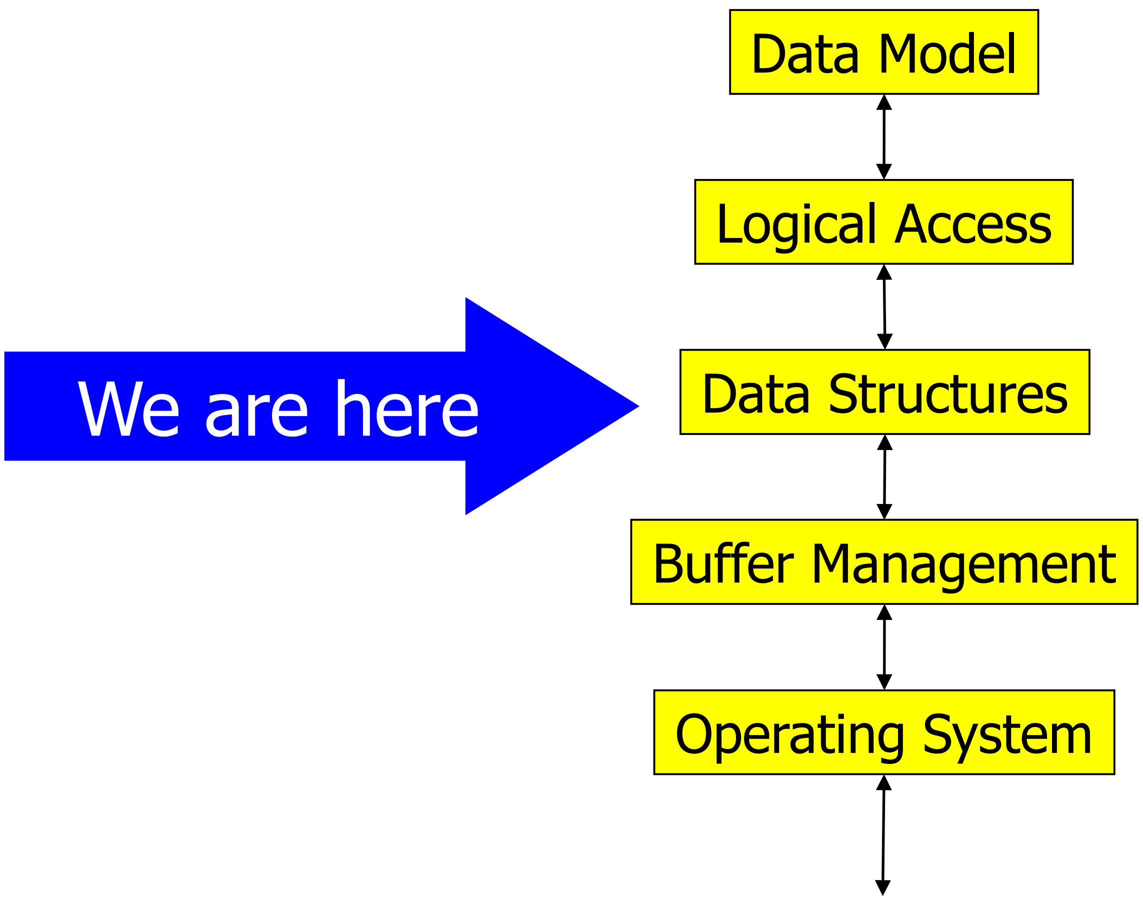
Datenbanksysteme II: File Structures

Ulf Leser

Content of this Lecture

- File structure
  - Heap files
  - Sorted files
  - Index Files
  - Hierarchical Index Files
  - B*-Trees

5 Layer Architecture



Files and Storage Structures

- We have
  - Records are stored in blocks without particular order
    - Makes INSERTs and DELETEs faster
  - Blocks are managed/cached by the buffer manager
  - Access records by TID through cache manager with adr-translation
- DBs mainly search records with certain properties
  - SELECT * FROM COSTUMER
    WHERE Name = "Bond"
  - SELECT * FROM ACCOUNT
    WHERE Account# < 1000

That's why SQL is called "declarative"

