# Datenbanksysteme II: Implementation of Database Systems

## Caching

## Storing Records in Files

## Simple Indexes

Material from

Prof. Johann Christoph Freytag

Prof. Kai-Uwe Sattler

Prof. Alfons Kemper, Dr. Eickler

Prof. Hector Garcia-Molina

# RAM versus IO

- Algorithms need to be designed and analyzed in a different way
- RAM model of computation
  - Access to data costs nothing
  - Only operations on the data count – comparison, arithmetic, counting, etc.
- IO model of computation
  - Operations cost nothing
  - Only access to data counts – reading & writing blocks
- Careful – sometimes both need to be considered

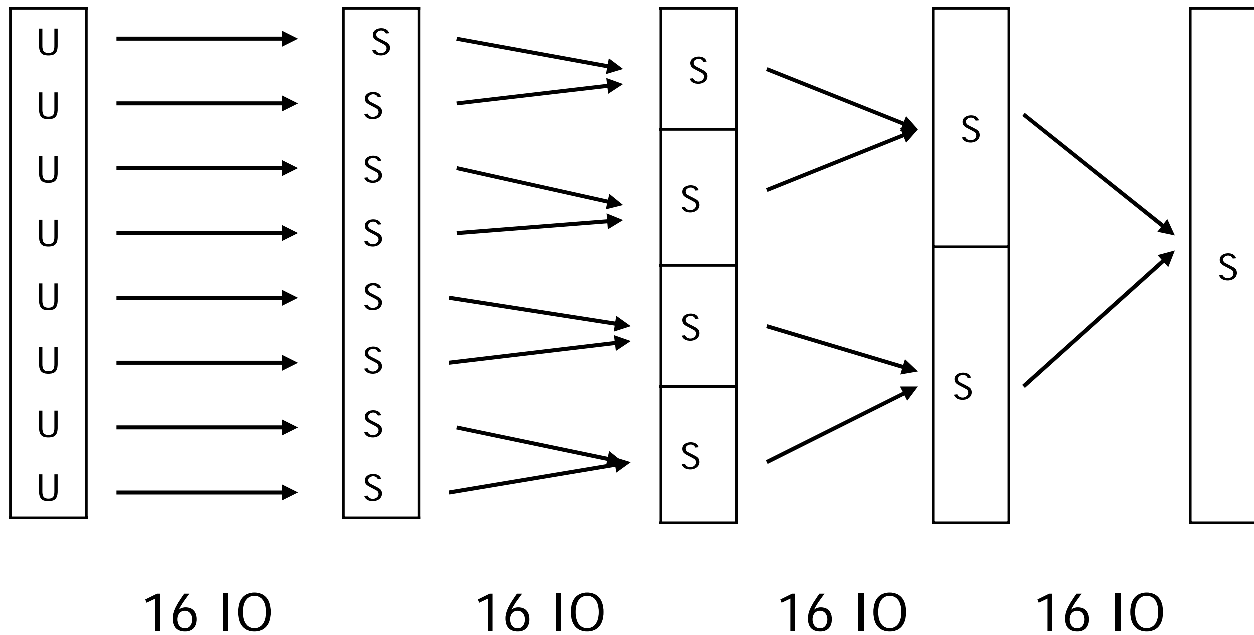
# Example Merge-Sort

- RAM model of Merge-Sort
- Idea
  - Two sorted lists of size n can be merged into one list in O(n)

| | | | |
|---|---|---|---|
| 1 | 3 | | 1 |
| 2 | 5 | | 2 |
| 4 | 6 | | 3 |
| 7 | 10 | | 4 |

  - Given an unsorted lists, work recursively
    - If list is of size 1, return (sublist is sorted)
    - Divide the list in two lists of equal size
    - Call MERGE-SORT for each list
    - Merge the sorted list
    - Complexity: $O(n*\log(n))$
      - Number of key comparisons

# Recursive Merge-Sort




- Total IO: 2*n*log(n)
  - n: Number of blocks
- Can we do better??

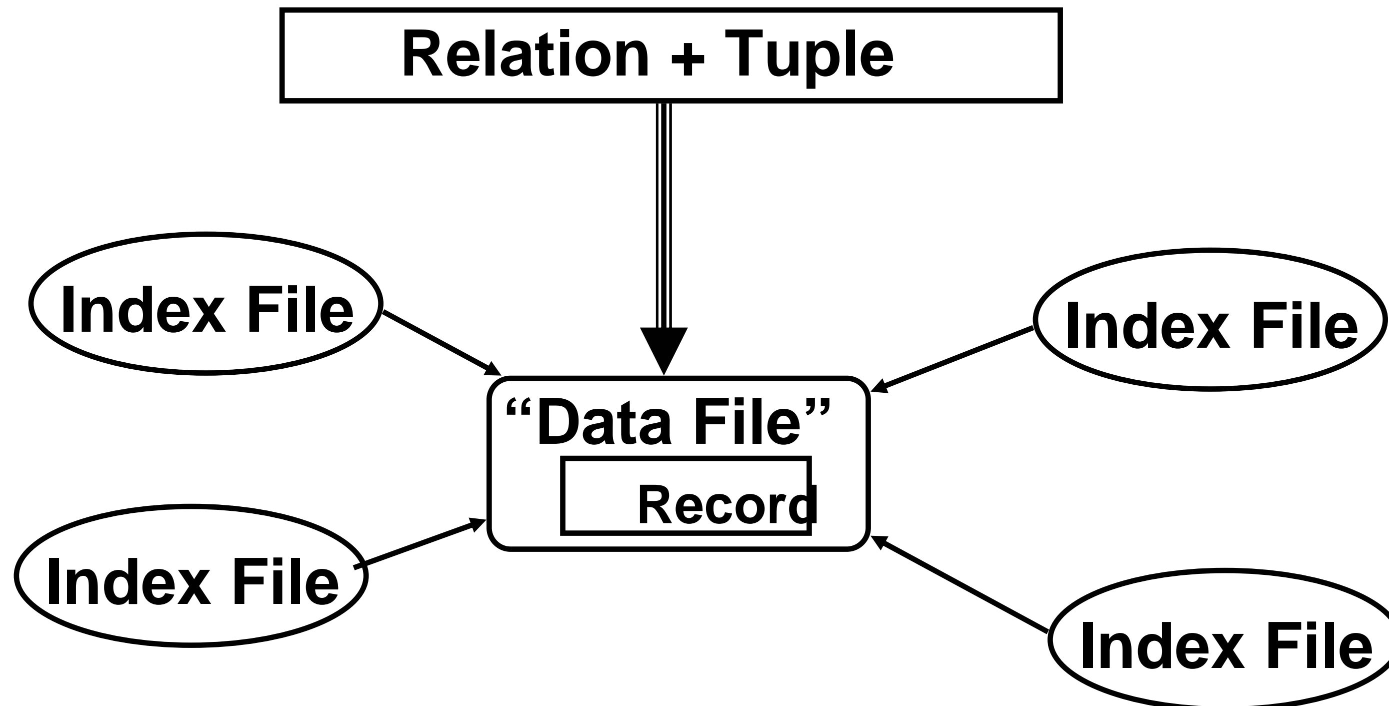
# Example cont'd

- Yes
- Load more than one block into main memory
  - Given unsorted file with n blocks
  - Main memory of m=b*n blocks
  - Read b blocks, sort in-memory, write
    - 2b IO; sorting is free
  - Iterate – generates x=n/b sorted files (runs)
    - Each block is read and written once: 2n IO
  - Merge x runs, opening all x input file at once
    - Each block is again read and written: 2n IO

# Example cont'd

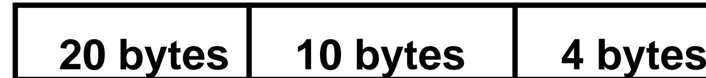
- Imagine we may have z+1 files open at a time
  - z blocks for reading and one block for writing
- Extension
  - Thus, we can sort z*b blocks using our method
    - Read and sort b blocks, each time generating one of z runs
  - Partition file in partitions of z*b blocks
  - Sort each partition, generating a mega-run
  - Open all mega-run in parallel and merge
  - If there are more than z mega-runs, recurse
- How much data can we sort now??

