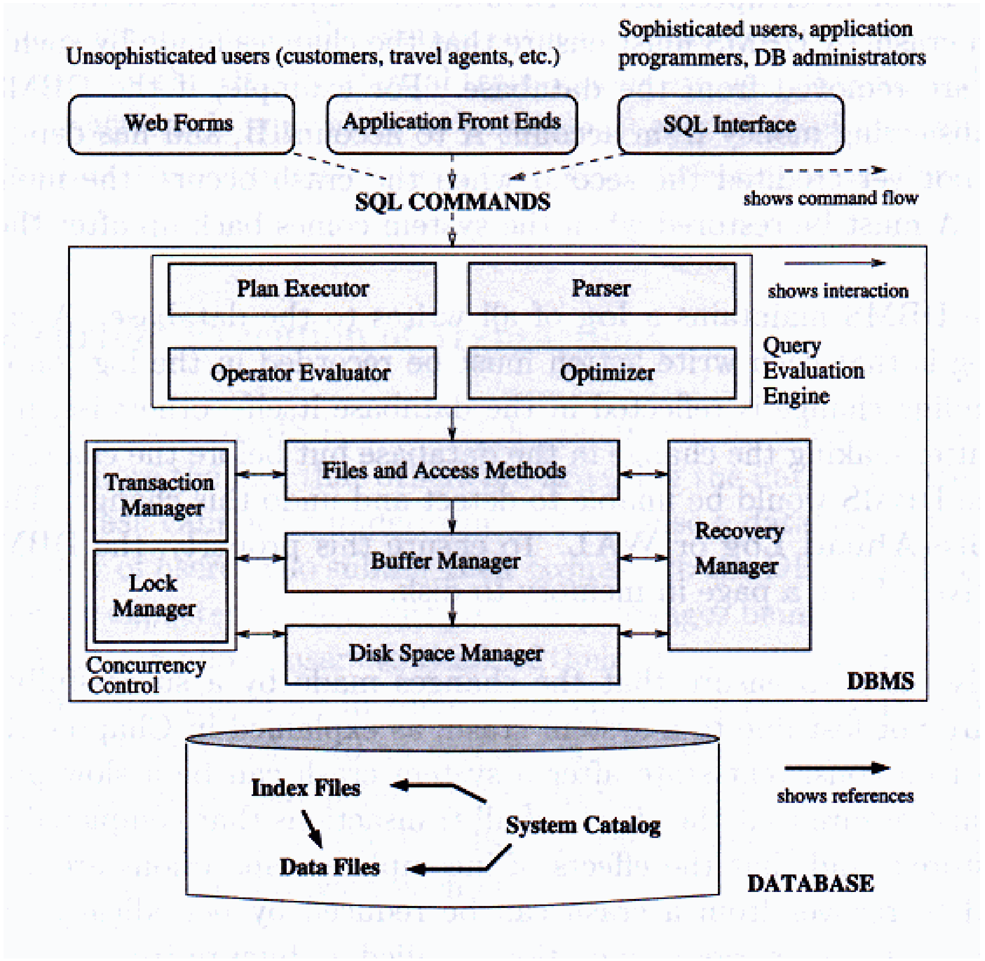
Storage and Indexes



Architecture of a DBMS

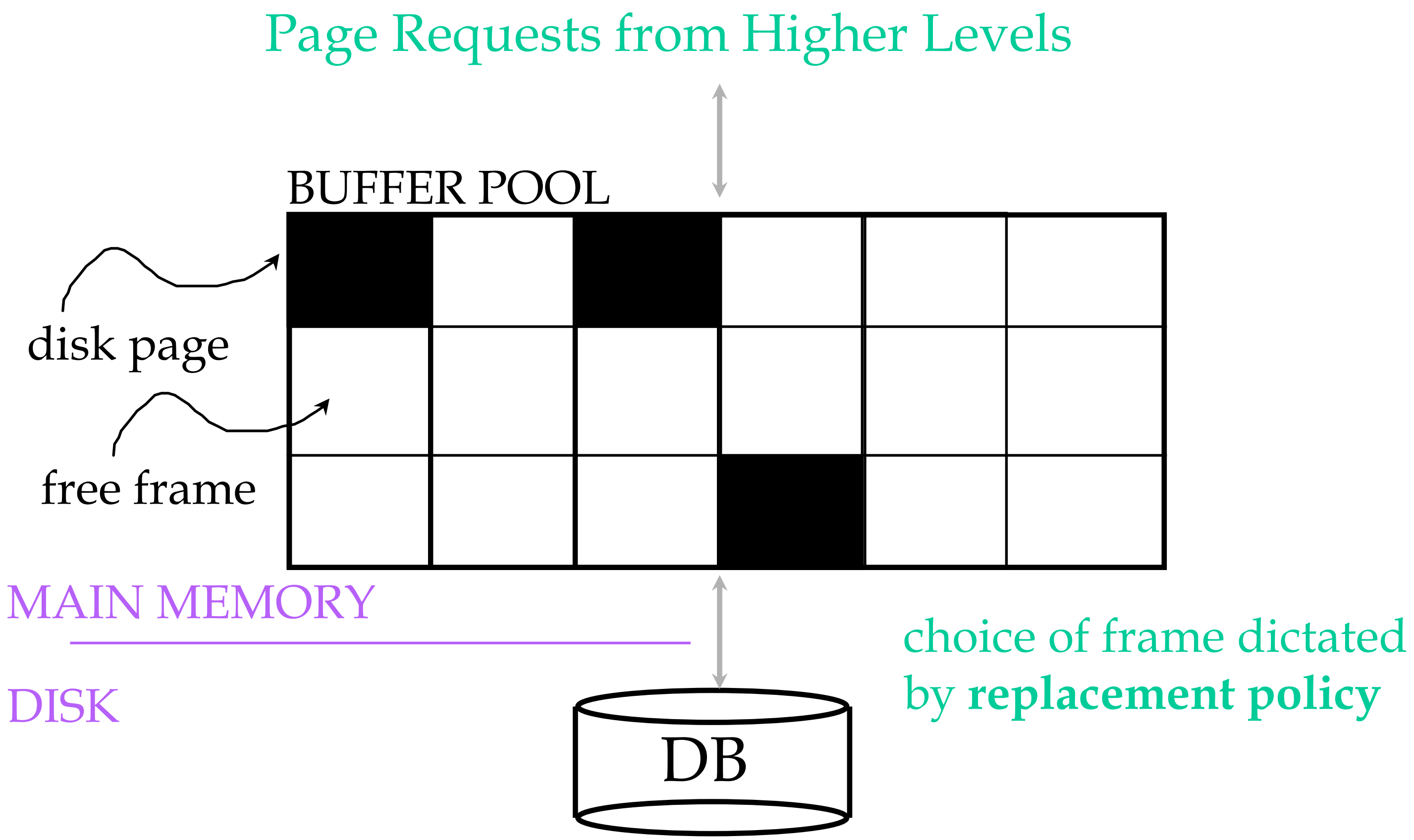
Disks and Files

- DBMS stores information on ("hard") disks.
  - A disk is a sequence of bytes, each has a ***disk address***.
  - READ: transfer data from disk to main memory (RAM).
  - WRITE: transfer data from RAM to disk.
  - Both are high-cost, relative to in-memory operations.
- Data are stored and retrieved in units called *disk blocks* or *pages*.
  - Each page has a fixed size, say 512 bytes. It contains a sequence of ***records***.
  - In many (not always) cases, records in a page have the same size, say 100 bytes.
  - In many (not always) cases, records implement relational tuples.

Why Not Store Everything in Main Memory?

- *Costs too much.* Ram is much more expensive than disk.
- *Main memory is volatile.* We want data to be saved between runs. (Obviously!)
- Typical storage hierarchy:
  - Main memory (RAM) for currently used data.
  - Disk for the main database (secondary storage).
  - Tapes for archiving older versions of the data (tertiary storage).

Buffer Management in a DBMS



Page Requests from Higher Levels
BUFFER POOL
disk page
free frame
MAIN MEMORY
DISK
DB
choice of frame dictated by **replacement policy**

- *Data must be in RAM for DBMS to operate on it!*
- *Table of <frame#, pageid> pairs is maintained.*

When a Page is Requested ...

