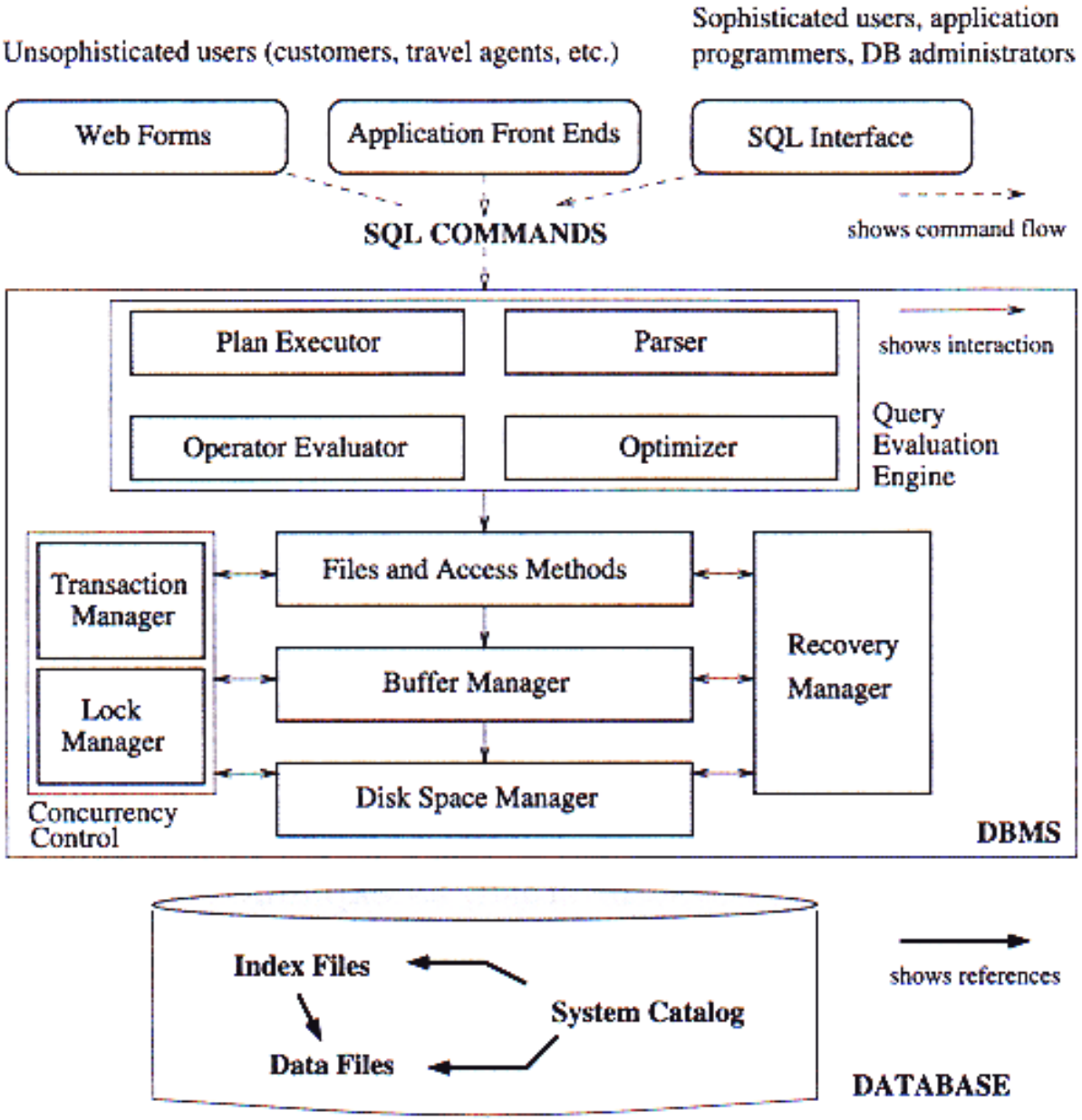
Storage and Indexes

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.

- 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**.
- Typically records in a page have the same size, say 100 bytes.
- Typically records implement relational tuples.

Why Not Store Everything in Main Memory?

- Costs too much.
  RAM is much more expensive than disk.

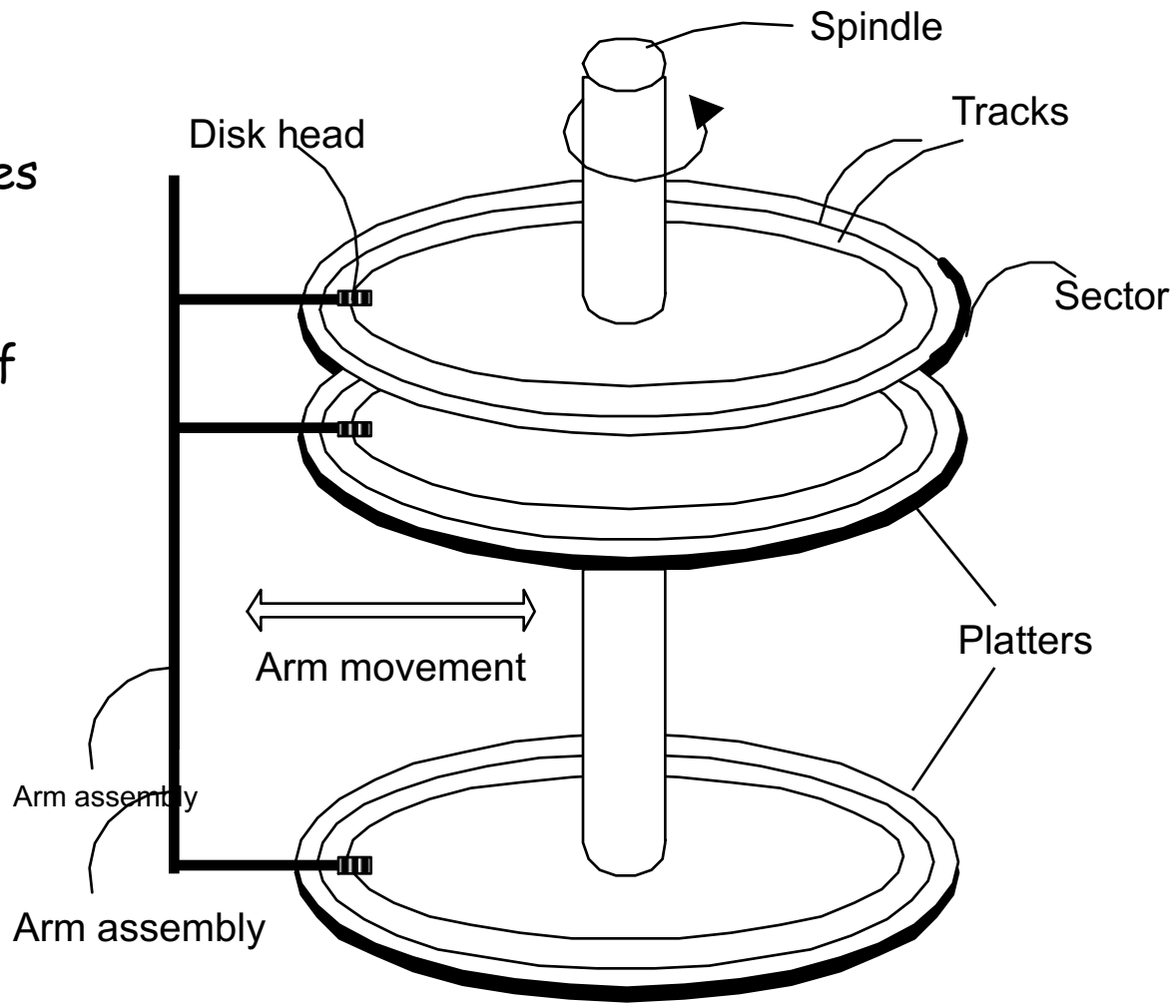
- Main memory is volatile.
  We want data to be saved between runs.

