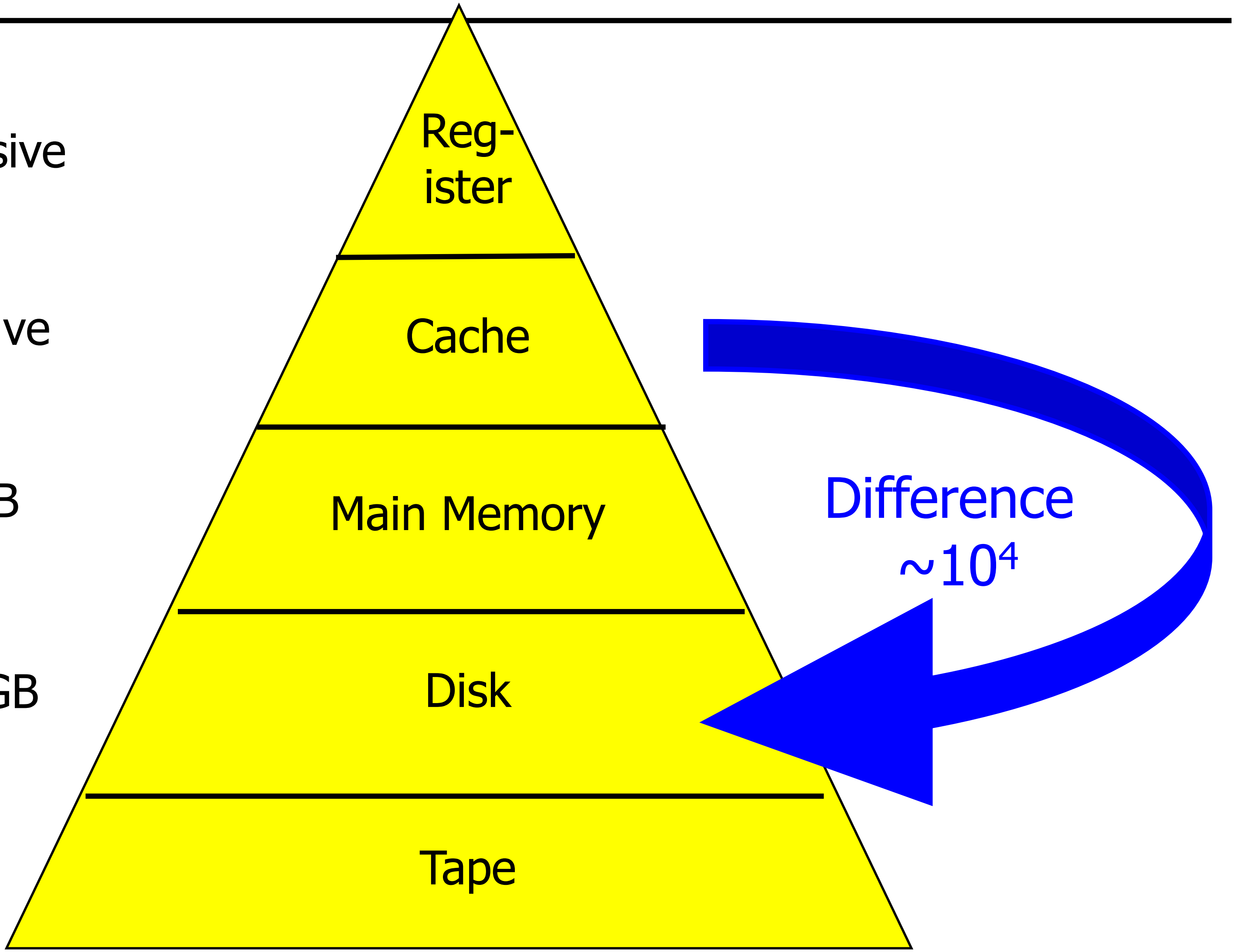
Datenbanksysteme II:
IO Complexity, Records & Blocks

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

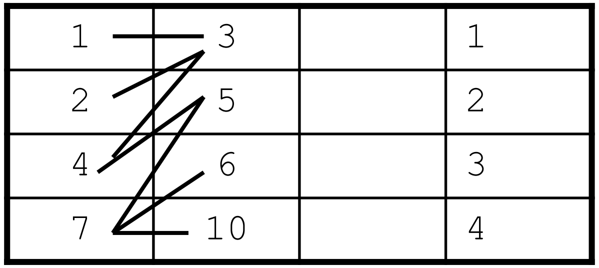
Consequences

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

- RAM model of computation
  - Access to data costs essentially nothing (O(1))
  - Only operations on the data count – comparison, arithmetic, etc.
- IO model of computation
  - Operations cost nothing (because CPU so much faster than HDD)
  - Only access to data counts – reading & writing blocks
- Beware: Sometimes both need to be considered
  - E.g. operations with non-linear complexity
  - That’s the setting in FONDA