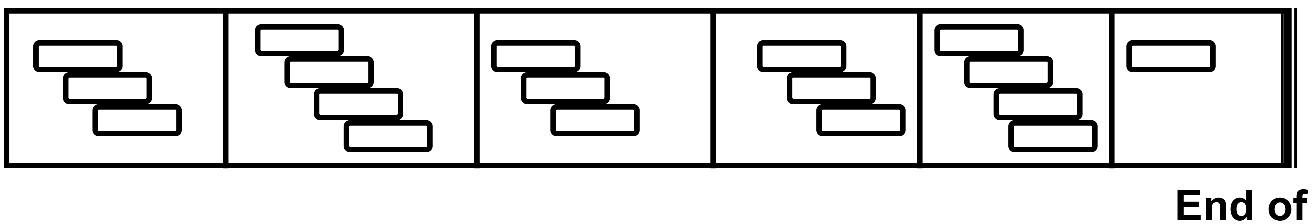
- This is not "access by TID"
- How can we quickly find all records matching a query?
  - Do we always need to scan all records in all blocks?

Preface

- In the following, we talk a lot about searching in lists
  - Unordered, ordered, hashing, trees, …
- What is different from classical (main-mem) algorithms?
  - Real data has duplicates
  - We are on a block device and count IO
  - Relational data is multidimensional
  - Later: We need to combine many search criteria
  - Differences random access – sequential IO

Sequential (Heap) File

- Records are stored sequentially in the order of inserts



- Insert always adds to end of file
- "Holes" occur if records are deleted
  - Can be reused by free-space management – complicated insert
- Minimal number of blocks: $b_{min} = \lceil n / r \rceil$
  - n = number of records, r = number of records per block
- Better to keep some space free for growing records
  - Fraction dep. on expected read/write ratio and record variability

Operations on Heap Files

- In the following: We assume highly selective searches
  - If most records are selected, scanning is hard to beat – see later
- Assume we have $b \geq b_{min}$ blocks
- Search by value of any attribute
  - b/2 IO in case of successful searching a PK (on average)
  - b IO in case of failure or searching non-unique values (always)
- Insert record without duplicate checking
  - Remember: relational model is per-se duplicate–free
  - Simple case: read last block, add, write last block: 2 IO
    - Free space management makes things more complicated
- Insert record with duplicate checking / delete record
  - b/2: for successful search and no insert (on average)
  - b+1: in case of search without success and insert

Content of this Lecture

- File structure
  - Heap files
  - Sorted files
  - Index Files
  - Hierarchical Index Files

Sorted Files

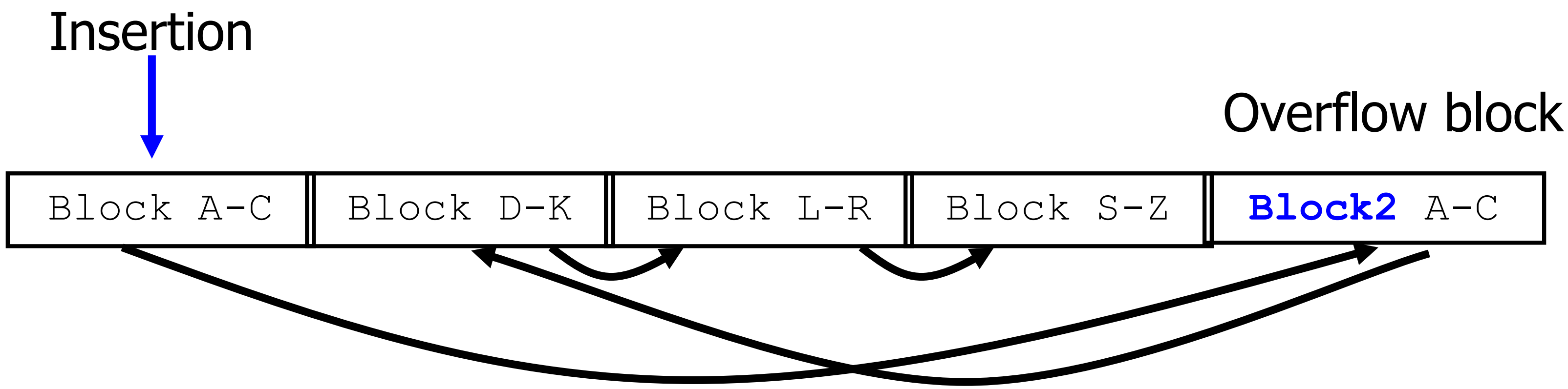
- Sort records in file according to some attribute
  - Fast searching when this attribute is search key
  - More complex management – order must be preserved
  - Not helpful when searching for other attributes
- Operations and associated costs
  - Search (using binsearch on blocks)
    - log(b) IO; searching in block is free (as always)
      - But: That's mostly random-access IO
  - Change / delete records based on value
    - First search in log(b)
    - Write changes / mark space as free
  - Insert record
    - First search correct position in log(b)
    - Then do what?