- 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).

Components of a Disk

Only one head reads/writes at any one time.

Block size is a multiple of *sector size* (which is fixed).



Accessing a Disk Page

- Time to access (read/write) a disk block:
  - seek time (moving arms to position disk head on track)
  - rotational delay (waiting for block to rotate under head)
  - transfer time (actually moving data to/from disk surface)

- Seek time and rotational delay dominate.
  - Seek time varies from about 1 to 20msec
  - Rotational delay varies from 0 to 10msec
  - Transfer rate is about 1msec per 4KB page

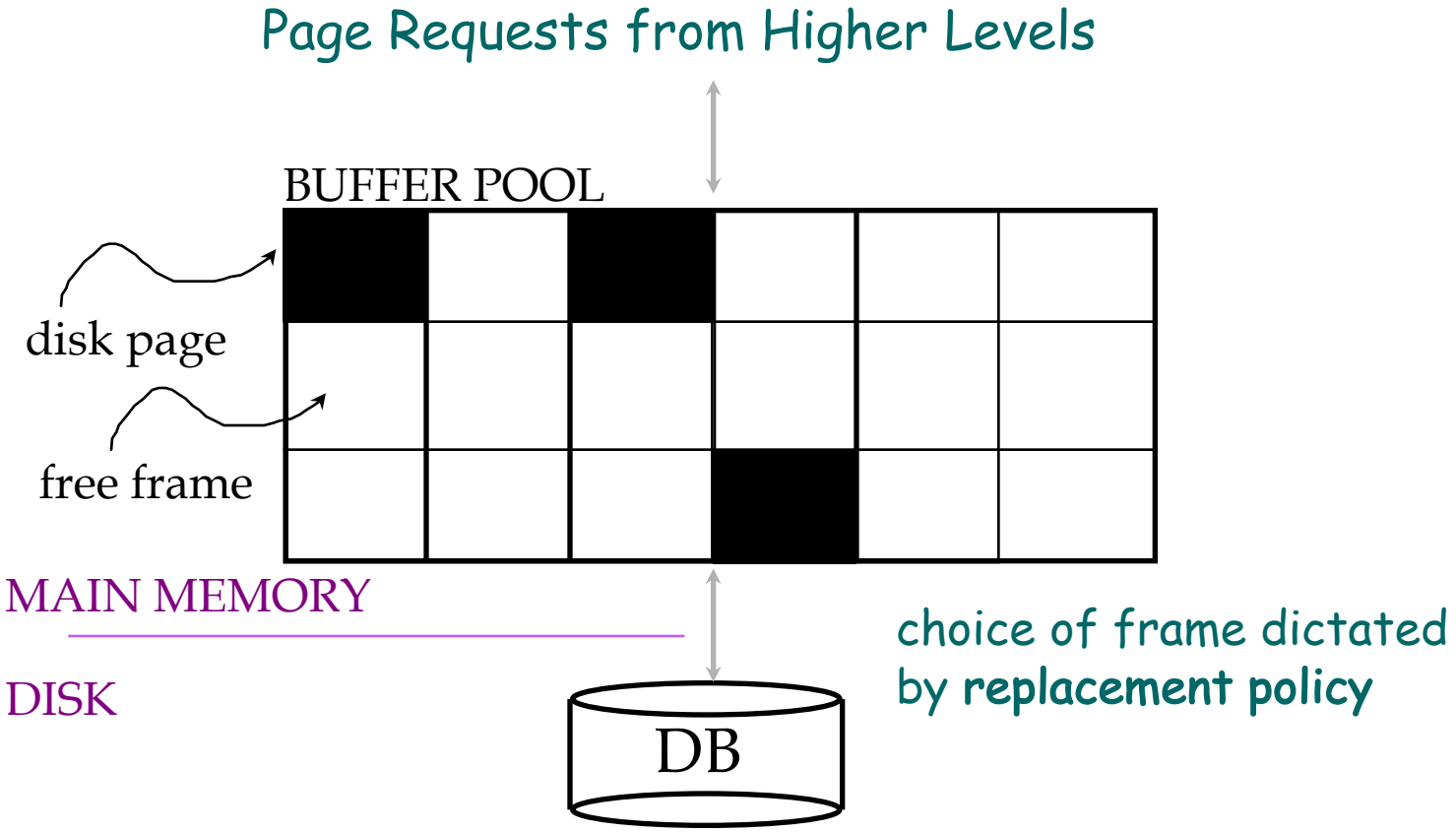
  - Accessing main memory location – 60 nanoseconds
- Key to lower I/O cost: reduce seek/rotation delays!

