4411

Database Management Systems

RAID Levels (con't)

- **Level 3**: Data is striped over *n* disks and an $(n+1)^{th}$ disk is used to store the exclusive or (XOR) of the corresponding bytes on the other *n* disks
  - The $(n+1)^{th}$ disk is called the parity disk
  - Chunks are bytes

Level 3 (con't)

- Redundancy increases reliability
  - Setting a bit on the parity disk to be the XOR of the bits on the other disks makes the corresponding bit on each disk the XOR of the bits on all the other disks, including the parity disk

    1 0 1 0 1   1 (parity disk)
  - If any disk fails, its information can be reconstructed as the XOR of the information on all the other disks

Level 3 (con't)

- Whenever a write is made to any disk, a write must by made to the parity disk

  New_Parity_Bit = Old_Parity_Bit XOR
  (Old_Data_Bit XOR New_Data_Bit)

- Thus each write requires 4 disk accesses
- The parity disk can be a bottleneck since all writes involve a read and a write to the parity disk

RAID Levels (con’t)

- **Level 5:** Data is striped and parity information is stored as in level 3, but
  - The chunks are disk blocks
  - The parity information is itself striped and is stored in turn on each disk
    - Eliminates the bottleneck of the parity disk
  - Level most often recommended for transaction processing applications

Controller Cache

- To further increase the efficiency of RAID systems, a controller cache can be used in memory
  - When reading from the disk, a larger number of disk blocks than have been requested can be read into memory
  - In *write back cache,* the RAID system reports that the write is complete as soon as the data is in the cache (before it is on the disk)
  - If all the blocks in a stripe are to be updated, the new value of the parity block can be computed in the cache and all the writes done in parallel

Data is stored in files

- Data is stored in files.
- Files usually have certain structure in order to facilitate efficient space use and fast retrievals and updates.
- Main file structures are:
  - Unsorted file (heap)
  - Sorted file
  - Indexed file (trees, hash tables).

Heap Files

- Rows appended to end of file as they are inserted
  - Hence the file is unordered
- Deleted rows create gaps in file
  - File must be periodically compacted to recover space

Transcript Stored as a Heap File

666666	MGT123	F1994	4.0
123456	CS305	S1996	4.0
987654	CS305	F1995	2.0

page 0

717171	CS315	S1997	4.0
666666	EE101	S1998	3.0
765432	MAT123	S1996	2.0
515151	EE101	F1995	3.0

page 1

234567	CS305	S1999	4.0
878787	MGT123	S1996	3.0

page 2

Heap File - Performance

- Assume file contains *F* pages
- *Inserting a row*:
  - Scan the file until either find the row (do not insert), or do not find it (and then insert at the end)
  - Avg. $F/2$ page transfers if row already exists
  - $F+1$ page transfers if row does not already exist
- *Deleting a row*:
  - Scan the file until either find the row (and then delete it), or not find it (nothing to be deleted).
  - Avg. $F/2+1$ page transfers if row exists
  - F page transfers if row does not exist

Heap File - Performance

- Query
  - scan the file
  - Organization efficient if query returns all rows and order of access is not important. E.g.,

    ```
    SELECT * FROM Transcript
    ```

  - Organization inefficient if a *few* rows are requested
    - Average *F*/2 pages read to get get a <u>single</u> row. E.g.,

```
SELECT  T.Grade
FROM  Transcript T
WHERE  T.StudId=12345 AND  T.CrsCode ='CS305'
            AND  T.Semester = 'S2000'
```

Heap File - Performance

- Organization inefficient when a subset of rows is requested: *F* pages must be read

```
SELECT  T.Course, T.Grade
FROM  Transcript T                              -- equality search
WHERE  T.StudId = 123456

SELECT  T.StudId, T.CrsCode
FROM  Transcript T                              -- range search
WHERE  T.Grade BETWEEN 2.0 AND 4.0
```

Sorted File

- Rows are sorted based on some attribute(s)
  - Typically use binary search to scan the file.
  - Equality or range query based on that attribute has cost log_2F to retrieve page containing first row.
  - Successive rows are in same (or successive) page(s) and cache hits are likely.
  - By storing all pages on the same track, seek time can be minimized.
- Example – Transcript sorted on *StudId* :

```
SELECT  T.Course, T.Grade
FROM  Transcript T
WHERE  T.StudId = 123456
```

```
SELECT  T.Course, T.Grade
FROM  Transcript T
WHERE T.StudId BETWEEN
          111111 AND 199999
```