RAM analysis of Merge-Sort

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



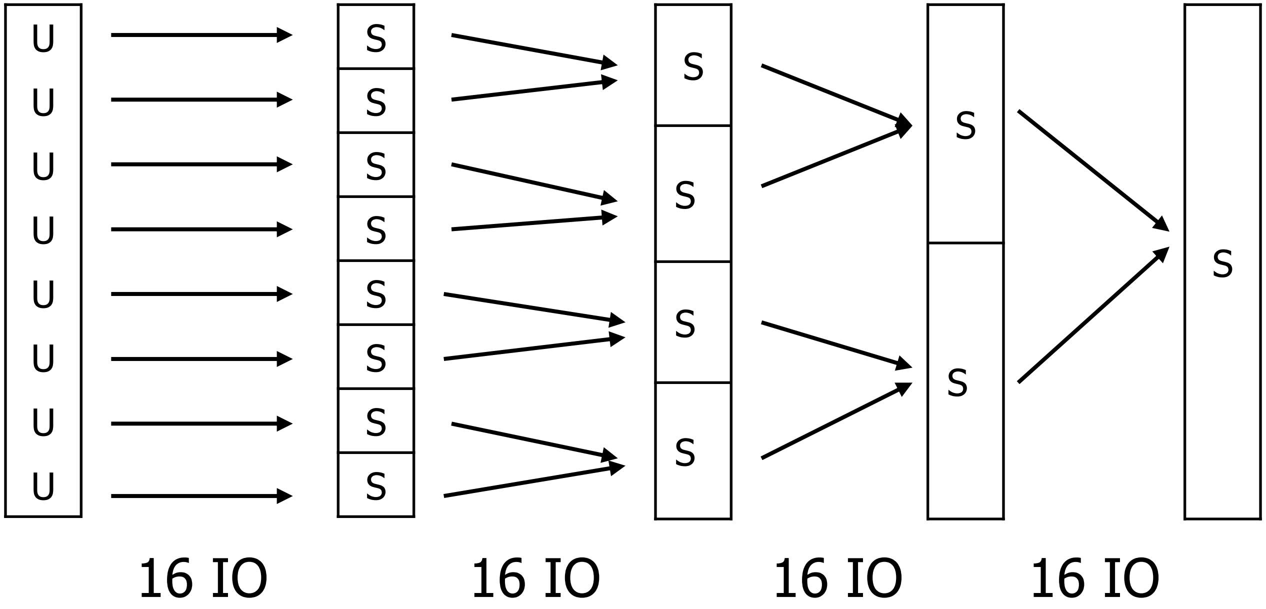
1	3		1
2	5		2
4	6		3
7	10		4

- 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

- Assume all data is in a sequence of blocks on disc
- 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, then reuse (1 IO)
  - In total, each block is read and written once – 4*n IO
- Now the recursive part of Merge Sort

Recursive merge-sort



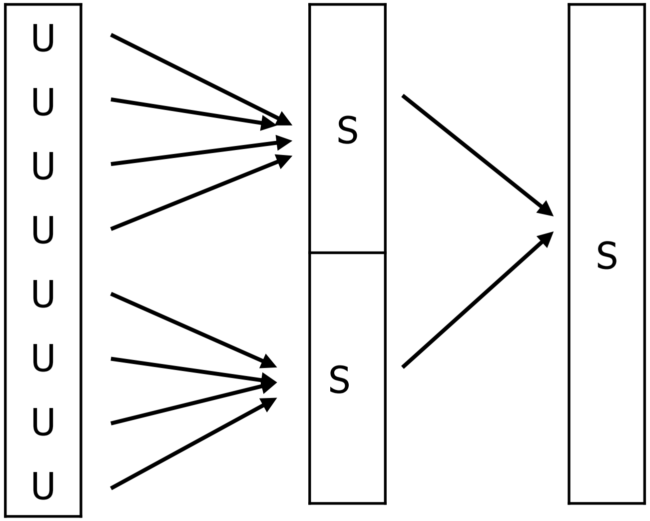
- Total IO: ~2*n*(log(n)+1)
  - n: Number of blocks; we count single block operations
    - No difference between random access and sequential IO
- How much main memory do we use?
  - Never more than three blocks
- Can we do better?

Example cont‵d

- Idea 1: 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
- Idea 2: Read concurrently from multiple files
  - Merge x runs in one step by opening all x files at once
  - Each block is again read and written: 2n IO
- Total (still): 4n IO, but … we are done