Arranging Pages on Disk

- Next block concept:
  - blocks on same track, followed by
  - blocks on same cylinder, followed by
  - blocks on adjacent cylinder

- If blocks in a file are arranged sequentially on disk (by `next'), we minimize seek and rotational delay.

Buffer Management in a DBMS



- 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 (updated), write it to disk
  - Read requested page into chosen frame
- Pin the page and return its address to the requestor.

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

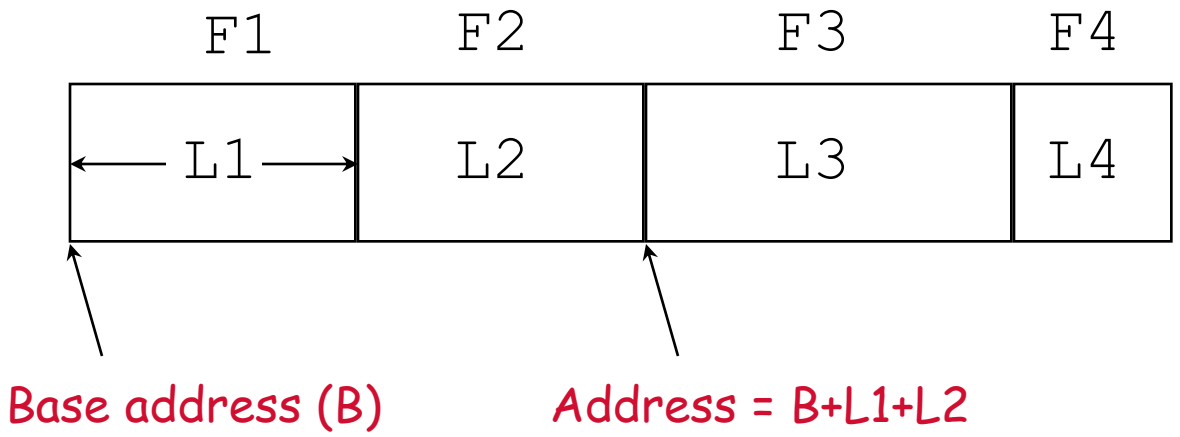
- To release a page, requestor of a page must unpin it, and indicate whether the page has been modified:
  - dirty bit is used for this.

- Page in pool may be requested many times,
  - a pin count is used. A page is a candidate for replacement iff *pin count* = 0.

- Concurrency control & recovery may entail additional I/O when a frame is chosen for replacement.

- Frame is chosen for replacement by a replacement policy: Least-recently-used (LRU), MRU etc.

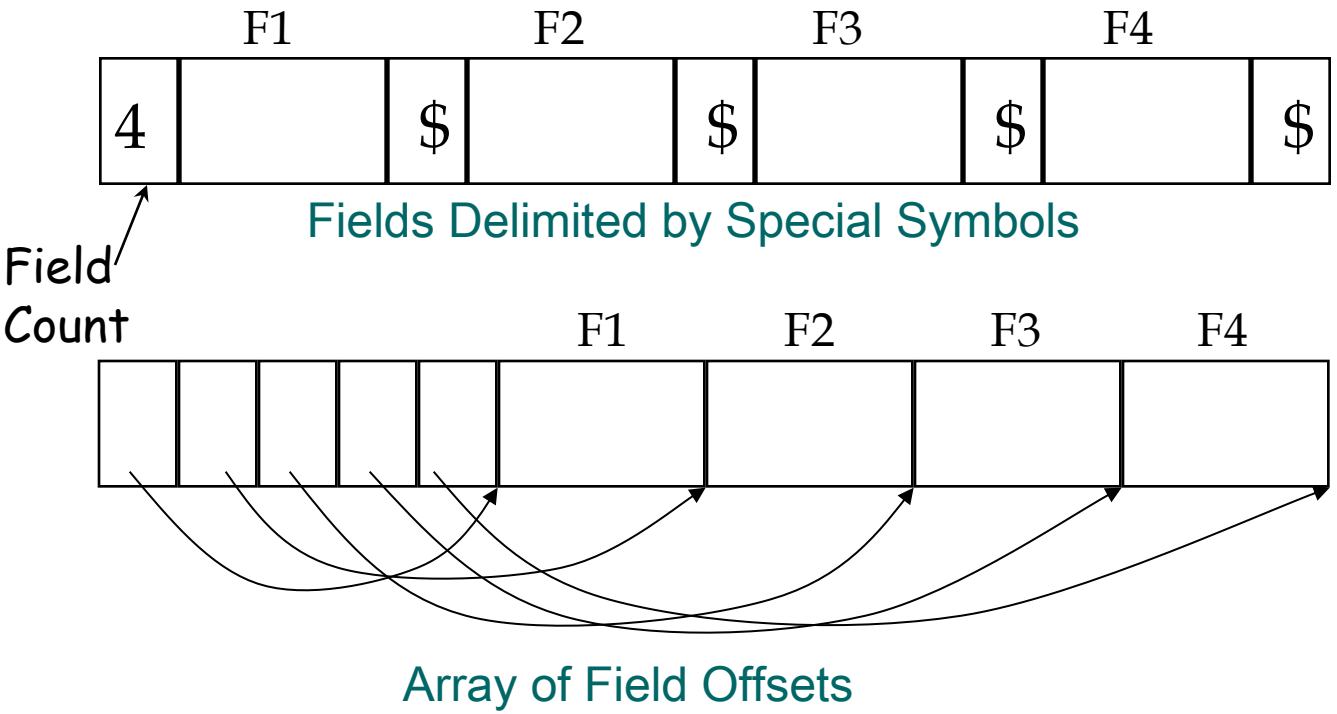
Record Formats: Fixed Length



- Information about field types stored in system catalogs.

