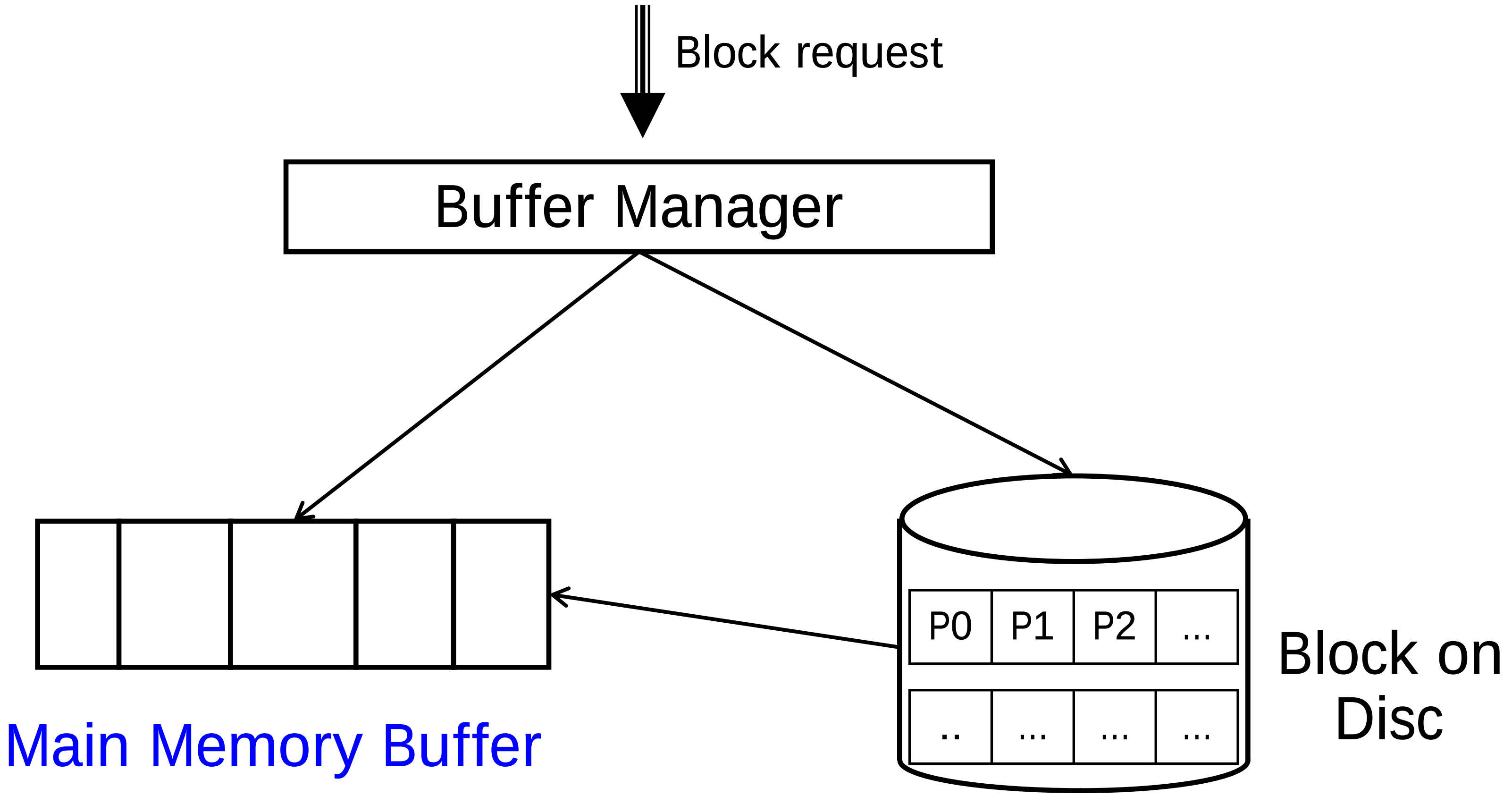
Datenbanksysteme II: Caching and File Structures

Ulf Leser

Content of this Lecture

- Caching
  - Overview
  - Accessing data
  - Cache replacement strategies
  - Prefetching
- File structure
- Index Files