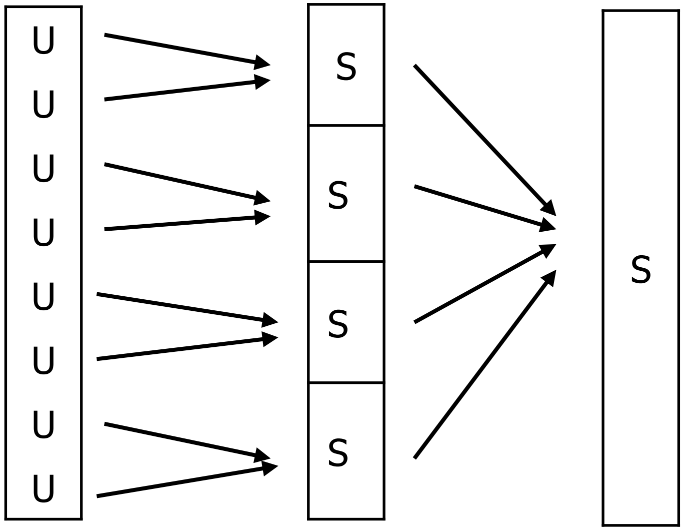
Towards linear IO

Idea 1:
Fill all your memory



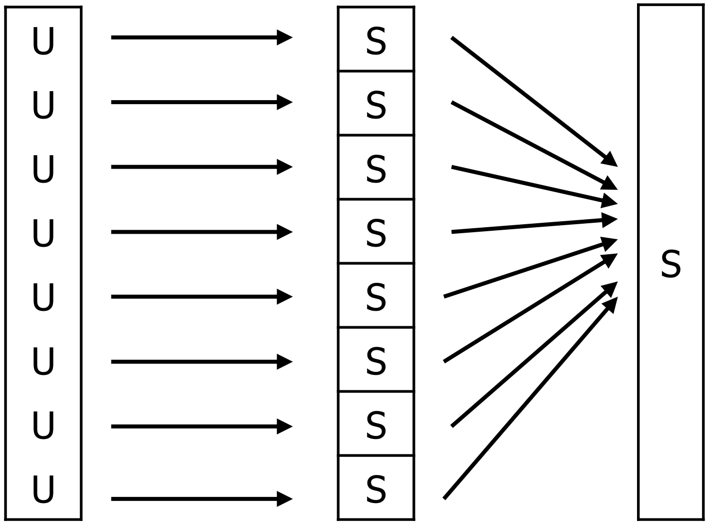
b=4: 32 IO

Idea 2:
Merge all runs at once



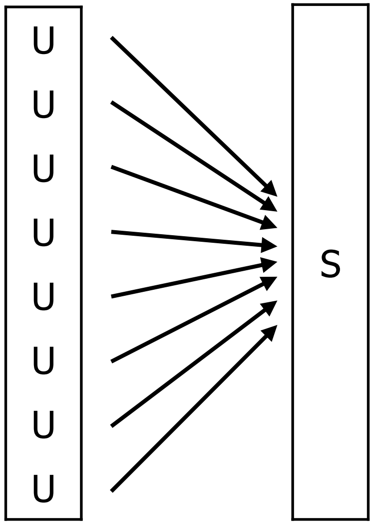
b=2: 32 IO

Works for all b



b=1: 32 IO

More Precisely: Works for all b<n

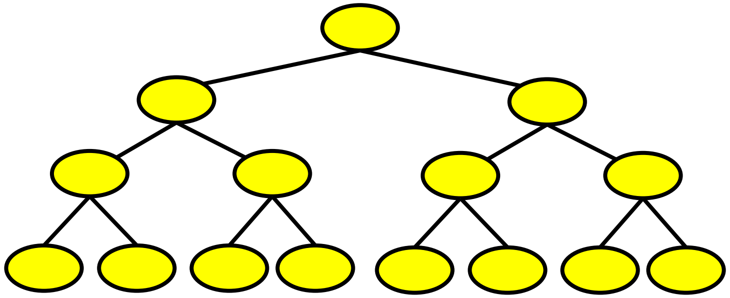


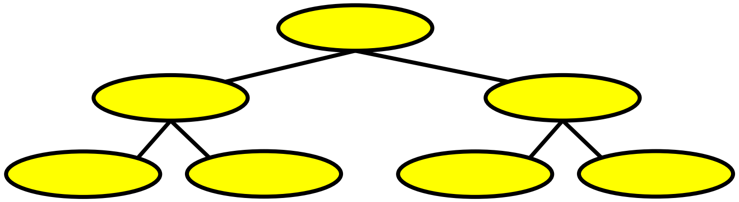
b=8=n: 16 IO

Blocked Multi-Way Merge-Sort

- Surprising: If b<n, total IO is 4n
  - And 2n otherwise
- Main Trick: Use concurrent reads
  - Parallel access is orthogonal to our cost model
- Concurrent reads help to “get away” from the logarithmic number of rounds
  - We don’t have log(n) deep 2-way trees, but a 2-layer b-way tree
  - Realistic (up to a certain point) with appropriate controllers, discs, …
- Result: Linear IO
  - But still O(n*log(n)) key comparisons

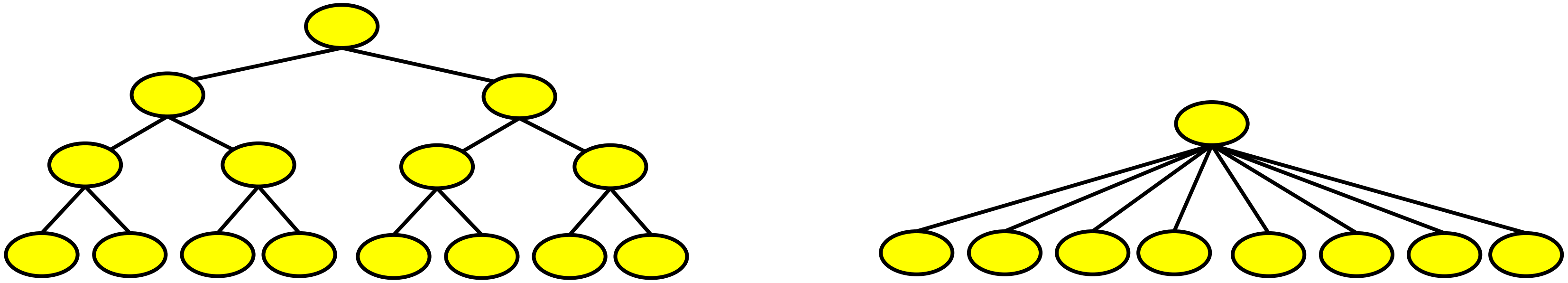
Illustration





- Classical, binary setup
  - Needs $\log_2(N)$ many rounds
  - Merges two blocks / runs in each node
- Binary with more memory
  - Height shrinks, but only by a constant factor

Illustration



- Classical, binary setup
  - Needs $\log_2(N)$ many rounds
  - Merges two blocks / runs in each node

