

Datenbanksysteme II: Implementation of Database Systems

Records & Blocks



Material von
Prof. Johann Christoph Freytag
Prof. Kai-Uwe Sattler
Prof. Alfons Kemper, Dr. Eickler



Stellenausschreibung SHK

Institut für deutsche Sprache und Linguistik / Korpuslinguistik (Anke Lüdeling)
Studentische Hilfskraft, 40 Stunden/ Monat, 01.06.2005 bis 31.05.2006

Erstellung einer Website und Suche auf einem Lernerkorpus

Ein Lernerkorpus ist eine Sammlung von Texten, die von Lernern des Deutschen als Fremdsprache geschrieben wurden. Die Daten sind auf vielen Ebenen mit Fehlercodes ausgezeichnet und liegen in einem XML-Format vor (multi-level, standoff). Das Korpus wird weiter aufgebaut, d. h. die Hilfskraft wird in einem engagierten Team arbeiten.

Anforderungen:

- Interesse an der Korpuslinguistik und der Erstellung von Korpora
- Programmiererfahrung und Teamfähigkeit
- Kenntnisse in XML, HTML und CSS
- Kenntnisse in Perl oder einer vergleichbaren Skriptsprache
- Kenntnisse in der Programmierung von Web-Applikationen
- Kenntnisse in UNIX / LINUX

<http://www.linguistik.hu-berlin.designato.de/korpuslinguistik/index.php>

Warum die Speicherhierarchie?

	Primär	Sekundär	Tertiär
Geschwindigkeit	schnell	langsam	sehr langsam
Preis	teuer	preiswert	billig
Stabilität	flüchtig	stabil	stabil
Größe	klein	groß	sehr groß
Granulate	fein	grob	grob

- Je schneller, desto teurer
- **Platzbedarf** – Chip, Platine, Gehäuse, eigenes Gerät, Bandroboter
- **Begrenzung Adressraum**: 32 bit = 4 Gigabyte
 - Hat ein Ende: 64 bit ~ $2 \cdot 10^{10}$ Gigabyte
 - Kostenunterschied bleibt bestehen
- Tertiärspeicher oft nicht mehr günstiger als Sekundärspeicher
 - Aber beliebige Erweiterbarkeit durch Wechselmedien (CDs, Tape)
 - Kein Controller-Engpass

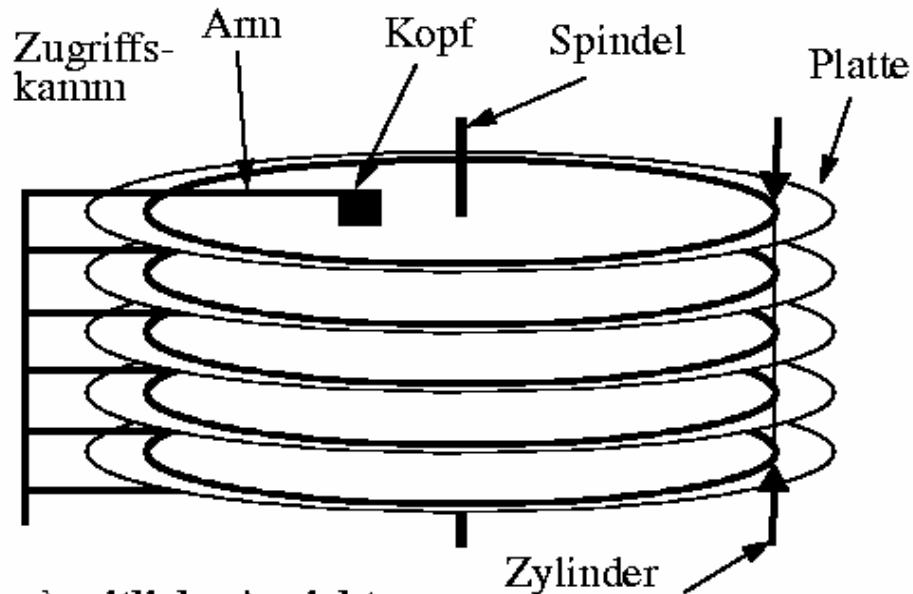
Magnet- bzw. Festplattenspeicher

Aufbau

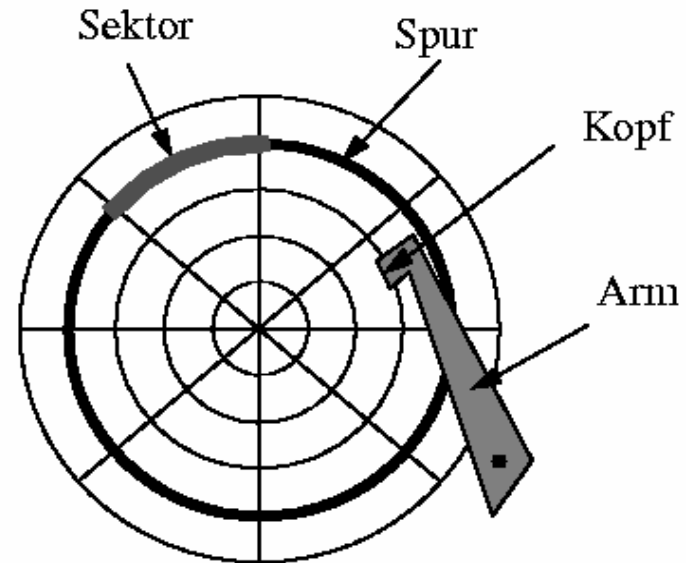
- mehrere gleichförmig rotierende Platten, für jede Plattenoberfläche ein Schreib-/Lesekopf
- jede Plattenoberfläche ist eingeteilt in Spuren
- die Spuren sind formatiert als Sektoren fester Größe (Slots)
- Sektoren (typischerweise 1 - 8 KB) sind die kleinste Schreib-/Leseinheit auf einer Platte

Adressierung

- Zylindernummer, Spurnummer, Sektornummer
- jeder Sektor speichert selbstkorrigierende Fehlercodes; bei nicht behebbaren Fehlern erfolgt automatische Abbildung auf Ersatzsektoren