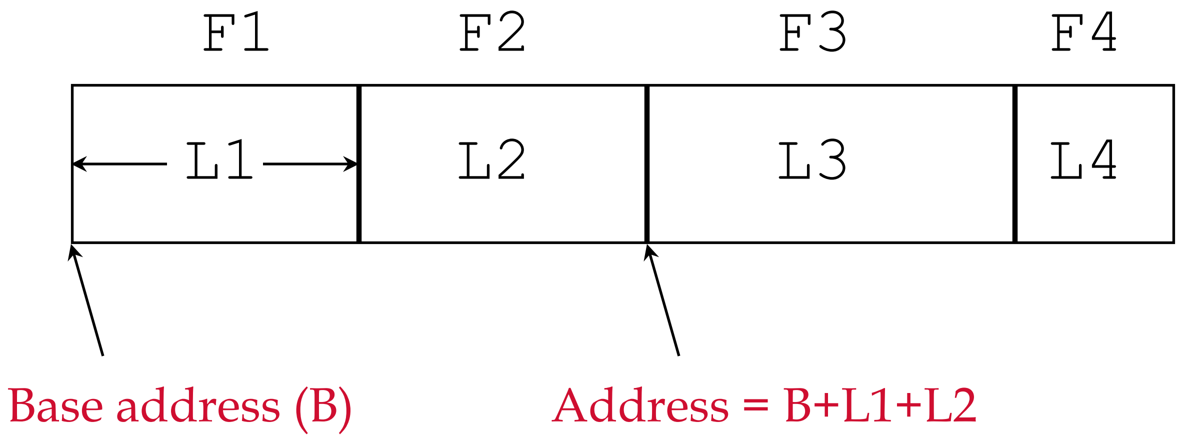
- If requested page is not in pool:
  - Choose a frame for *replacement*
  - If frame is dirty, write it to disk
  - Read requested page into chosen frame
- *Pin* the page and return its address.

If requests can be predicted (e.g., sequential scans) pages can be pre-fetched several pages at a time!

More on Buffer Management

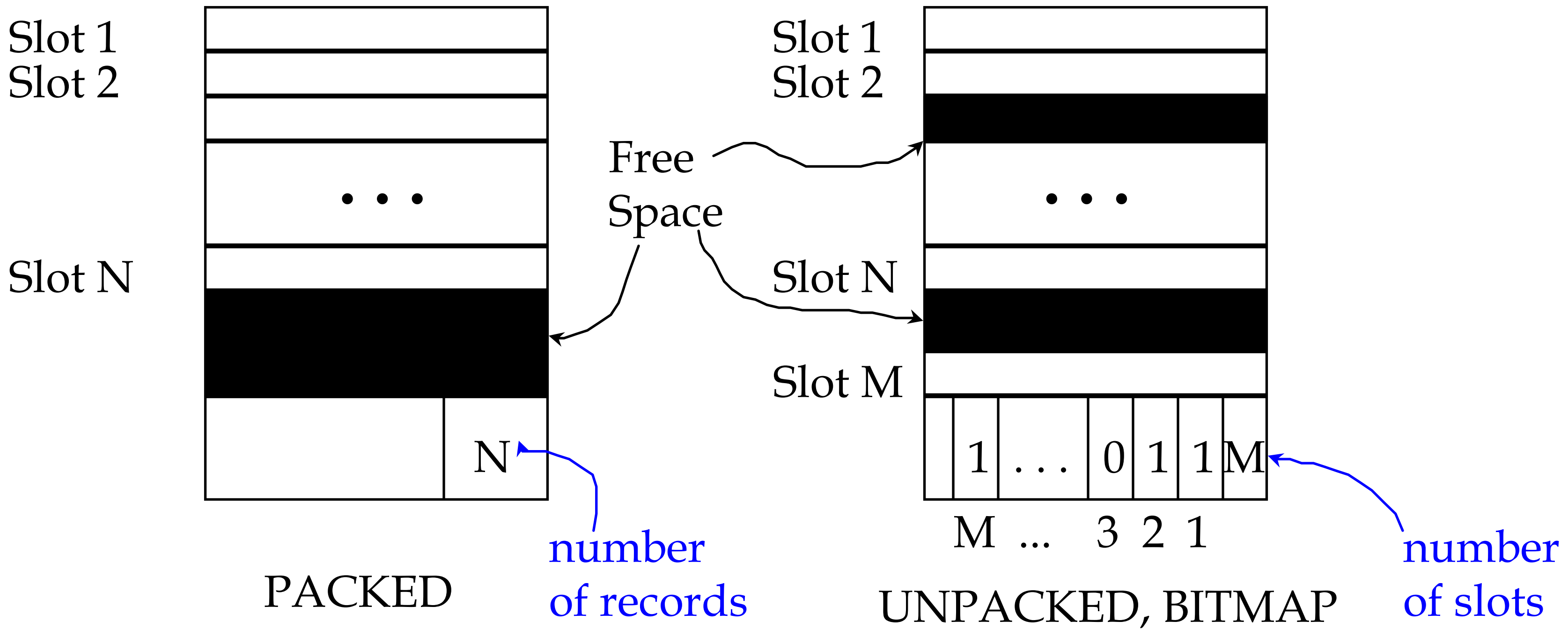
- Requestor of page must unpin it, and indicate whether a page has been modified:
  - *dirty* bit is used for this.
- Page in pool may be requested many times,
  - a *pin count* is used. A page is a candidate for replacement iff *pin count* = 0.
- CC & recovery may entail additional I/O when a frame is chosen for replacement. (*Write-Ahead Log* protocol; more later.)
- Frame is chosen for replacement by a *replacement policy:*
  - Least-recently-used (LRU), Clock, MRU etc.