- With concurrent reads
  - Height becomes a constant: 2

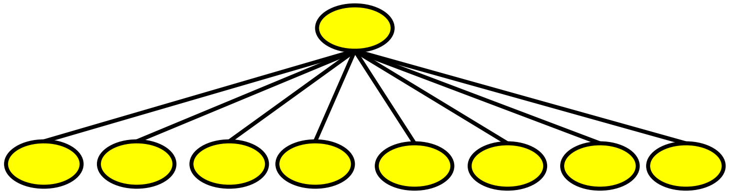
Limits

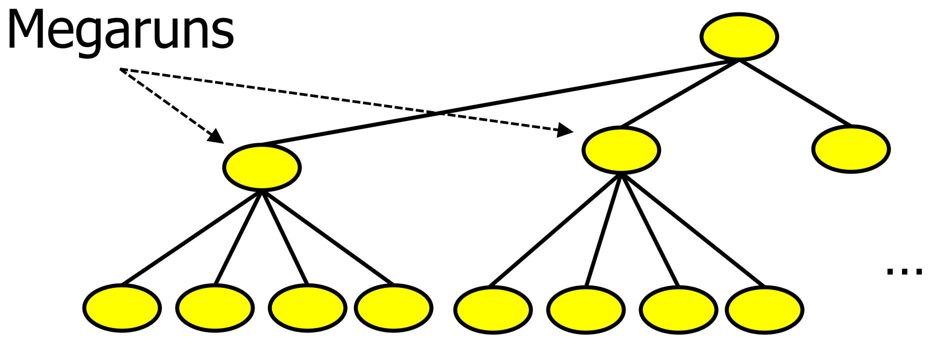
- Of course, there is a practical limit: How many blocks can we open at once and read in parallel without delay?
- Problem 1: We need to have many files open at a time
  - Example: 1M blocks, b=2
  - Generates 500K runs of size 2 each
  - We probably cannot open 500K files at once
- Problem 2: We need to hold n/b+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 problems 1 and 2
  - The limiting factor is min(b, number_open_files)
  - Assume that b=min(b, number_open_files)
    - Ignore 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
- We are back at logarithmic complexity
- But – when will this take effect?
  - We digest b^2 blocks at once!

Illustration





- Needs one file handle and one block memory for each edge
- Constant height only if
  - Unlimited number of opened files
  - Unlimited main memory

- Assume b=4
- Tree again has logarithmic height
- But at base b: $\log_b(n)$
- The additional in-memory cost for merging b values is ignored in our cost model

Analysis

- Without mega-runs
  - One run sorts b blocks; we can read b files in parallel
  - Hence, we can sort b^2 blocks (hard limit)
  - Suppose
    - Block size=4096B, 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 4GB free memory: 2E13 records = 2 "Peta-records"
- With mega runs
  - In one mega-run (=partition), we sort b^2 blocks
  - Using 1 more level of mega runs, we can sort b partitions of size b^2
  - Sorts 100.000^3*20 = 2E16 records = 4000 petabyte
  - With 128GB free memory: 6E28 records

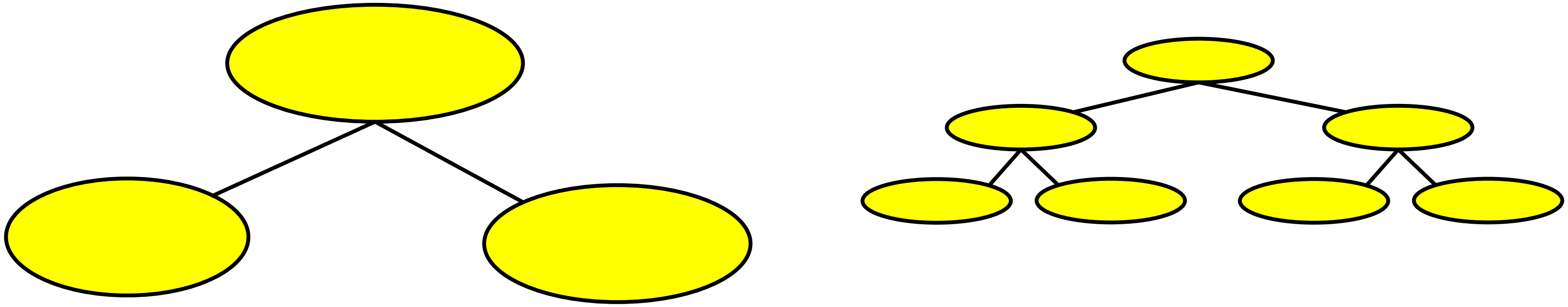
Sequential IO

- We ignored differences between random access / sequential reads/writes
  - Differences are not captured by our IO model
  - Opening n/b files at once and reading them block-by-block – much random access
- How can we maximize sequential IO?

Block Sequences

- Don’t read/write blocks one-at-a-time
- Work on sequences of consecutive blocks in each run
  - For instance, 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
    - Write b/3 blocks in one sequential write
  - Merge n/b runs by every time reading b/(n/b+1) blocks of each run
- Effect is the stronger, the larger b
  - Memorize: Always try to use all memory you have

Illustration



- Binary with more memory
  - Height shrinks compared to block-wise
  - But much random IO
- Binary with sequential IO
  - Height grows again
  - But much less random access and many more sequential reads

Asynchronous Read/Write

- Anything else to optimize?
  - What does the machine do while waiting for (slow) IO?

- 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

Take-Home Messages

- Sorting is linear on disc in terms of IO even for extremely large data sets
- Always try to use all memory you can get
- In practice: Consider all players
  - Does the disk controller cache tracks?
  - How many blocks can be read in O(1) in parallel? Congestion?
  - Is b a constant, or can we request memory dynamically?
  - Which parts can be implemented asynchronously?
  - …

