Datenbanksysteme II: Complexity, Records & Blocks

Ulf Leser

Content of this Lecture

- IO complexity model
- Records and pages
- Referencing tuples
- BLOBs and free space lists
- Example: Oracle block structure

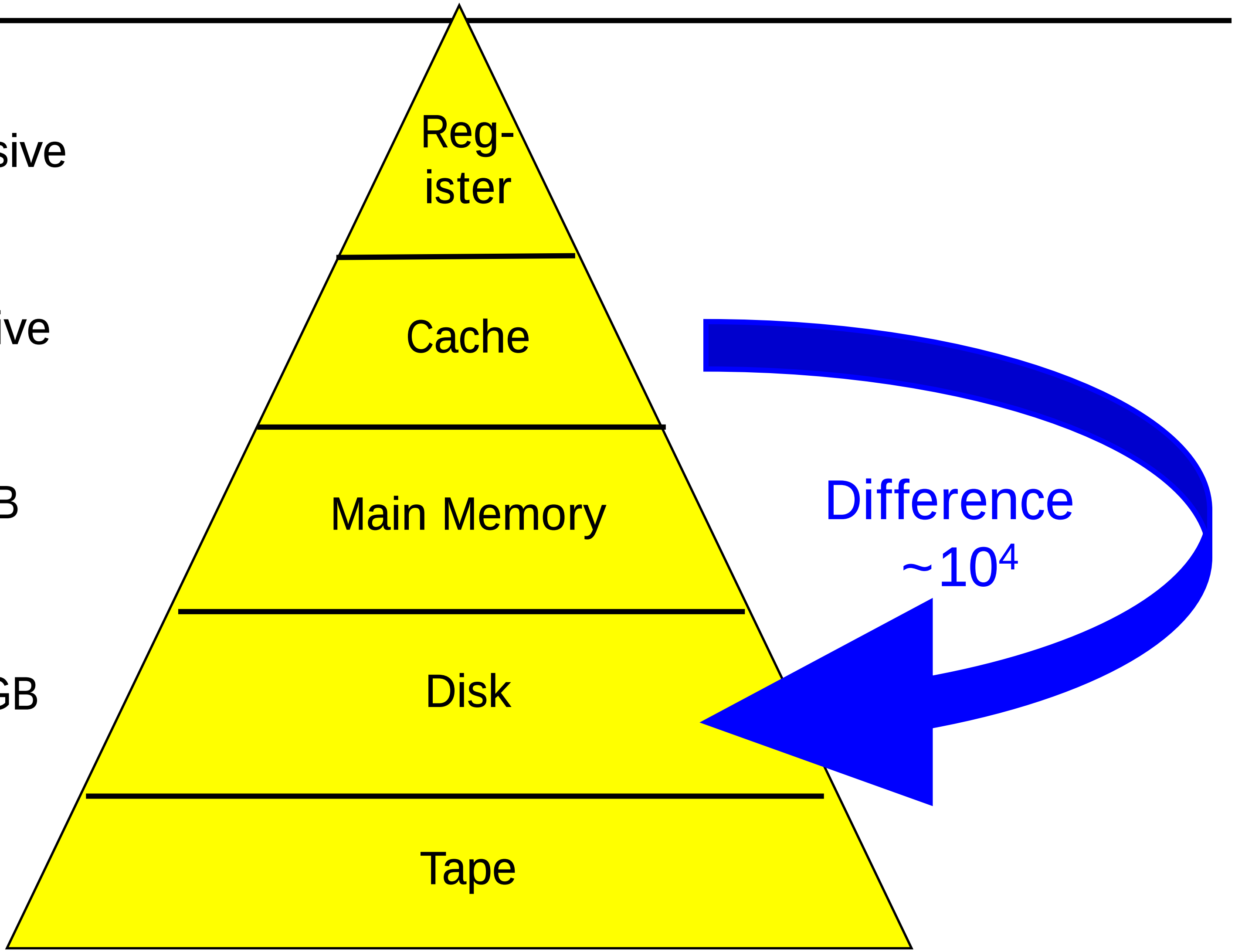
Again

Really expensive

Very expensive

~ 15 € / GB

~ 0,04 € / GB



Consequences

- Depending on the mode of data access, algorithms need to be designed and analyzed differently

- RAM model of computation
  - Access to data costs nothing (O(1))
  - Only operations on the data count – comparison, arithmetic, etc.
- IO model of computation
  - Operations cost nothing (as long as it is linear ...)
  - Only access to data counts – reading & writing blocks
- Beware: Sometimes both need to be considered
  - E.g. operations with non-linear complexity

RAM analysis of Merge-Sort

- Basis: Two sorted lists of size n can be merged in O(n)