a) seitliche Ansicht

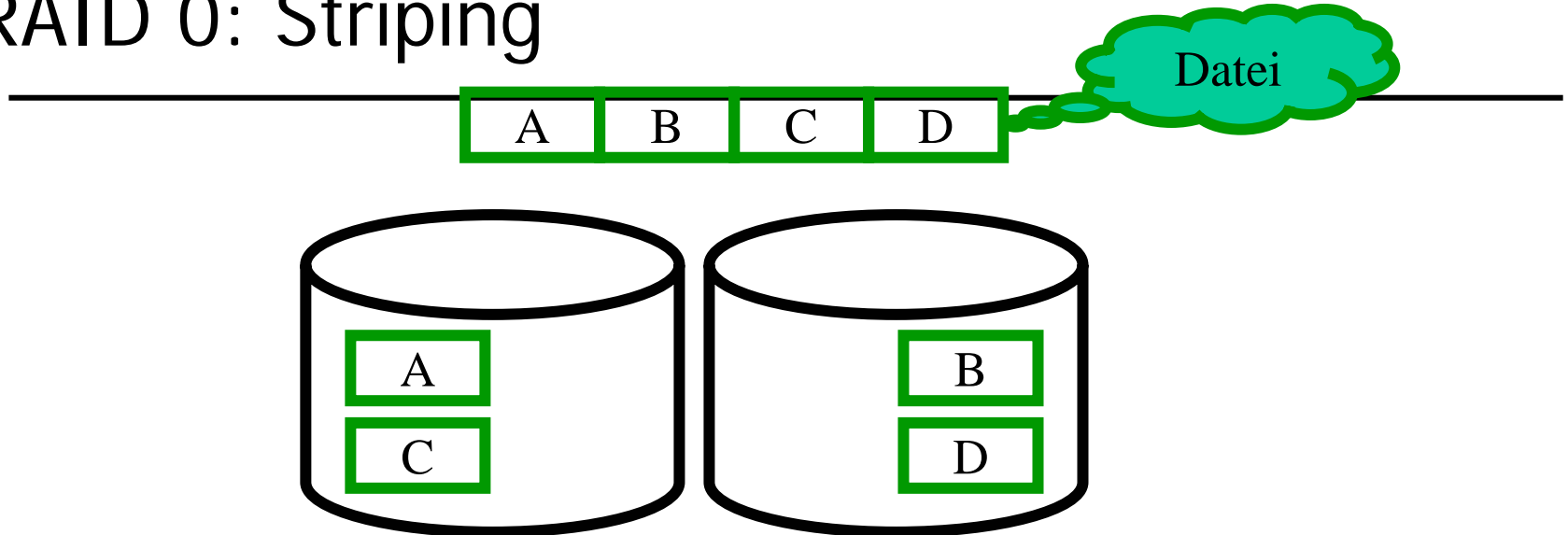


b) Draufsicht

Random versus Sequential IO

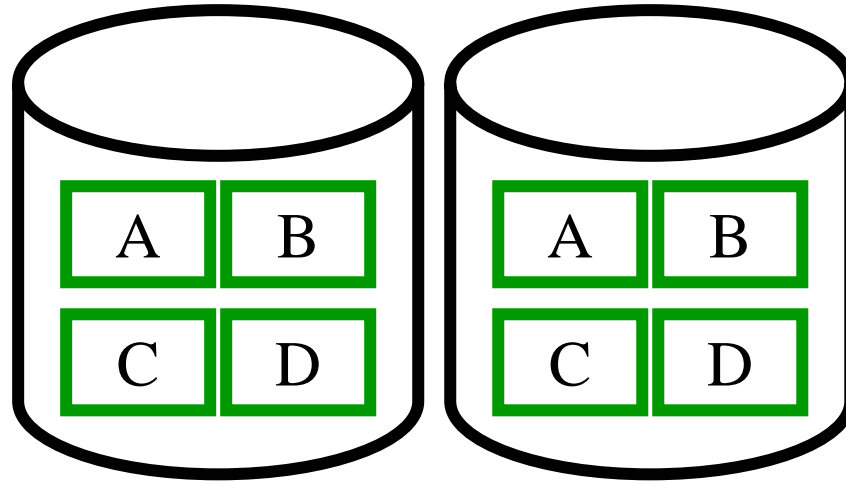
- Aufgabe: 1000 Blöcke à 4KB sind zu lesen
 - $T_s = 5\text{ms}$, $T_r = 3\text{ms}$, $U = 15\text{ MB/s}$
- Random I/O
 - Jedes mal Arm positionieren
 - Jedes mal Latenzzeit
 - → $1000 * (5\text{ ms} + 3\text{ ms}) + \text{Transferzeit von 4 MB}$
 - → $> 8000\text{ ms} + 300\text{ms} \sim 8\text{s}$
- Sequential IO
 - Einmal positionieren, dann nur noch sequentiell lesen
 - → $5\text{ ms} + 3\text{ms} + \text{Transferzeit von 4 MB}$
 - → $8\text{ms} + 300\text{ ms} \sim 1/3\text{ s}$
- Unterschied: **ein bis zwei Größenordnungen**
 - Wichtige für Query Optimierung
 - Index bei $>5\%$ Anfrageselektivität nicht mehr benutzen

RAID 0: Striping



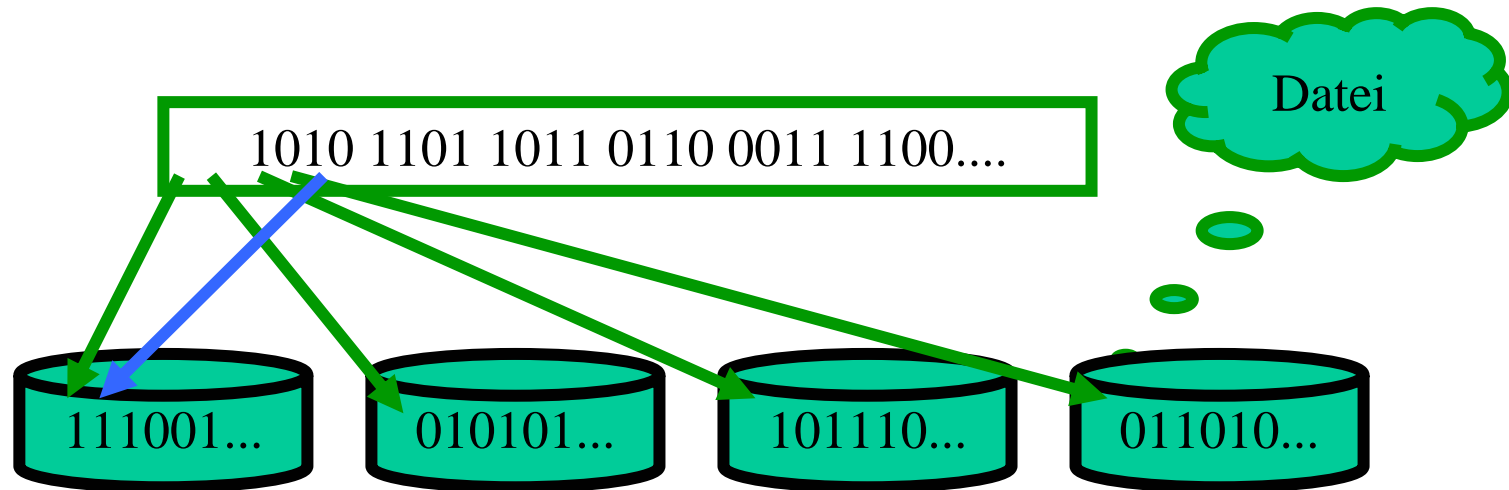
- **Doppelte Bandbreite** beim sequentiellen Lesen der Datei
 - Zyklische Verteilung der Blöcke, wenn alle Blöcke mit gleicher Häufigkeit gelesen/geschrieben werden
 - Alternative: Verteilung nach Zugriffshäufigkeit (Optimierungsproblem)
 - Zugriffshäufigkeit aber i.d.R. nicht bekannt
- Keine Beschleunigung für Lesen eines einzelnen Blocks
- **Datenverlust** wird immer wahrscheinlicher, je mehr Platten man verwendet

RAID 1: Spiegelung (mirroring)



- **Datensicherheit** durch Redundanz aller Daten
 - Keine Hilfe bei Bitfehlern – wer hat recht?
- Doppelter Speicherbedarf
- Lastbalancierung beim Lesen (wie RAID0)
- Außerdem „konkurrierendes“ Lesen einer 1-Block Datei möglich – der schnellere Kopf gewinnt
- Beim Schreiben müssen beide Kopien geschrieben werden
 - Kann parallel geschehen