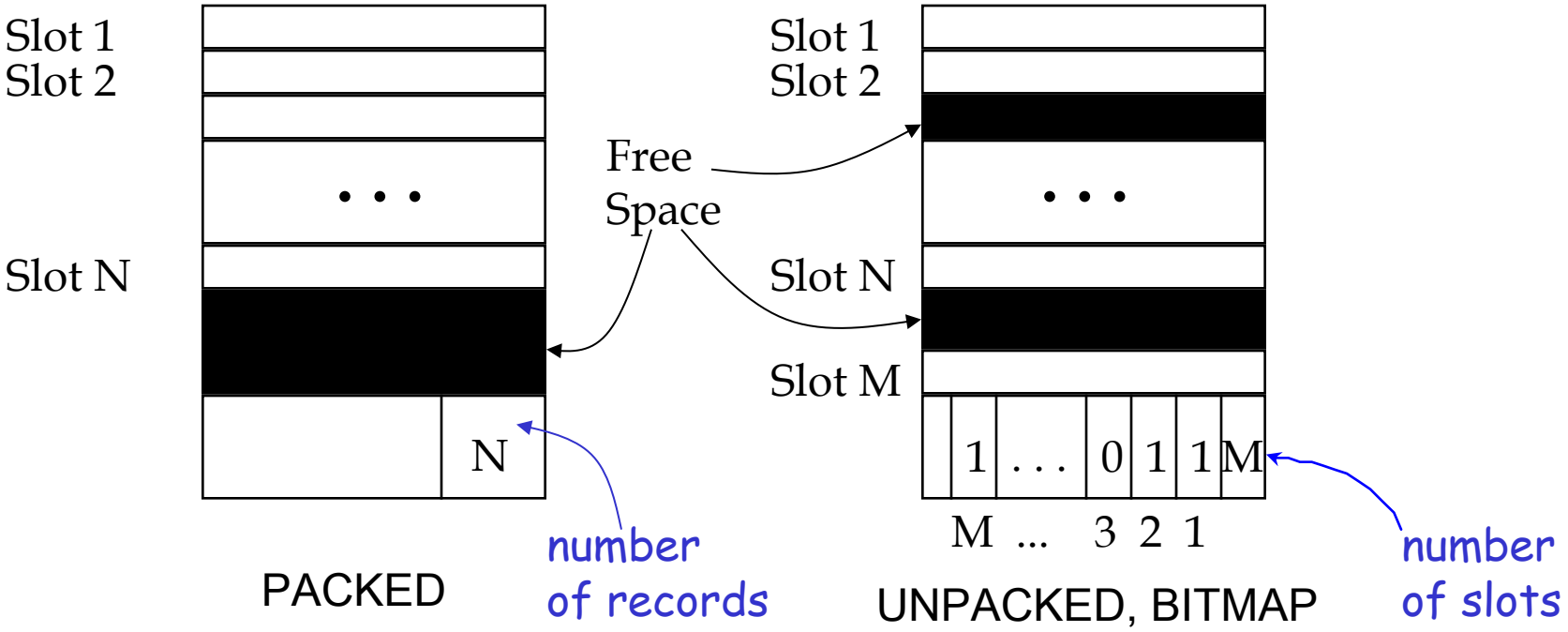
Record Formats: Variable Length

Two alternative formats (# fields is fixed):



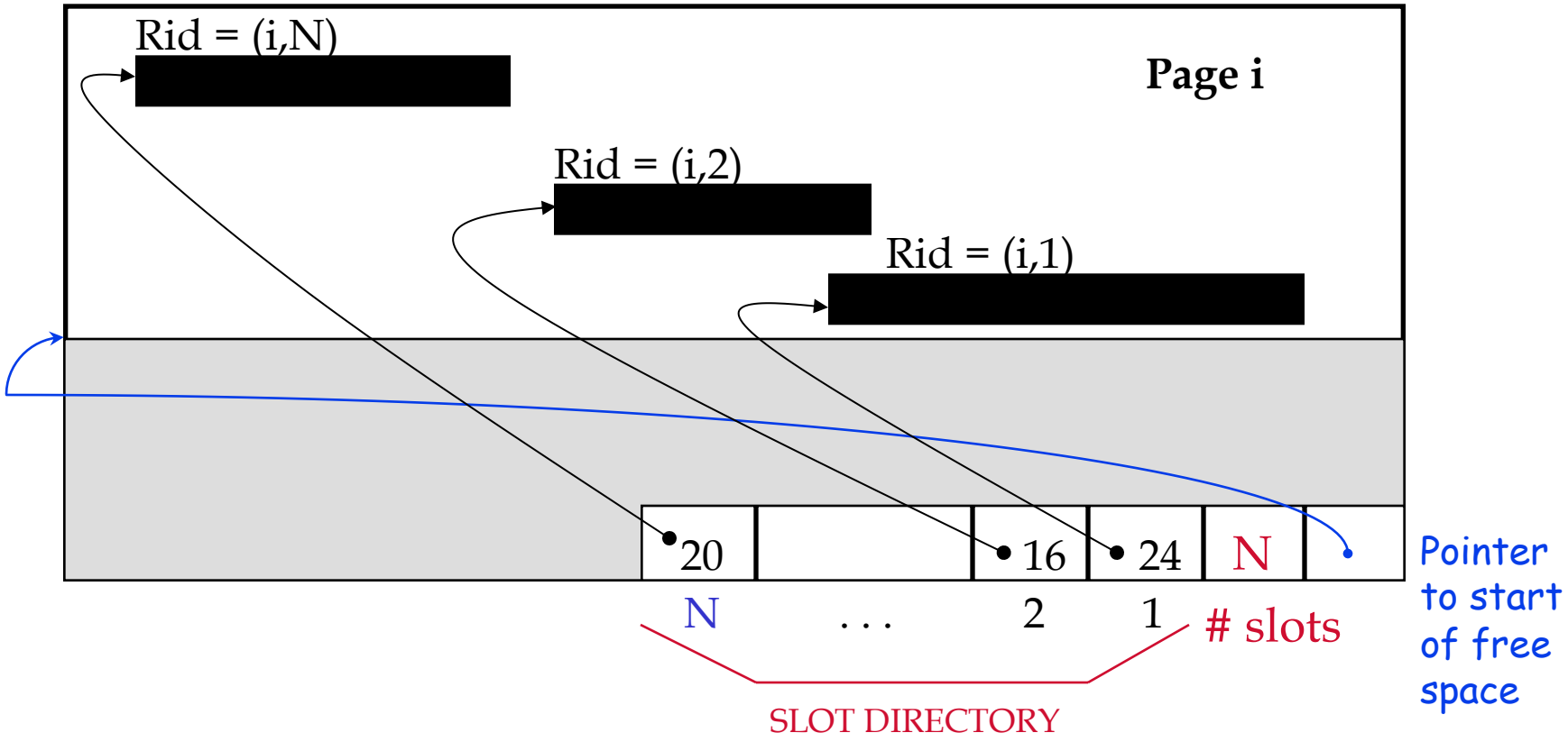
☛ Second format offers direct access to i'th field

Fixed Length Records



☛Record id = <page id, slot #>.