Inserting in a Sorted File

- General: Reserve free space in every new blocks
  - Don't fill blocks to 100% when allocated first time
  - Chances increase that later insertions can be handled in the block
- Option 1: Use space available in block
  - 1 IO for writing
- Option 2: Move records through blocks to free space
  - Enormously expensive – read/write entire file
- Option 3: Check neighbors
  - See X blocks down and X blocks up in the file
  - When space is found, in-between records need to be moved
  - Cost: depends on how far we need/want to look
    - +4 IO if X=1 (two more reads, two more writes)
  - If no place found: Use option 2 or 4

Overflow Blocks

- Option 4: Generate overflow blocks
  - Create a new “overflow” block and insert record
  - Requires that blocks are connected by pointers
    - Sorted table scan possible only if blocks are chained in disc order
    - But: Overflow blocks will not be in disc order
  - When block is added at end of file
    - Sequential table scan still possible, but not in order of attribute
  - In heavy RW tables, block order will be completely destroyed



Properties Sorted Files

- Additional cost for keeping order
  - INSERT requires log(b) search first
  - Overflow blocks create more random-access IO
- We can sort by only one search key
  - Searching on other attributes requires linear scans
    - With more random-access
  - Many ideas: See multi-dimensional indexes
- But: Search time grows only logarithmically with b
  - For 10.000.000 blocks, we need ~23 IO
  - But all random access
- Can we do better?

Idea 1: Interpolated Search: Build Histograms

- Partition key value range into buckets
- Count number of keys in each bucket
- Searching: Start at estimated position of search key
  - Example: Search "Immel", [A-C]=7500, [D-F]=6200, [G-I]=3300
  - Estimated position: 7500+6200+(3300/3)*2 + …
  - Continue with local search (e.g. exponential) at estimated position
- Advantages
  - Very little IO if data is uniformly distributed – exact estimates
  - Small space consumption when few buckets are used
    - But: the more buckets (higher granularity), the better the estimates
- Disadvantages (see later for ideas)
  - Histograms (statistics) need to be maintained
  - Choosing optimal bucket number and range is difficult

Content of this Lecture

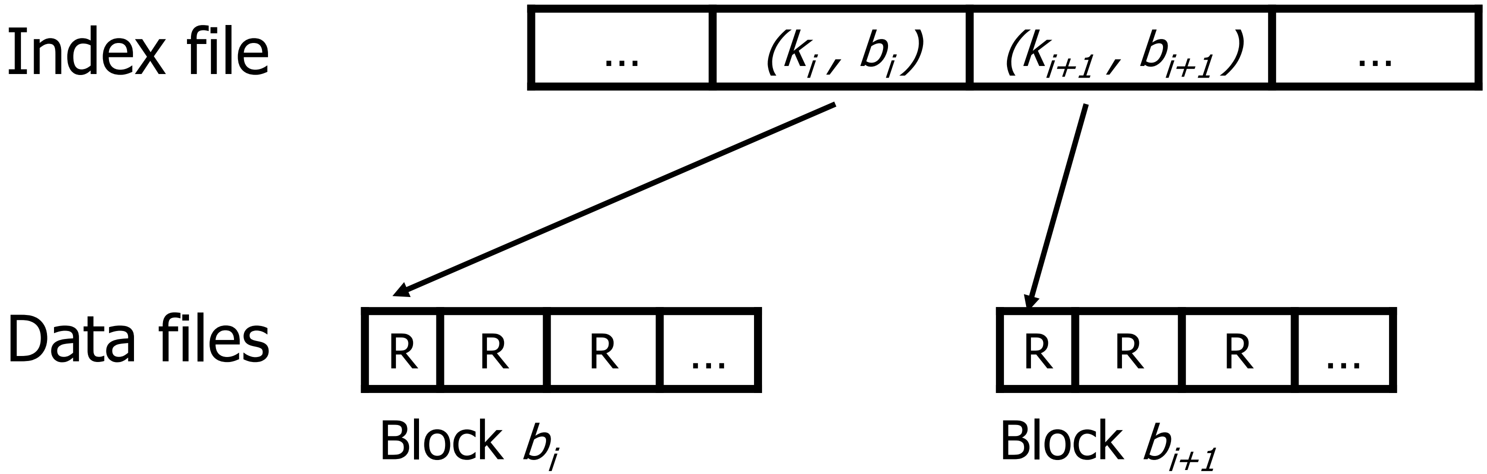
- File structure
  - Heap files
  - Sorted files
  - Index Files
  - Hierarchical Index Files