# Physical Representation



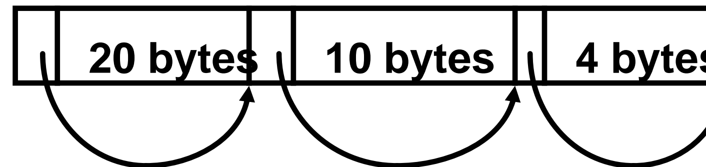
# Record Formats

- Fixed length records

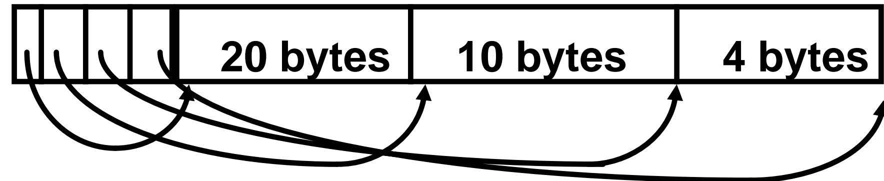



- Variable length records
  - Mark end of attributes

  - Indicator of length



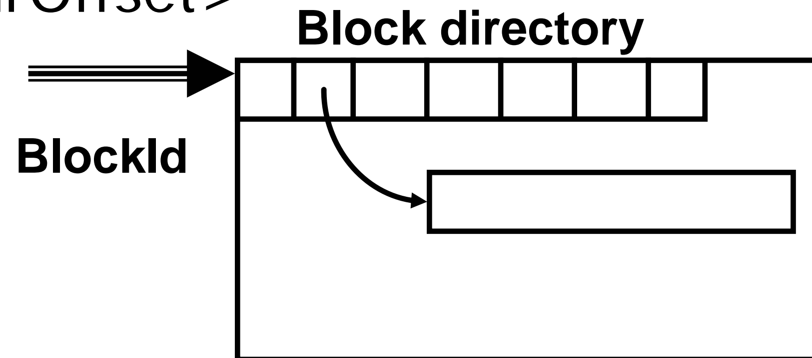

  - Record dictionary




- Don't be afraid of variable length records
  - More freedom in data modeling
  - Less space consumption
  - Additional work is manageable (but nut negligible)
  - Cost for locating tuple by pointer (instead of computing address) dominated by IO

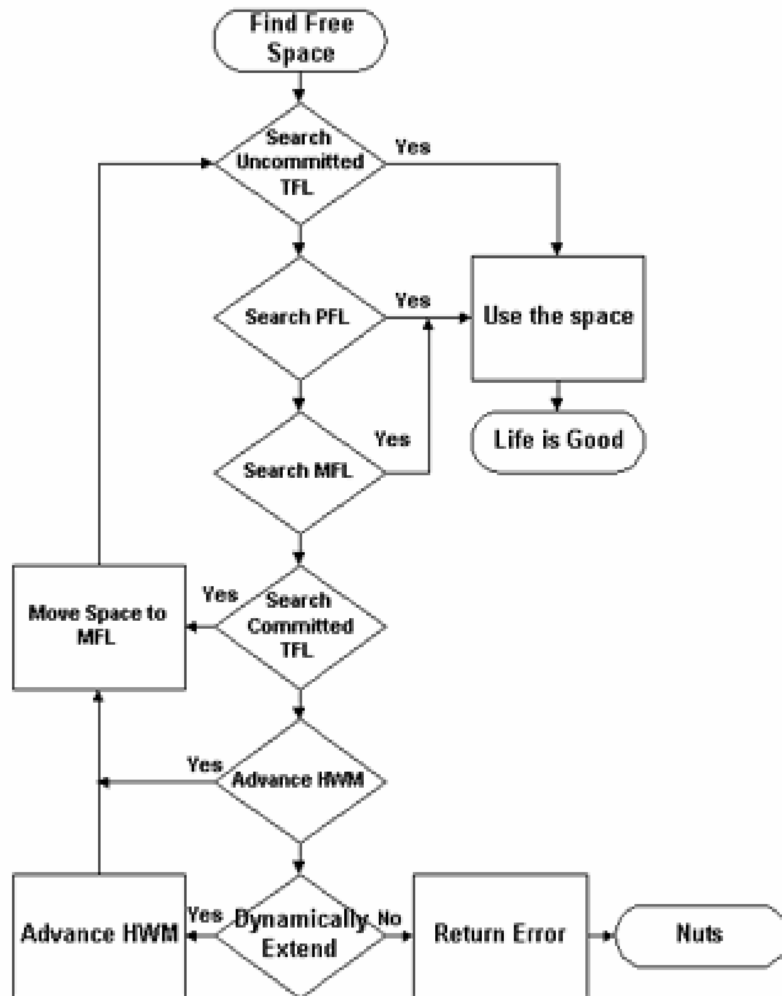
# Record Addressing (cont.)

- Block directory (tuple table):
  - TID = <BlockId, @DirOffset>




- Method of choice
  - References remain stable when tuples move in block
  - Requires storage of block directory with each block
    - Some overhead
  - How can we move across blocks without updating pointers??