Ignoring IO cost is a bad idea

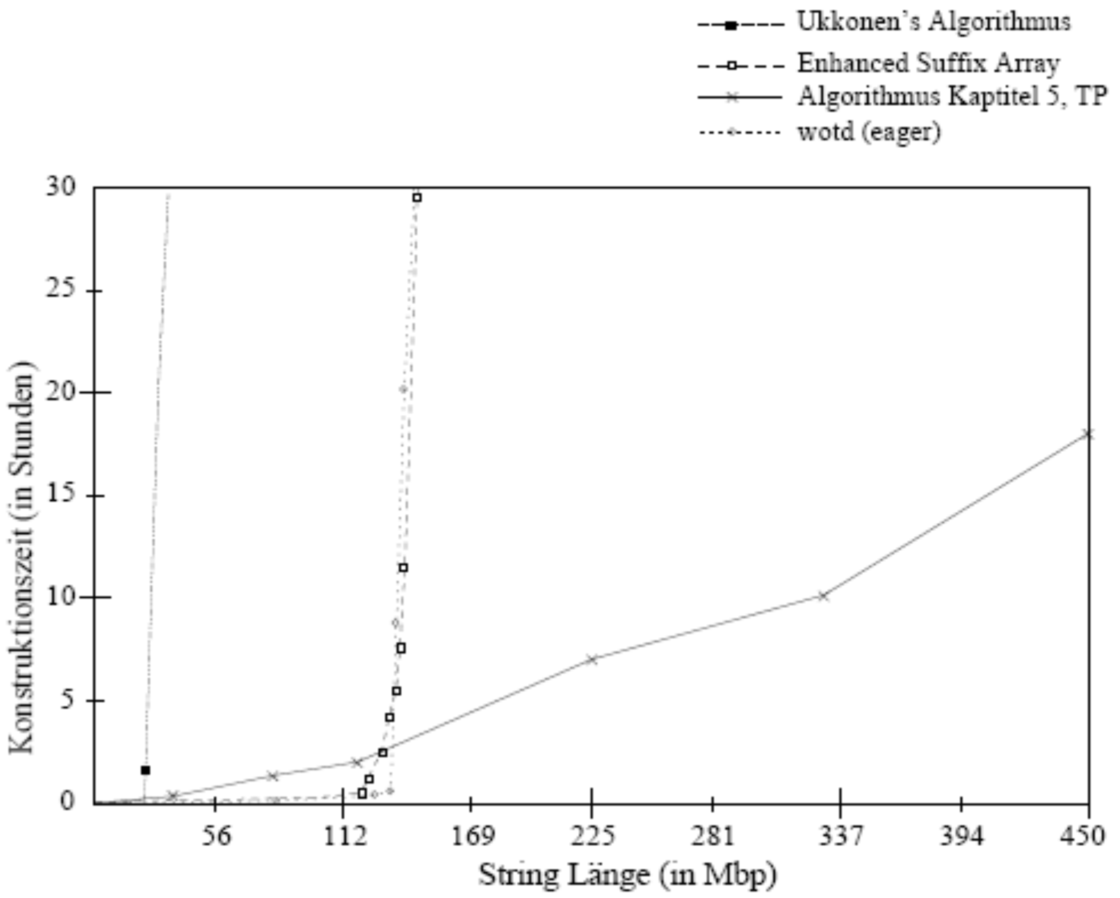
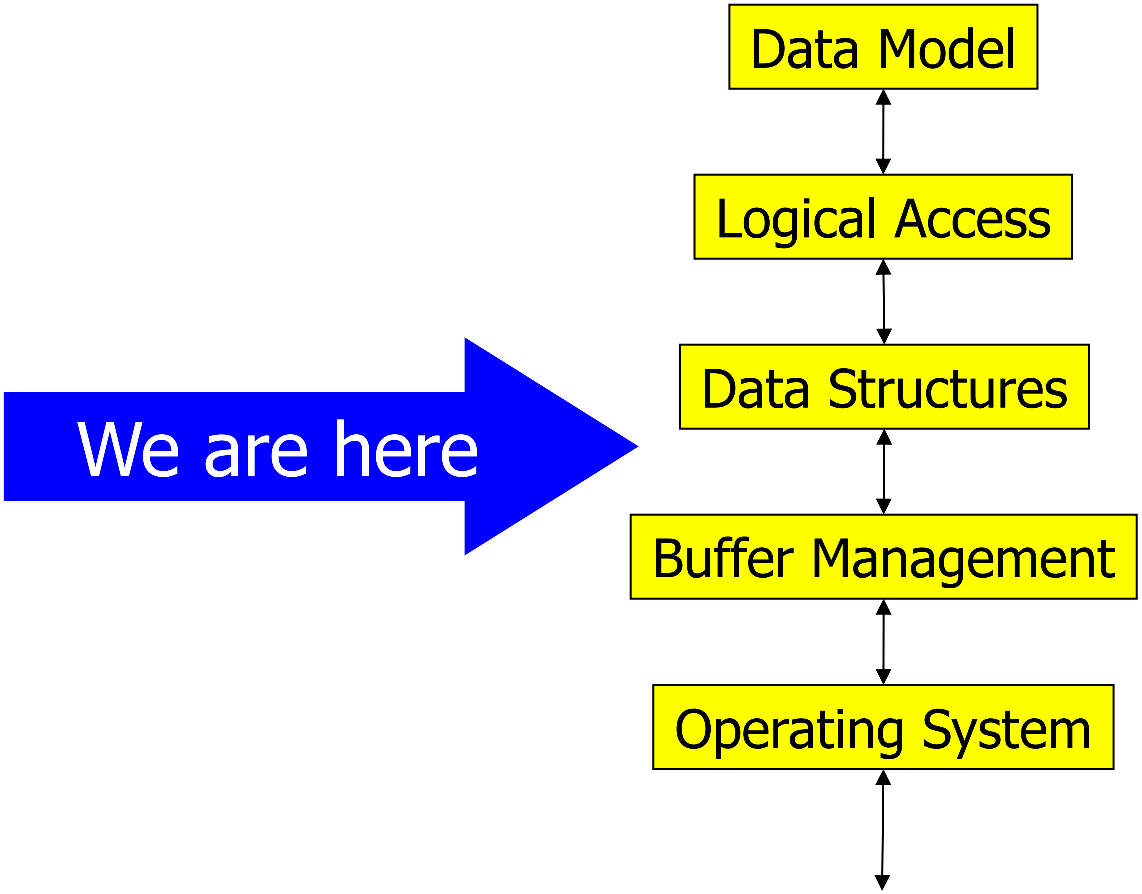