Record Formats: Fixed Length



- Information about field types same for all records in a file; stored in *system catalogs.*

Page Formats: Fixed Length Records

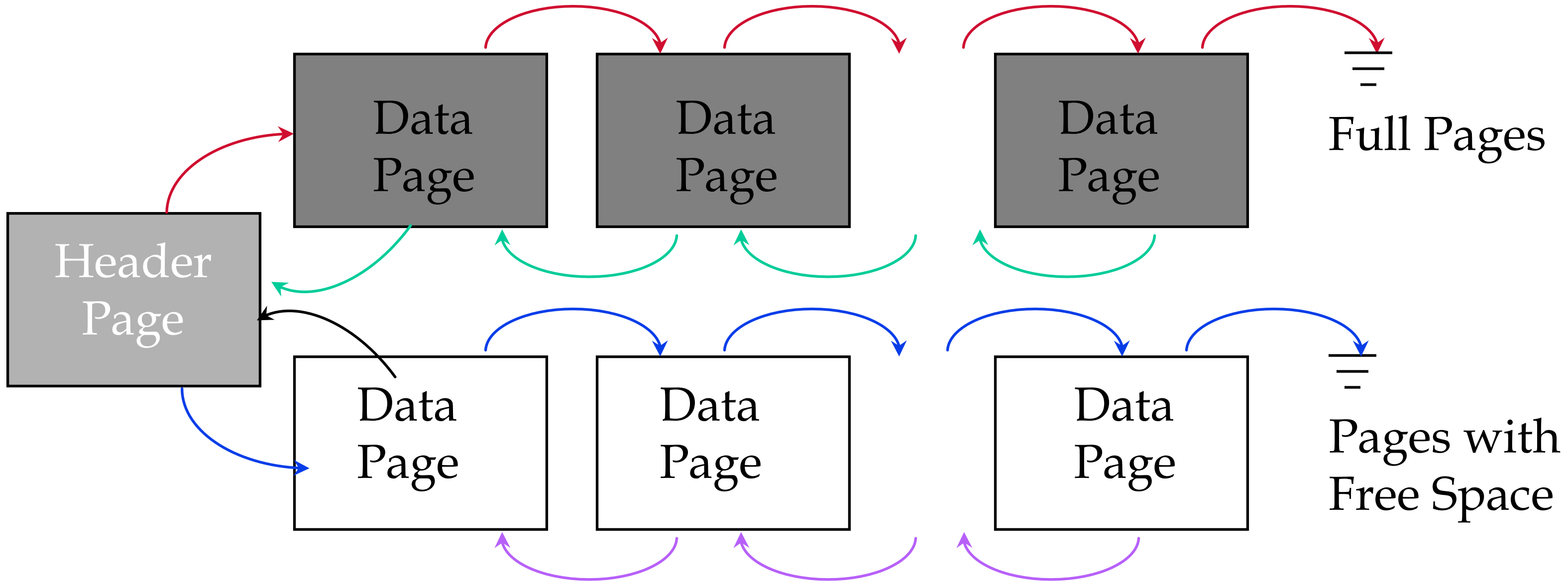


Record id = <page id, slot #>. In the first alternative, moving records for free space management changes rid; may not be acceptable.