Variable Length Records



☛ Can move records on a page without changing rid; so, attractive for fixed-length records too.

Files of Records

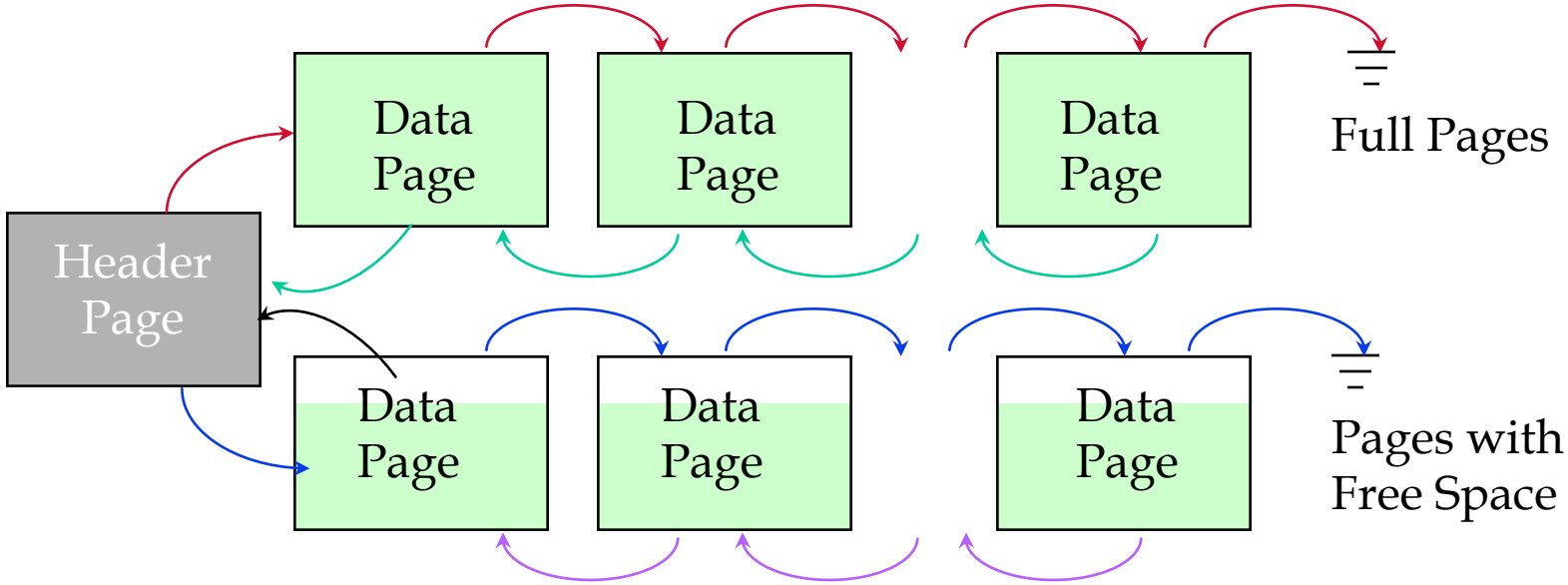
- Page or block is OK when doing I/O, but higher levels of DBMS operate on *records*, and *files of records*.

- 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)