Caching = Buffer Management



IO Buffering

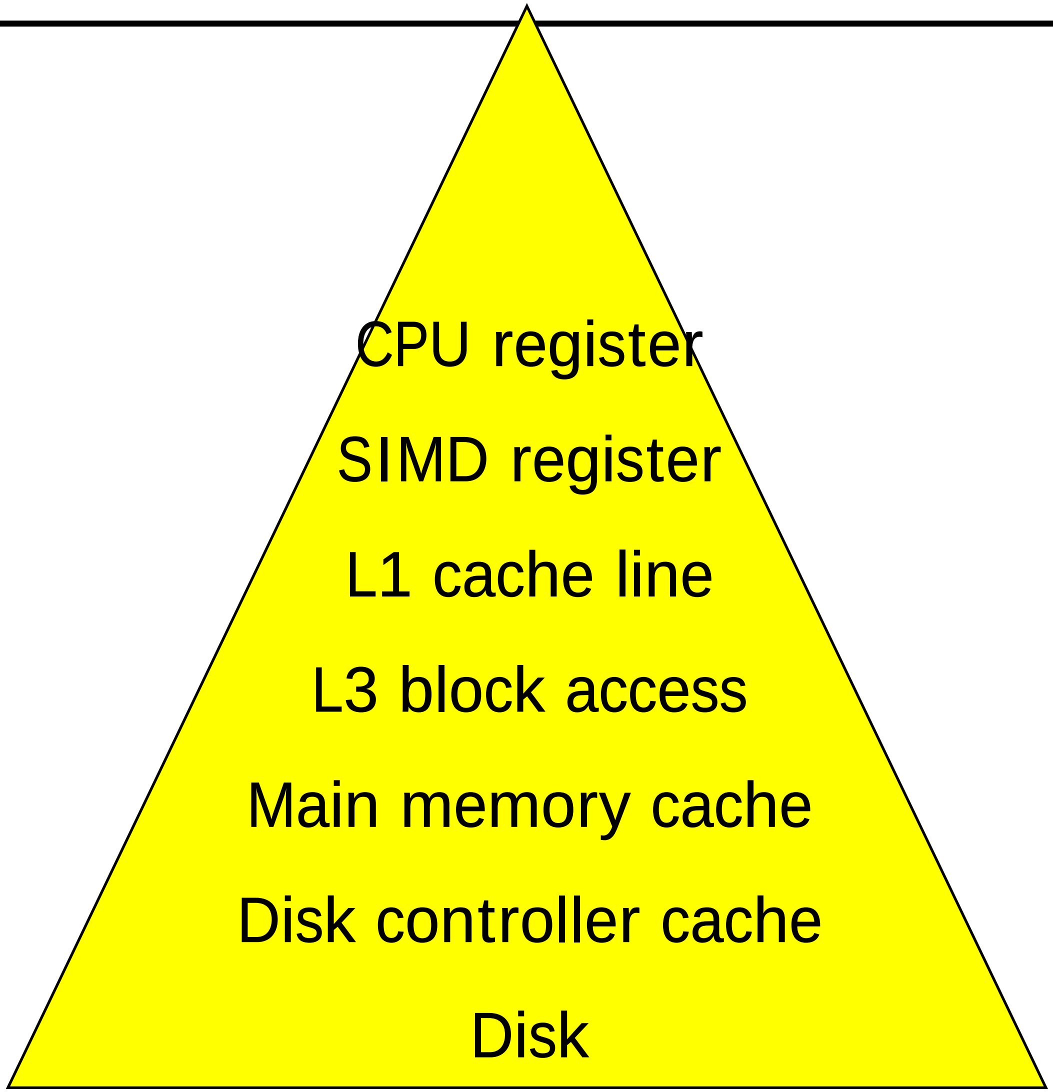
- RDBMS requests block Y from disk buffer manager
- Buffer manager checks if …
  - Y in cache: Grant access
  - Y not in cache
    - No free space in buffer?
      - Choose block Z in buffer
      - If Z has been changed – write Z to disc
      - Mark Z as free and proceed
    - Free space available?
      - Load Y into free space
      - Grant access

Address rewriting

Cache replacement policy

Cache fetch policy

Storage Hierarchy



General Method

- Level X requests block Y from level X+1
- Buffer manager of X+1 checks if …
  - Y in cache: Grant access

Address rewriting

  - Y not in cache
    - No space available?
      - Choose block Z in buffer
      - If Z has been changed – write Z to disc
      - Mark Z as free and proceed

Cache replacement policy

    - Space available?
      - Load Y into free space
      - Write into free space
      - Grant access

Cache fetch policy

Finding a Block

- We need to check if block Y is in buffer
  - Y is logical block ID in a virtual address space
- Possibilities
  - Memory blocks store their logical block ID
    - Find Y: Search all blocks (slow, no global data structures)
  - Mapping table "logical block ID" – "physical block address"
    - List data structure for all BlockIDs in buffer
      - Sorted array, linked list, sorted linked list, hashing, ...
    - Find Y: Fast, but requires synchronized access

Access with a TID

- By delegation: x:=getData(TID,10)
- By hardwired pointer: adr := getAddr(TID); x:=adr[10]

- Pinned tuples: References to location in main memory exist
  - Direct access possible
  - Record must not be moved
    - Would require adaptation of all references
  - Block must not be replaced without destroying existing pointers
- Unpinned tuples: No references to location exist
  - Every access requires one indirection
  - Tuple may be moved
  - Block may be written

Content of this Lecture

- Caching
  - Overview
  - Accessing data
  - Cache replacement strategies
  - Prefetching
- File structure
- Index Files

Caching Strategies – Going Wrong

- Imagine a nested loop join
  - Outer relation A has 10 blocks, inner relation B has 6 blocks
- Buffer size 6 blocks
- Assume Caching with FIFO (first in – first out)
  - Cache is filled with A1 and B1, B2, B3, B4, B5
  - Loading B6 replaces A1
  - For next inner loop, A1 must be loaded again, replacing B1
  - For loading A2, B2 is replaced, B1 replaces B3, ...
  - Altogether: 70 reads
- FIFO is a typical OS caching strategies
- DB needs to be able to control cache behavior

Caching Strategies – Better Strategy

- Imagine a nested loop join
  - Outer relation A has 10 blocks, inner relation B has 6 blocks
- Buffer size 6 blocks
- Proceed as follows
  - Cache is filled with A1 and B1, B2, B3, B4, B5
  - Loading B6 replaces B5
  - For next outer loop, A2 replaces A1
  - Inner loop: B1-B4+B6 without replacement
  - B6 replaces B5
  - ...
  - Altogether: 1+6+9+9 = 25 block reads

Caching Aspects

- What to manage?
- How much to load?
  - Optimal strategy ensures block is in buffer before request
  - “Block-at-a-Time” versus “Read ahead”
- What to replace?
  - Cache replacement strategies
- Good caches requires information flow from DB layer to buffer manager
  - Example: Reading complete relation (read ahead)
  - Example: Executing a “Nested Loop Join” (fix outer-loop blocks)