Transcript Stored as a Sorted File

111111	MGT123	F1994	4.0
111111	CS305	S1996	4.0
123456	CS305	F1995	2.0

page 0

123456	CS315	S1997	4.0
123456	EE101	S1998	3.0
232323	MAT123	S1996	2.0
234567	EE101	F1995	3.0

page 1

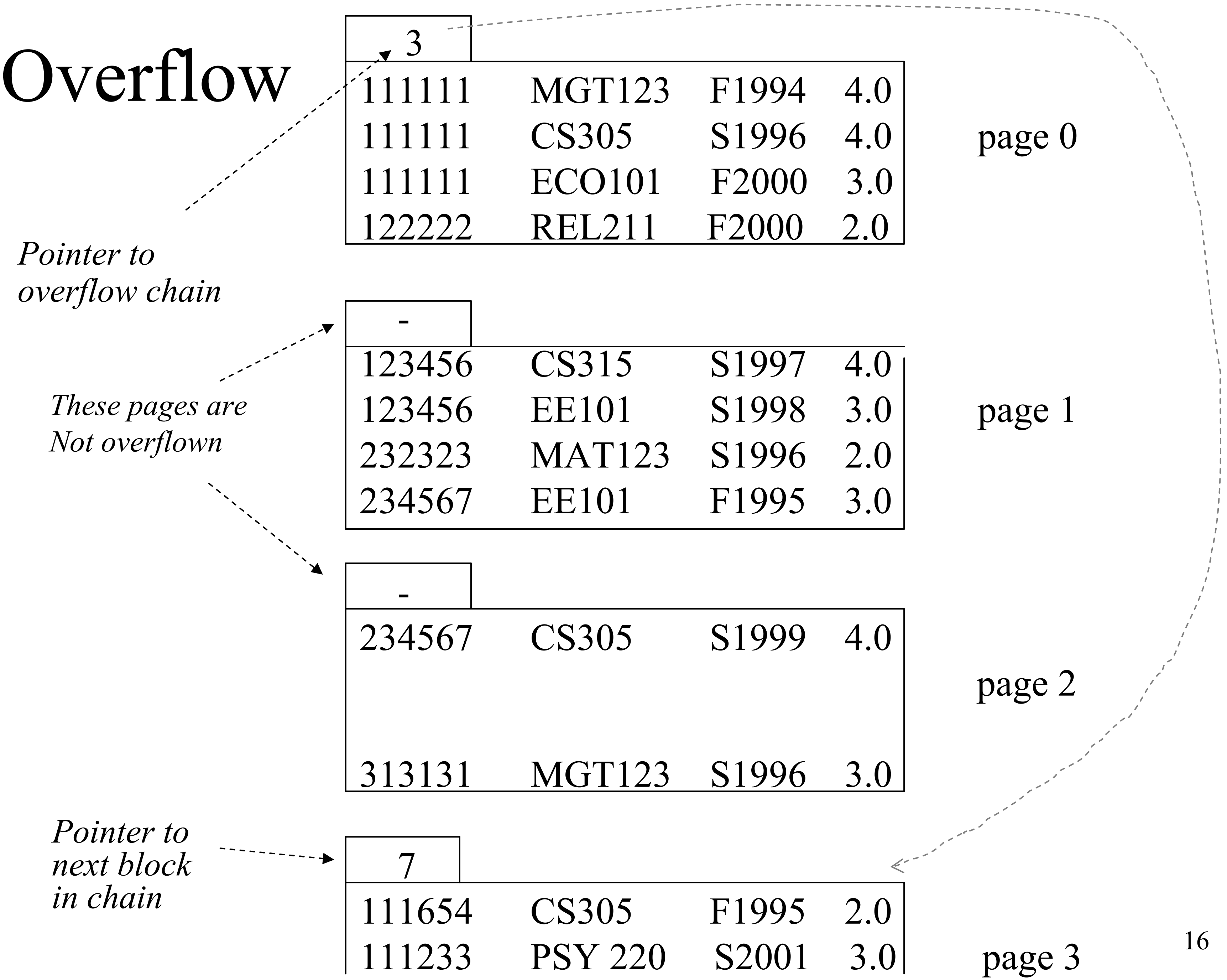
234567	CS305	S1999	4.0
313131	MGT123	S1996	3.0

page 2

Maintaining Sorted Order

- **Problem**: After the correct position for an insert has been determined, inserting the row requires (on average) $F/2$ reads and $F/2$ writes (because shifting is necessary to make space)
- **Partial Solution 1**: Leave empty space in each page (how much space is determined by a *fillfactor)*.
- **Partial Solution 2**: Use *overflow pages* (*chains*).
  - Disadvantages:
    - Successive pages no longer stored contiguously
    - Overflow chain not sorted, hence cost no longer $log_2 F$

Overflow



Pointer to overflow chain
These pages are Not overflown
Pointer to next block in chain
3
-
-
7
page 0
page 1
page 2
page 3

page 0 (pointer: 3)

111111	MGT123	F1994	4.0
111111	CS305	S1996	4.0
111111	ECO101	F2000	3.0
122222	REL211	F2000	2.0

page 1 (pointer: -)

123456	CS315	S1997	4.0
123456	EE101	S1998	3.0
232323	MAT123	S1996	2.0
234567	EE101	F1995	3.0

page 2 (pointer: -)

234567	CS305	S1999	4.0
313131	MGT123	S1996	3.0

page 3 (pointer: 7)