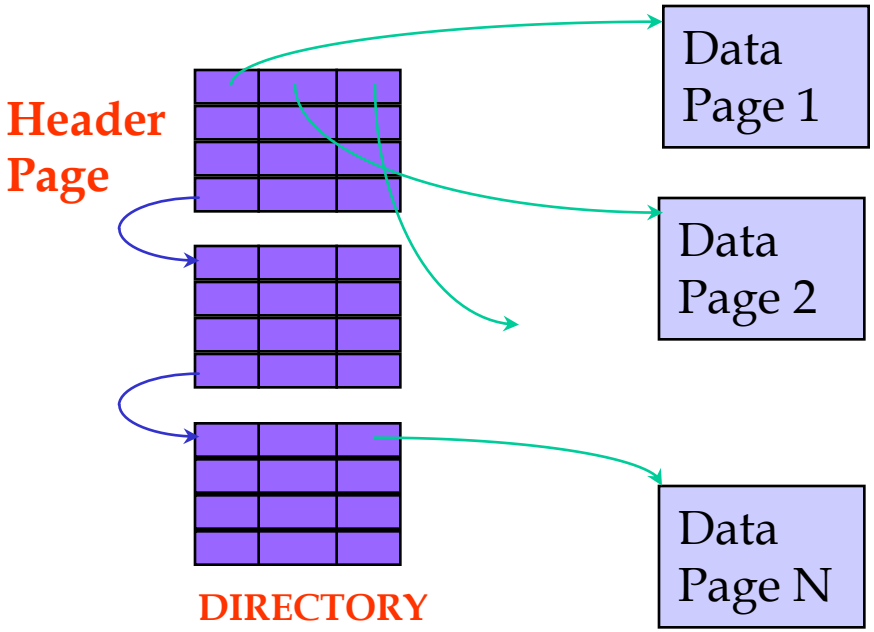
Heap File -(randomly orderd file)-

Implemented as a List



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

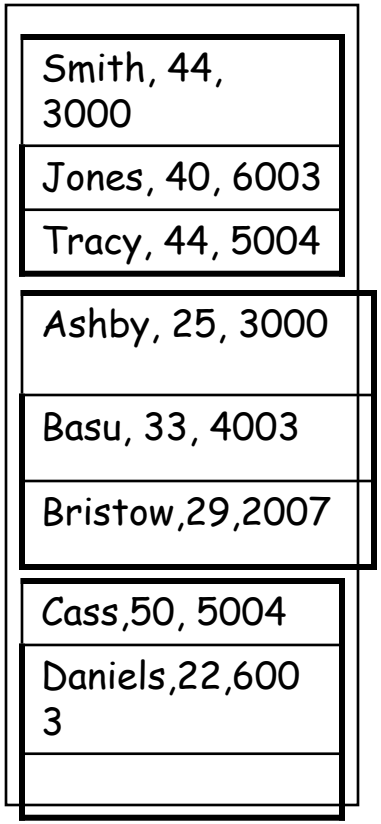
Heap File Using a Page Directory



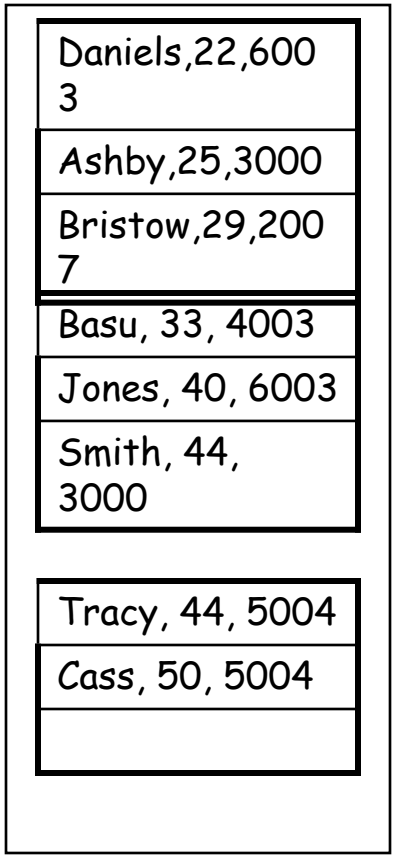
- The entry for a page can include the number of free bytes on the page.
- The directory is a collection of pages
- Can easily search for a page with enough space for a record to be inserted.

Alternative File Organizations

- Heap (random order) files: Suitable when typical access is a file scan retrieving all records.
- Sorted Files: Best if records must be retrieved in some order, or only a `range' of records is needed.
- Indexes: Data structures to organize records via trees or hashing.
  - Like sorted files, they speed up searches for a subset of records, based on values in certain ("search key") fields
  - Updates are much faster than in sorted files.



Heap file (randomly Ordered)



Sorted file on Age field

Indexes

- 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**

- Given k, an index helps to retrieves all data entries **k***

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.

Note: rid is record id

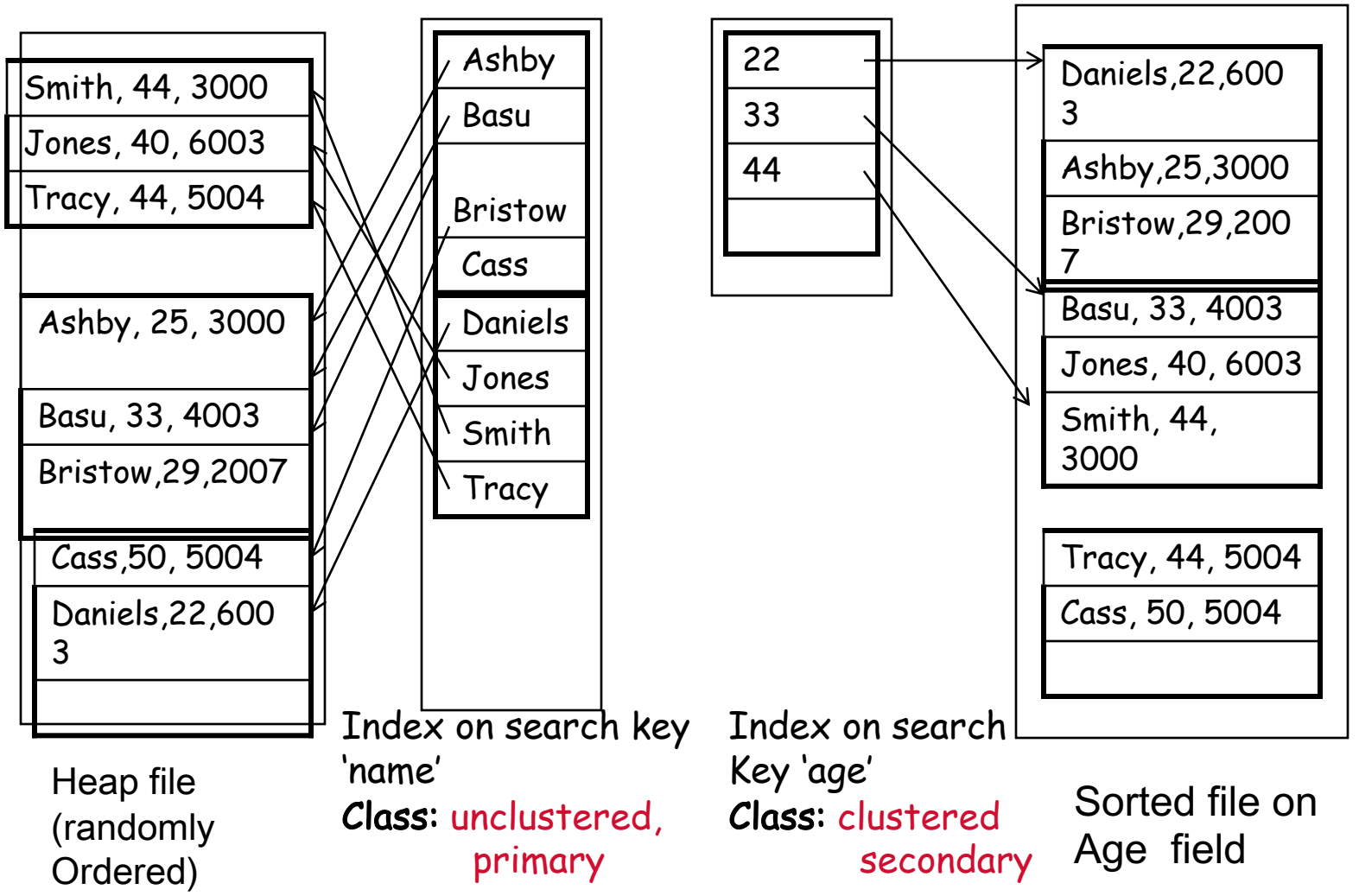
Index Classification

- Primary vs. secondary: If search key contains primary key, then it is a 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 in the index, then it is called a 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 (randomly Ordered)