Granularity of Cached Units

- Blocks (default): OS blocks or database blocks
- Records: Not used because “sub-IO” cost
- Chunks
  - Group blocks into larger “chunks”
  - Less administration cost at buffer manager (buffer lists)
  - IO on chunks can exploit sequentially placed blocks on disk
  - Good for very large operations (large table joins or sorts)
  - [Disk controller automatically imitates chunking]
- Tables
  - Fix all blocks of heavily used tables
  - E.g.: System catalog, Oracles CACHE parameter

Pre-fetching

- Load blocks not yet needed but probably soon
- Examples
  - If block from relation is requested, also load next blocks
    - Possible full table scan?
  - If object is accessed, also load referenced objects
    - Not implemented in RDBMS, but successful in OODBMS
- Disc pre-fetching – if sector is requested, read entire track
- Pre-fetching requires replacement of multiple blocks
- Using sequential and asynchronous (non-blocking) IO, pre-fetching costs little and can save a lot of time

General Replacement Strategies

- Properties of blocks
  - Age(s)
    - Time since block was loaded
    - Last time accessed
  - Living references
  - Demand: Number of accesses over (recent) time

Verfahren	Prinzip
FIFO	älteste Seite ersetzt
LFU (least frequently used)	Seite mit geringster Häufigkeit ersetzen
LRU (least recently used)	Seite ersetzen, die am längsten nicht referenziert wurde (System R)
DGCLOCK (dyn. generalized clock)	Protokollierung der Ersetzungshäufigkeiten wichtiger Seiten
LRD (least reference density)	Ersetzung der Seite mit geringster Referenzdichte

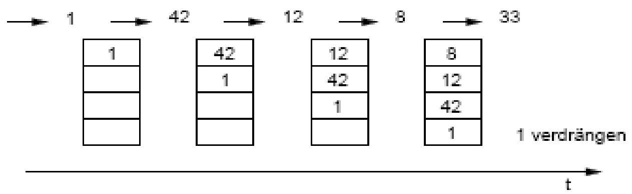
- Trade-offs
  - Young blocks have few refs, but are involved in current operations
  - Old blocks have many refers, but might get out-of-use right now
- Practice
  - Query / Operator-specific strategies (explicit pinning)
  - Use / weight multiple properties

Lessons

- Many general caching strategies have been (and are still) developed
- Simple strategies are surprisingly good
  - LRU or even random
  - Commercial databases: Mostly LRU
    - With fixing of blocks and special tricks for large operations

Implementing LRU with a LRU queue

- When block is requested
  - Critical operation:
    Search blockID in queue
    - Very often performed
  - Implemented with two lists
    - Queue sorted by least access
    - Maintain pointers on first and last position
    - Hashmap: BlockIDs to queue positions (quasi-constant time)
- Access block: Delete and push on top of queue
- Evict block: Remove from bottom of queue
- Load block: Add at top of queue



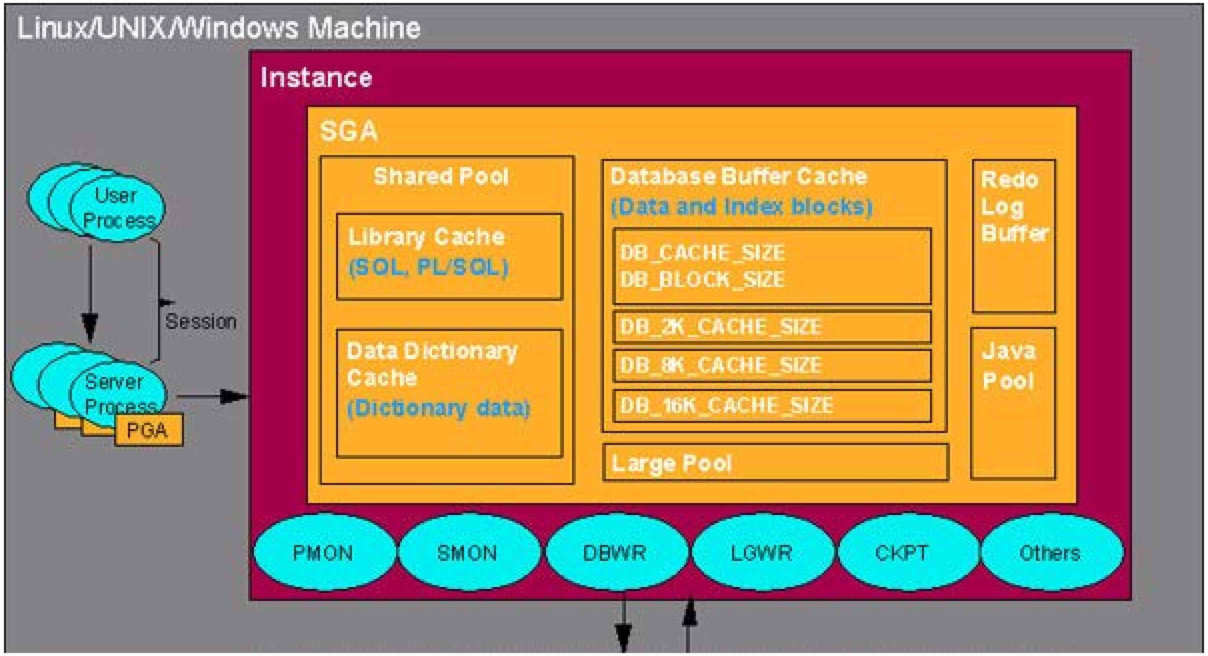
Other Cache Issues

- Be aware: Your data is not written immediately
  - Cache manager needs to check if writing before replacement is necessary (dirty flag)
  - With caching, data stays on volatile device much longer than without
  - Special care required – recovery strategies
- Cache consistency in distributed systems
  - If more than one system caches, data may become stale
  - Requires some form of synchronization
- Cache consistency in multi-TX systems
  - If more than one TX changes data, multiple versions of a block may exist
  - Requires some form of synchronization