111654	CS305	F1995	2.0
111233	PSY 220	S2001	3.0

Index

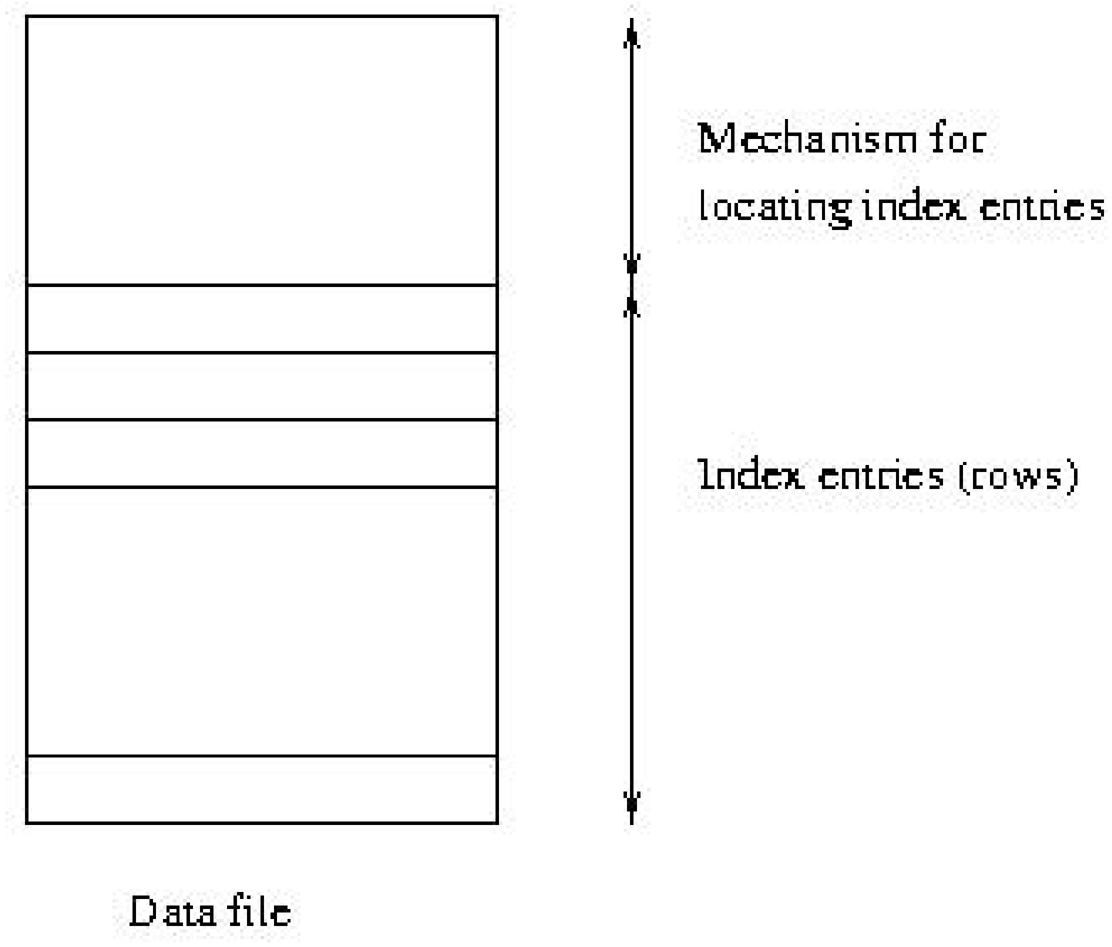
- Mechanism for efficiently locating row(s) without having to scan entire table
- Based on a ***search key***
- <u>Don’t confuse</u> candidate key with search key:
  - Candidate key: *set* of attributes; *guarantees* uniqueness
  - Search key: *sequence* of attributes; *does not guarantee* uniqueness –just used for search
  - For example, for a table STUDENT, the primary key may be *stNumber*, and the search key may be *stGPA*.

Index Structure

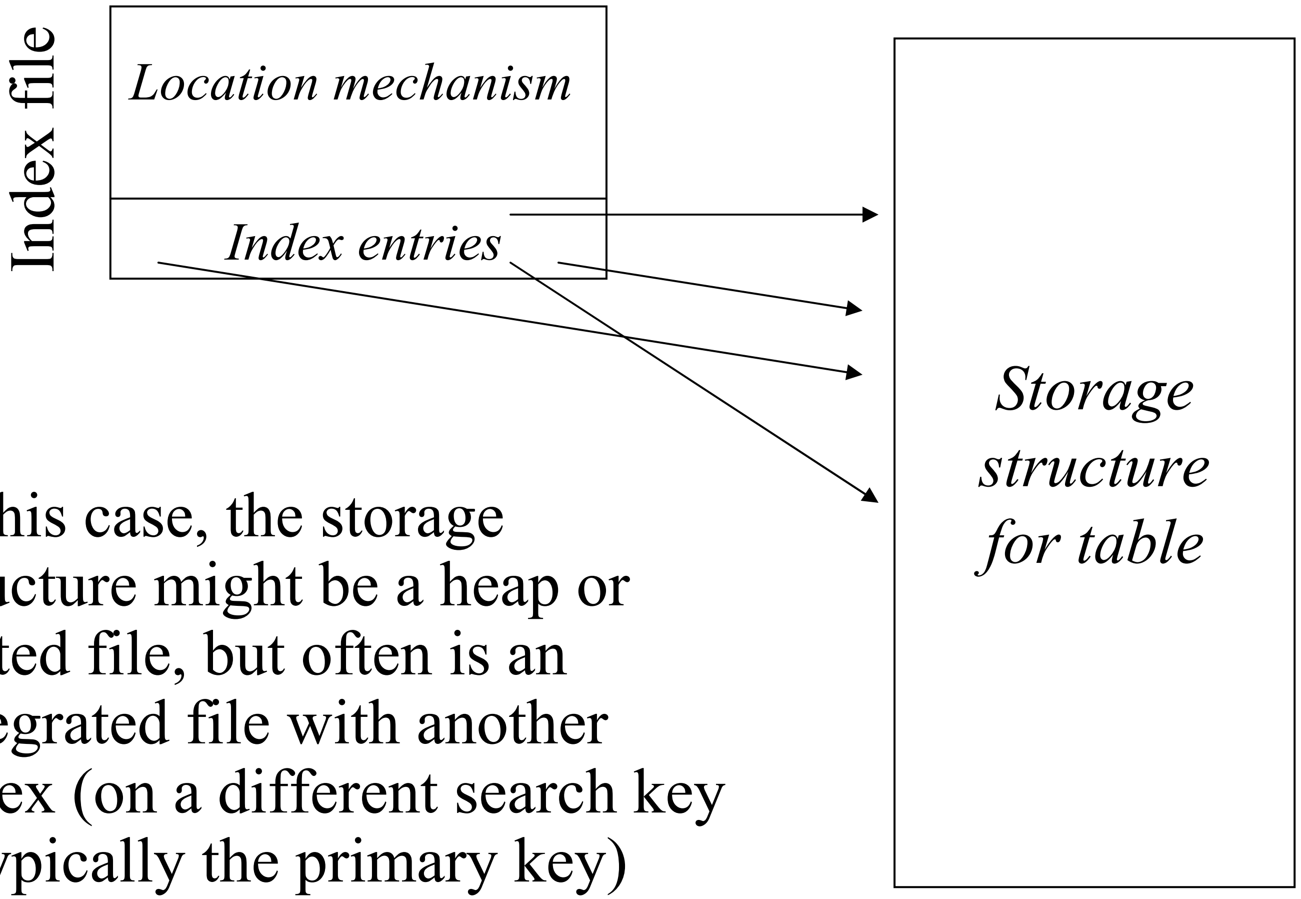
- Contains:
  - *Index entries*
    - Can contain the data tuple itself (index and table are ***integrated*** in this case); or
    - Search key value and a pointer to a row having that value; table stored separately in this case – ***non-integrated*** index
  - *Location mechanism*
    - Algorithm + data structure for locating an index entry with a given search key value
  - Index entries are stored in accordance with the search key value
    - Entries with the same search key value are stored together (hash, B-tree)
    - Entries may be sorted on search key value (B-tree)