# Life is complex

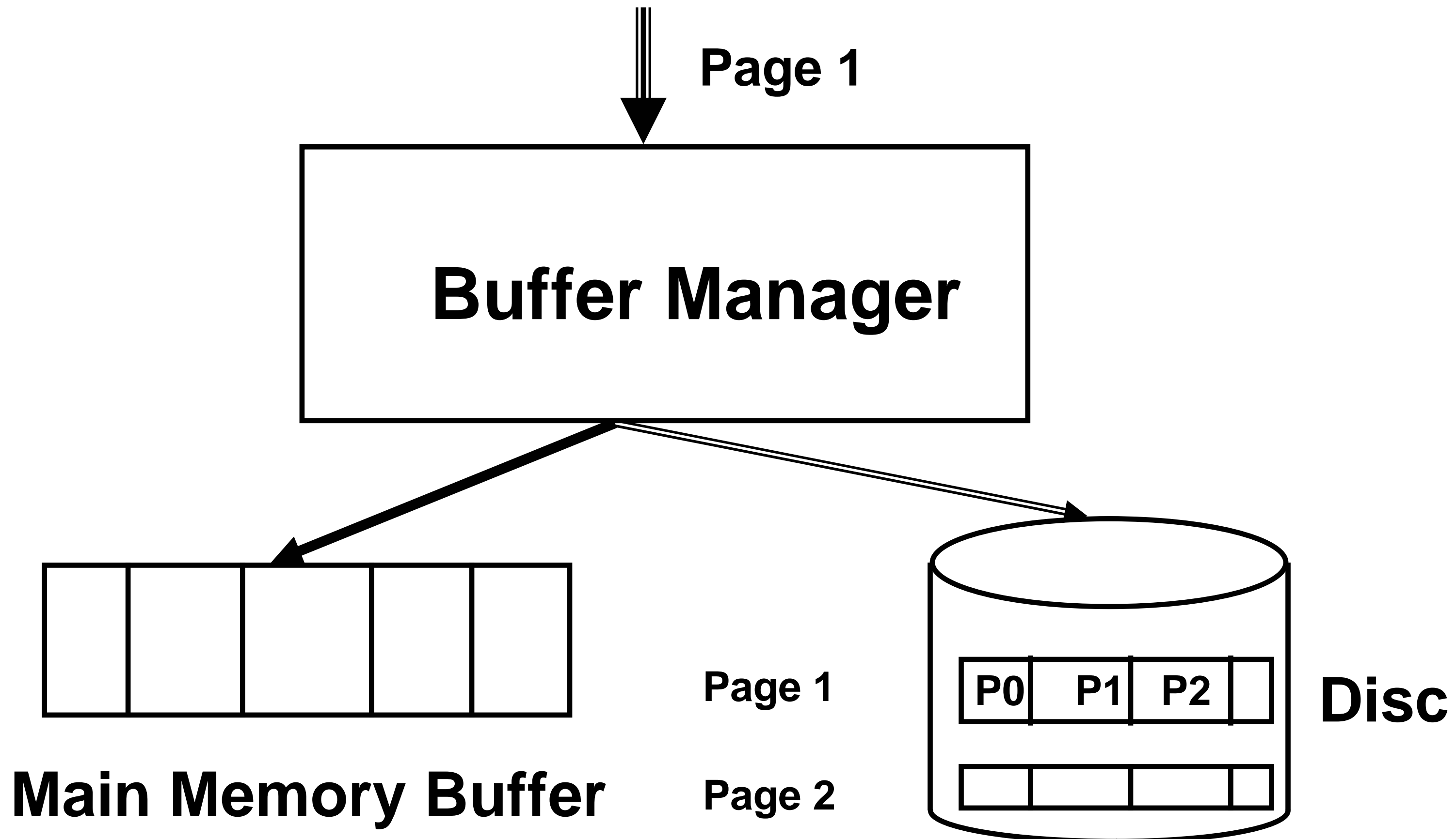



- Oracle procedure for finding free space
- Free space is administered at the level of segments
  – Logical database objects
- Explanation
  – TFL: transaction free list
  – PFL: process free list
  – MFL: master free list
  – HWM: High water mark

# Content of this Lecture

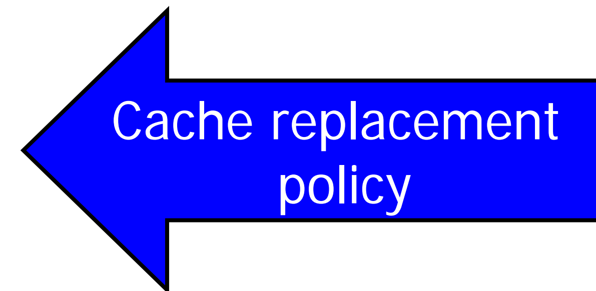
- Caching
- File structure
  - Heap files
  - Sorted files
- Some indexing
  - Index files
  - Index sequential files
  - Hierarchical index sequential files
  - (B tree)

# Caching = Buffer Management



# General Method

- Level X requests block Y from level X+1
- Buffer manager (of X+1) checks
  - Y in cache
    - Grant access
  - Y not in cache
    - No space available?
      - Choose block Z
      - If Z has been changed – write Z to disc (better: write to level X+2)
    - Space available?
      - Request Y from level X+2
      - Write into free space
      - Grant access



# Finding a Page

- Level X requests block Y from level X+1 …
- We need to check if Y is in buffer
  - Y is logical block ID in a virtual address space
- Possibilities
  - Memory pages store their logical block ID
    - Search all pages (slow, no administration)
  - Maintain mapping "logical block ID" – "physical page address"
    - Not possible to hold table with all block Ids
    - Use list data structure: unsorted array, linked list, sorted linked list, hashing, …
    - Fast, but requires some administration

# False Caching Strategies

- Imagine a nested loop join
  - Outer relation A, 10 blocks
  - Inner relation B, 6 blocks
- Buffer size 6 blocks
- Caching with FIFO (first in – first out)
  - Cache is filled with A1 and B1, B2, B3, B4, B5
  - Loading B6 replaces A1
  - For next outer loop, A1 must be loaded again, replacing B1
  - For loading B1, B2 is replaced, ...
- FIFO is a typical OS caching strategies
- DB needs to be able to control cache behavior
  - Fix A1
  - Read and replace 5 blocks at once
  - (Blocked nested loop later)

# Caching Aspects

- Really optimal strategy: Required page always in buffer before request arrives
  - Not possible in general
- Buffer replacement strategies
  - Which page to replace in case buffer is full?
  - "Block-at-a-Time" or "Read ahead" strategy
- Desirable: Cooperation between DB and buffer manager
  - Example: Reading complete relation
    - Asynchronous "Read ahead" advantageous
  - Example: Executing a "Nested Loop Join"
    - Knowledge about outer and inner relation can help
- Cooperation with transaction manager
  - Usage of Pin/Unpin Operation (later)

# Granularity

- Blocks
  - Normally used
  - OS blocks or database blocks
- Records
  - Not used because replacement / reloading to costly
- Chunks
  - Group blocks into larger “chunks”
  - Less administration cost
  - Loading / writing chunks can exploit sequentially placed blocks on disk
  - Good for very large operations (large table joins or sorts)
- Tables
  - Fix all blocks of heavily used tables
  - E.g.: system catalog, Oracles CACHE parameter

# Semantic Caching

- Cache query result, not disc blocks
- 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
  - When can a query be answered using results of one or more other queries?
  - Query containment, “answering queries using views”
- Very complicated for write operations
  - Cached result blocks are not IO blocks
  - Duplicated caching: result blocks for reading, IO blocks for writing
- Semantic caching not used by any commercial database today
  - Note: Normal caching can mimic semantic caching
  - If Q1 executed after Q2, blocks from Q2 are in cache
  - But: Computations need to be repeated (e.g. aggregation)

# Other Cache Issues

- Cache replacement policy needs to be chosen carefully
  - Least recently used (timestamp)
    - Simple, popular, and effective
  - Least often used (counter), cyclic, …
- Data not written immediately
  - Cache manager needs to check if writing is necessary
    - Dirty flag
  - Data stays on volatile device much longer than without caching
  - Special care required – recovery strategies
- Cache consistency
  - Parallel servers, clustered databases
  - If more than one system caches, data becomes outdated
  - Requires synchronization protocols

# Pre-fetching

- Load blocks not yet needed, but hopefully soon
- Examples
  - If block from chunk is requested, load entire chunk
    - If possible by sequential read from disc
  - If page from relation is requested, also load next pages
    - Possible full table scan?
  - If object is accessed, also load referenced objects
    - Not implemented in RDBMS, but very successful in OODBMS
  - Disc-based pre-fetching – if sector is requested, read entire track
    - Try to buffer entire track
- Using sequential and asynchronous, non-blocking reads, pre-fetching often costs little and can save a lot of time

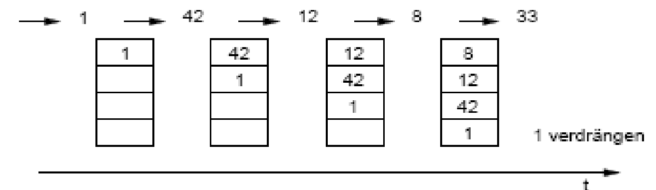
# Caching strategies

- Properties of pages
  - Age
    - Time since page was loaded
    - Last time accessed
  - Living references
    - Referenced pages might still be necessary in future
  - Trade-off
    - Young pages cannot have many references, but are involved in current operations
    - Old pages have collected many references, survived a long time - constant use
    - Mix age and references
- Many caching strategies have been (and are still) developed
- Simple strategies are surprisingly good
  - LRU or even random
  - Commercial databases: Mostly LRU
    - With fixing of pages and special tricks for large operations