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

Index on search Key 'age'
Class: clustered secondary

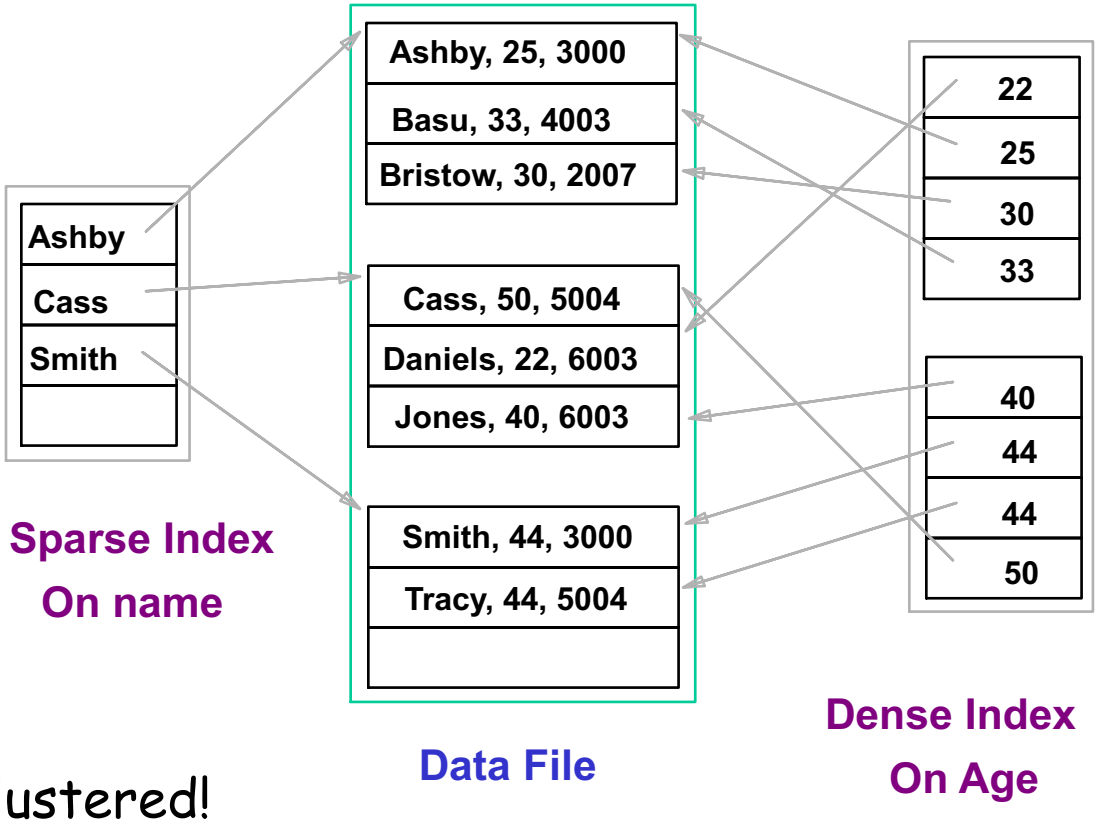
Sorted file on Age field

This index uses alternative 2 <k, rid>

This index uses alternative 2 <k,rid>

Index Classification

- Dense vs. Sparse:
- If there is at least one data entry in the index per search key value, then dense.



Every sparse index is clustered!

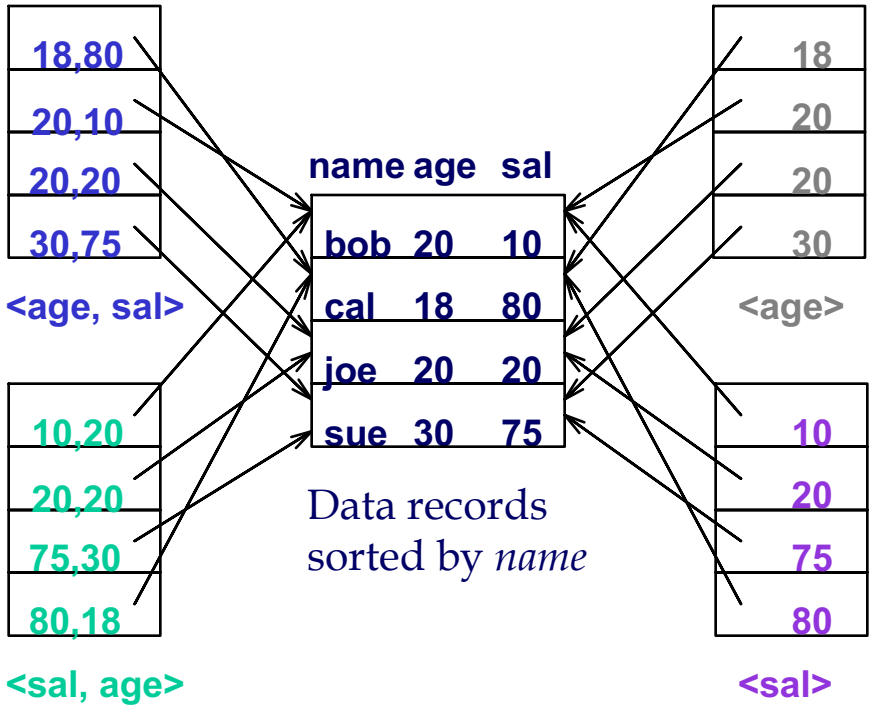
Index Classification

Composite Search Keys:

a combination of fields.

- Equality query: every field value is equal to a constant value.
  E.g. <sal,age> index:
  - age=30 and sal =75
- Range query: Some field value is not a constant.
  E.g.:
  - age =30;
  - age=30 and sal > 10

Data entries in an index is sorted by the search key to support range queries.