Integrated Storage Structure

Contains table and (main) index

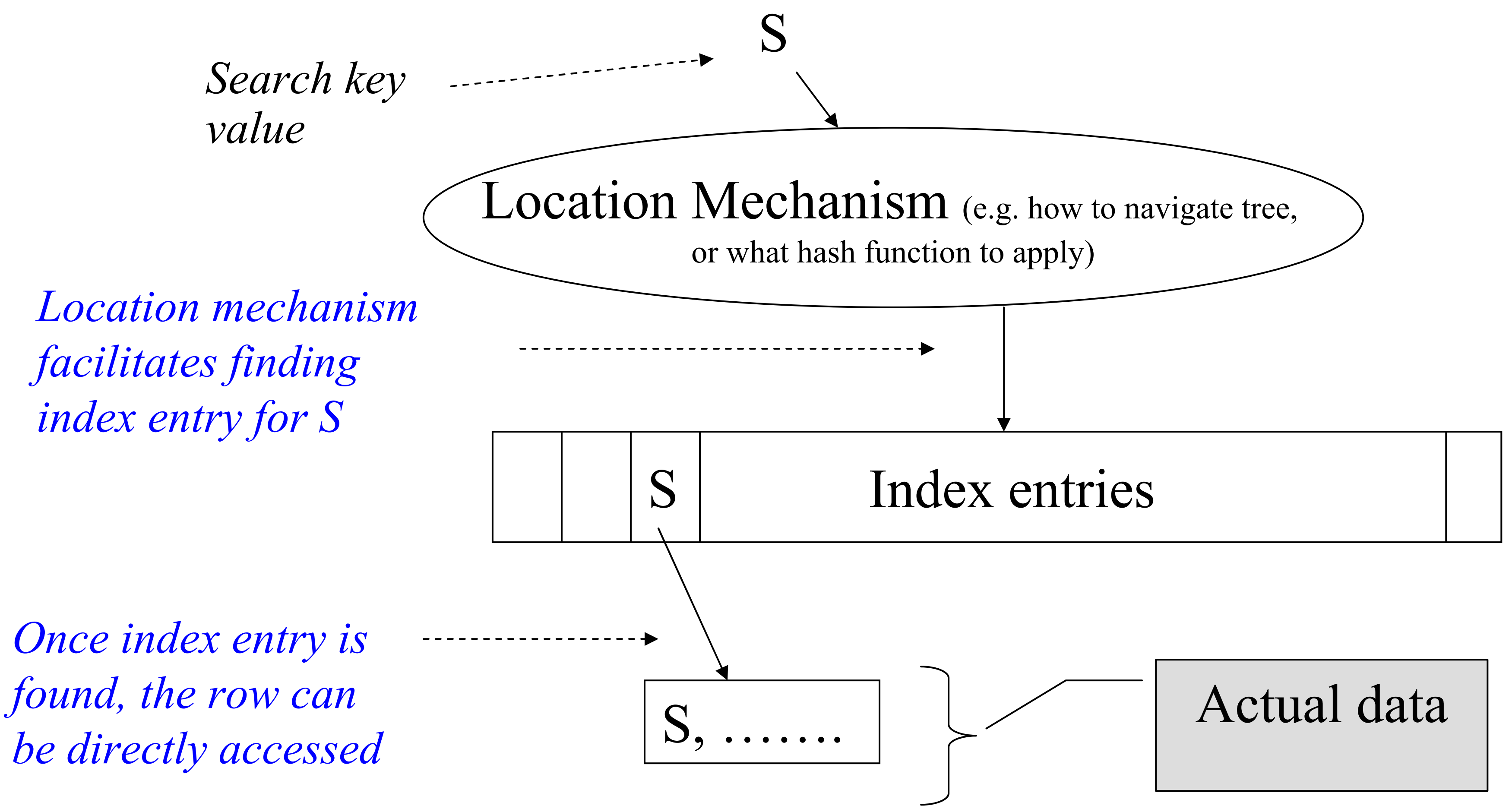


Index File With Separate Storage Structure



In this case, the storage structure might be a heap or sorted file, but often is an integrated file with another index (on a different search key – typically the primary key)