Semantic Caching: Cache query result

- Example
  - Q1: “Select name from person where age>45”
  - Q2: “Select * from person where age>18”
  - Q1 can be answered using result tuples from Q2
- Powerful but complicated technique
  - Can a query be answered using results of one or more other q’s?
  - Query containment, “answering queries using views”
- Very complicated for write operations
  - Cached result blocks are not IO blocks
- Semantic caching not used by any real DB today
  - Note: Normal caching sometimes “mimics” semantic caching
  - If Q1 executed after Q2, blocks from Q2 are in cache
  - But: Computations need to be repeated (e.g. aggregation)

Many Tasks Compete for Main Memory

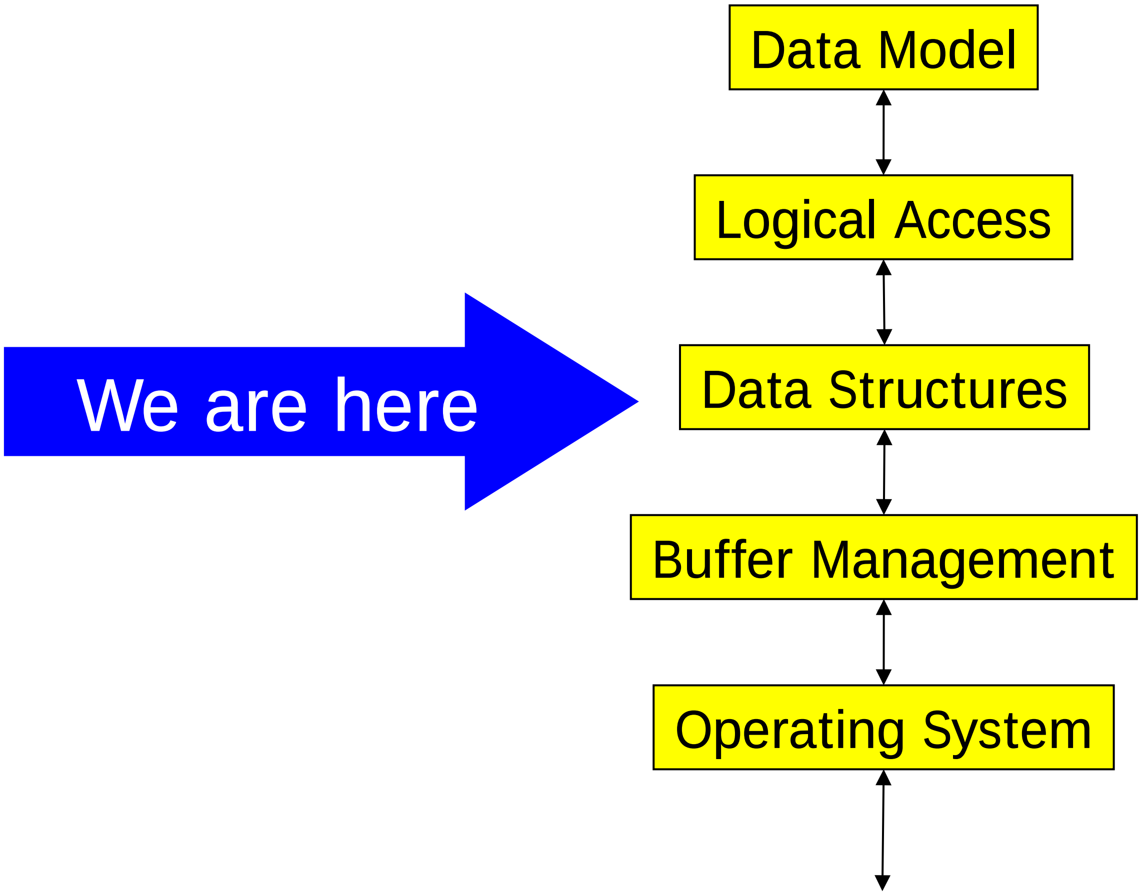


- SGA: System global area
  - Processes communicate through SGA
  - Requires locking of main memory structures – latches
- Library cache: buffers SQL prepared statements using LRU
- Java pool: area for java stored procedures
- Each process additionally gets its PGA (process global area)
- Each area is limited and can become a bottleneck

Content of this Lecture

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

5 Layer Architecture

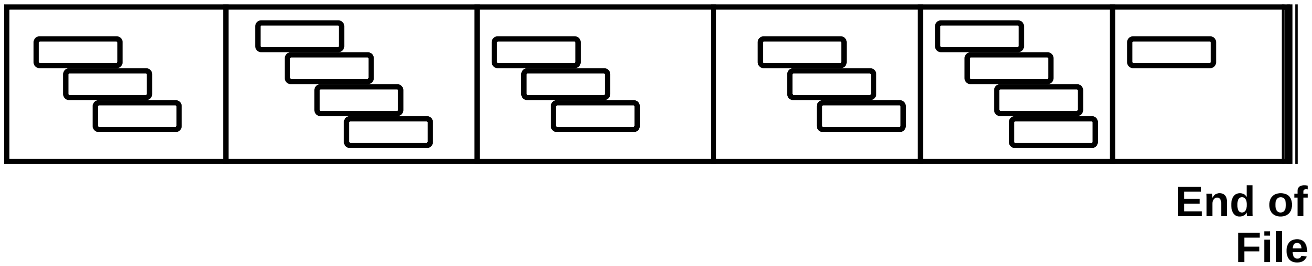


Files and Storage Structures

- We have
  - Records are stored in blocks
  - Blocks are managed/cached by the buffer manager
  - Access records by TID through cache manager with adr-translation
- But DBs usually search records with certain properties
  - SELECT * FROM COSTUMER
    WHERE Name = "Bond"
  - SELECT * FROM ACCOUNT
    WHERE Account# < 1000
- This is not "access by TID"
- There must be more clever ways than scanning