Abbildung 6.3: Entwicklung der Laufzeiten im Vergleich zu anderen Algorithmen

Content of this Lecture

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

5 Layer Architecture



We are here

Data Model

Logical Access

Data Structures

Buffer Management

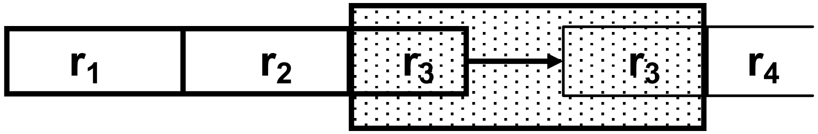
Operating System

Storing relational data

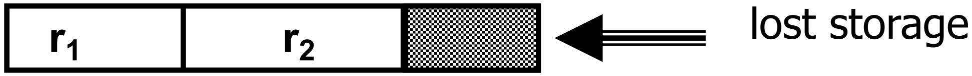
- Fundamental elements:
  Records (or tuples) consisting of typed attributes (or fields)
- We need to
  - Quench records on pages
  - Find all attribute values of a given tuple
  - Find a record in a page
  - Find a page (next lecture)
  - This is physical, record-at-a-time access
    - And not the set-based semantics of SQL
- Central issue: Stable record references

Quenching Records

- Fields (and thus records) can have fixed or variable length
- Mapping of records to pages
  - "Spanned Record"



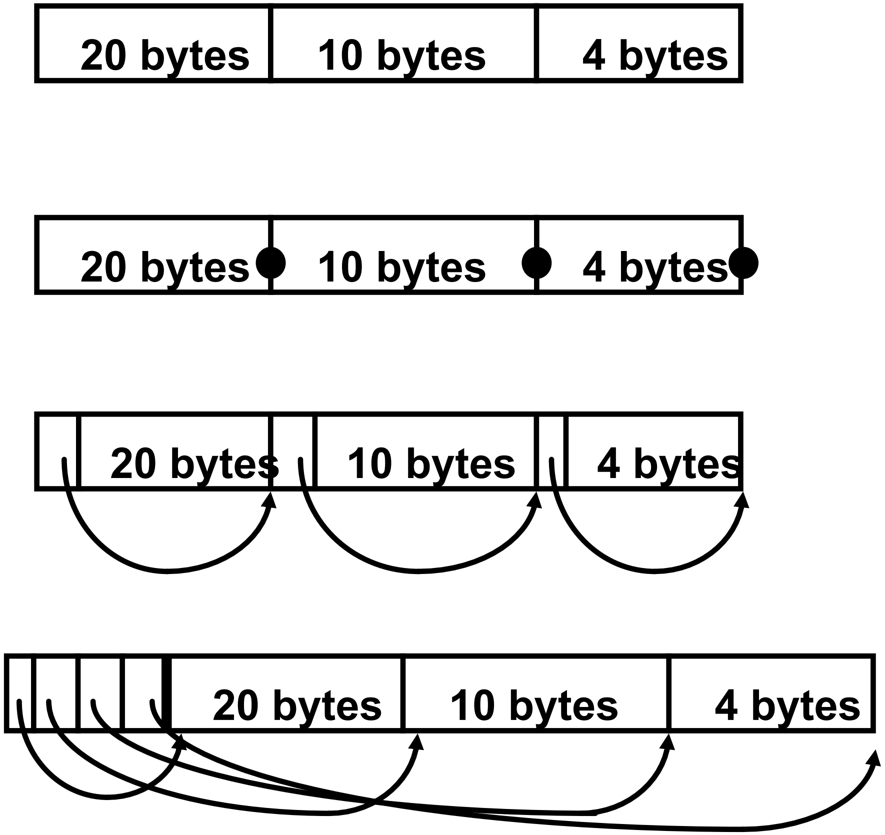
  - "Unspanned Record"



- Spanning records?
  - Requires two (or even more) IO operations
  - Transaction management on block level much more difficult
  - Offers better space utilization
  - Allows arbitrarily large records
- Practice: Avoid spanning records
  - But how to handle oversize records?

Adressing Fields of a Records

- Assume records with k fields and n byte total
- V1: Fixed length records: Store as array
- Variable length records
  - V2: Mark end of fields
    - Space: n+k; requires special end symbol; access O(n)
  - V3: Store lengths of fields
    - Space: n+k*|len|; requires fixed |len|; access O(k)
  - V4: Use record dictionary
    - Space: n+k*|ptr|; requires fixed |ptr|; access O(2)



Variable Length Records

- Don’t be afraid of variable length records (up to a certain length)
  - Varchar
  - More freedom in data modeling
  - Enables much better space utilization
  - Additional work for DB is manageable
- Think twice when using very large fields
  - Images, XML files, graphs (in DB2), varchar (512MB), …
  - Do not fit in single pages – usual techniques don’t work
  - Need special support by the RDBMS (CLOBs, BLOBs)

Storing NULL‘s

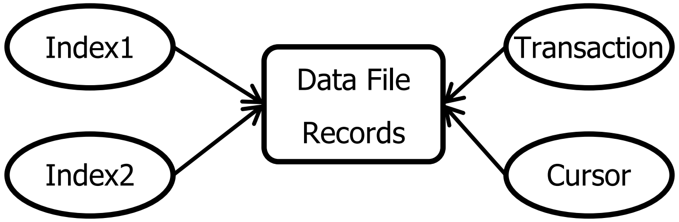
- NULL has special semantics
  - Assume z=NULL; then, the following is not the same in SQL
    - if (z) then XXX else YYY;
    - If (z) then XXX; if (not z) then YYY;
  - Not at all the same: z=“” and z=NULL
    - Purposefully no value given versus … (unclear)
- The many meanings of NULL
  - Not known, not defined, no value at the moment, …
- NULLs as field values need special techniques
  - We need to discern “” from NULL
  - For fixed length, with end marks, length indicator: Use special symbol (otherwise unused)
  - For 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