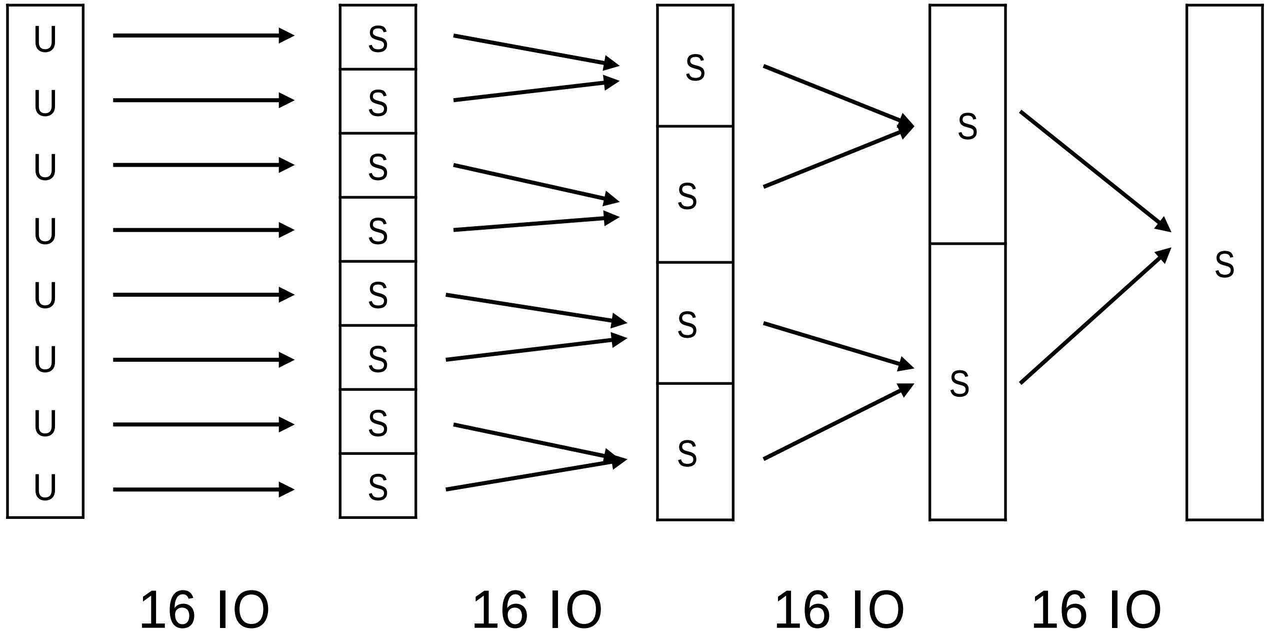
1	3		1
2	5		2
4	6		3
7	10		4

- Merge-Sort:
  – If list is of size 1, return (sublist is sorted)
  – Else, divide list in two lists of equal size
  – Call MERGE-SORT for each sublist
  – Merge the sorted list
- Complexity
  – O(n*log(n)) when measuring number of key comparisons

IO Analysis of Merge-Sort

- Basis: Two sorted lists on disc consisting of n blocks each 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 – 4*n IO
- Let's apply the recursive algorithm

Recursive merge-sort

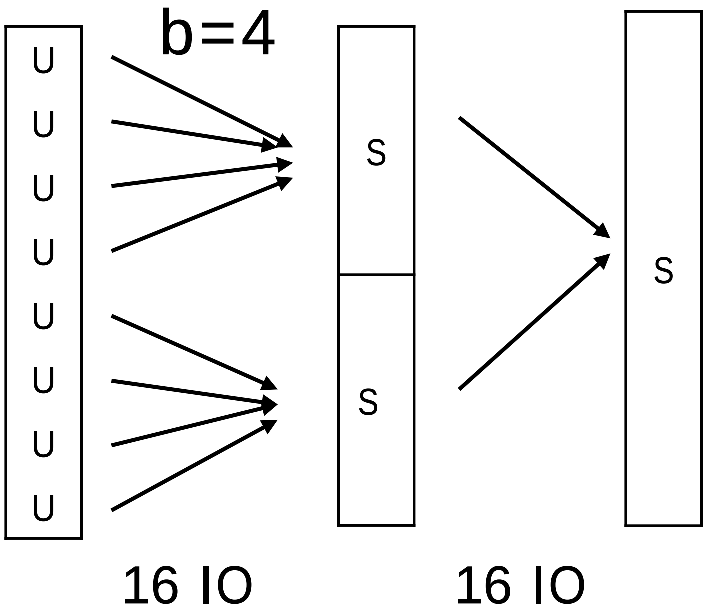


- Total IO: 2*n*(log(n)+1)
  - n: Number of blocks
- How much main memory do we need?
  - Just three blocks
- Can we do better?

Example cont‘d

- Idea: Load more than one block into main memory
  - Unsorted file with n blocks, main-memory M of size |M|=b blocks
  - Read b blocks from file, sort in-memory, write
    - 2b IO; sorting is free; needs in-place sorting algorithm
  - Repeat until file is read entirely; generates x=n/b sorted files (runs)
    - Total IO: Each block is read and written once: 2n IO
- Merge x runs in one step by opening all x files at once
  - Each block is again read and written: 2n IO
- Total: 4n IO

Blocked Multi-Way Merge-Sort



- b=4: 32 block IO
- b=3: ?
- b=8: ?

- Trick: Many concurrent reads
  - We do not measure time here, so parallel IO is not essential for analysis
  - But parallel reads are realistic with appropriate controllers, discs, ...
- Concurrent reads help to “get away” from the logarithmic number of rounds
  - We remove the assumption that only two blocks can be accessed at once
- Result: Linear IO
  - But still O(n*log(n)) comparisons

Limits

- If b<n, total IO is 4n
- But there is a limit (we are cheating)
  - During merge phase
  - Assume b=1: We would have to read 8 blocks, but b=1!
  - Problem 1: We need to have many files open at a time
    - Example: 1M, b=2
    - Generates 500K runs of size 2 each
    - We probably cannot open 500K files at once
  - Problem 2: We need to hold x+1 blocks in main memory
    - We will not be able to load 500K blocks in memory in case b=2
    - We could load a block, take first record, load next block ...
- Solutions?