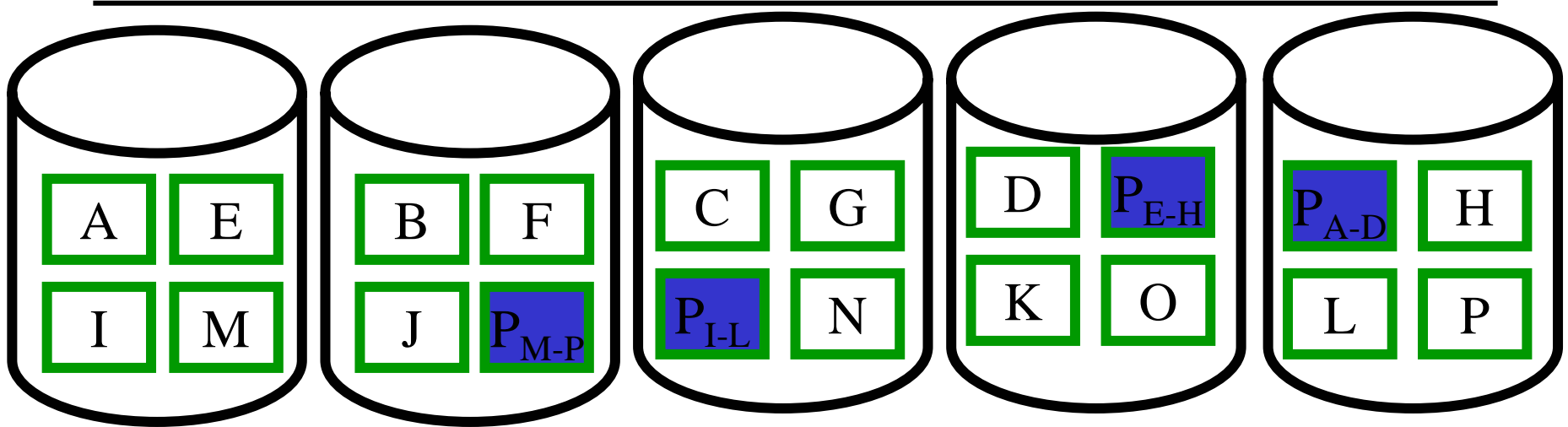
RAID 2: Striping auf Bit-Ebene



- Striping auf Bit- (oder Byte-) statt Blockebene
- Idealerweise höherer Durchsatz schon beim Lesen **einzelner Blöcke**
 - Aber: Lesen eines Sektors dauert i.d.R. genauso lange wie Lesen 1/8 Sektors
 - Also muss Aufteilung von **Blöcken auf Sektoren** beachtet werden
 - Typisch: 8-16 KB Blöcke, 512 Byte Sektoren = 16-32 Sektoren pro DB Block
- Keine Beschleunigung für mehrere parallele Leseoperationen
 - Jedes Lesen/Schreiben braucht **alle Platten**
- Verschlechterte Ausfallsicherheit



RAID 5: RAID4, Verteilung der Parity



- Parityblock immer auf der Platte, die keinen der Par-Blöcke enthält
- Wesentlich bessere Lastbalancierung als bei RAID 4
 - Paritätsplatte kein Flaschenhals mehr
 - Schreiben erfordert X-1 Platten für die Daten und 1 weitere Platte für Parity
- Wird in der Praxis häufig eingesetzt
 - Typischerweise langsamer als RAID10
 - Dafür deutlich Platz sparender
- Guter Ausgleich zwischen Platzbedarf und Leistungsfähigkeit



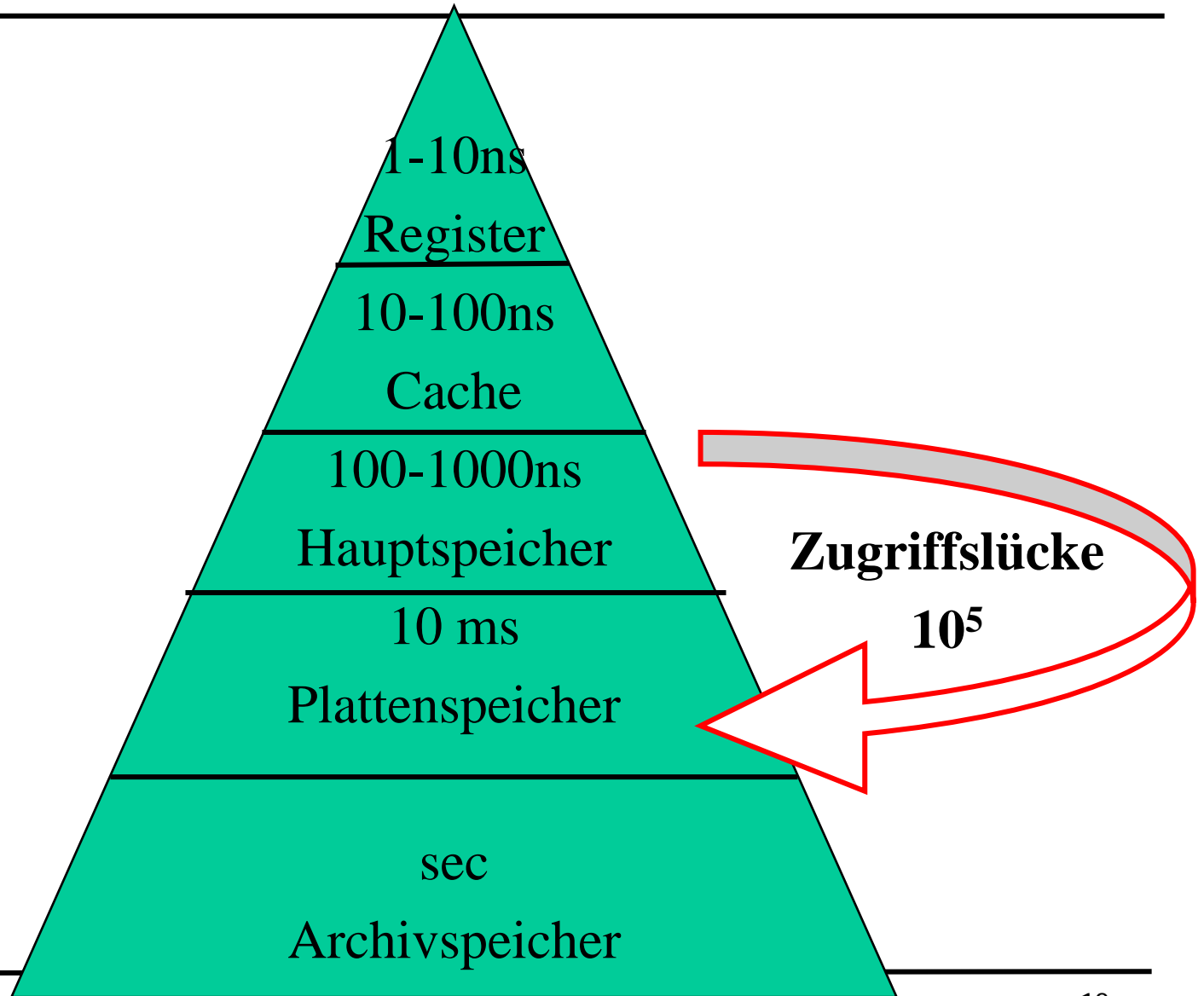
In reality it is more complex

- Software / Hardware RAID, Database RAID, Volume manager, SAN
- Parallelization by **distributing tablespaces**
 - System tablespace on separate disk
 - Or: locally, tablespace-managed data dictionary
 - Separate data and index tablespaces
 - Separate disk for REDO Logs
- Parallelization by **distributing one tablespace**
 - Distribute different files in one tablespace on different discs
- Parallelization by **distributing a single table**
 - Distribution of segments
 - **Partitioning** - “semantic” distribution of table data
 - All sales prior to 2005 on one disk, all sales this year on another disk
 - One disk for sales in 2005, 2004, 2003, ...
- Complex dependencies

Content of this Lecture

- Secondary management revisited
- Records, pages, segments
- Referencing tuples
- BLOBs and free space lists
- Oracle block structure