Files of Records

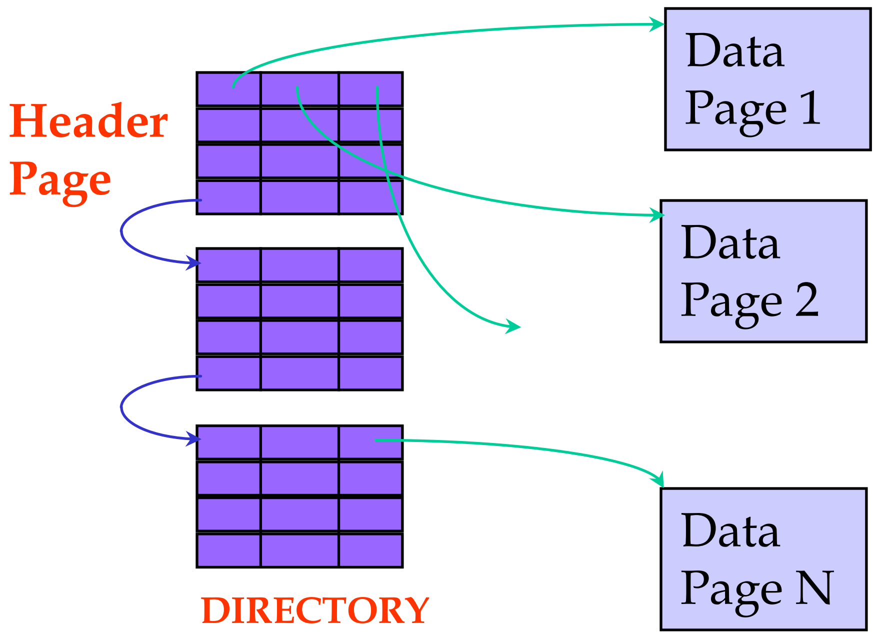
- Page or block is OK when doing I/O, but higher levels of DBMS operate on *records*, and *files of records*.
- FILE: A collection of pages, each containing a collection of records. Must support:
  - insert/delete/modify record
  - read a particular record (specified using *record id*)
  - scan all records (possibly with some conditions on the records to be retrieved)

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!*

Indexes

- 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* (minimal set of fields that uniquely identify a record in a relation).
- An index contains a collection of *data entries*
- A data entry is denoted as **k*,** where **k** is a search key value and ***** tells where to find the record containing **k**
- Index must support efficient retrieval of all data entries **k*** with a given key value **k**
- Structure of data entry in more detail