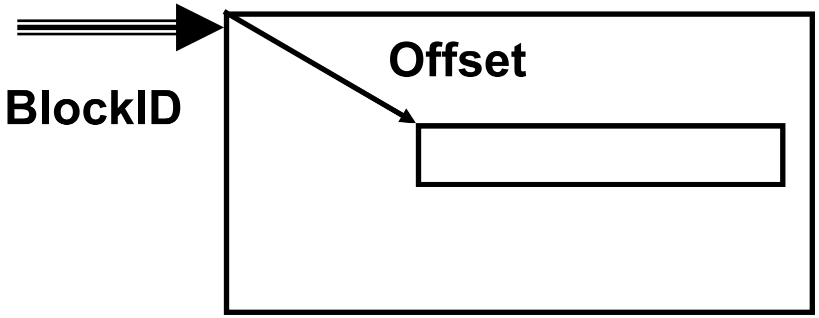
- Tuples are identified by tuple ID (TID) (or RID)
- At system level, tuples need to be addressable
  - Must allow to locate records: Block and location in block
  - References from indexes, transaction contexts, cursors, …
  - Must be unique and immutable
    - Uniqueness for identification
    - Immutable for stable references



- Still, physical location should be changeable
  - For growing tuples, for improving free space management, during block reorganization, …
- Semi-physical referencing: Decoupling TID from location

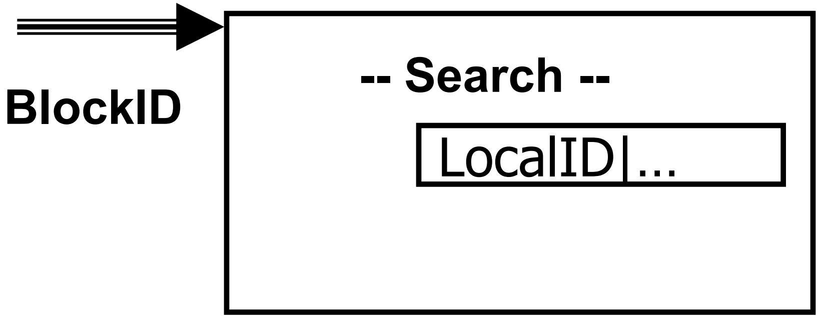
Addressing a Record in a Page

- Option 1: TID = <BlockID, Offset>



- Good: direct acc. in page
- Bad: no moving

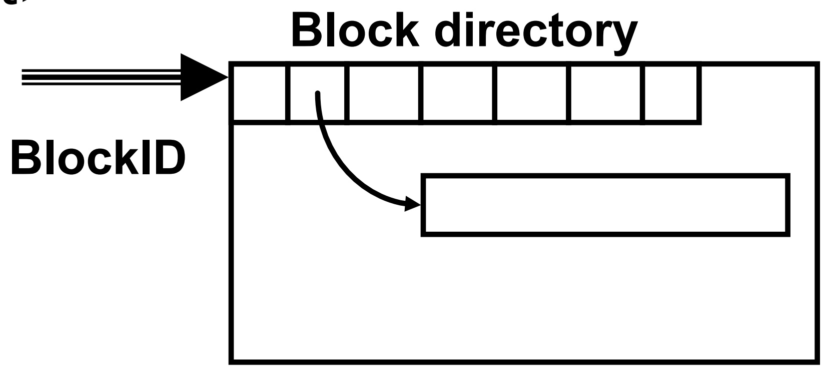
- Option 2: TID = <BlockID, LocalID>, then search



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

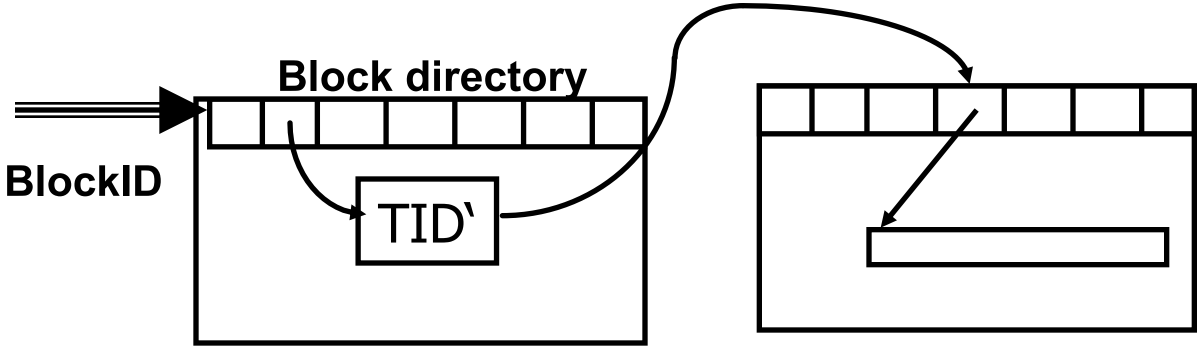
Using a Block Directory

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



- Method of choice
  - TID remains stable when tuple moves within block
  - No scan, only 2 indirections
- Requires management of block directory within each block (requires space; must be locked; …)
- How to move across blocks?