Index Structure


Search key value
S
Location Mechanism (e.g. how to navigate tree, or what hash function to apply)
Location mechanism facilitates finding index entry for S
S
Index entries
Once index entry is found, the row can be directly accessed
S, …….
Actual data

Indices: The Down Side

- Additional I/O to access index pages (except if index is small enough to fit in main memory)
- Index must be updated when table is modified.
- SQL-92 does not provide for creation or deletion of indices
  - Index on primary key generally created automatically
  - Vendor specific statements:
    - CREATE INDEX ind ON Transcript (*CrsCode*)
    - DROP INDEX ind

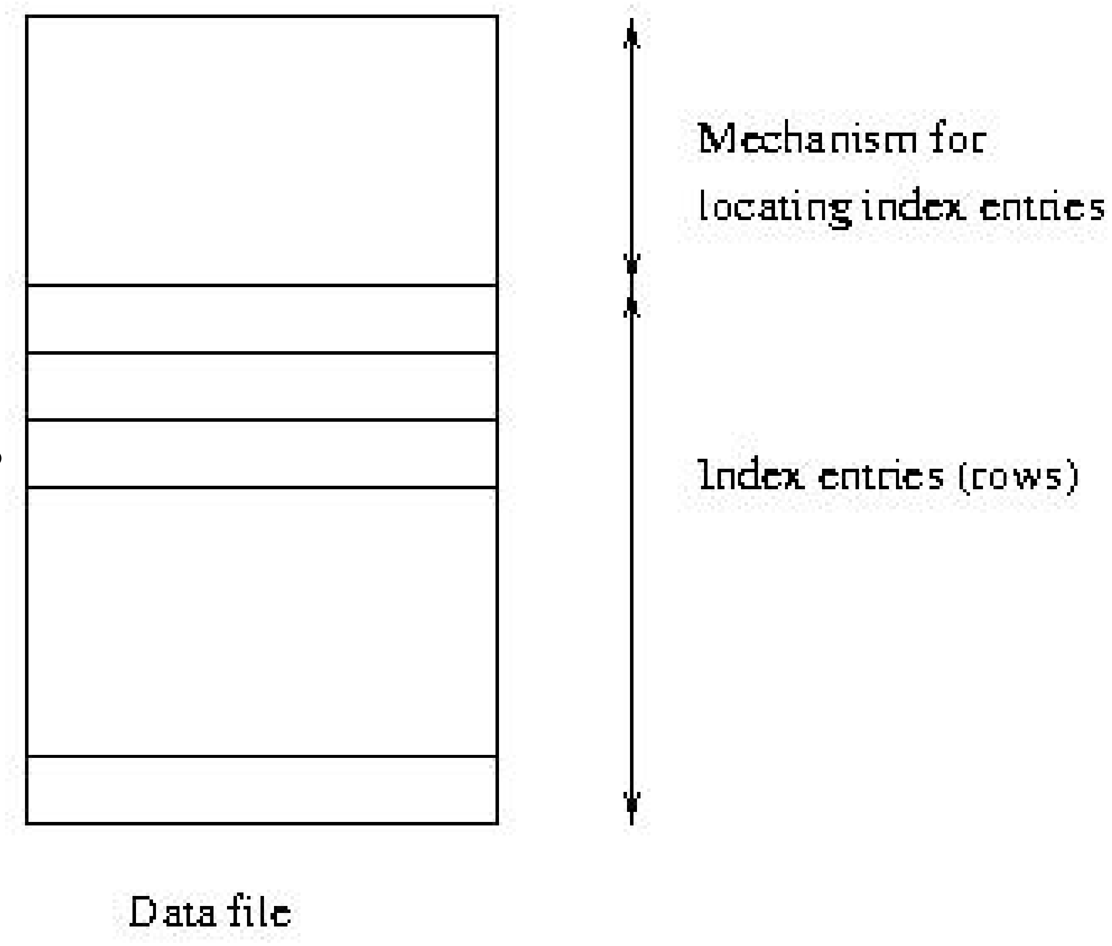
Clustered Index

- *Clustered index*: index entries and actual data rows are ordered in the same way
  - An integrated storage structure is always also clustered (since rows and index entries are the same)
  - The particular index structure (eg, hash, tree) dictates how the rows are organized in the storage structure
    - *There can be at most one clustered index on a table, and it is usually called the* main *index.*
  - CREATE TABLE generally creates an integrated, clustered (main) index on primary key