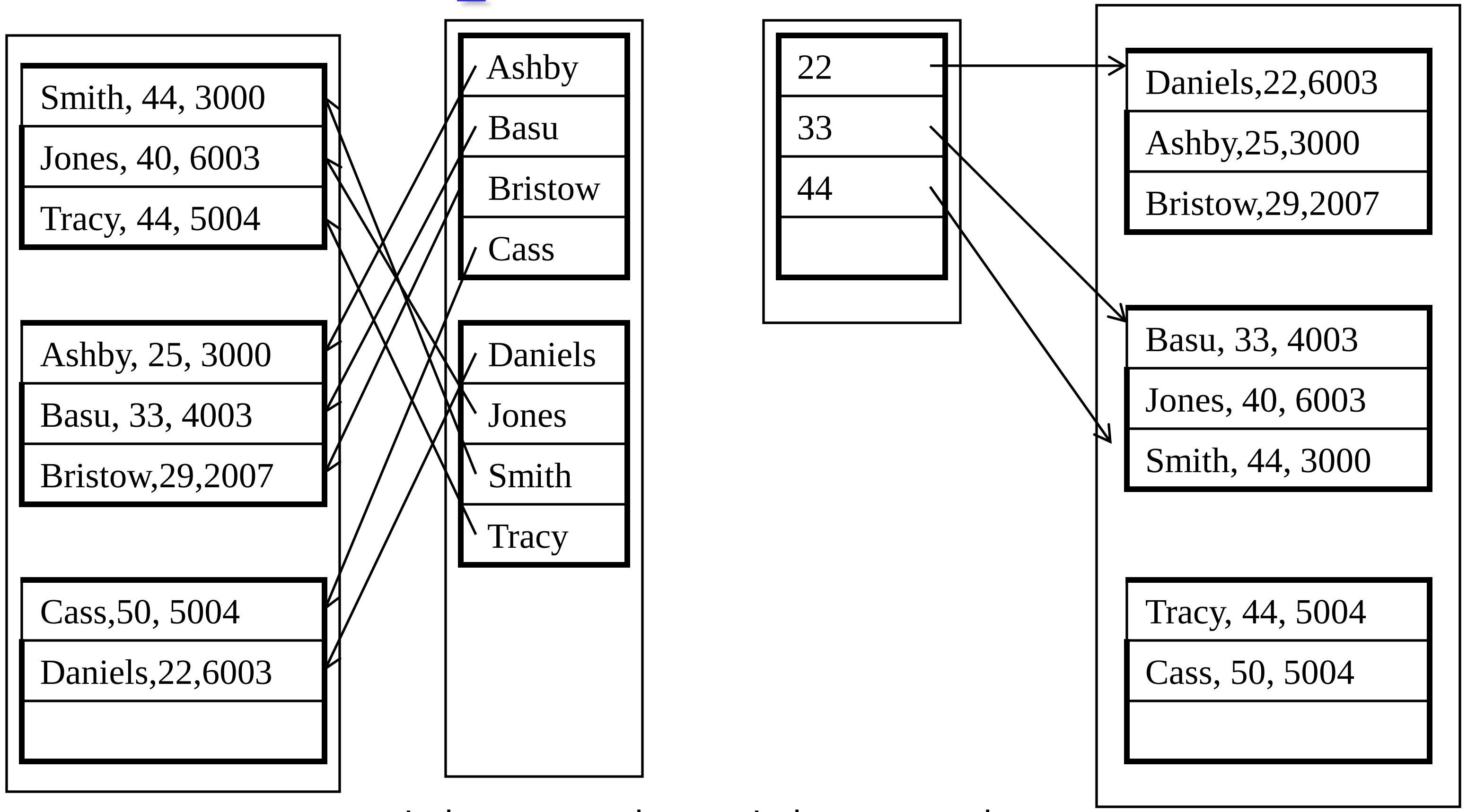
Alternatives for Data Entry k* in Index

- Three alternatives:
  1. <k, data record with search key value k> (not often used)
  2. <k, rid of data record with search key value k> (often used)
  3. <k, list of rids of data records with search key value k>
- Examples, assuming field ‘name’ is the search key
  1. <“Lin Wang”, (“Lin Wang”, 25, “12 First Street”, 26094359)>
  2. <“Lin Wang”, 10101> where 10101 is the rid of a record that contains “Lin Wang”
  3. <“Lin Wang”, 10101, 10111, 11010> where 10101, 10111, 11010 are records which all contain “Lin Wang”.
- Note:
  – for alternative 1, data entries are actually data records.
  – rid is record id

Index Classification

- *Primary* vs. *secondary*: If search key contains primary key, then called primary index, otherwise secondary.
  - *Unique* index: Search key contains a candidate key.
- *Clustered* vs. *unclustered*: If order of data records is the same as, or `close to', order of 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!

Examples of Indexes



Heap file

Index on search key 'name'
Class: unclustered, primary

This index uses alternative 2

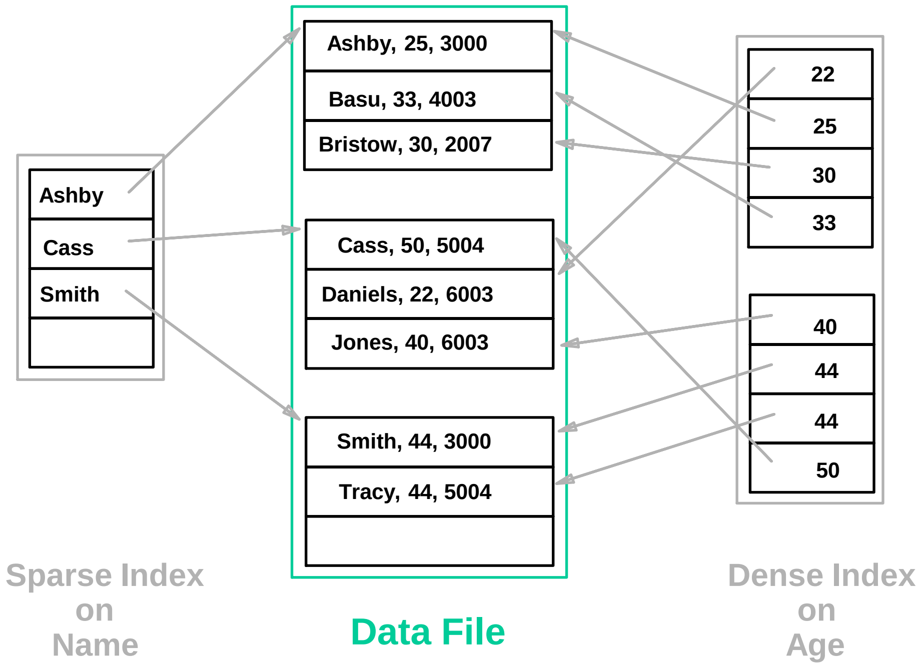
Index on search Key 'age'
Class: clustered secondary

This index uses alternative 2

Sorted file

Index Classification (Contd.)

- *Dense vs. Sparse:* If there is at least one data entry per search key value (in some data record), then dense.
  - Alternative 1 always leads to dense index.
  - Every sparse index is clustered!
  - Sparse indexes are smaller; however, some useful optimizations are based on dense indexes.