Idea 2: Decrease b

- Keep only essential info in less blocks
- Use additional file (index) storing only keys and TIDs
- Searching: (Bin-)search index, then access data by TID
- Advantages
  - Data file need not be sorted any more
    - Faster inserts in data file, but additional cost for updating index
      - Integer keys: Fixed-length index entries; strings: Use fixed-length prefix
  - Faster search due to smaller records and less blocks: $b_{index} < b_{records}$
  - Several indexes can be build for different attributes
    - More flexibility, more update cost
- Disadvantages
  - More files to manage, lock, recover, …
  - No more fast sorted scans of entire table (e.g. for merge-join)

Further Decrease b: Index Sequential Files

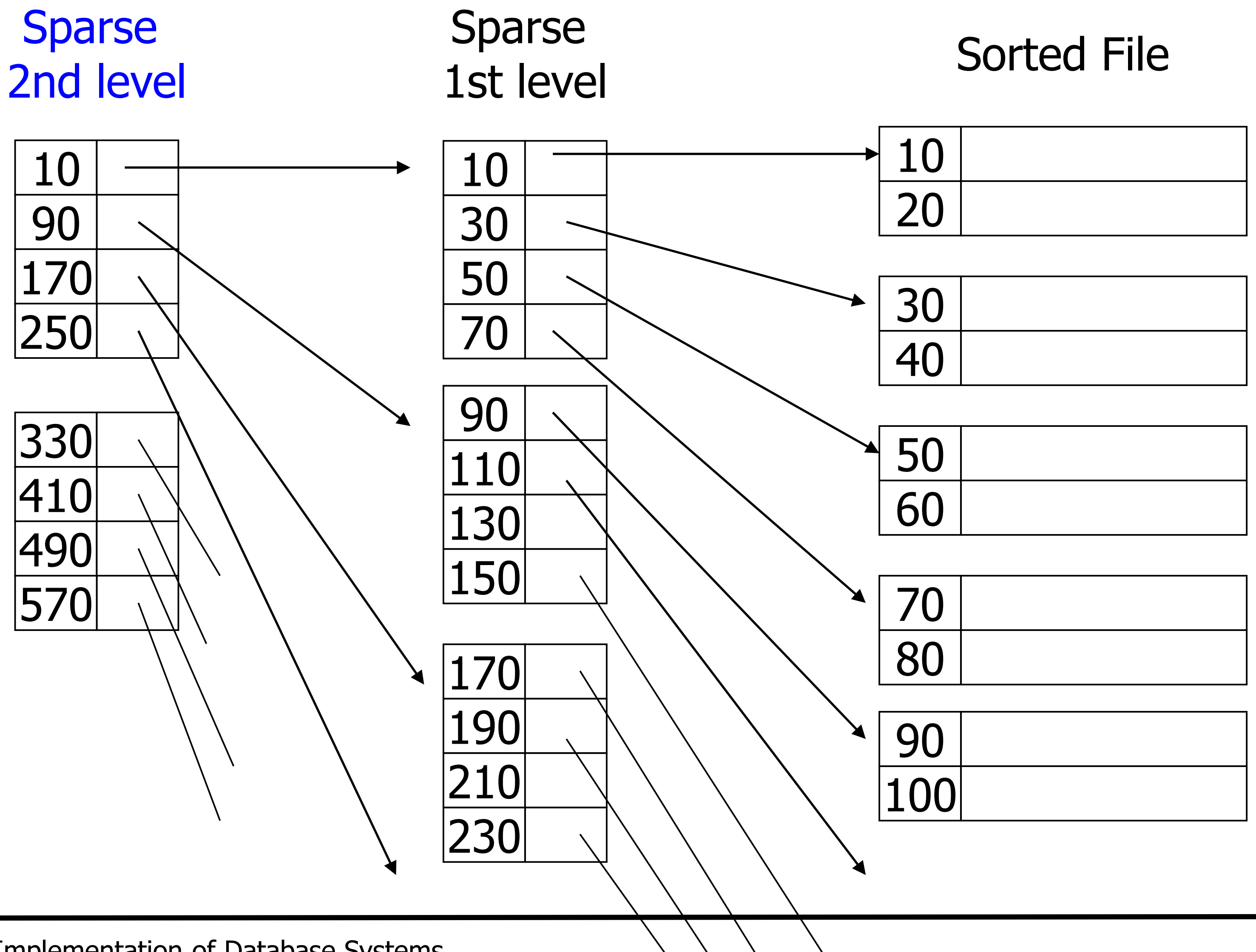
- Data file has records sorted on key
- Index stores pairs (first key, pointer) for each data block
  - Sparse index: Only put first key per block in index
- Constraint (k_i, ptr): For all k in ptr↑: $k_i \leq k < k_{i+1}$