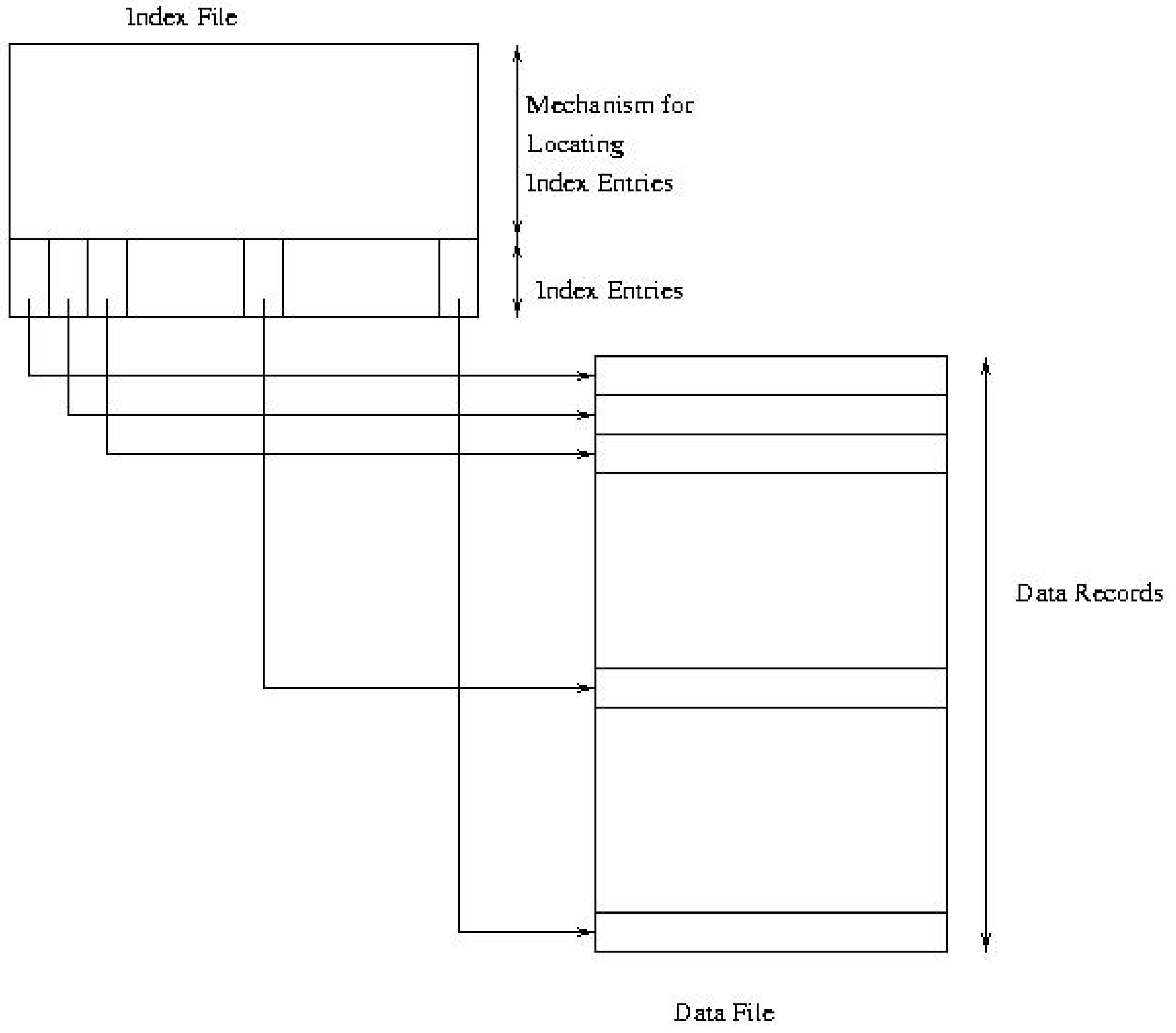
Clustered and integrated Index

Storage structure contains table and (main) index; rows are contained in index entries.

Each entry contains both search key and data.