Mega-Runs

- Solution for problem 2
  - Forget the one block we need for writing (makes math easier)
  - Thus, we can sort b*b blocks using our method
    - Read and sort b blocks, each time generating one of b runs
  - Partition file in partitions of b^2 blocks
  - Sort each partition, generating a mega-run
  - Open all mega-runs in parallel and merge
  - If there are more than b mega-runs, apply recursively
- How much data can we sort now?

Analysis

- Without mega-runs
  - One run sorts b blocks; we can read b files in parallel
  - Hence, we can sort b^2 blocks
  - Suppose
    - Block size=4096, record size=200: ~20 records per block
    - Main memory: 512 MB, ~400MB free: ~100.000 blocks (b=100.000)
    - Sorts 100.000^2*20 = 200.000.000.000 records
- With mega runs
  - In one mega-run (=partition), we sort b^2 blocks
  - Using 1 level of mega runs, we can sort b partitions of size b^2
  - Sorts 100.000^3*20 = 2E16 records = 4000 petabyte
- Small server: MM=4GB; b=1000000 => Sorts 4E6 PB
  - With how much IO?

Sequential IO

- We forgot differences between random / sequential access
  - Limitation: These are not captured by our IO model
- How can we maximize sequential IO?

Block Sequences

- Don't read/write blocks one-at-a-time
- Work on 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
- Anything else to optimize?
  - What does the machine do when waiting for (slow) IO?

Asynchronous Read/Write

- Use non-blocking, asynchronous IO
- Divide each third in two partitions
- Work with one partition; when done, issue IO request and continue with other partition while IO is happening to refill first partition
- Takes into account that main memory operations are not really free