Schon wieder: Speicherhierarchie



Consequences

- 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 $O(n)$

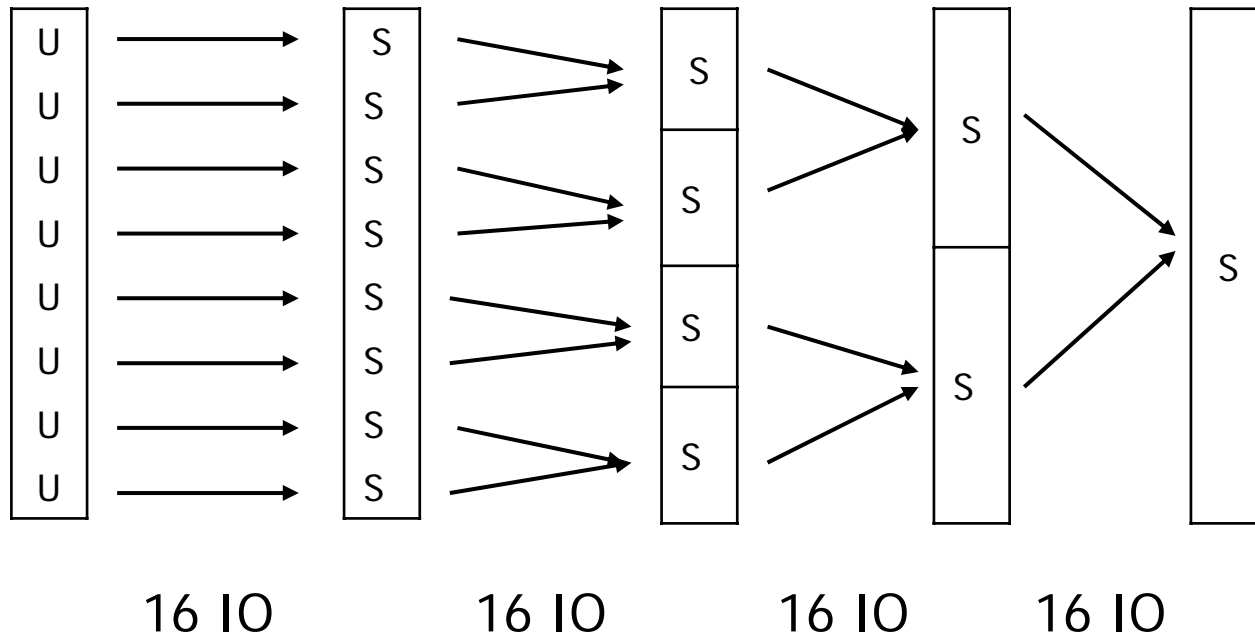
1	—	3		1
2	—	5		2
4	—	6		3
7	—	10		4

- 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: $O(n \cdot \log(n))$
 - Number of key comparisons

Example cont'd

- IO model of Merge-Sort
 - We don't consider caching here
- Of course, two sorted lists on disc consisting of n blocks can be merged in $O(n)$ IO operations
 - Read first blocks of each list (2 IO)
 - Merge both sorted blocks into one output block (0 IO)
 - If end of one input block is reached, read next block (1 IO)
 - If output block is full, write to disc (1 IO)
 - In total, each block is read and written once – $2 \cdot n$ IO
- Let's apply the recursive algorithm

Recursive merge-sort

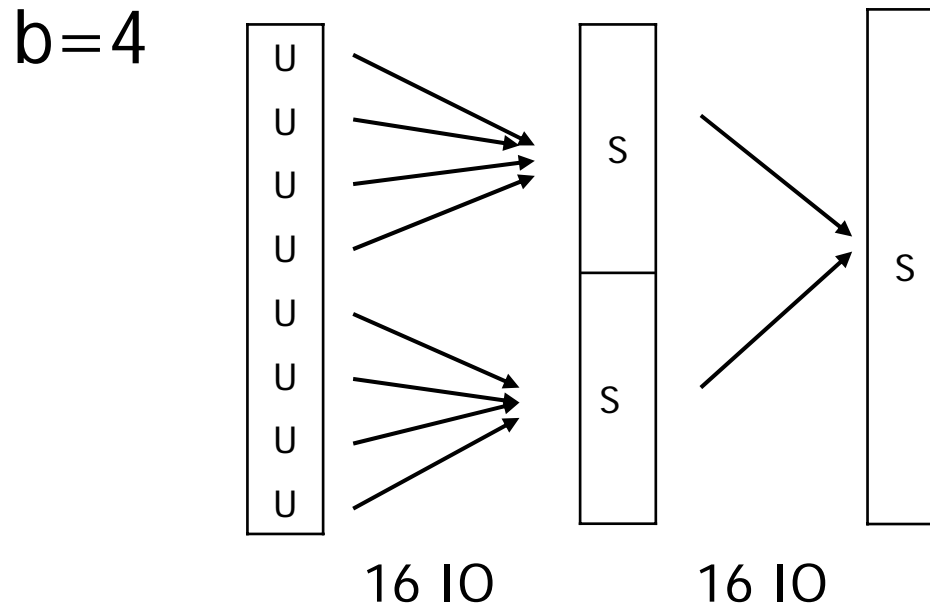


- Total IO: $2 * n * \log(n)$
 - n: Number of blocks
- Can we do better??

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

Blocked merge-sort



- $b=4$: Total IO 32 block reads/writes
- $b=3$: ??
- $b=2$: ??
- $b=8$: ??

Example cont'd

- Total IO: $4n$
 - If second phase necessary
- But there is a limit here
 - During Merge phase
 - We need to have **many files open** at a time
 - Example: $n=1024$ blocks, $b=2$
 - Generates 512 runs of size 2 each
 - We probably cannot open 512 files in parallel
 - We need to hold $x+1$ records in main memory
 - If b is small, we might not be able to hold 512 keys in main memory at a time

- Extension??

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
- If number of tuples is limiting, algorithm needs to be adapted
- Now also consider difference between random / sequential access – further improvements??

Example cont'd

- Don't read/write blocks always one-at-a-time
- Always treat **sequences of consecutive blocks**
 - Merge two sorted lists by every time reading **$b/3$ blocks of each file**
 - Two third for reading, one third for writing
 - Only read another $b/3$ blocks, when first exhausted
 - We might have already written one sequence in the meantime
 - Write $b/3$ blocks in one sequential write
 - Merge x runs by every time reading $b/(x+1)$ blocks of each run
- Further optimization (for two file case)
 - Use **non-blocking, asynchronous disc IO**
 - Divide each third in two partitions
 - Work with one partition; when done, issue IO request and continue with other partition while IO is performed
 - Considers that **main memory operations are not really free**