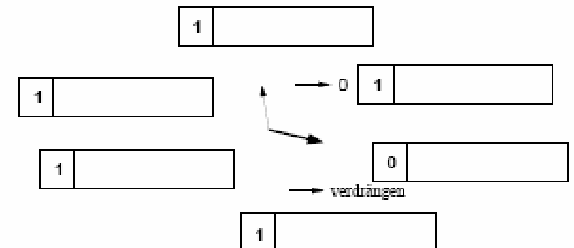
| Verfahren | Prinzip |
|---|---|
| FIFO | älteste Seite ersetzt |
| LFU (least frequently used) | Seite mit geringster Häufigkeit ersetzen |
| LRU (least recently used) | Seite ersetzen, die am längsten nicht referenziert wurde (System R) |
| DGCLOCK (dyn. generalized clock) | Protokollierung der Ersetzungshäufigkeiten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

# Implementing LRU

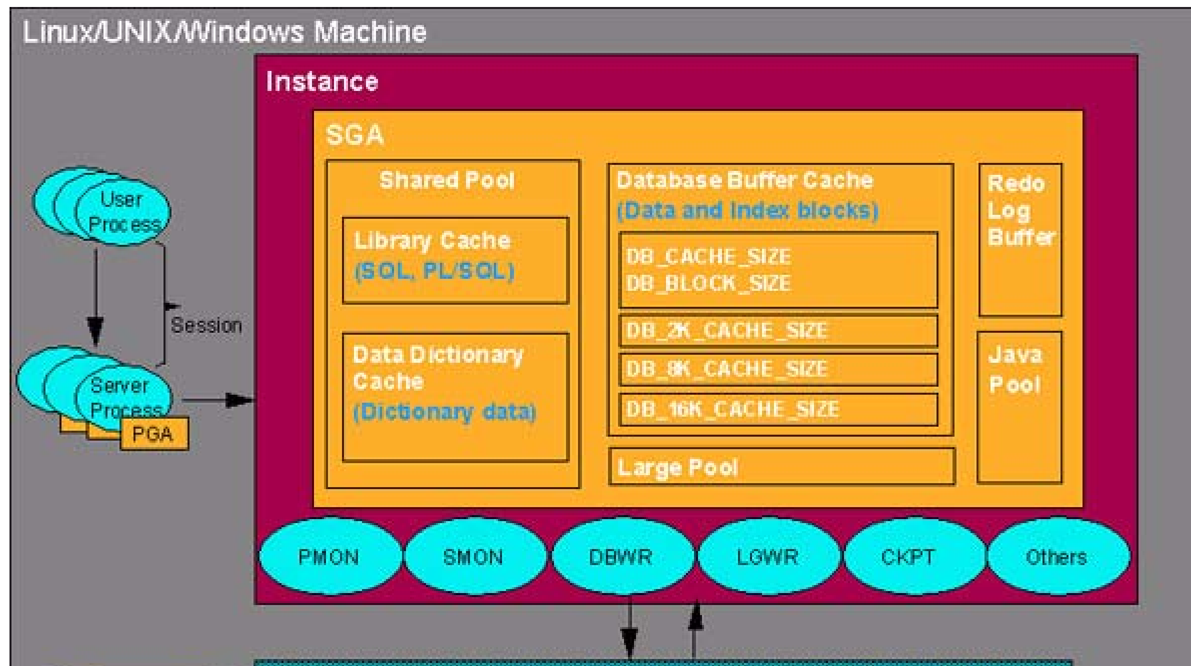
- Use a LRU queue
  - Page is used
    - Search pageID in queue
    - If found, delete from queue
    - Push pageID on top of queue
  - Page needs to be replaced
    - Pop pageID from bottom of queue




- Alternative – CLOCK ("2nd chance" caching)
  - Use "Used" bit B in each page
  - Page is used – set B:=1
  - Page needs to be replaced
    - Cycle through pages
    - If used=1, change to 0 and move pointer to next page
    - If used=0, replace this page and move pointer to next page



# Many Tasks Compete for Main Memory

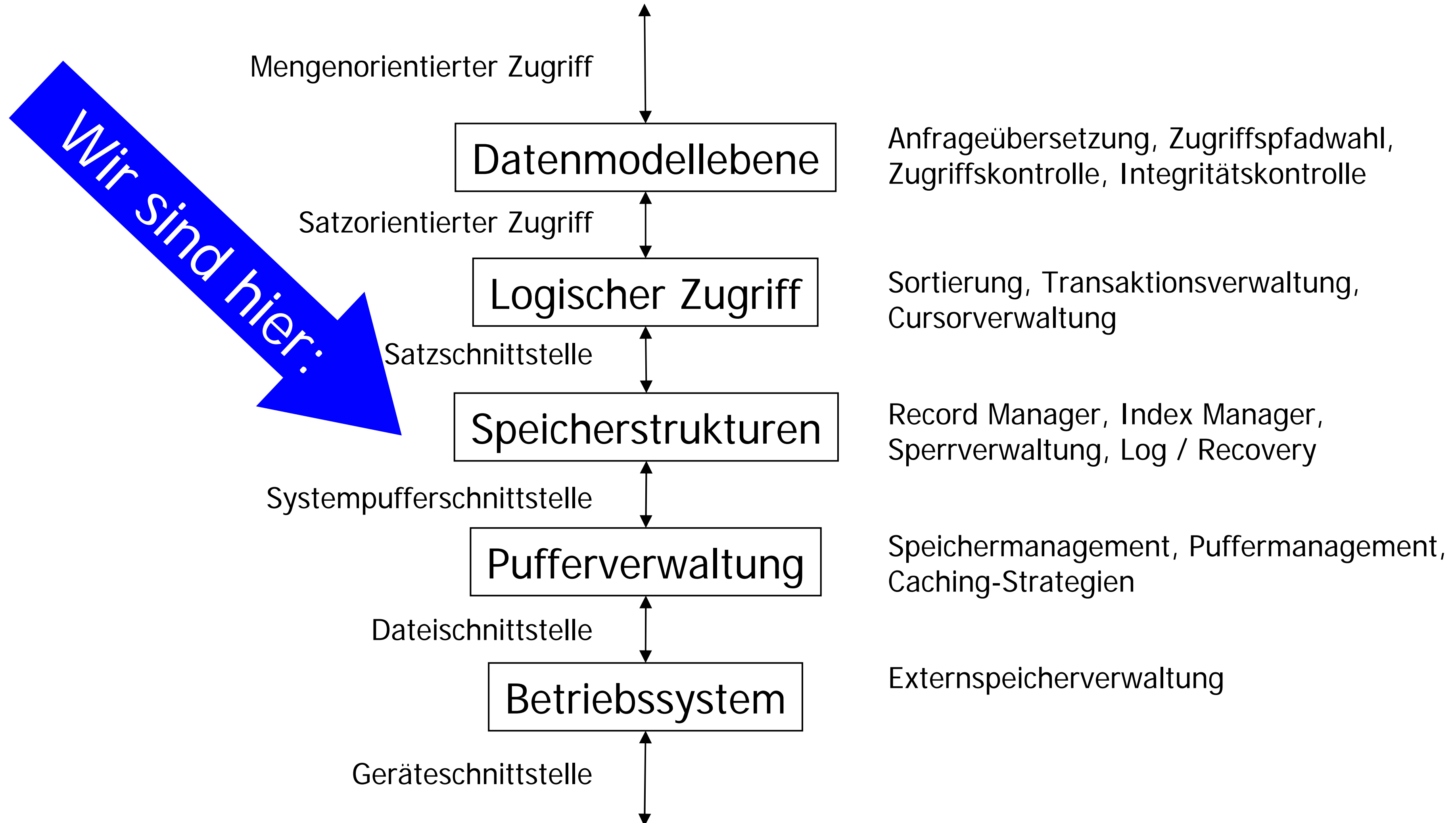



- SGA: System global area
  - Processes communicate through SGA
  - Requires locking of main memory structures – latches
- Library cache: buffers SQL prepared statements using LRU
- Java pool: area for java stored procedures
- Each process additionally gets its PGA (process global area)
- Each area is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