Searching in Index-Sequential Files

- Search key in index using binsearch, then access correct block by TID
- Advantages
  - Index has only few keys: $b_{index} << b_{records}$
    - Assume 10.000.000 records of size 200, |blockID|=10, |search key|=20, block size=4096
    - Number of blocks b= 10.000.000*200/4096 ~ 500.000
    - Access if kept sorted: log(500.000) ~ 19 IO
    - Index-seq file: log(500.000*(10+20)/4096) ~ 12 IO +1 for data
  - Chances that index fits into main memory
- Disadvantages
  - Only possible for one attribute (data file must be sorted)
  - More administration (compared to heap file)

Even Better: Multi-Level Index Files



Hierarchical Index-Sequential files

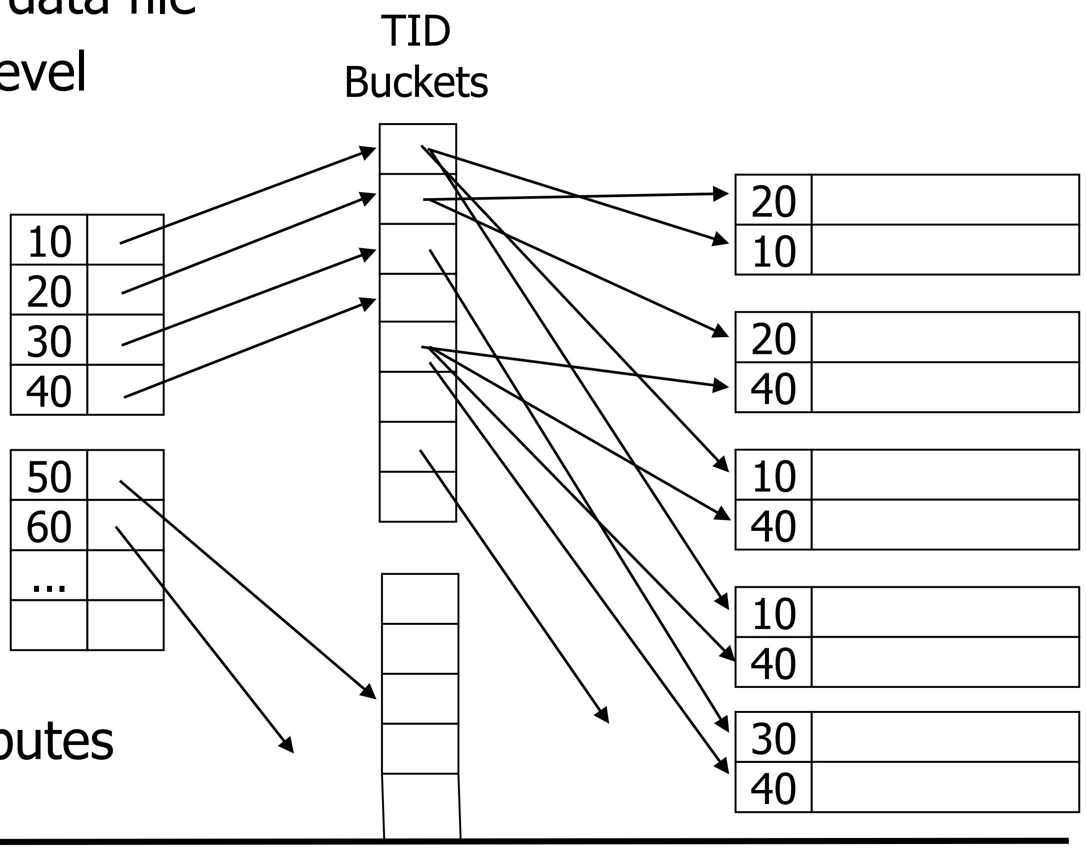
- Build a sparse, second-level index on the first-level index
- Advantages
  - Access time is reduced further
    - Assume 10.000.000 records of size 200, |blockID|=10, |search key|=20, block size=4096, b = 500.000
    - Index-seq file: log(500.000*(10+20)/4096) = 12+1 block IO
    - With second level: log(3662*(10+20)/4096) = 5+2 blocks IO
    - With three levels: log(28*(10+20)/4096) = 1+3
  - Higher levels are very small – cache permanently
- With more than one level, inserting becomes tricky
  - Either degradation (overflows) or costly reorganizations
  - Better: B-trees (later)

Index Files and Duplicates

- What happens if search key is not unique?
- Index file may
  - Store duplicates: one pointer for each record
  - Ignore duplicates: one pointer for each distinct value
    - Smaller index file
    - Requires sorted data file
    - "Semi-sparse" index
- Index degradation
  - If only few distinct values exist, every search selects many TID
    - E.g. index on Boolean attributes – index has only two different entries
  - Semi-sparse index leads to less IO