Sequential (Heap) File

- Records are stored sequentially in the order of inserts



- Insert always add to end of file
- "Holes" occur if records are deleted
- Minimal number of blocks : b = ⌈n / R⌉
  – With n = number of records, R = number of records per block
- Better to keep some space free for growing records
  – Fraction depends on expected read/write ratio

Operations on Heap Files

- In the following: We assume highly selective searches
  – Only a few records qualify
- Search with value
  – b/2 block IO in case of successful searching a PK (on average)
  – b block IO in case of failure or searching non-unique values
- 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 list 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

Deleting Records

- First issue: File fragmentation
  - Move records in block to gather larger chunks of free space
  - In case of underflow: Remove blocks
    - And change block translation table
- Second issue: Dangling pointers
  - In case of deletes, existing references (indexes) need to considered
  - Option 1: Update references
    - Requires to keep a list of all active references per record
    - One record deletion results in multiple physical deletions
  - Option 2: Use tombstones
    - Only mark record as deleted (e.g. null in block-dir)
    - References are updated only when used
    - Very fast at deletion time, some effort later

Sorted Files

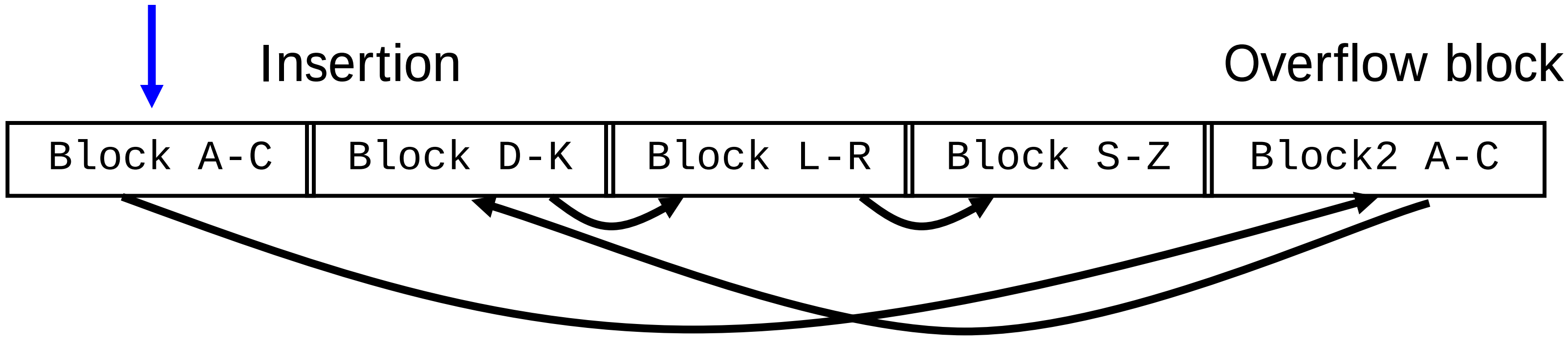
- Sort records in file according to some attribute
  - Faster searching when this attribute is search key
  - More complex management – order must be preserved
- Operations and associated costs
  - Search (using binsearch on blocks)
    - log(b) IO; searching in block is free (as always)
      - Note: That’s mostly random IO!
  - Change / delete record 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 additional IO for writing
- Option 2: Check neighbors
  - See X blocks down and X blocks up in the file (usually X=1)
  - When space is found, in-between records need to be moved
    - Add change block translation table
  - Cost: depends on how far we need/want to look
- Option 3: ?

Overflow Blocks

- Option 3: Generate overflow blocks
  - Create a new, “orthogonal” overflow block and insert record
  - When blocks are connected by pointers
    - Sorted table scan still possible as blocks are chained in correct order
    - New block will not be in sequential physical order
  - When block is added at end of file
    - Sequential-IO table scan still possible, but not in order of attribute
    - Requires that continuous space is reserved for growing tables
      - Oracles “Extent”



Disadvantages

- Some cost of keeping order (INSERT requires log(b) search first, management of overflow blocks, ...)
- Only one search key
  – Searching on other attributes requires linear scans
  – See multi-dimensional indexes
- Search time grows logarithmically with b
  – For 10.000.000 blocks, we need ~23 IO
- 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 “Hampel”, [A-C]=7500, [D-F]=6200, [G-I]=3300
  - Estimated position: 7500+6200+(3300/3)*2 + ...
  - Continue with local search around 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
  - Histograms (statistics) need to be maintained (see later)
    - Updates and synchronized: Potential bottleneck for update operations on multiple records in the same bucket
  - Choosing optimal bucket number and range is difficult