Ignoring IO cost is a bad idea

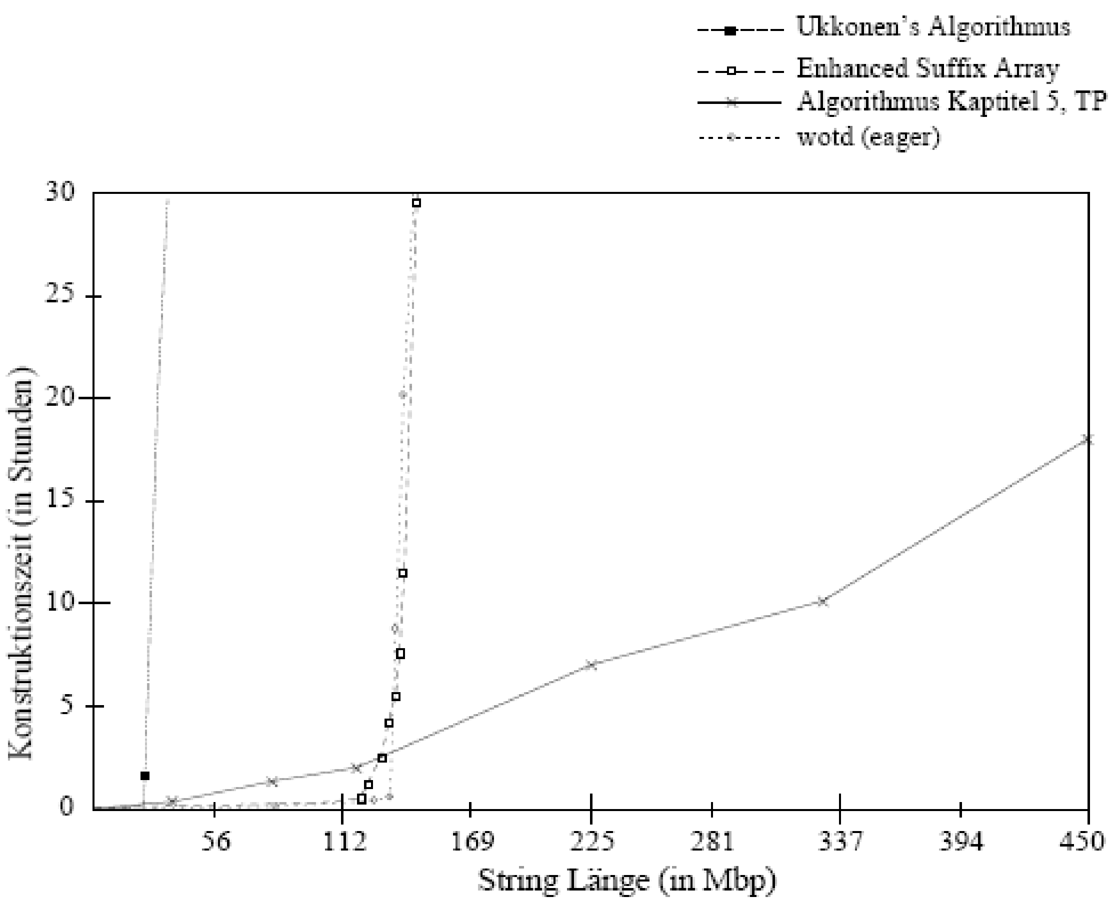
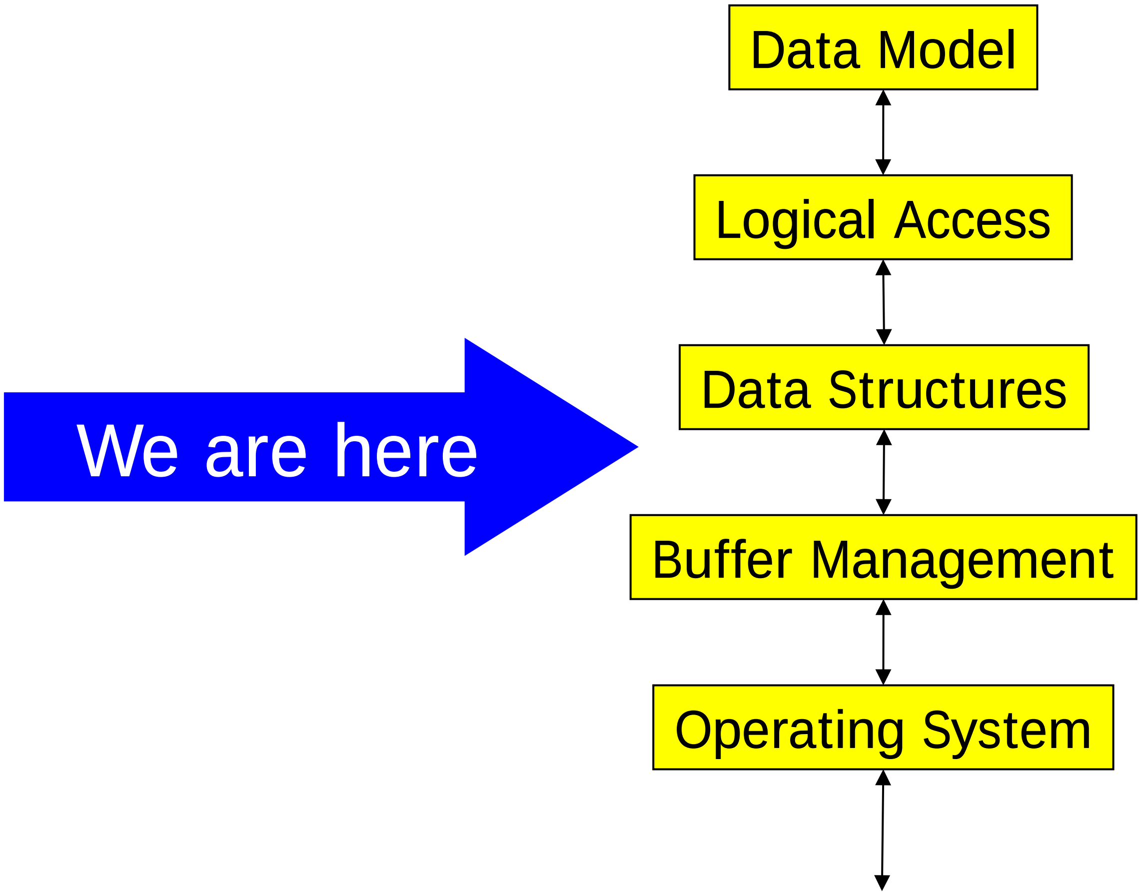


Abbildung 6.3: Entwicklung der Laufzeiten im Vergleich zu anderen Algorithmen

Content of this Lecture

- IO complexity model
- Records and pages
- Referencing tuples
- BLOBs and free space lists
- Example: Oracle block structure