Recall – Ignoring IO cost is a bad idea

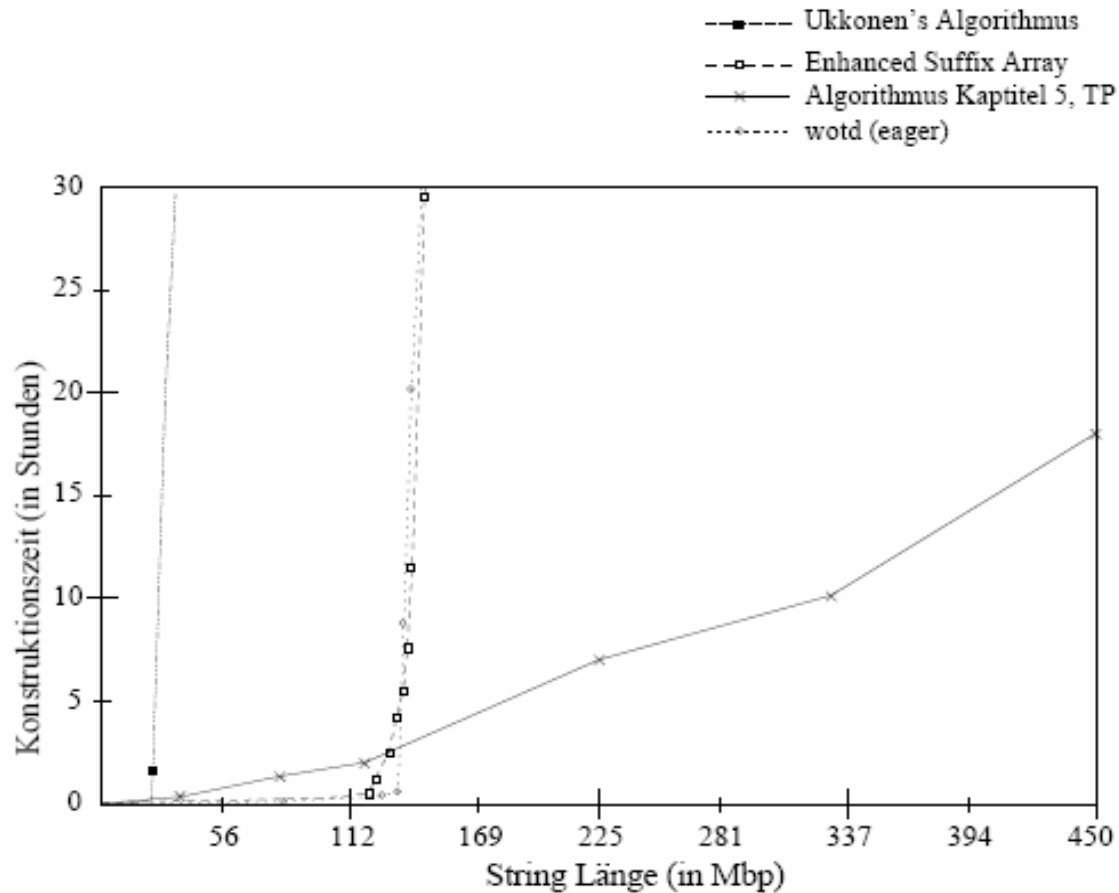
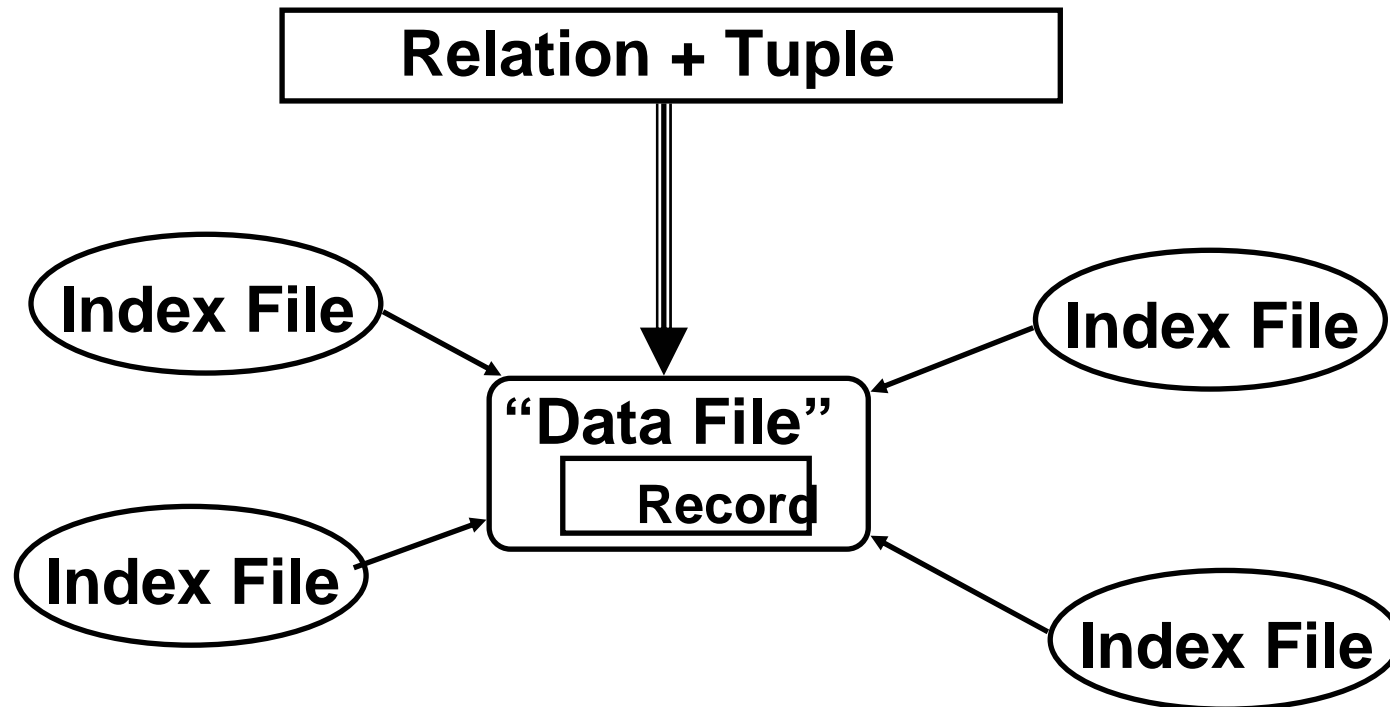


Abbildung 6.3: Entwicklung der Laufzeiten im Vergleich zu anderen Algorithmen

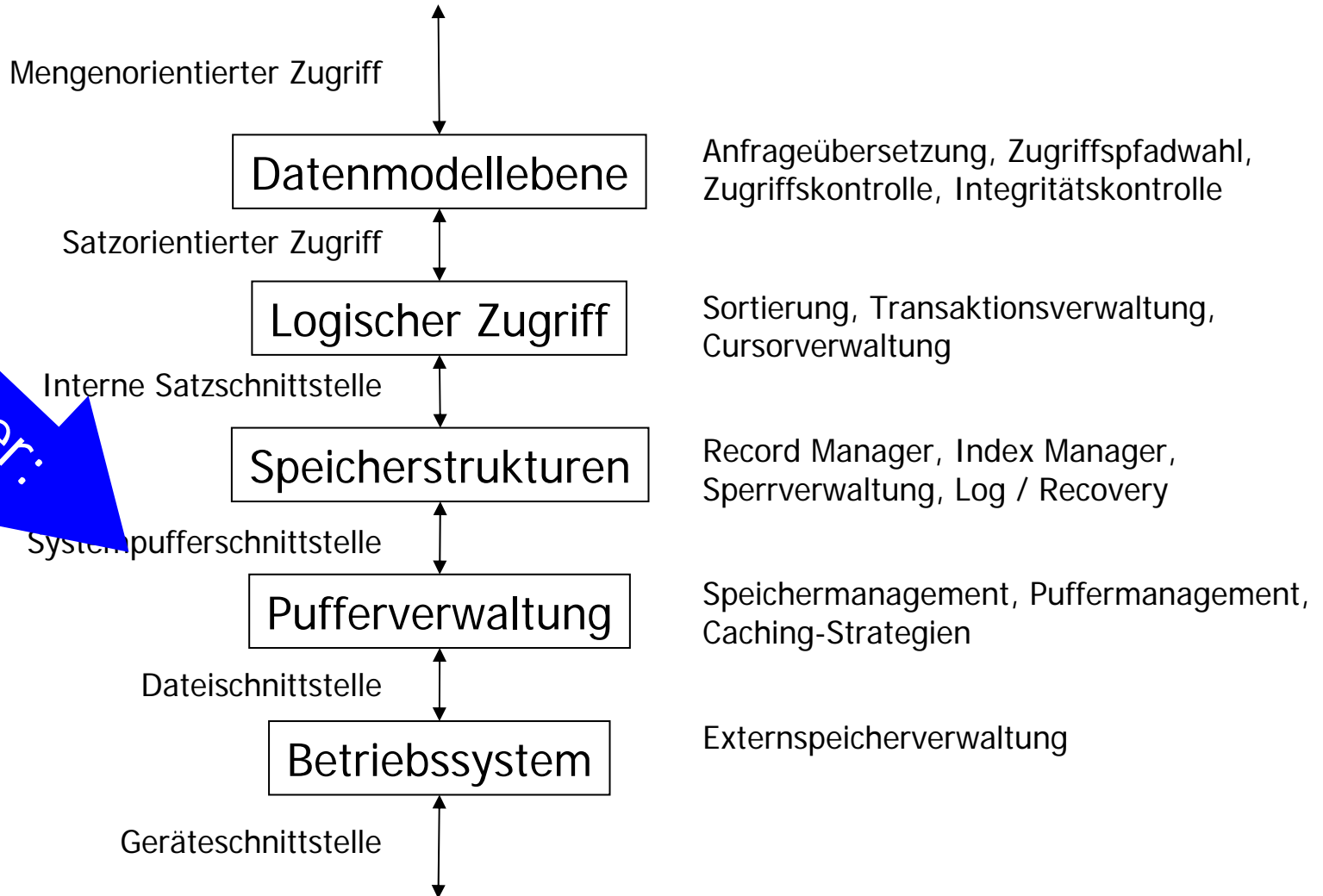
Storing relational data

- Fundamental elements of storage in RDBMS
 - **Records**, consisting of
 - typed **attributes** or fields
- Tasks
 - Mapping records to pages
 - Mapping pages to OS-blocks
 - Skipped; usually, X pages form one block
 - Mapping OS-blocks to sectors
 - Now skipped; performed by disc controller
 - Reading/writing sectors (and hence records) from/to disc
- Plus clever tricks to speed things up
 - How is the data structured inside a block?
 - Indexing, hashing, ...
 - Discussed later