Content of this Lecture

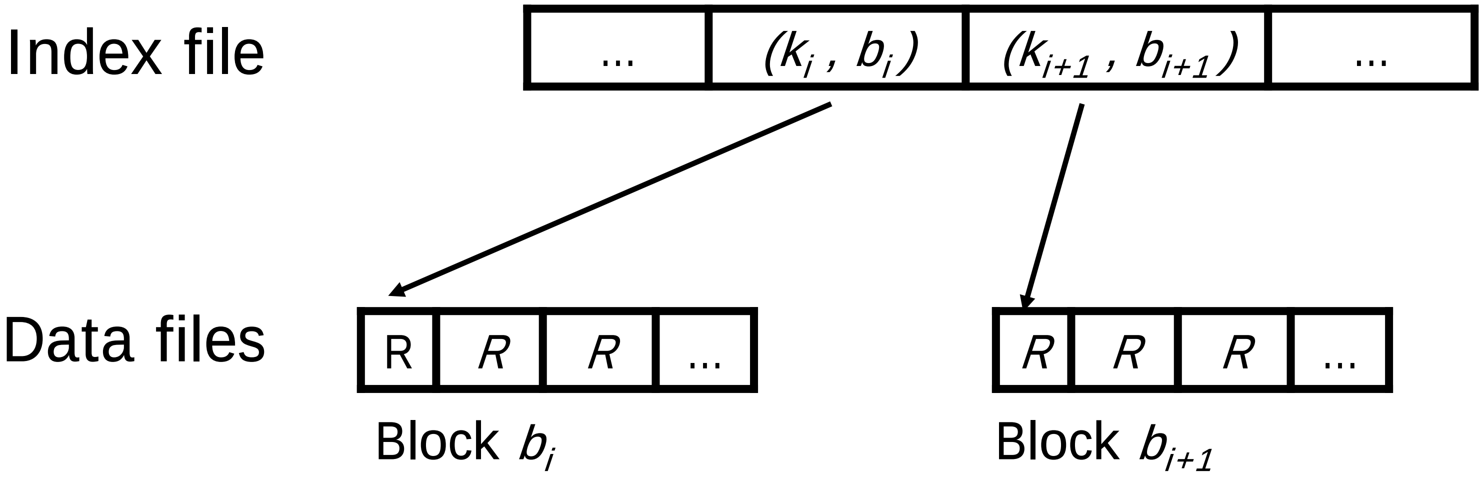
- Caching
- File structure
  - Heap files
  - Sorted files
- Index Files

Idea 2: Decrease b: 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
      - I:nteger keys: Fixed-length index entries; strings: Use fixed-length prefix
    - But no fast sorted scans anymore (e.g. for sort-merge join)
  - Faster search due to smaller records and less blocks: $b_{index} < b_{records}$
  - Several indexes can be build for several attributes
    - More flexibility, more update cost
- Disadvantages
  - More files to manage, lock, recover, ...
  - Advantage shrinks if many tuples are selected (e.g. range queries)

Further Decrease b: Index Sequential Files

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



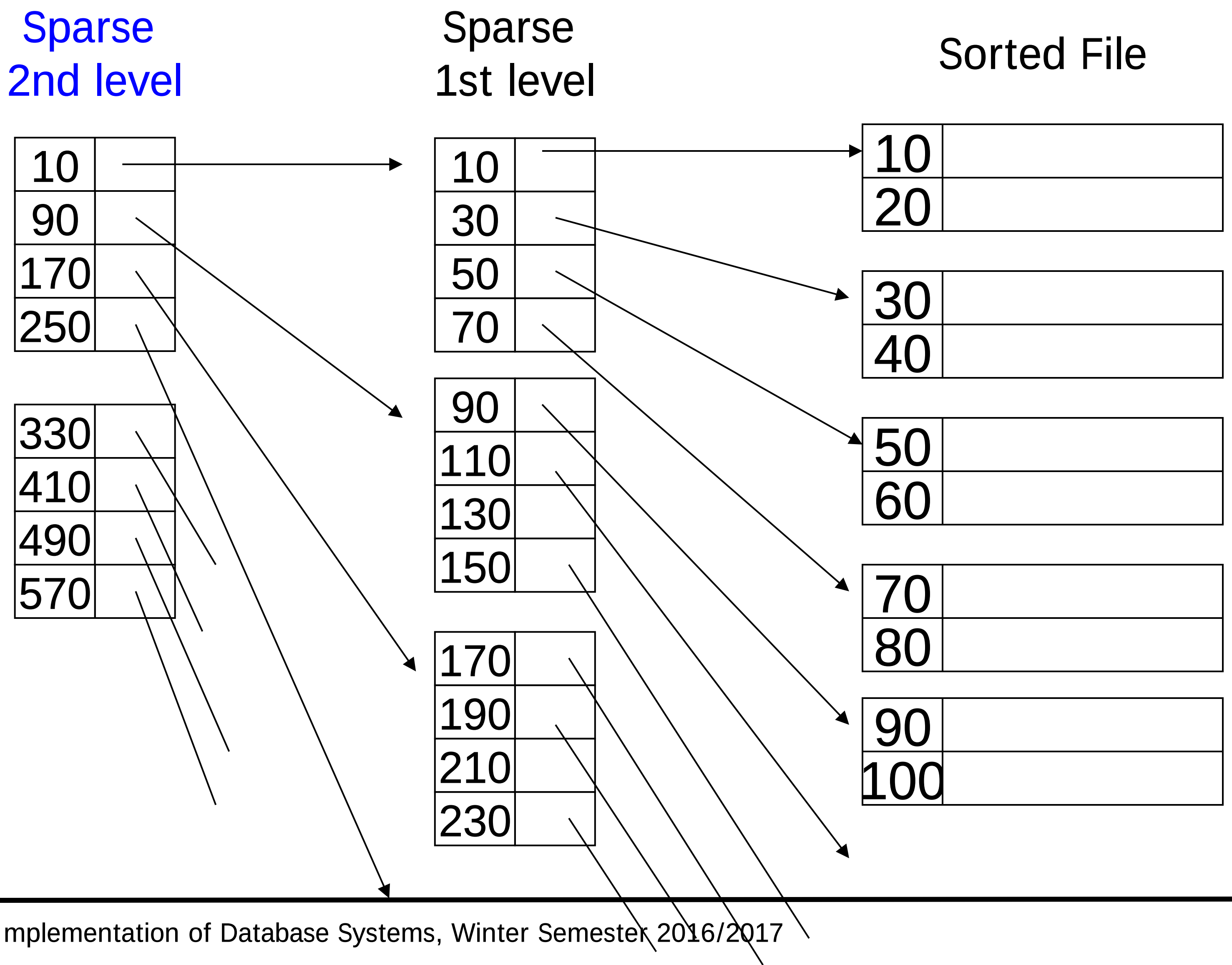
Searching in Index-Sequential Files

- Search key in index using binsearch, then access 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 (mostly) into main memory
- Disadvantages
  - Only possible for one attribute (data file must be sorted)
  - More administration (compared to heap file)