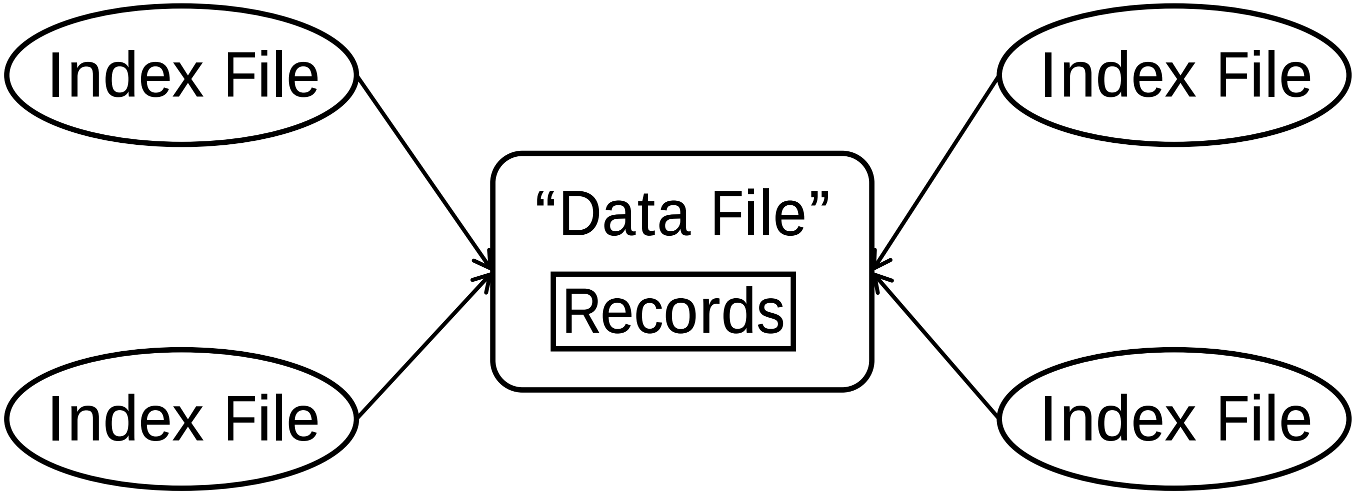
5 Layer Architecture



Storing relational data

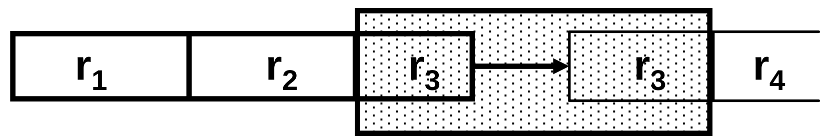
- Fundamental elements:
  Records (or tuples) consisting of typed attributes (or fields)
- We need to
  - Quench records on pages
  - Find attribute values of a given tuple
  - Find a record in a page
  - Find a page (next lecture)
- Central: Stable record references

Data and Indexes

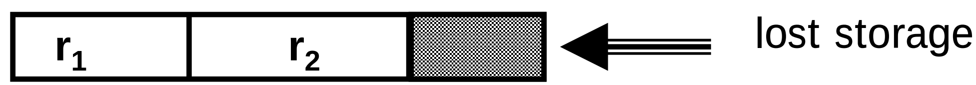


Quenching Records

- Tuple = Record; fixed or variable length
- Mapping of records to pages
  - "Spanned Record"



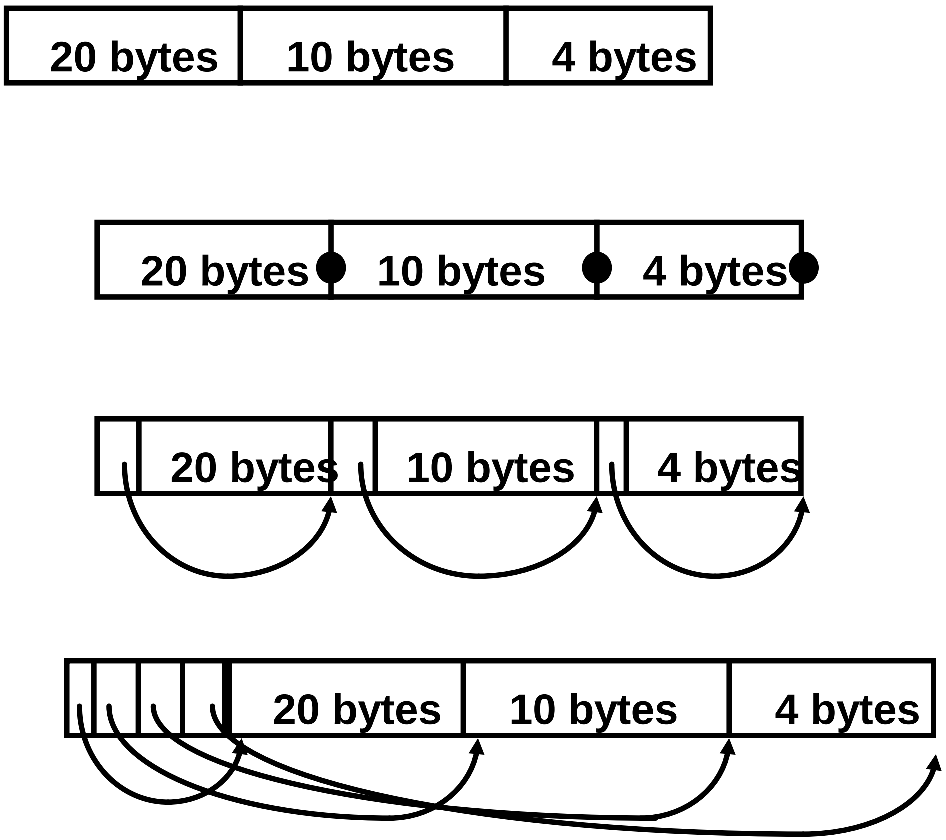
  - "Unspanned Record"



- Evaluation
  - Requires two (or more) IO operations
  - Transaction management on block level much more difficult
  - Offers better space utilization
- Summary: Avoid spanning records
  - But how to handle oversize records?