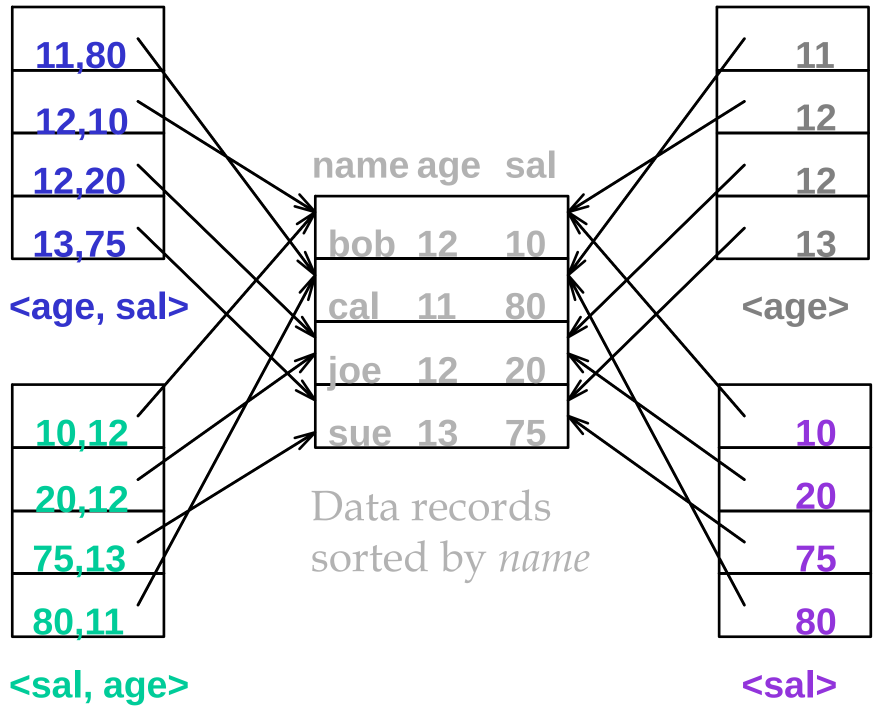


Index Classification (Contd.)

- *Composite Search Keys*: Search on a combination of fields.
  - Equality query: Every field value 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 to support range queries.

Examples of composite key indexes using lexicographic order.



Data records sorted by *name*

Data entries in index sorted by <*sal,age*>

Data entries sorted by <*sal*>

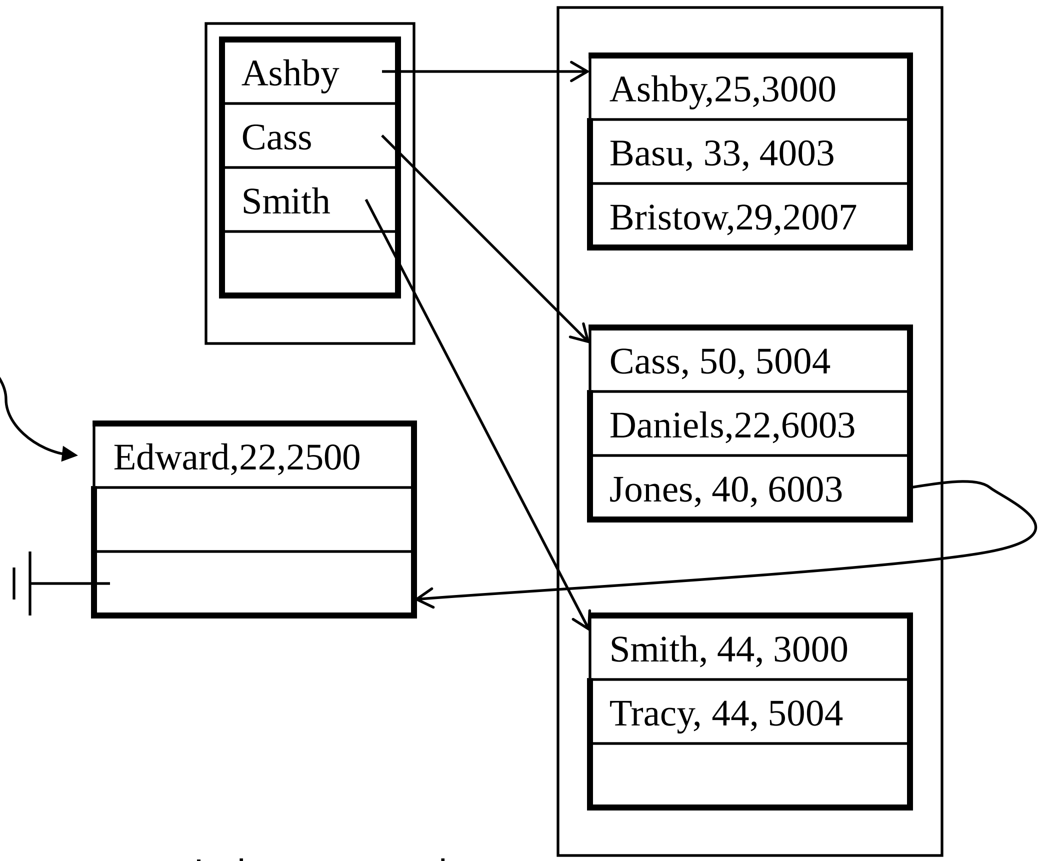
How to build an index (search key value k)