Index-Sequential Files: Other Operations

- Insert record r with key k
  - Search for block b_i with $k_i \leq k \leq k_{i+1}$
  - Free space in block? Insert r; done
  - Else, either check neighbors
    - Index needs to be updated, as block's first keys change
  - ... or create overflow blocks
    - Option 1: New block not represented in index; index not updated
      - More IO when searching data, as overflow blocks need to be followed
    - Option 2: Index is updated (more IO at time of insertion)
      - We need to insert into the index – leave free space in index blocks!
- Ideas for improving search further?

Multi-Level Index Files



Hierarchical Index-Sequential files

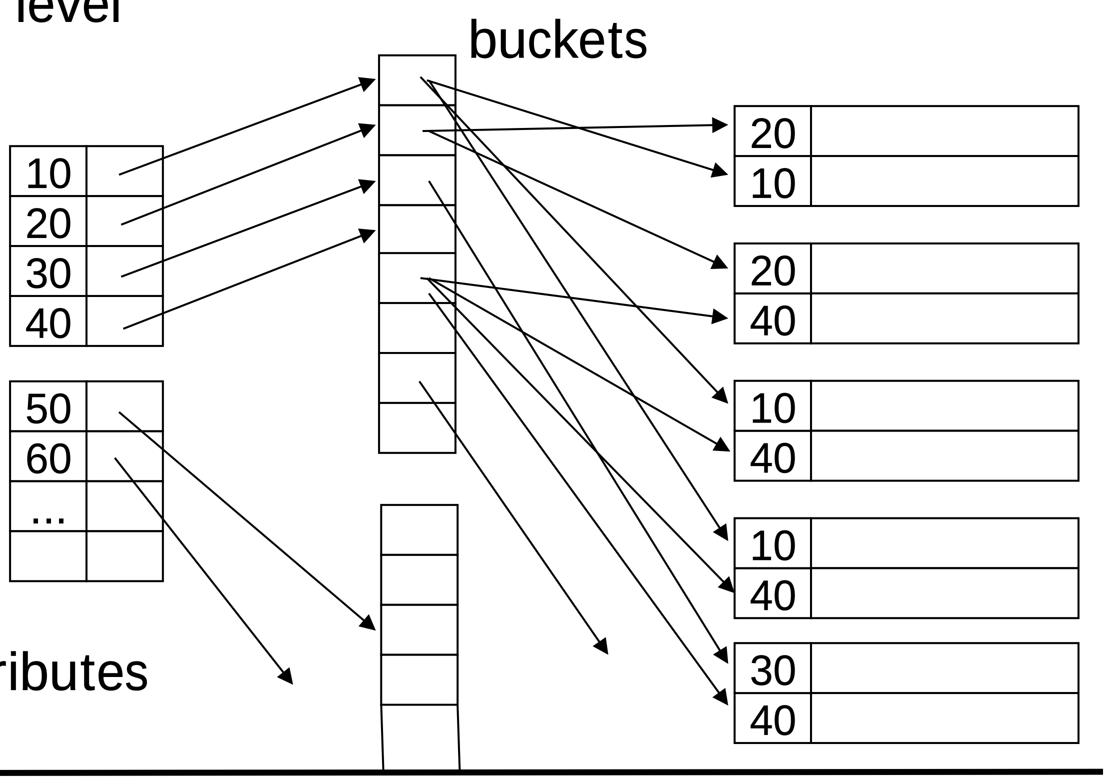
- Build a sparse, second-level index on the first-level index
  - First level may be spare or dense
  - All but the first level must be sparse; why?
- Advantages
  - Access time reduces further
    - Assume 10.000.000 records of size 200, |blockID|=10, |search key|=20, block size=4096, b = 500.000
    - 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
  - Alternative: B-trees (later)

Index Files and Duplicates

- What happens if search key is not unique in relation?
- 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