Physical Representation

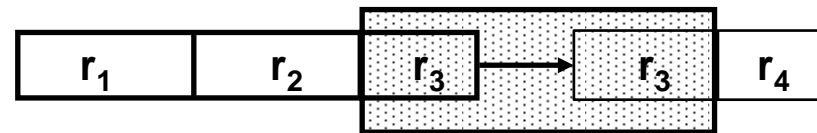


5 Schichten Architektur



From records to pages

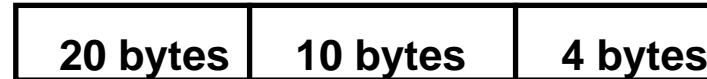
- Tuple = Record
 - Fixed length
 - Variable length
- Mapping of records to pages:
 - “Spanned Record”:
 - “Unspanned Record”:



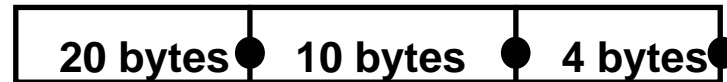
- **Avoid spanning record**
 - Requires two read operations
 - Transaction management on block level much more difficult
 - But leads to better space utilization
- It still happens
 - Tuples larger than block size: BLOBs, CLOBs, STRING fields
 - Update of short tuples where additional space not available in block

Record Formats

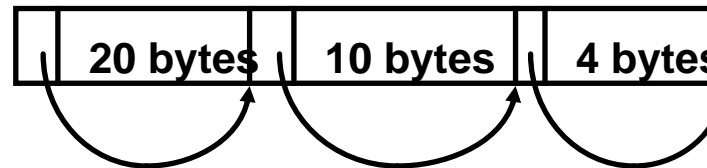
- Fixed length records



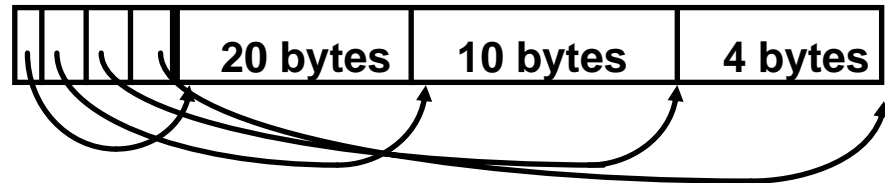
- Variable length records
 - Mark end of attributes



- 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 not negligible)
 - Cost for locating tuple by pointer (instead of computing address) dominated by IO

Storing NULL's

- NULL has **special semantics**
 - $Z := \text{NULL}$; then, the following is not the same
 - if (z) then XXX else YYY;
 - If (z) then XXX; if (not Z) then YYY;
 - Not at all the same: $Z = ""$ and $Z = \text{NULL}$
 - Purposefully no value given versus never thought of a value
- The **many meanings of NULL**
 - Not known: Age of person
 - Not defined: Driver license of child
 - No value: Price of car not on sale
- Storing NULLs in record fields
 - Problems with strings – separate from empty string
 - Fixed length, with end marks, length indicator
 - Use **special (and otherwise unused) binary characters**
 - Record dictionary: set pointer to NULL

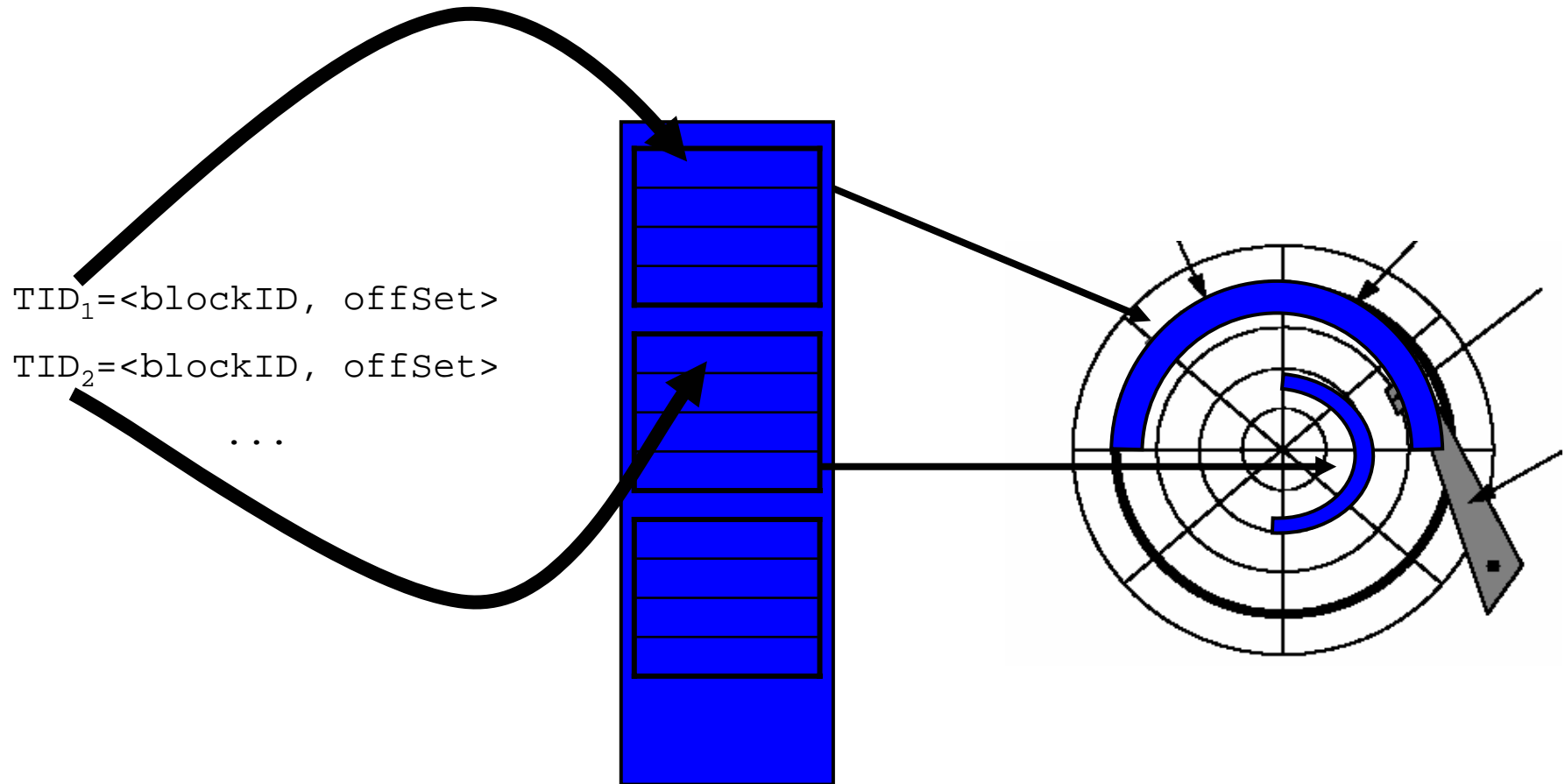
Storing Records on Disc

- Tuples need to be addressable in a **stable manner**
 - References from indexes
 - Overflow records (spanned records)
- Table on disc consists of blocks/extents
 - Extents (multiple blocks) are reserved if table grows
- Record are identified by **tuple ID (TID)**
 - Block number
 - Address of tuple in block
- When requesting a TID
 - See if block is in buffer
 - If not, request block from disc
 - If necessary, free block space in buffer first
- Difference foreign key – TID??