# Content of this Lecture

- Caching
- File structure
  - Heap files
  - Sorted files
- Some indexing
  - Index files
  - Index sequential files
  - Hierarchical index sequential files
  - (B tree)

# 5 Schichten Architektur

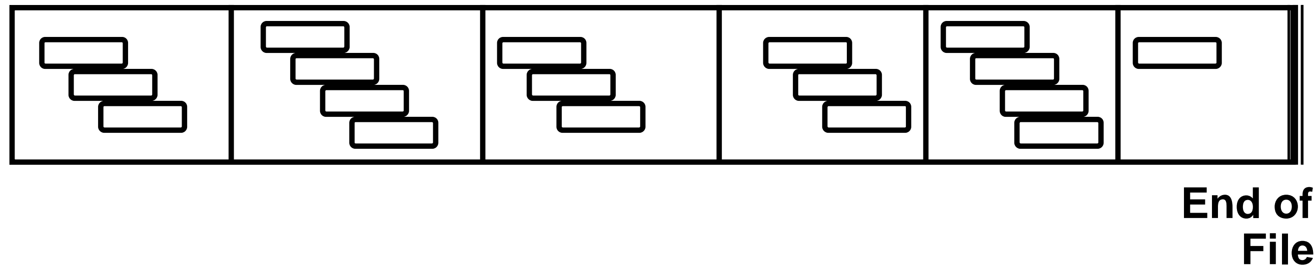


# Files and Storage Structures

- We have
  - Records are stored in blocks
  - Blocks are managed/cached by the buffer manager
  - Access to records by TID possible through buffer manager
- Usually, we want records with certain properties
  - SELECT * FROM COSTUMER
    WHERE Name = "Bond"
  - SELECT * FROM ACCOUNT
    WHERE Account# < 1000
- There must be more clever ways than scanning all records...

# Sequential (Heap) File

- Records are stored sequentially according to the order of insert operations




- Given:
  - Number of records: n
  - Number of records per block: R
- Minimal number of blocks : $b = \lceil n / R \rceil$
  - Usually better to keep some space free for growing records
    - Depends on read/write ratio
  - "Holes" appear if records are deleted

# Operations on Heap Files

- Search with value
  - b/2 block IO in case of success (average)
  - b block IO in case of failure
- 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
  - b/2: for successful search and no insert (average)
  - b+1: in case of search without success and insert
- Change / delete one record based on value
  - b/2+1: in case of success (average)
  - b: for searching without success

# Deleting Records

- First issue: file reorganization
  - Move records in block to gather larger chunks of free space
  - Possibly, check neighbors in case of underflow and remove blocks
- Second issue: dangling pointers
  - Existing references need to be updated
    - Especially indexes
  - Option 1: Update references at once
    - Requires to keep a list of all references
    - Single record deletion results in multiple physical deletions
  - Option 2: Use tombstones
    - Only mark record as deleted
      - E.g. null in block directory
    - References are updated only when used
    - Very fast when deletions are rare events

# Sorted file

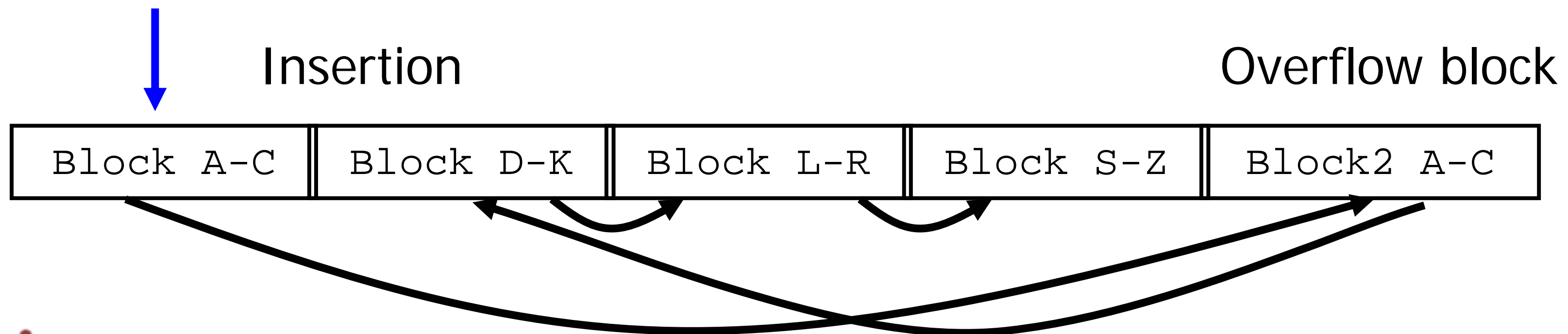
- Sort records in file according to some attribute
  - Faster searching when this attribute is search key
  - More complex management – order must be preserved
- Operations and associated costs
  - Search (using binsearch on blocks)
    - log(b) block IO; searching in block is free as always
  - Change / delete record based on value
    - First search in log( b)
    - Write changes / mark space as free
  - Insert record
    - First search in log(b)
    - Then do what??

# Inserting in a sorted file

- General: Reserve free space in 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
  - Won, costs 1 IO for writing
- Option 2: Check neighbors
  - See X blocks down and X blocks up the list (usually X=1)
  - When space is found, some records need to be moved
  - Cost: depends on how far we need/want to look
  - Beware of pinned records – update references
- Option 3: ??

# Inserting in a sorted file 2

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



# Improving Sorted Files

- Disadvantage of sorted files
  - Administration cost of keeping order
  - Only one search key
    - Searching on other attributes requires linear scans
  - Search time grows logarithmically with b
    - For 10.000.000 records, we already need ~23 IO
  - Can we do better??

# Improving Sorted Files

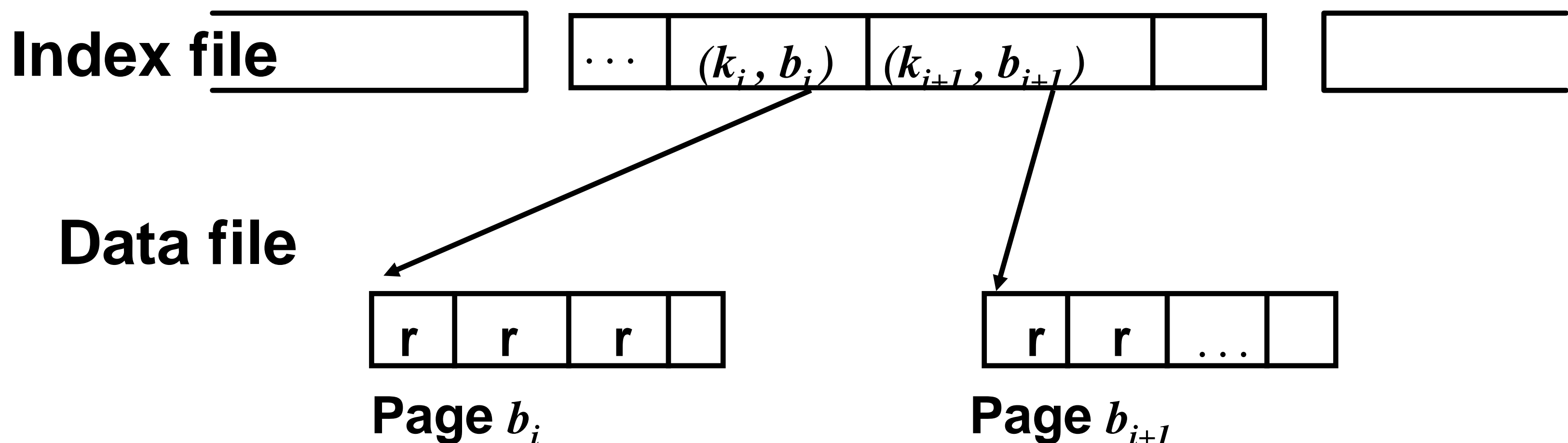
- Be better than logarithmic
  - If distribution of data is known, build histograms
    - Partition attribute range into buckets
    - Count number of keys in each bucket
  - Searching: estimate position of search key in file using buckets
  - Advantages
    - For uniformly distributed data, very good access time
      - Telephone book
    - Small space consumption when few buckets are used
      - But: the more buckets (higher granularity), the better access time
      - Tradeoff – buckets also require main memory
  - Disadvantages
    - Histograms (statistics) need to be maintained
      - Statistics become bottleneck for update operations
    - Only possible for one attribute (file still need to be sorted)
    - Choosing bucket number is very difficult
      - Buckets of unequal size – more administration
      - Finest granularity: size of index, but still only probabilistic gains