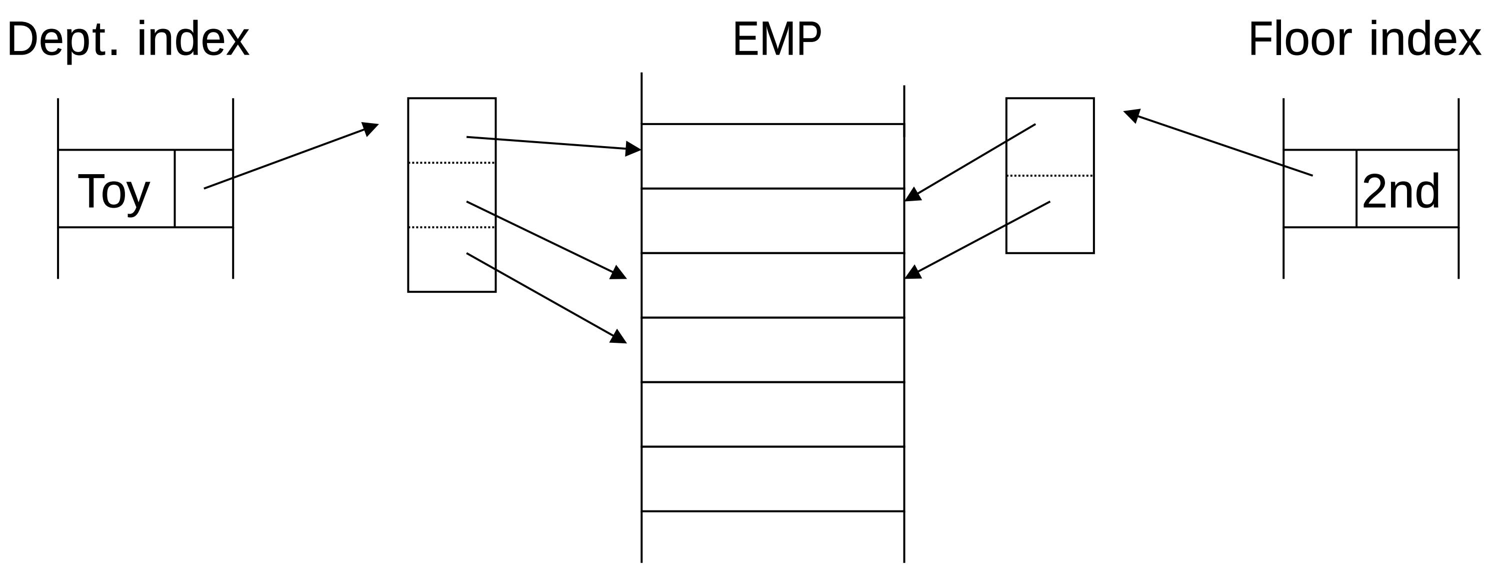
Secondary Index Files

- 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
  - Buckets hold TIDs sorted by index key
  - Buckets don't store key values
  - Advantageous for low cardinality attributes



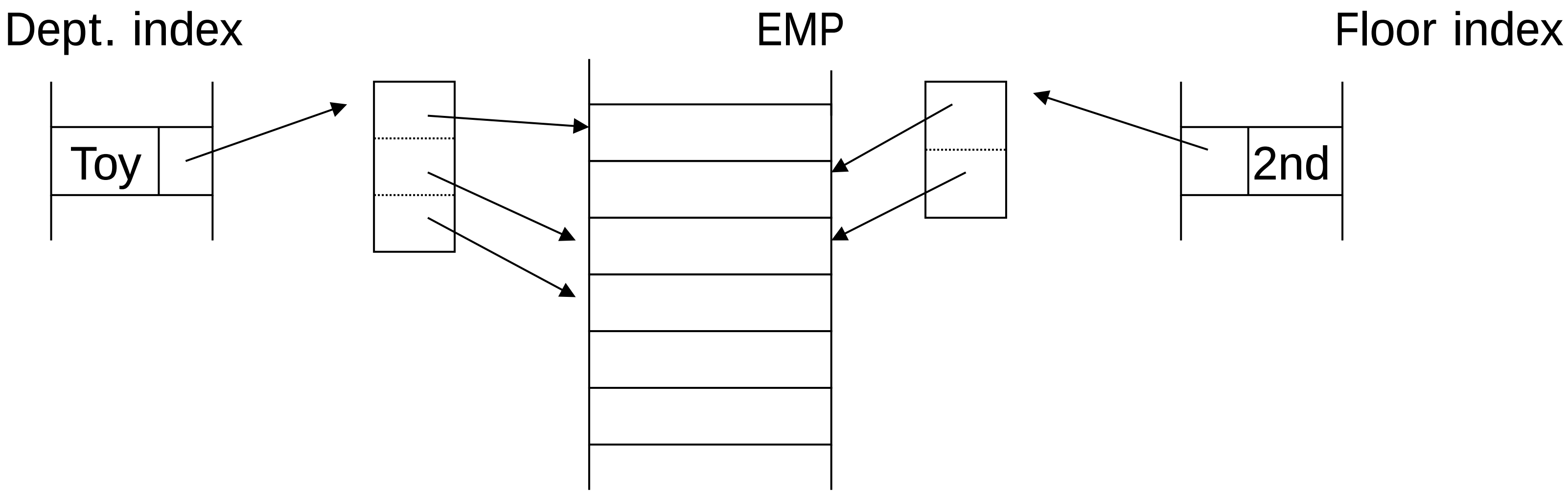
Buckets for Secondary Index Files

- Index stores keys and ptr to buckets; buckets store TIDs
- Good if many TIDs with same attribute value exist
- That's essentially a persistent hash partitioning
- Compute joins and AND's by intersecting TID-lists



Example

- Query: “All employees in TOY dept. located on 2nd floor”
  - Use floor index to find TID-list L_1
  - Use department index to load TID-list L_2
  - Compute $L = L_1 \cap L_2$
  - Load employee data only for TIDs in L
- Advantage increases with more conditions (STAR join)



Indexes in Oracle

- Per default: Secondary B* tree indexes
- Data files usually are heap files
  - Exception: Index-organized tables (IOT)
  - Recommended only for “read-only” tables
- No primary indexes
  - Do not confuse with primary key – there is always an index on a primary key (why?)
- Cluster index – cluster two 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

Multiple Sorts

- Use primary index (index-organized file) on sorted file
- Build secondary index including all attributes of the table in desired second order
  - Example: employee (ID, name, dep#, income)
  - Create IOT employee (ID, name, dep#, income)
    - Sorted by ID
  - Create index on employee (name, ID, dep#, income)
    - Sorted by name
- Maintained by database
  - Doubled space consumption
  - Faster queries
  - Increased cost for UPDATE, DELETE, INSERT

Excursion: Indexing Text

- Information retrieval
  - Searching documents with words
  - 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
  - Doesn’t allow for WORD queries
  - 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