TIDs and foreign keys

- Foreign key is a logical value, TID is a physical address
 - FKs are looked up in index
 - Gives TID – block/data is loaded
- Foreign key is a value visible for developer, TID (usually) not
 - Never use TIDs as references – might change
- Foreign key is an integrity constraint, not a pointer
 - May join foreign key with any other value in the database as well
- Foreign keys are used instead of pointers (TIDs) to allow for moving records with little adaptation
 - Only DB-managed pointers need to be considered

Storing Records on Disc

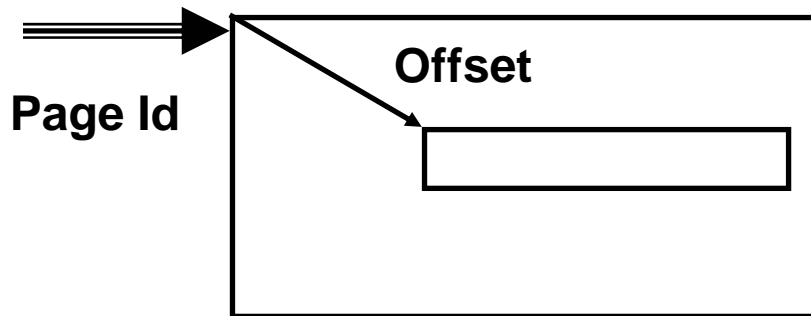


Tuple Identifiers

- Requirements
 - TID must be **unique and invariant**
 - Uniqueness for unique identification
 - Invariance for keeping references alive
 - E.g. indexes (“dangling pointer” problem)
- **“Pinned” record**
 - Record is at physical location
 - References to this record exist via physical location
- **“Unpinned” record**
 - No **physical reference** exists
- Aim: decouple TID from physical location
 - Makes tuples **moveable within and across pages**
 - Necessary for improving management of free space

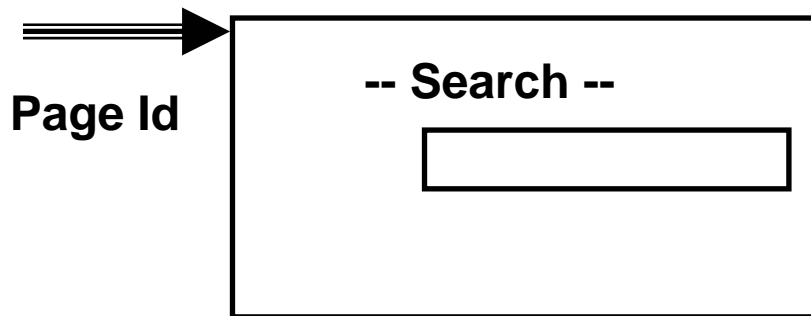
Record Addressing

- Mapping alternatives
 - absolute addressing: $TID = \langle PageId, Offset \rangle$



Good: direct access
Bad: no de-coupling

- absolute addressing + search: $TID = \langle PageId \rangle$

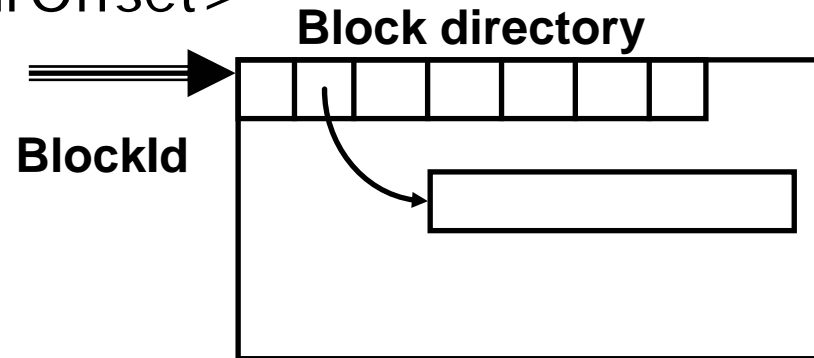


Good: not much
Bad: - searching necessary
- no de-coupling

Record Addressing (cont.)

- **Block directory** (tuple table):

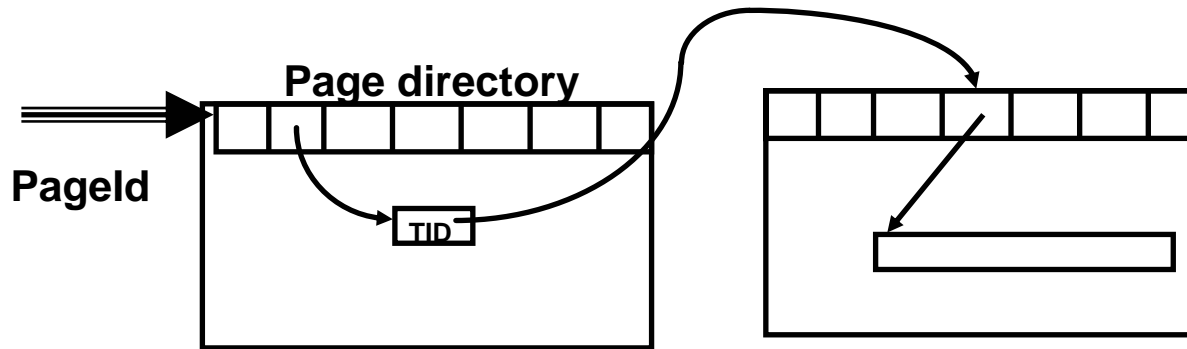
- TID = <BlockId, @DirOffset>



- 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??

Block moves



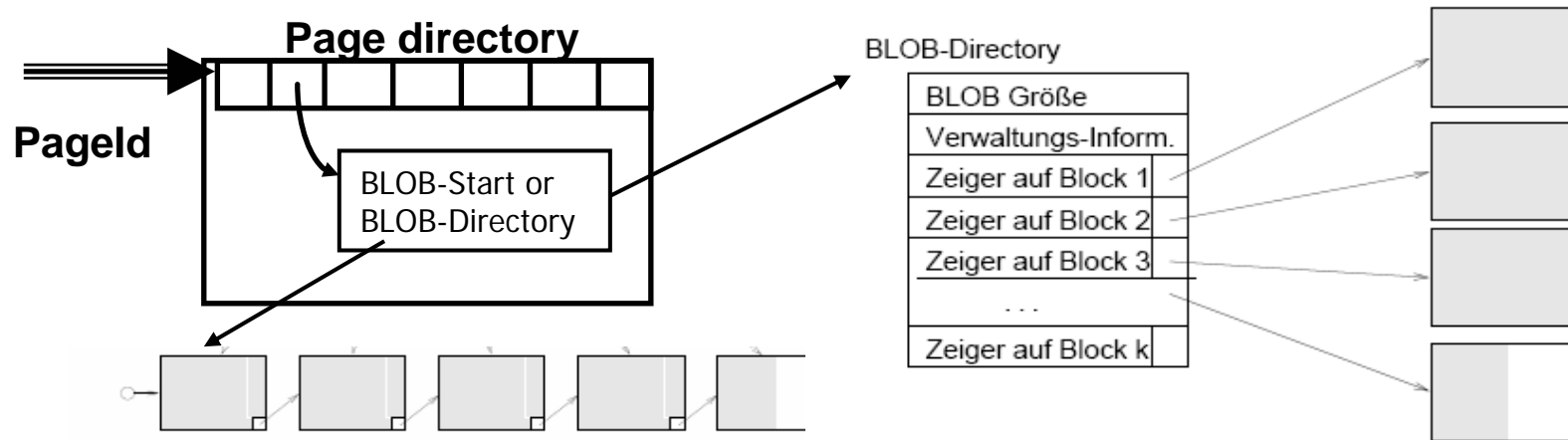
- Replace tuple with pointer
 - Another TID
- When further move is necessary, adapt TID pointer in first block
 - No chaining of references
 - Requires **mostly two block** accesses
- If tuples change blocks frequently, still degeneration on table level
 - Too many 2-block-accesses
 - **Re-organization necessary**
 - Find and change all references to TID



