good in others:*

- – <u>Heap files:</u> Suitable when typical access is a file scan retrieving all records.
- – <u>Sorted Files:</u> Best for retrieval in *search key* order, or for a `range' of records.
- – <u>Clustered Files:</u> Clustered B+ tree file with search key
- – <u>Heap file w/ unclustered B+ tree index</u>
- – <u>Heap file w/ unclustered Hash index</u>

Heap (Unordered) Files

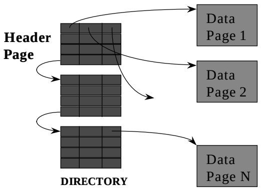
- Simplest file structure
  - – contains records in no particular order.
- As file grows and shrinks, disk pages are allocated and de-allocated.
- To support record level operations, we must:
  - – keep track of the *pages* in a file
  - – keep track of *free space* on pages
  - – keep track of the *records* on a page
- There are many design alternatives for these.

Heap File Implemented as a List



- The header page id and Heap file name must be stored someplace.
- Each page contains 2 `pointers' plus data.

Heap File Using a Page Directory



- The entry for a page can include the number of free bytes on the page.
- The directory is a collection of pages; linked list implementation is just one alternative.
  - *Much smaller than linked list of all HF pages*!

Quick and Dirty Cost Model

We ignore CPU costs, for simplicity:

- **B:** The number of data pages
- **R:** Number of records per page
- **D:** (Average) time to read or write disk page
- Measuring number of page I/O's ignores gains of pre-fetching and sequential access; thus, even I/O cost is only loosely approximated.
- Average-case analysis; based on several simplistic assumptions.

🖂 **Good enough to show the overall trends!**

Some Assumptions in The Analysis

- Single record insert and delete.
- Equality selection - exactly one match (Question: what if more or less???).
- Heap Files:
  - Insert always appends to end of file.
- Sorted Files:
  - Files compacted after deletions.
  - Selections on search key.
- Clustered Files:
  - 67% page occupancy

Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

	Heap File	Sorted File	Clustered File
Scan all records			
Equality Search			
Range Search			
Insert			
Delete			

Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

	Heap File	Sorted File	Clustered File
Scan all records	BD	BD	1.5 BD
Equality Search			
Range Search			
Insert			
Delete			

Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

	Heap File	Sorted File	Clustered File
Scan all records	BD	BD	1.5 BD
Equality Search	0.5 BD	(log$_2$ B) * D	(log$_F$ 1.5B) * D
Range Search			
Insert			
Delete			

Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

	Heap File	Sorted File	Clustered File
Scan all records	BD	BD	1.5 BD
Equality Search	0.5 BD	(log$_2$ B) * D	(log$_F$ 1.5B) * D
Range Search	BD	((log$_2$ B) + #match pg)*D	((log$_F$ 1.5B) + #match pg)*D
Insert			
Delete			

Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

	Heap File	Sorted File	Clustered File
Scan all records	BD	BD	1.5 BD
Equality Search	0.5 BD	(log$_2$ B) * D	(log$_F$ 1.5B) * D
Range Search	BD	((log$_2$ B) + #match pg)*D	((log$_F$ 1.5B) + #match pg)*D
Insert	2D	((log$_2$B)+B)D *(because R,W 0.5)*	((log$_F$ 1.5B)+1) * D
Delete			

Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

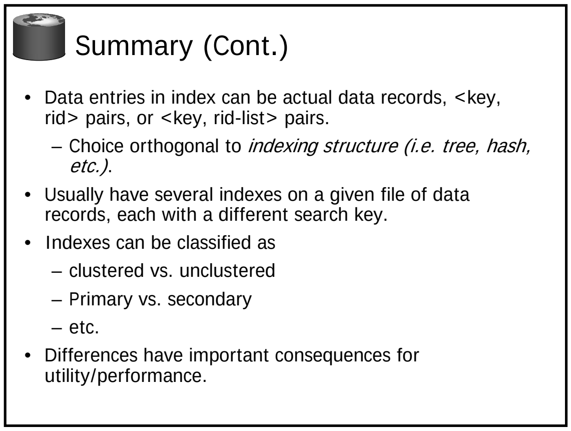
	Heap File	Sorted File	Clustered File
Scan all records	BD	BD	1.5 BD
Equality Search	0.5 BD	(log$_2$ B) * D	(log$_F$ 1.5B) * D
Range Search	BD	((log$_2$ B) + #match pg)*D	((log$_F$ 1.5B) + #match pg)*D
Insert	2D	((log$_2$B)+B)D	((log$_F$ 1.5B)+1) * D
Delete	0.5BD + D	((log$_2$B)+B)D *(because R,W 0.5)*	((log$_F$ 1.5B)+1) * D

Summary

- Variable length record format with field offset directory offers support for direct access to i'th field and null values.
- Slotted page format supports variable length records and allows records to move on page.
- File layer keeps track of pages in a file, and supports abstraction of a collection of records.
  - Also tracks availability of free space
- Catalog relations store information about relations, indexes and views. (*Information that is common to all records in a given collection.*)

Summary (Cont.)

- 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.
  - Hash-based indexes only good for equality search.
  - Sorted files and tree-based indexes best for range search; also good for equality search. (Files rarely kept sorted in practice; B+ tree index is better.)



Summary (Cont.)

- Data entries in index can be actual data records, <key, rid> pairs, or <key, rid-list> pairs.
  - Choice orthogonal to *indexing structure (i.e. tree, hash, etc.)*.
- Usually have several indexes on a given file of data records, each with a different search key.
- Indexes can be classified as
  - clustered vs. unclustered
  - Primary vs. secondary
  - etc.
- Differences have important consequences for utility/performance.