Index only – no need to retrieve data records

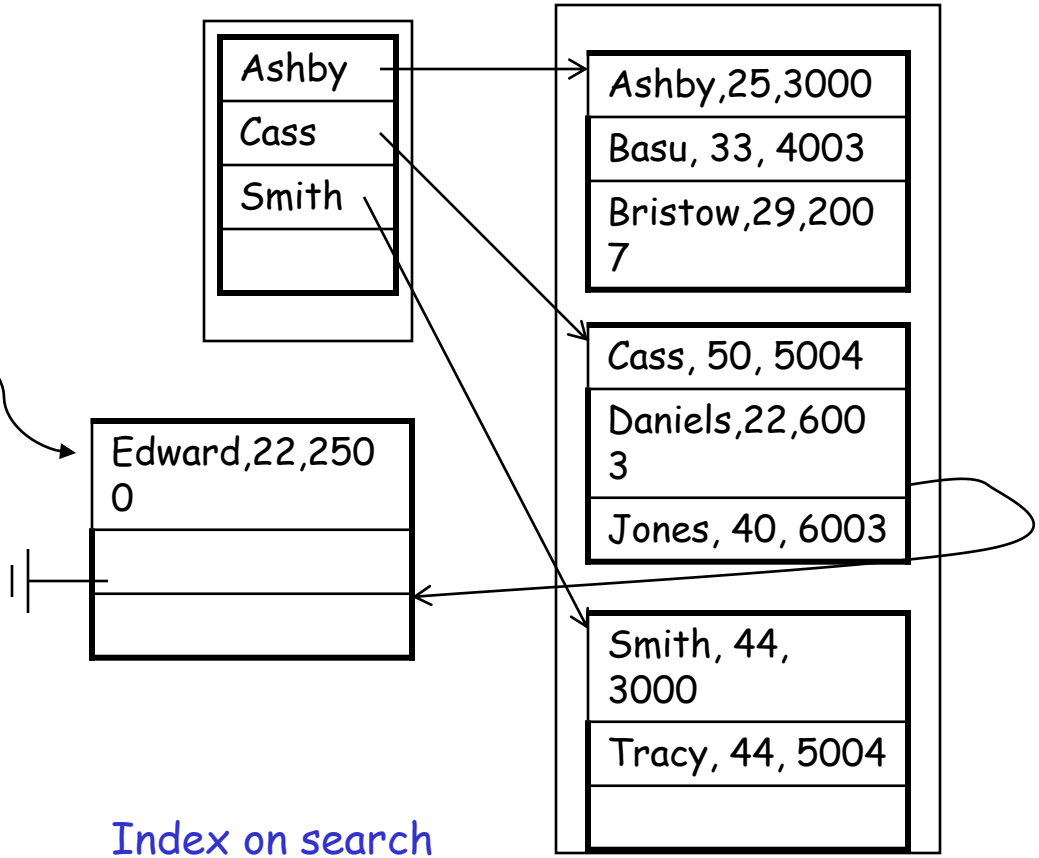
How to build an index (search key value k)

- 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

Overflow page in Clustered index

Insert record:
(Edward, 22, 2500)

We must use an overflow page



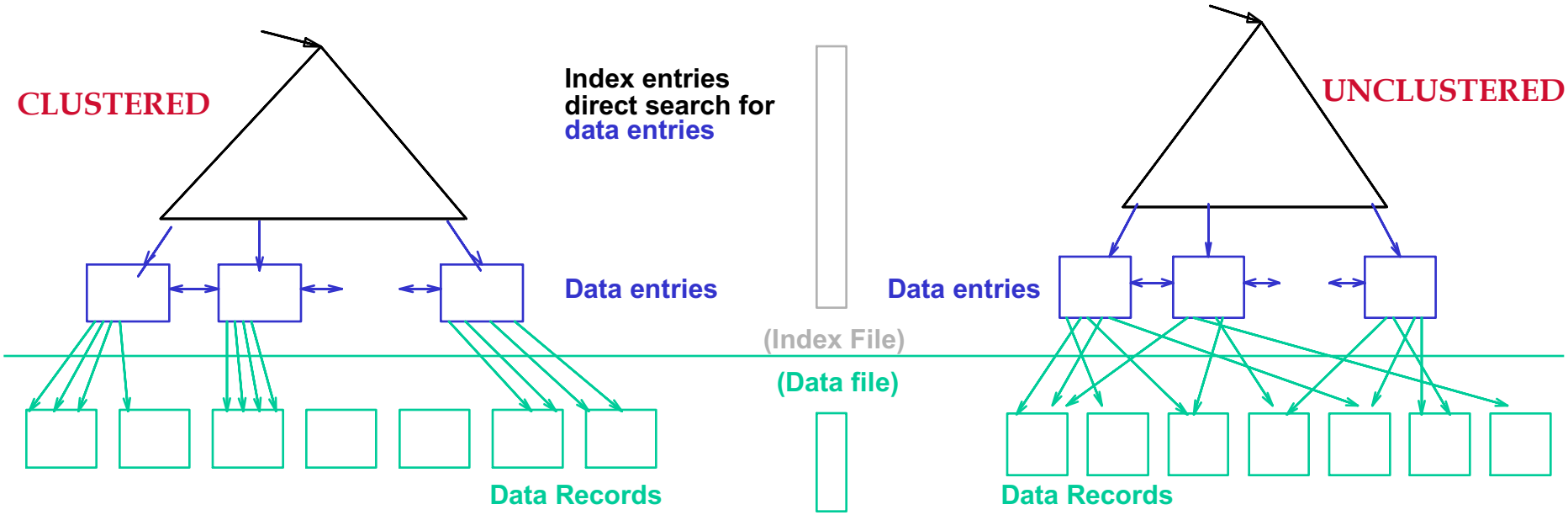
–Note: overflow pages may be needed for inserts. (Thus, order of data records is 'close to', but not identical to, the sort order.)

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
- 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



Choice of Indexes

- One 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.

  - Obviously, to do this we must understand how a DBMS evaluates queries and creates query evaluation plans!

- Before creating an index, must also consider the impact on the updates
  - 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.
  - 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.
  - 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.