Storing BLOBs

- BLOB/ CLOB : Binary / Character large Objects
 - Images, video, music
 - Documents, XML
- Often allowed to up to 4 Gigabyte in size
- BLOBs do not fit into a block, page, segment, ...
- BLOBs are stored in separate data structures
 - Ever read from JDBC?
 - Access much harder than to other attributes
- Need other reference concept

Storing BLOBs

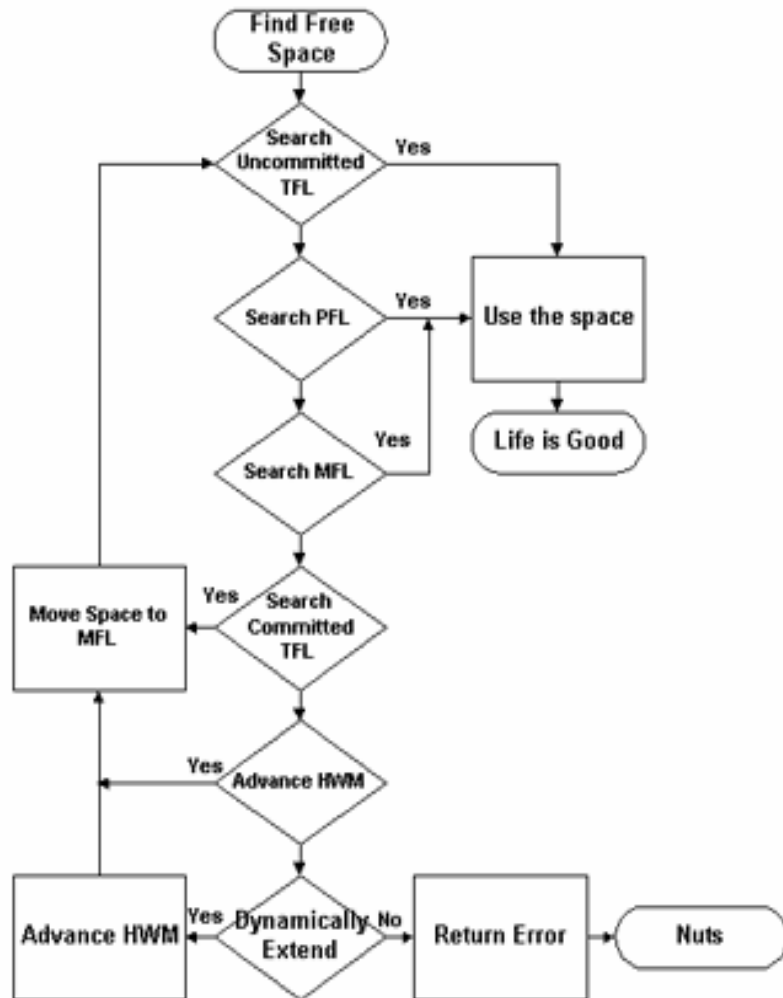


- Very good for **sequential reads**
- Do not try to get something from inside the BLOB
- No limitation in size
- BLOB can be managed in file system
 - Danger: Deletion of file – dangling pointer
- Bad for sequential read (video)
 - Even if blocks are sequential on disc – DB doesn't know
 - IO Controller prefetching helps
- Good for **access into BLOB** (e.g. large XML files)
- Size is limited if block directory is limited

INSERT – Finding free space

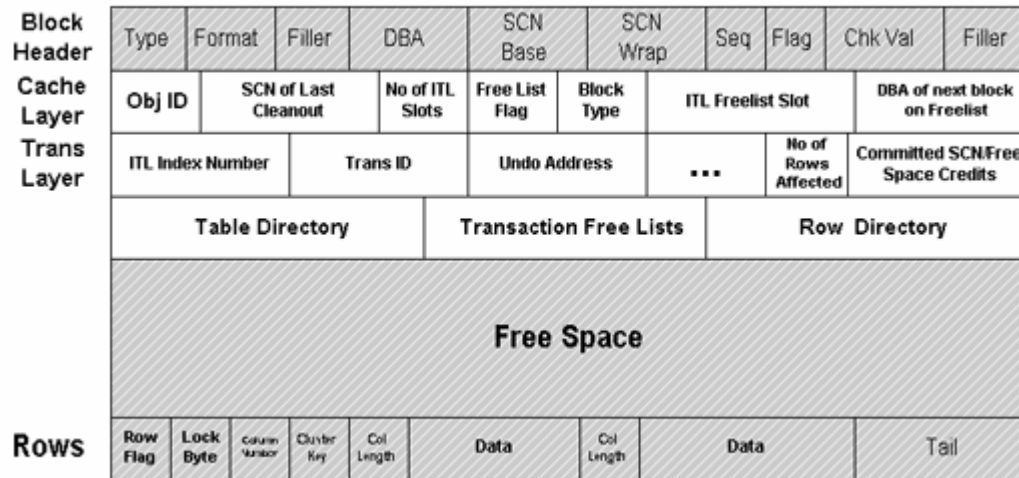
- What happens if record is deleted?
 - Mark record as deleted in block directory
 - Compress block or leave “hole” in block
 - In either case, free space is left
- INSERT a record
 - Possibility 1: Always into last page
 - No space reuse (apart from updates)
 - Requires reorganization from time to time
 - Possibility 2: Find space
 - First space – first space that is large enough
 - Remember: tuples usually have variable size!
 - Best space – first space that fits (almost) perfectly
 - Requires management of [free space list](#)

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

Oracle Block Structure



- DBA: Data Block Header: block address (global and relative in tablespace)
- Block type: data, index, redo, ...
- Table directory: tables in this block (for clustered data)
- Row directory: offset of tuples in block
- ITL: Interested transaction list – list of locks on rows in block
 - There is no „lock manager“ in Oracle
 - ITL grows and shrinks – “IRL wait”, INITTRANS, MAXTRANS
 - Locks are not cleaned upon TX end – next TX checks Trans-ID



Creating a table

```
CREATE TABLE
"SCOTT"."EMP"
(EMPNO NUMBER(4,0), ...)
PCTFREE 10
PCTUSED 40
INITRANS 1
MAXTRANS 255
NOCOMPRESS
LOGGING
STORAGE( INITIAL 65536
          NEXT 1048576
          MINEXTENTS 1
          MAXEXTENTS ...

          PCTINCREASE 0)
TABLESPACE SYSTEM
```

- PCTFREE
 - Not filled by inserts (reserved for updates) –avoids row chaining
 - Block is removed from free list
- PCTUSED
 - Low mark before block is put into free list
- INITRANS
 - Initial space reserved for TX-locks in blocks
- MAXTRANS
 - Max space reserved for TX-locks in blocks
- NOCOMPRESS
- LOGGING
 - generates REDO information
- INITIAL
 - Space of first extent
- NEXT
 - Size of next extent
- MINEXT
 - N# of extents allocated immediately (each size INITIAL, but total space not continuous)
- MAXEXT
 - Max. n# of extents
- PCTINCREASE
 - Increase of NEXT size