Adressing Fields of a Records

- Assume records with k fields and n byte



- V1: Fixed length records
- Variable length records
  - V2: Mark end of fields
    - Space: n+k; requires special end symbol; access by scan
  - V3: Store lengths of fields
    - Space: n+k*|len|; requires fixed |len|; access by hops
  - V4: Use record dictionary
    - Space: n+k*|ptr|; requires fixed |ptr|; direct access

Variable Length Records

- Practical hint: Don't be afraid of variable length records

- More freedom in data modeling
- Enables much better space utilization
- Additional work is manageable

Storing NULL‘s

- NULL has special semantics
  - Assume z=NULL; then, the following is not the same in SQL
    - if (z) then XXX else YYY;
    - If (z) then XXX; if (not z) then YYY;
  - Not at all the same: z=“” and z=NULL
    - Purposefully no value given versus ... (unclear)
- The many meanings of NULL
  - Not known, not defined, no value at the moment, ...
- NULLs as field values
  - Fixed length, with end marks, length indicator
    - Use special symbol (otherwise unused)
    - Always make sure to be able to discern “” from NULL
  - Record dictionary: set pointer to NULL

Content of this Lecture

- IO complexity model
- Records and pages
- Referencing tuples
- BLOBs and free space lists
- Example: Oracle block structure

Referencing Tuples

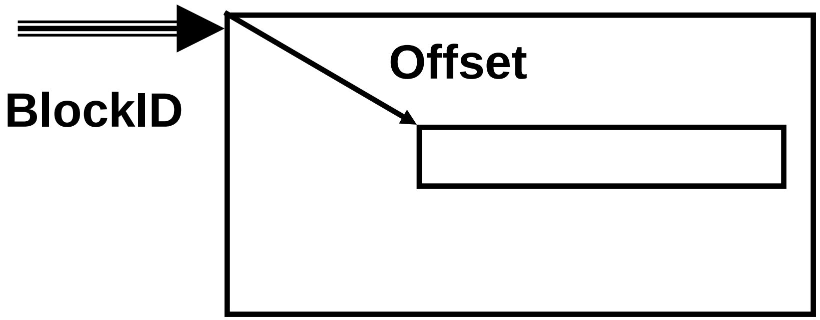
- At system level, tuples need to be addressable
  - E.g. references from indexes, transaction contexts, ...
  - TID should be unique and immutable
    - Uniqueness for unique identification
    - Immutable for keeping references alive
- Still, physical location should be changeable
  - For growing tuples, for improving free space management, during block reorganization, ...
  - Moves can be within or across pages
- Requires some form of decoupling of TID from physical location (semi-physical referencing)

TID Concept

- Tuples are identified by tuple ID (TID)
  - Must encoded: Block + location in block
  - Different options – next slides
- When requesting a tuple by its TID
  - Determine block
  - See if block is in buffer
  - Yes – return physical block address
  - No
    - If necessary, free space for block in buffer first
    - Load block; translate blockID in physical address
      - Virtualized memory
  - All performed by the cache manager (next lecture)

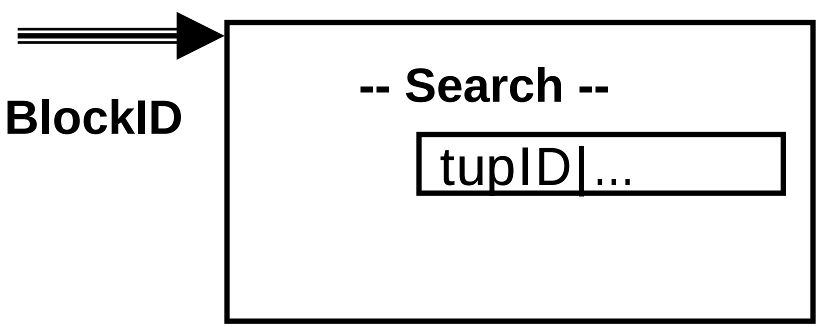
Addressing a Record in a Page

- Option 1: TID = <BlockID, Offset>



- Good: direct access
- Bad: no moves possible

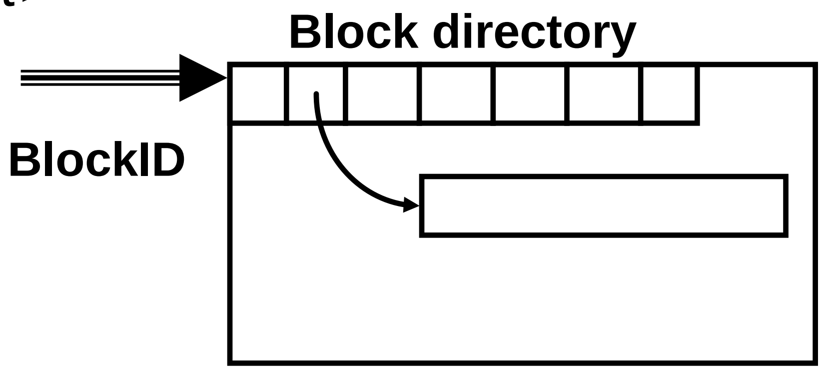
- Option 2: TID = <BlockID, tupID>, then search