  - But selects blocks in random IO – scan might be cheaper

Secondary Index Files

- Primary ind.: Index on attribute on which data file is sorted
- Secondary index: Index on any other attribute
  - Cannot exploit order in data file
  - Must be dense at first level
- Improvement:
  Use intermediate buckets only for TIDs
  - Buckets hold TIDs sorted by index key
  - Buckets don't store key values
  - Advantageous for low cardinality attributes



Indexes in Oracle

- Data files are heap files
  - Exception: Index-organized tables (IOT)
    - Recommended only for "read-only" tables
  - Every primary key is indexed automatically
  - Every UNIQUE attribute is indexed automatically
  - Default: B* tree
    - Alternatives: Multidim index, bitmap index, user-defined
- Join index: Index on attribute of foreign table with FK/PK
- Cluster index (DB2) – cluster tables and index common key
  - Example: Cluster department and employee on common depNum
  - Tuples with same depNum will go into same data block
  - Cluster index: Create index on depNum (~ persistent join)
  - Oracle has no clustered indexes – use index-organized tables

Content of this Lecture

- File structure
  - Heap files
  - Sorted files
  - Index Files
  - Hierarchical Index Files
- Excursion: Indexing texts

Excursion: Indexing Text

- Information retrieval
  - Searching documents with keywords
  - Typically, each document is represented as “bag of words”
  - Queries search for documents containing a set of words
- Naïve relational database way fails
  - Indexed varchar2(64KB) attribute containing text
  - Not efficient for keyword queries (INSTR())
  - We cannot store each word in an extra column
- Alternatives?

Inverted Lists

- Build a secondary, bucketed index on the words
- Find documents by intersecting buckets
  - Enables AND, NOT or OR