- Unclustered (must be dense):
  - Primary: each data entry k* points to the single record that contains k
  - Secondary:each data entry k* points to all the records that contain k
- Clustered (primary or secondary):
  - Sort both data file and index file on the search key
  - Each data entry k* points to the ***first*** record that contains k
  - Note: overflow pages may be needed for inserts. (Thus, order of data records is 'close to', but not identical to, the sort order.)

Overflow page in Clustered index

Insert record:
(Edward, 22, 2500)

We must use an overflow page

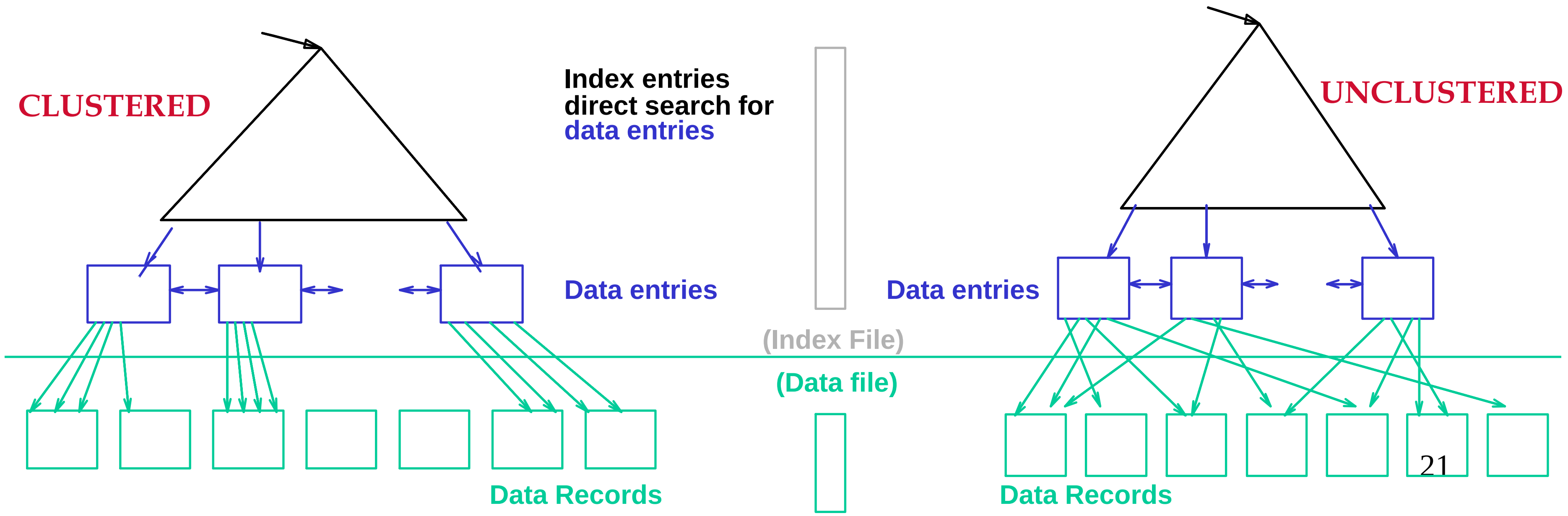


Index on search
Key 'name'
Class: clustered
primary

Sorted file on
name field

More complex index structures

- Sometime the index file itself may be too large to be accessed efficiently, then
  - View the index file as a data file
  - Build an additional index on this data file
  - This idea can be applied repeatedly
  - Solution: tree structured index structure (to be discussed in Chapter 10.)



Choice of Indexes

- Approach: Consider the most important queries in turn. Consider the best plan using the current indexes, and see if a better plan is possible with an additional index. If so, create it.
  - Obviously, this implies that we must understand how a DBMS evaluates queries and creates query evaluation plans!
  - For now, we discuss simple 1-table queries.
- Before creating an index, must also consider the impact on updates in the workload!
  - Trade-off: Indexes can make queries go faster, updates slower. Require disk space, too.

Index Selection Guidelines

- 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; can also help on equality queries if there are many duplicates.
- 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 important queries.
    - For index-only strategies, clustering is not important!
- Try to choose indexes that benefit as many queries as possible. Since only one index can be clustered per relation, choose it based on important queries that would benefit the most from clustering.