- Good: Moving within block
- Bad: Requires a block scan; tupID must be managed

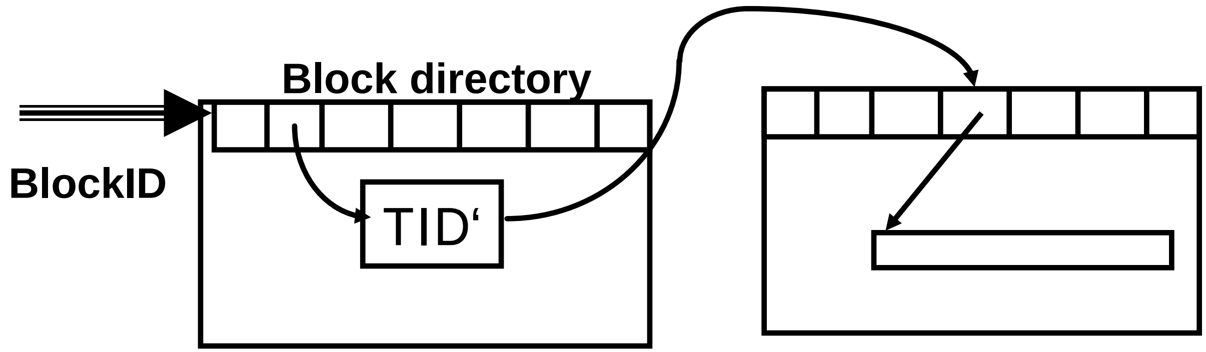
Using a Block Directory

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



- Method of choice
  - TID remains stable when tuples move within blocks
  - No scan, only 2 indirections
- Requires management of block directory within each block (requires space; must be locked; ...)
- How to move across blocks (without updating pointers)?

Delegation



- Replace tuple with TID': Another TID, used only internally
- Upon further moves, only adapt pointer (TID')
  - No chaining of references
  - Accessing tuple requires at most two block IO
- Might incur degeneration
  - Too many 2-block-accesses
  - Incentive for periodic re-organizations

TIDs versus Foreign Keys

- Foreign key is a logical value at the data model layer, TID is a semi-physical value at in internal layer
- FKs are looked-up in an index, TID are translated into physical addresses
- Foreign key is visible to developers, TID (usually) not
  - Do not use TIDs as foreign keys – will change during query processing
- Foreign key is an integrity constraint, not a pointer
  - May join foreign key with any other value in the database as well

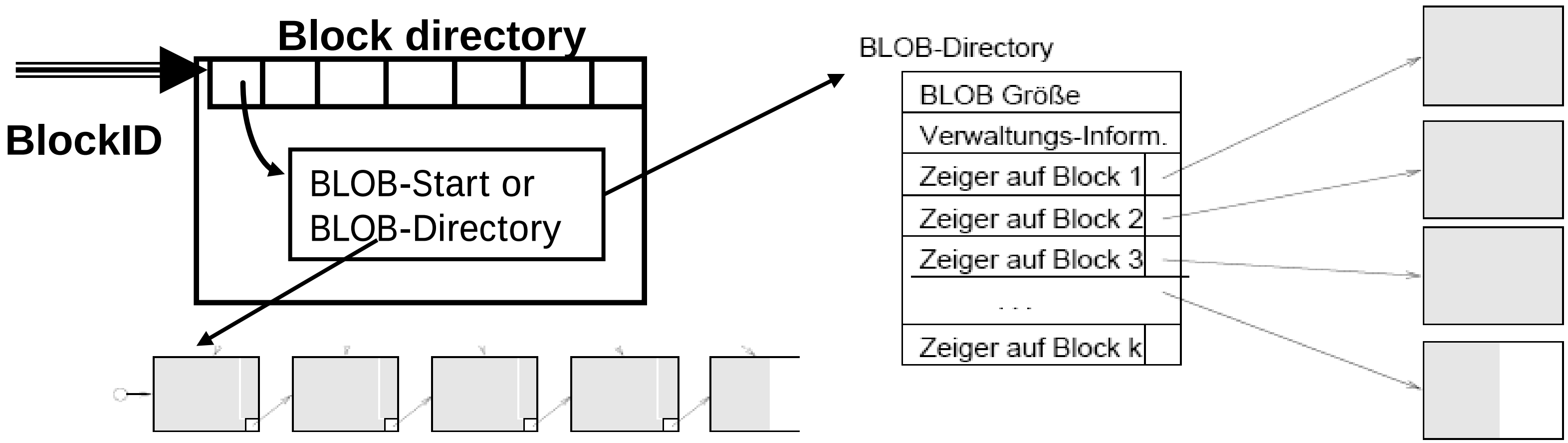
Content of this Lecture

- IO complexity model
- Records and pages
- Referencing tuples
- BLOBs and free space lists
- Example: Oracle block structure

Storing BLOBs

- BLOB/ CLOB : Binary / Character large Objects
  - Images, video, music, PDF, ...
- May have gigabyte in size (depending on DBMS)
- Do not fit into a block, page, segment, ...
- BLOBs typically are stored in separate data structures
  - Ever read a BLOB through JDBC?
  - Access much harder than for ordinary attributes
- May be managed by file system or by DBMS (tablespaces)
- If managed by file system: File may be deleted, other access credentials, ...