# Improving Sorted Files

- Decrease b
  - Use additional index file
    - Stores only search keys and TIDs
    - Searching: Binsearch in index, then additional IO for accessing data block
      - Only better if "additional IO" is not getting too much
  - Advantages
    - Data file need not be sorted any more
    - Due to smaller records, $b_{index} < b_{records}$
      - Considerable faster access
    - Indexes files can be build on several attributes
  - Disadvantages
    - More files to manage, to lock, to recover
    - Generates TID pointers – pinning becomes more severe
    - Potentially slower range queries than with sorted file
      - We quickly find the TIDs
      - But additional IO for accessing the data

# Improving Sorted Files

- Decrease b even further: index sequential files
  - Data file has records sorted on key
  - Additional index file stores (first key, pointer) pairs for each data blocks
  - Index record ($k_i$, b-ptr ): For all k in b-ptr : $k_i \leq k \leq k_{i+1}$
  - Sparse index: Not every search key is represented



# Index-Sequential Files

- Searching
  - Search key in index file using binsearch
  - Load data block and search in block
- Advantages
  - Not every key represented: $b_{index} << b_{records}$
    - Assume 10.000.000 records of size 200, |TID|=10, |search key|=20, block size=4096
    - Number of blocks b= 10.000.000*200/4096 = 500.000
    - Sorted file: log( 500.000) = 19 block IO
    - Index-seq file: log( 500.000*(10+20)/4096) = 12 block IO (+1)
  - Chances that index fits (mostly) into main memory
- Disadvantages
  - Only possible for one attribute
    - Data file need to be sorted
  - More administration

# Operations on Index-Sequential Files

- Initialize
  - Sort data file (including space on pages for future inserts)
  - Generate a sorted index file with b entries of type (k , b)
- Search for key k
  - Search in index file entry ($k_i$ , $b_i$ ) with : $k_i \leq k \leq k_{i+1}$
  - Search in page $b_i$ of data file
- Insert a record in data file
  - Search for page $b_i$ with $k_i \leq k \leq k_{i+1}$
  - If no space in block, 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
- Ideas for improving search further??

# Hierarchical Index-Sequential files

- Build a sparse, second-level index on the first-level index
  - All but the first level must be sparse; why??
  - First level may be spare or dense
- Advantages
  - Access time reduces further
    - Assume 10.000.000 records of size 200, |TID|=10, |search key|=20, block size=4096
    - Number of blocks b= 10.000.000*200/4096 = 500.000
    - Sorted file: log( 500.000) = 19 block IO
    - Index-seq file: log( 500.000*(10+20)/4096) = 12 block IO (+1)
    - With second level: log( 3600*(10+20)/4096) = 5 blocks IO (+2)
    - With more levels: log( log( log( ...))) = ... +X
  - Higher levels are very small – cache permanently
- Using more than 2 levels leads to B-trees (later)

# Multi-Level Index Files

Sparse 2nd level | Sparse 1st level | Sorted File

| Sparse 2nd level |
| --- |
| 10 |
| 90 |
| 170 |
| 250 |
| 330 |
| 410 |
| 490 |
| 570 |

| Sparse 1st level |
| --- |
| 10 |
| 30 |
| 50 |
| 70 |
| 90 |
| 110 |
| 130 |
| 150 |
| 170 |
| 190 |
| 210 |
| 230 |

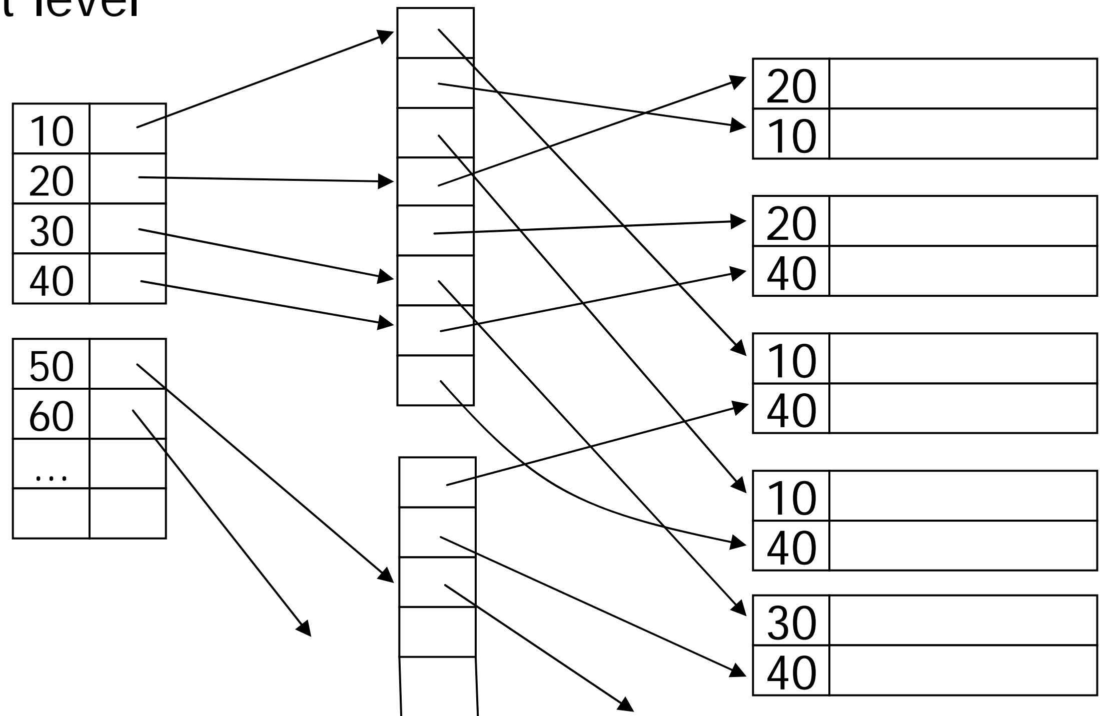
| Sorted File |
| --- |
| 10 |
| 20 |
| 30 |
| 40 |
| 50 |
| 60 |
| 70 |
| 80 |
| 90 |
| 100 |

# Index Files and Duplicates

- What happens if search key is not key of relation?
  - Values of keys may appear more than once
- 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
      - All records with same key must be one after the other in the data file
    - "Semi-sparse" index
- Index degradation
  - If only few distinct values exist, all keys are essentially the same
  - Semi-sparse index becomes much better
    - Extreme case: index on "retired" – two values, semi-sparse index has only two entries