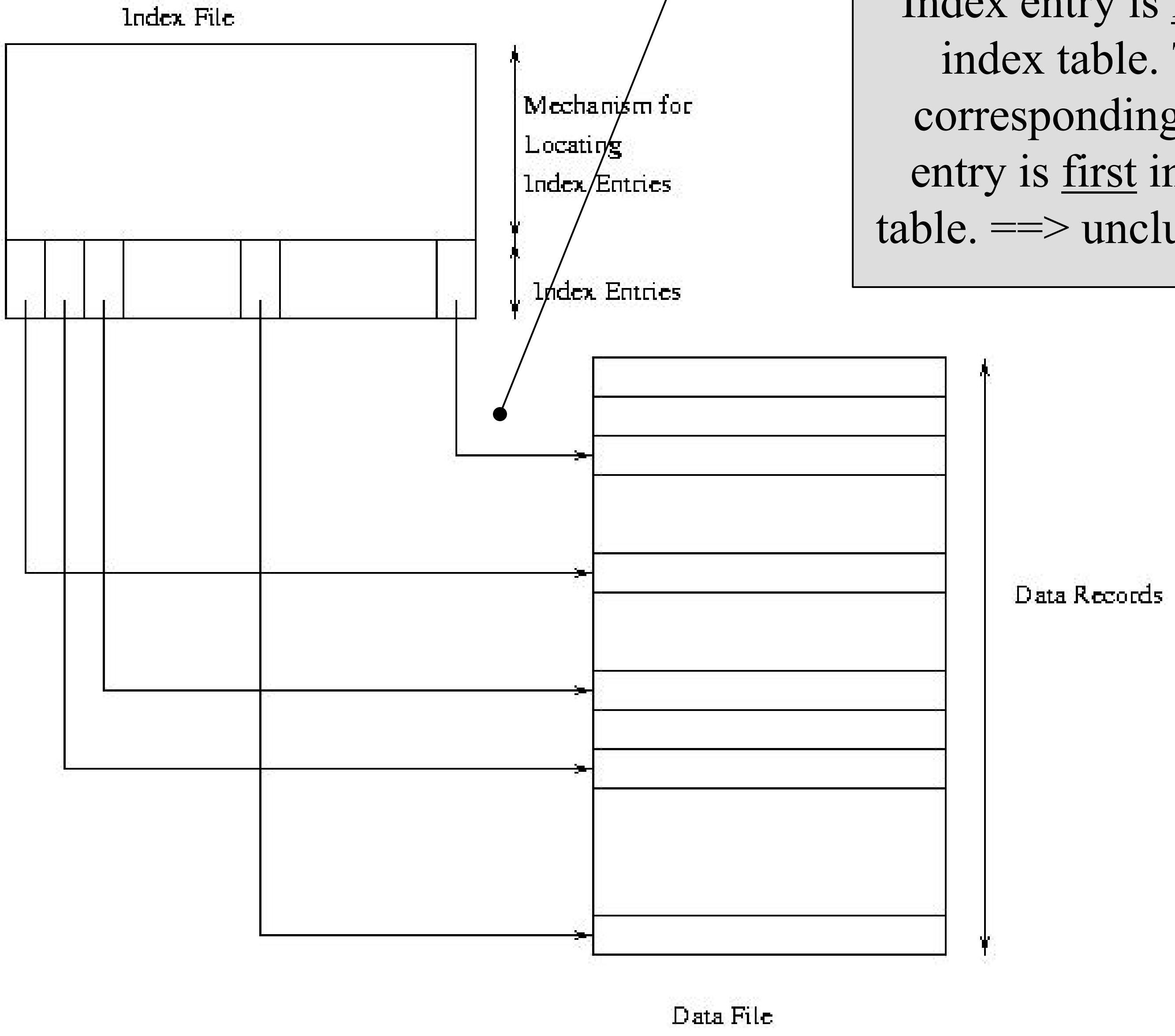
Clustered (and non-integrated) Index



Unclustered Index

- Unclustered (secondary) index: index entries and actual data rows are not ordered in the same way
- A secondary index might be clustered or unclustered with respect to the storage structure it references
  - It is generally unclustered (since the organization of rows in the storage structure depends on main index)
  - There can be many secondary indices on a table
  - Index created by CREATE INDEX is generally an unclustered, secondary index

Unclustered Secondary Index



Index entry is last in index table. The corresponding data entry is first in data table. ==> unclustered.

Clustered Index

- Good for range searches when a range of search key values is requested
  - Use location mechanism to locate index entry at start of range
    - This locates first row.
  - Subsequent rows are stored in successive locations if index is clustered (not so, if unclustered)
  - Minimizes page transfers and maximizes likelihood of cache hits