Storing BLOBs



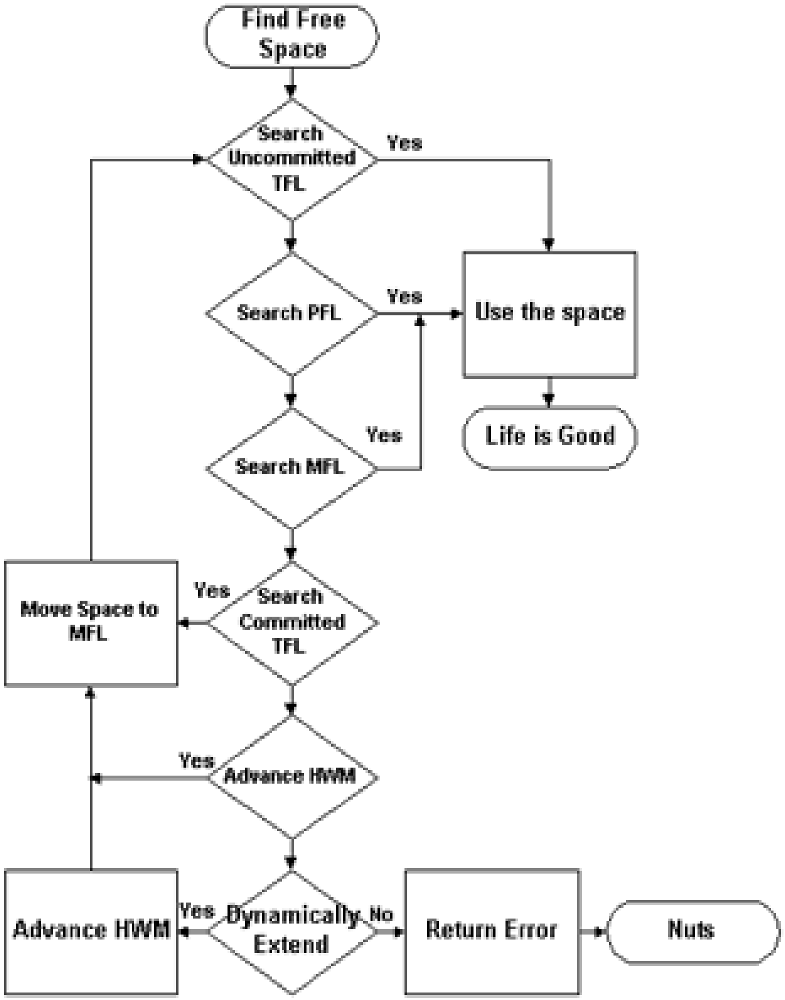
- Allows sequential reads
  - If blocks are really sequential on disk
- Difficult to seek specific positions inside BLOB
- No limitation in size

- No sequential read (video)
  - Use block chaining on top
- Good for accessing specific positions within BLOB
  - Large XML files
- Size limited through dir size

INSERT – Finding Free Space

- What happens if a 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 block
    - No space reuse (apart from updates)
    - Requires periodic reorganizations to ensure sufficient space utilization
  - Possibility 2: Try to find free space inside blocks
    - Must be large enough (simple for fixed-size tuples)
    - Many possible strategies: Next free space? Best fitting space? Space in block with is most underutilized?
    - Requires management of free space list per logical storage unit

Life is complex

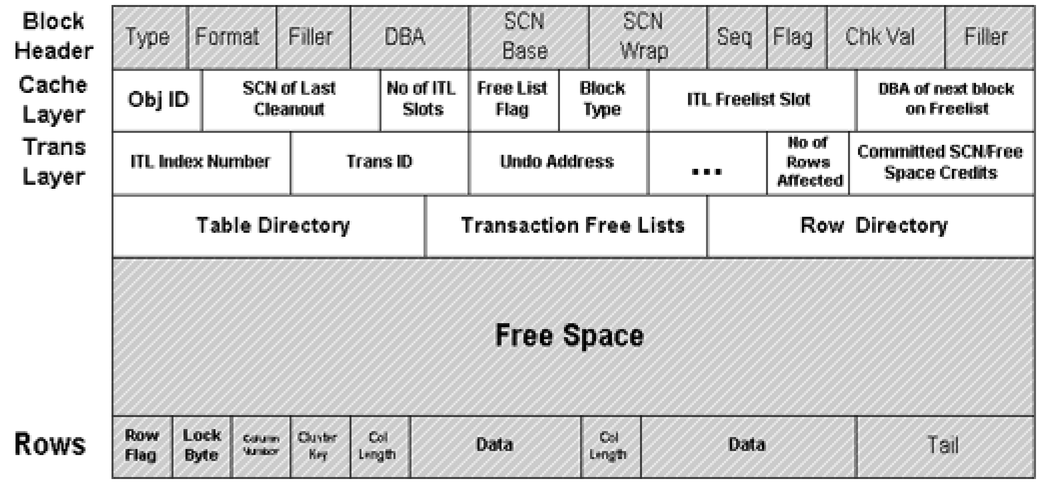


- Oracle procedure for finding free space
- Free space is administered at the level of segments
  - Logical database objects
- Explanation
  - TFL: transaction free list
  - PFL: process free list
  - MFL: master free list
  - HWM: High water mark

Content of this Lecture

- IO complexity model
- Records and pages
- Referencing tuples
- BLOBs and free space lists
- Example: Oracle block structure

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 – locks on rows in block
  - There is no „lock manager“ in Oracle
  - ITL grows and shrinks – “ITL wait”, INITTRANS, MAXTRANS
  - Locks are not cleaned upon TX end – next TX checks TX-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
- PCTUSED: Low mark before block is put into free list
- INITTRANS: Initial space reserved for TX-locks in each block
- MAXTRANS: Max space reserved for TX-locks
- NOCOMPRESS
- LOGGING: generates REDO or not
- INITIAL: Size of 1st extent
- NEXT: Size of next extent
- MINEXT: Number of extents allocated immediately (each size INITIAL, but total space not continuous)
- MAXEXT: Max. number of extents
- PCTINCREASE: Increase of NEXT size