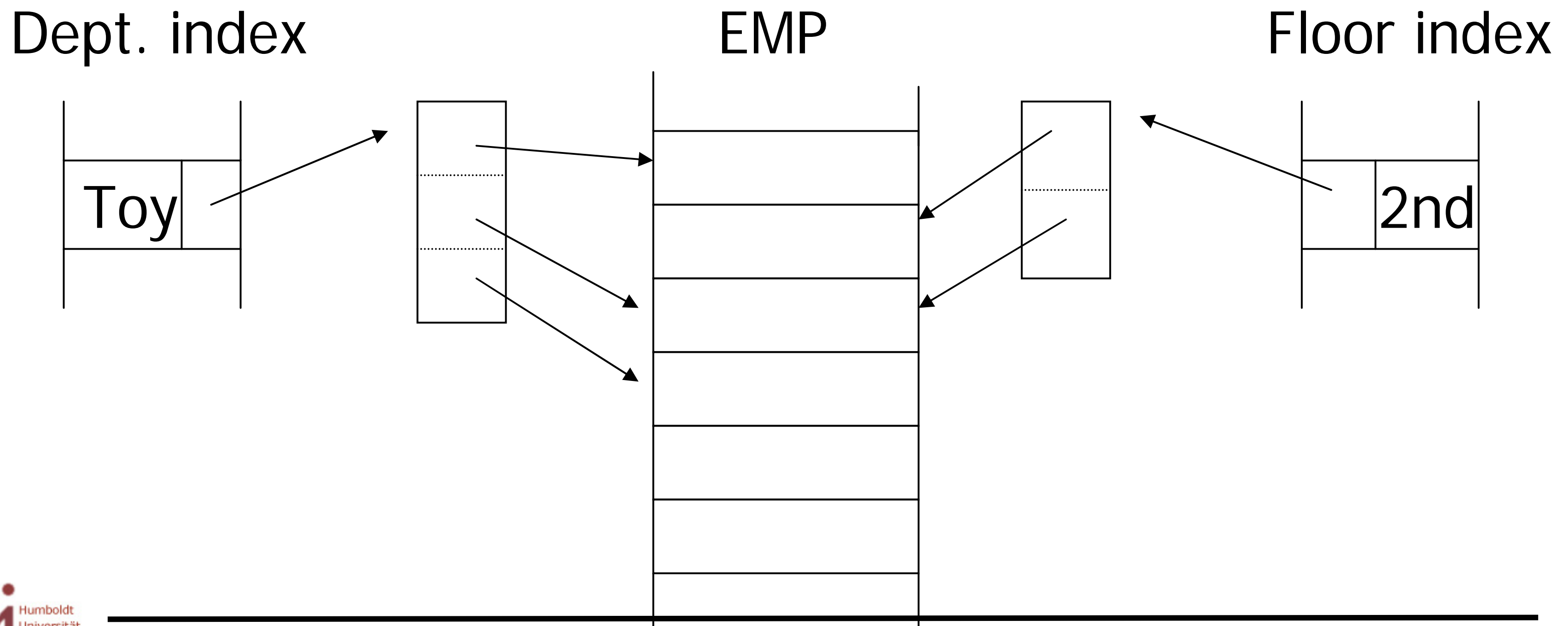
# Secondary Index Files

- Primary index: Index on the attribute on which the data file is sorted
- Secondary index: Index on any other attribute
  - Cannot exploit order of data file
  - Must be dense at first level (also for duplicates)
- Improvement: Use intermediate buckets
  - Buckets hold TIDs sorted by index key
  - Buckets do not store values



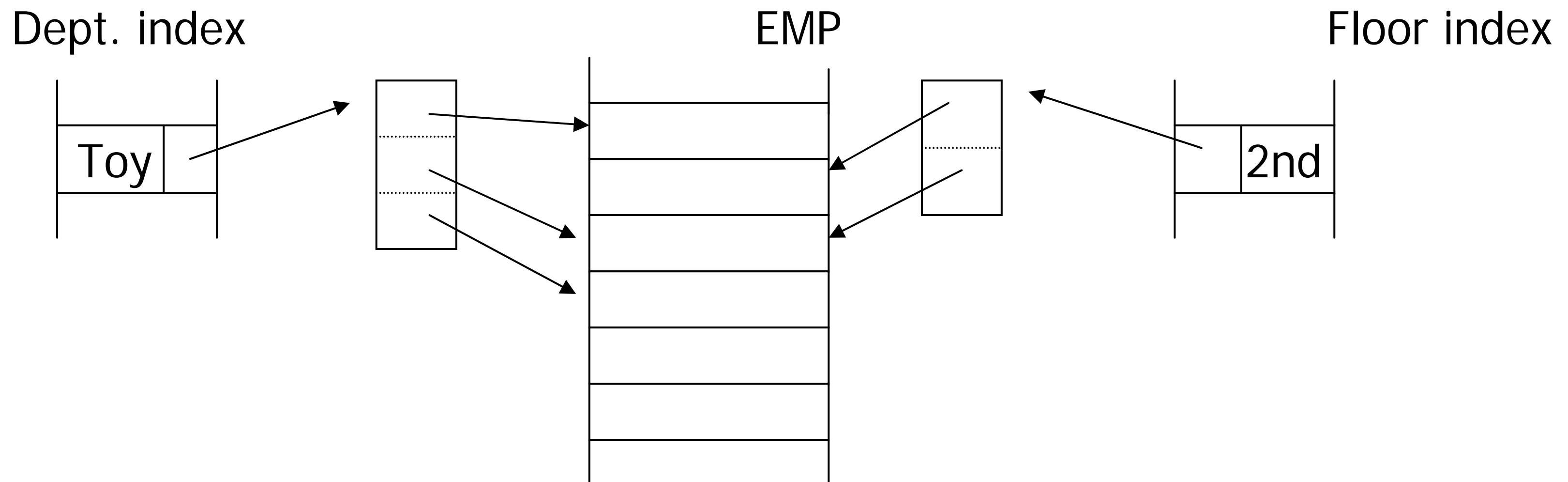
# Buckets for Secondary Index Files

- Buckets have advantages
  - Index stores keys and pointer to buckets; buckets only store TID
    - If many duplicates exist, this saves same space
  - Compute ANDed conditions by intersecting TID sets from buckets



# Buckets and Queries

- Query: "All employees in TOY dept. sitting on 2nd floor"
  - Use floor index to find TID set 1
  - Use department index to load TID set 2
  - Intersect both sets
  - Load employee data only for intersection
  - Advantage increases for 5,6,7,... attribute conditions (STAR join)



# Indexes in Oracle

- Per default: secondary B* tree indexes
  - Also R-tree (later) and bitmap index (DWH)
- Data files usually are heap files
  - Keeping order too expensive
- Exception: Index-organized tables
  - These are sorted files
  - 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 indexes
  - A cluster index is an index in a cluster – not a clustered index
    - E.g. cluster tables department and employee and index common attribute "dep_number"
  - There are no clustered indexes – use index-organized tables
- Can we store a table sorted on two different keys??