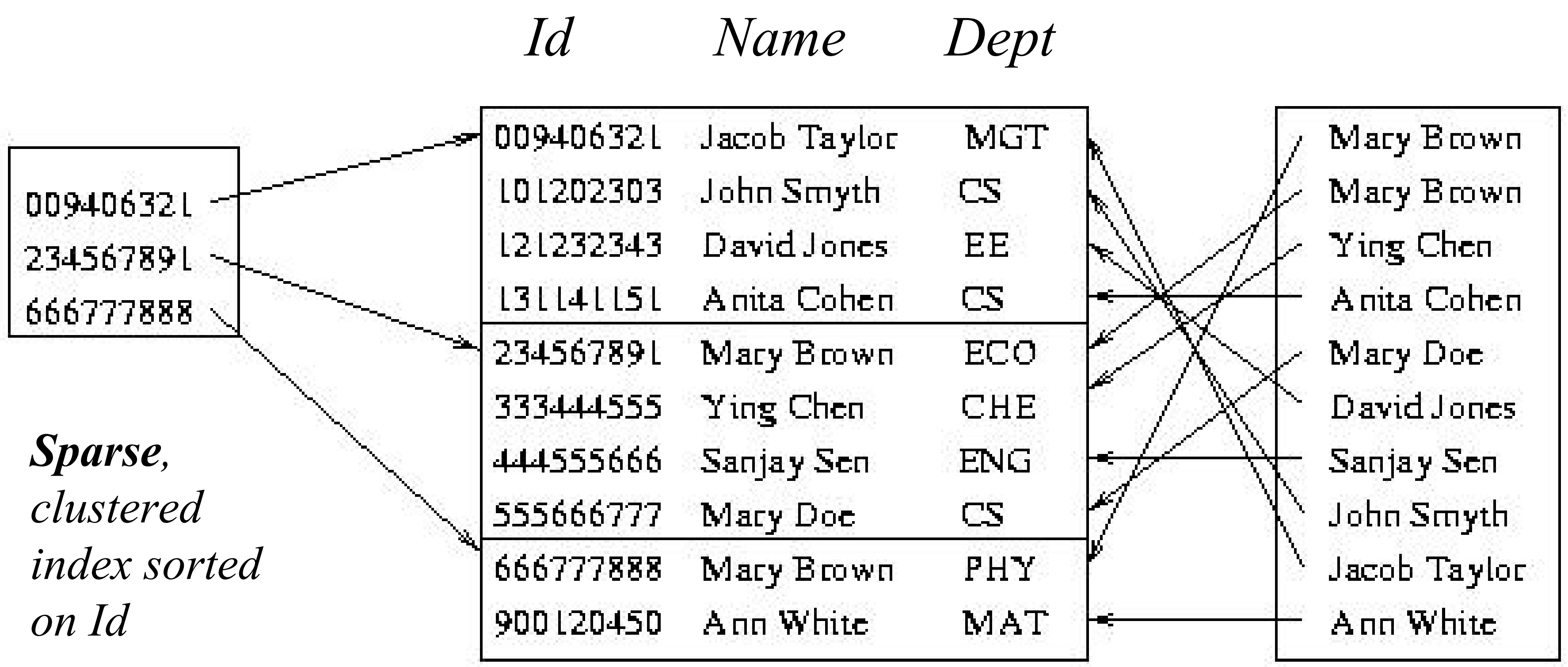
Example – Cost of Range Search

- Assume, data file has 10,000 pages, 100 rows in search range
- Page transfers **for table rows** (assume 20 rows/page):
  - Heap: 10,000 page transfers (**entire file must be scanned**)
  - File sorted on search key: $\log_2 10000 + (5 \text{ or } 6) \approx 19$ (**log cost to locate first entry in index, and then 5 or 6 I/Os to transfer the 100 table rows that follow the first row**).
  - Unclustered index: ≤ 100 (**100 transfers in the worst case, assuming that each row incurs a separate page transfer**).
  - Clustered index: 5 or 6 (**assuming that index file is in memory**)

Sparse vs. Dense Index

- *Dense index*: has index entry for each data record
  - Unclustered index *must* be dense
  - Clustered index need not be dense
  - Sparse index must be clustered.
- *Sparse index*: has index entry for each page of data file

Sparse Vs. Dense Index



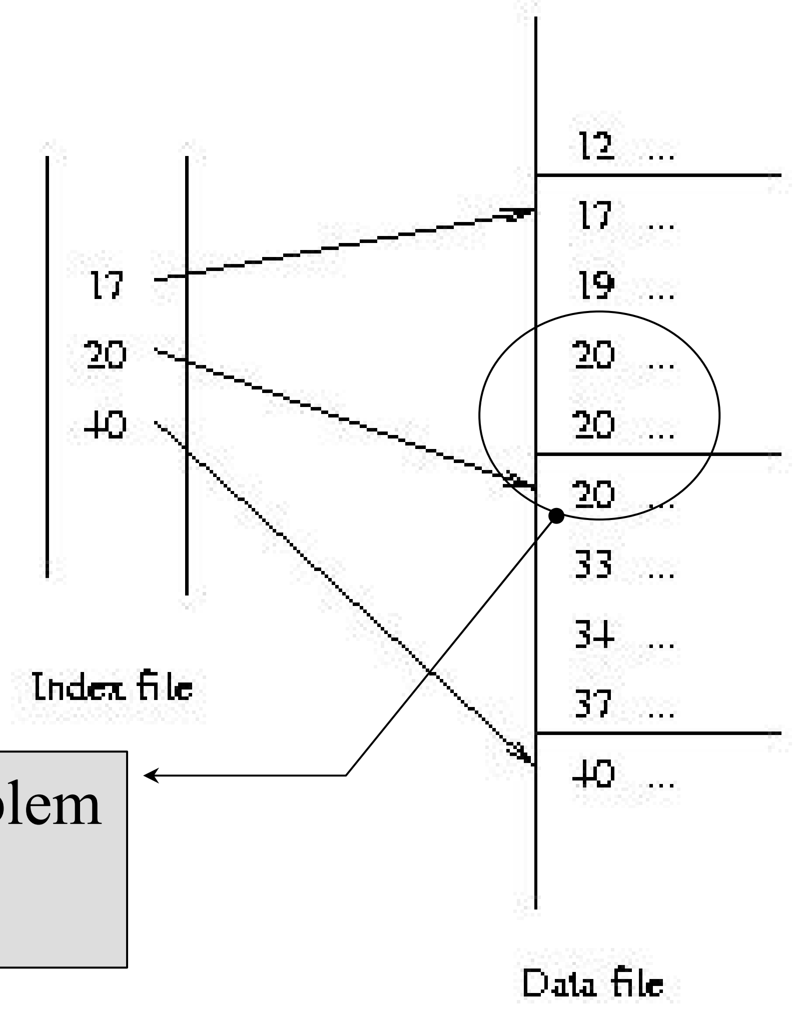
***Sparse**, clustered index sorted on Id*

Data file sorted on Id

***Dense**, unclustered index sorted on Name*

Sparse Index

Search key should be candidate key (and therefore, <u>unique</u>) of data file (else additional measures required)



Searching for '20' causes a problem if search key is not unique.

Tree indices

- Tree indices can be inspired from binary search …
- Binary search: equivalent to a BST (binary search tree) search.
- BSTs have nodes, with each node holding one key (or one record).
- How about if have a 'MST' (Multiway Search Tree) instead of a BST?
- Such trees actually have been thought of before and they are called m-way trees.

M-way trees

- Idea: pack records into blocks, say 7 records per block.
- Possible gain:
  - Instead of one I/O per record, we do one I/O per block (which contains 7 records).
  - … *transparencies* …