# Multiple Sorts

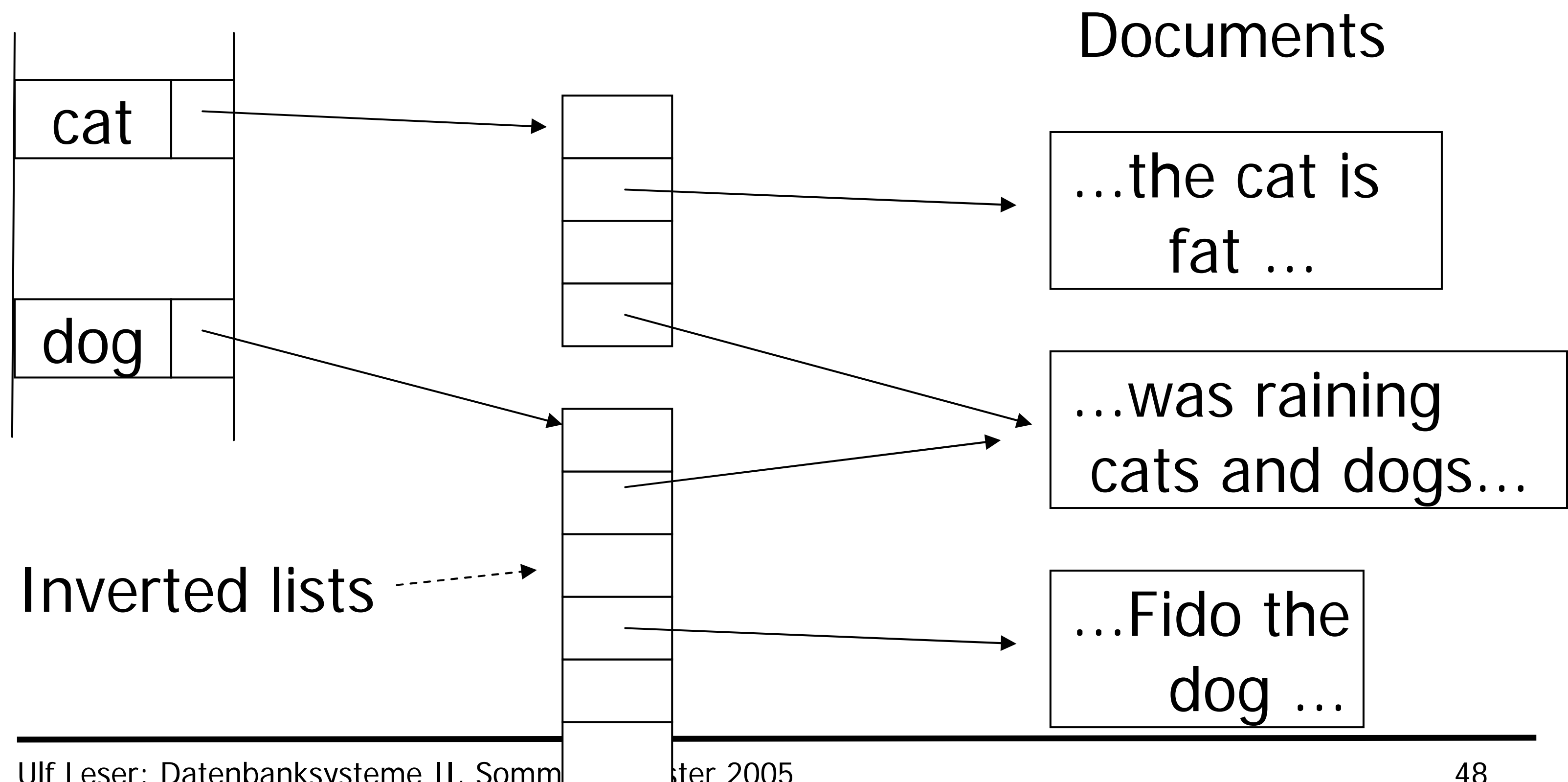
- Use primary index on sorted file
  - Oracle: index-organized file
- Build secondary index including all attributes of the table in desired second order
  - Example: employee (ID, name, dep#, income)
  - Create IOT employee (ID, name, dep#, income)
    - Sorted by ID
  - Create index on employee (name,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

# Excursion: Indexing Text

- Mainly 2 fields of informatics are concerned with searching
  - Databases
  - Information retrieval
- Information retrieval
  - Searching documents
  - Typically, each document is represented as “bag of words”
  - Queries search for documents containing a set of words
    - Google-style queries
- 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??

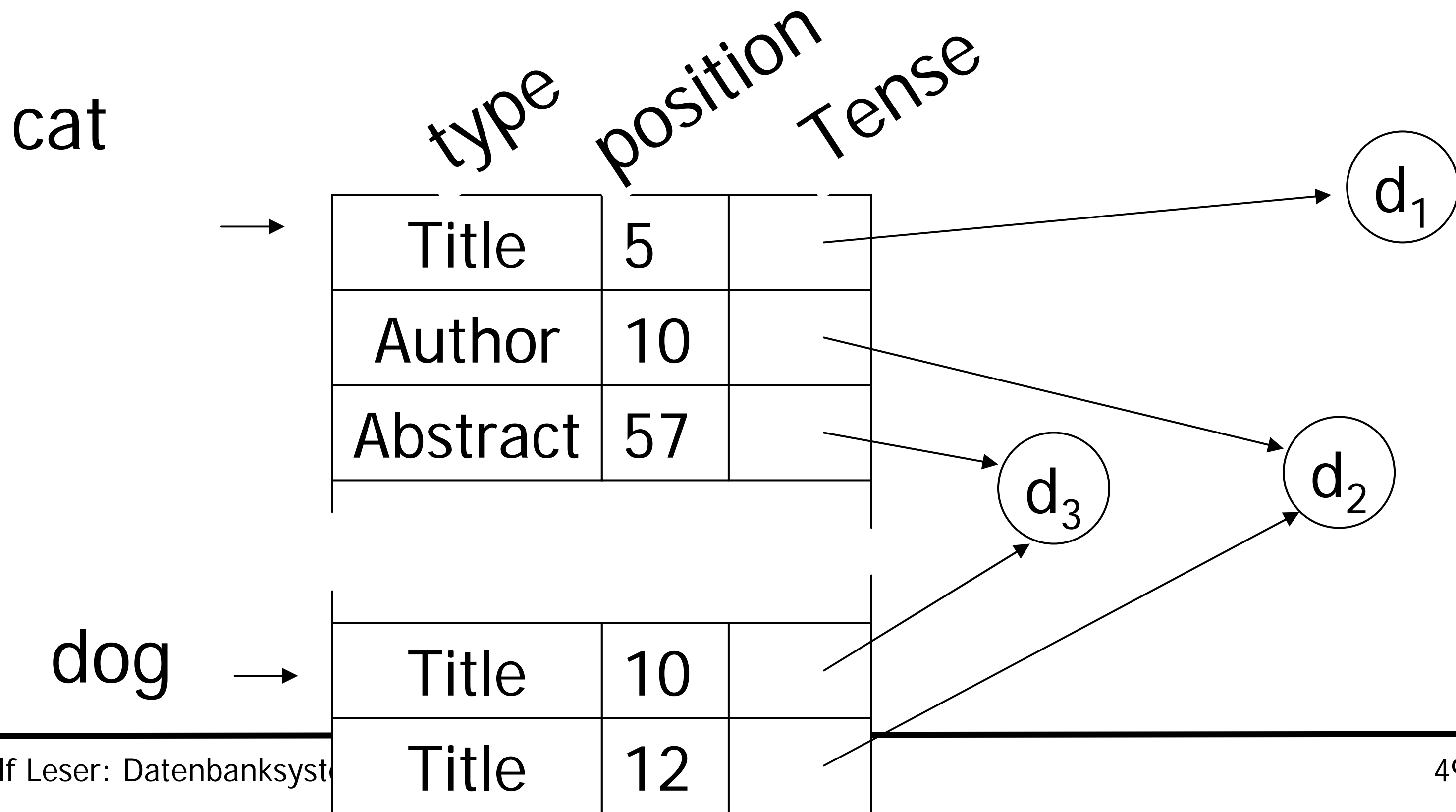
# Inverted Lists

- Build a secondary, bucketed index on the words
  - Indexing all words, i.e. parts of an attribute = impossible in RDBMS
  - Find documents by intersecting buckets
    - Also supports AND NOT or OR



# Improvements: Positional index

- Store position in sentence with each word
  - Allows for textual neighborhood queries
- Store other meta-information with each word
  - Location in document, grammatical info, …



# There is much more in IR

- Stop words
- Truncation and lemmatization
- Sentence splitting
- Document conversion
  - Including figures and tables
  - Preserving document structure
- Weighting of words
  - Vector space model
- Query expansion
  - Using thesauri
- …