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, TIDs are essentially direct physical addresses
- FKs are visible to developers, TID (usually) not
  - Can be accessed in some systems, but think twice before using for anything – there are no guarantees
- FK is an integrity constraint, not a pointer
  - May join foreign key with any other value in the database as well

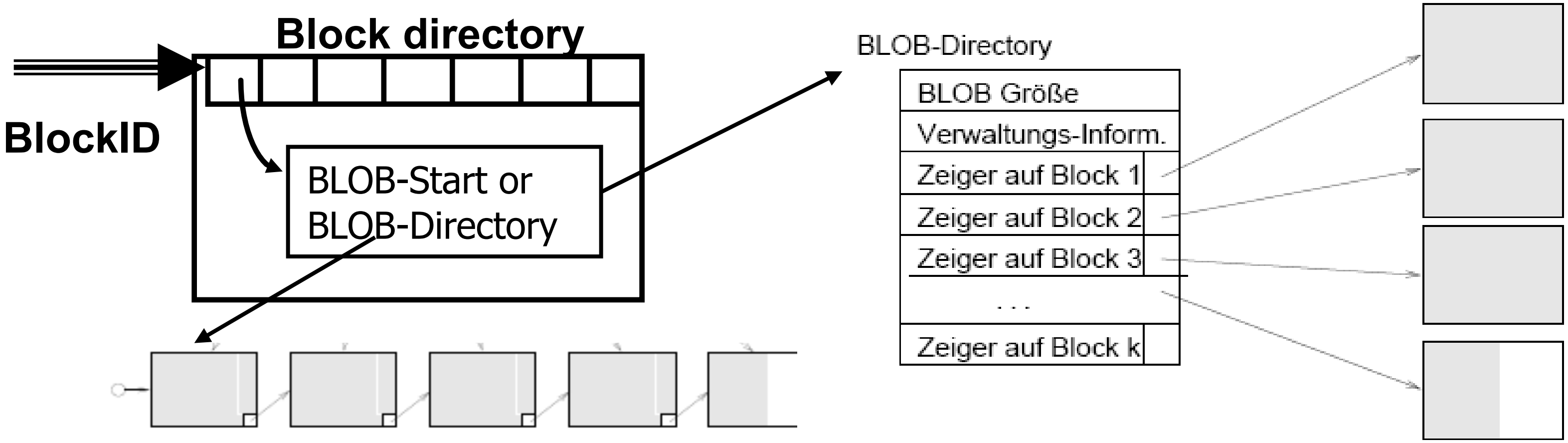
Content of this Lecture

- 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



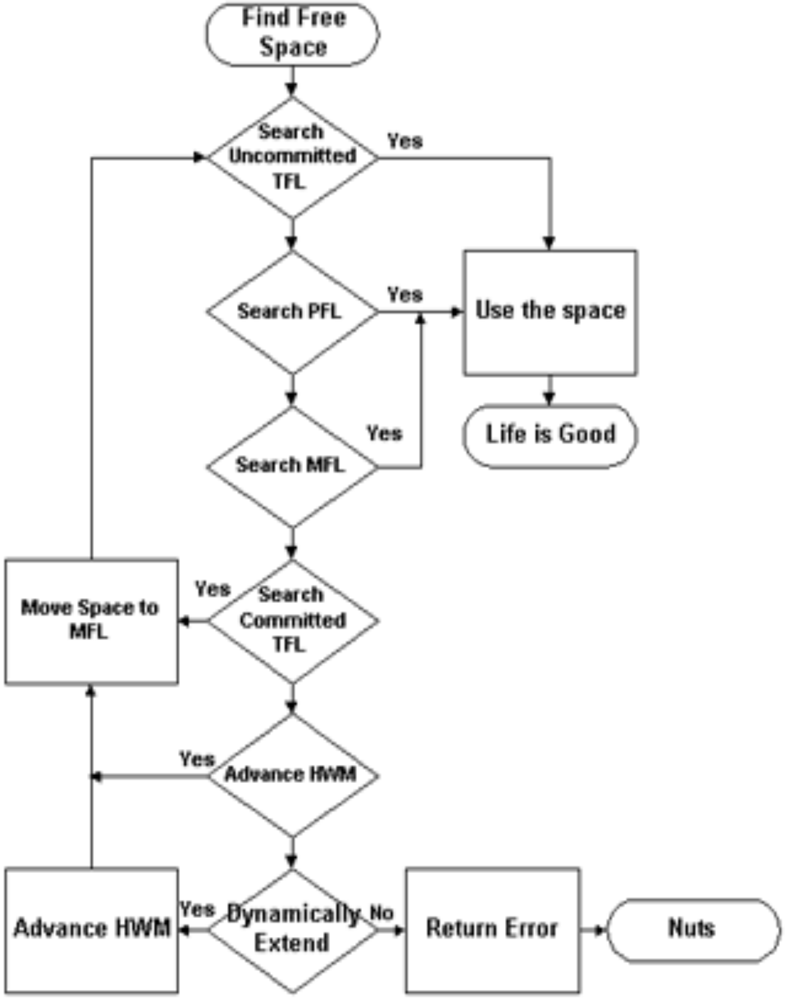
- Blob-chain
  - Allows sequential reads
    - If blocks are really sequential on disk
  - Difficult to seek specific positions inside BLOB
  - No limitation in size

- Blob-directory
  - No sequential reads
    - Potential “hicks” while reading
  - Easier to move to specific positions
  - Size limited through dir size

INSERT – Finding Free Space

- What happens if a record is deleted?
  - Mark record as deleted in block directory (tombstone)
  - Compress block or leave "hole" in block
  - In either case, free space is left
- INSERT a record
  - Possibility 1: Always into last block of table files
    - Always increases Highwater mark
    - No space reuse (apart from updates)
    - Requires periodic reorganizations to ensure sufficient space utilization
  - Possibility 2: Try to find free space inside blocks
    - Space 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? Space in cached blocks?
    - Requires management of free space lists per logical storage unit

Life is complex

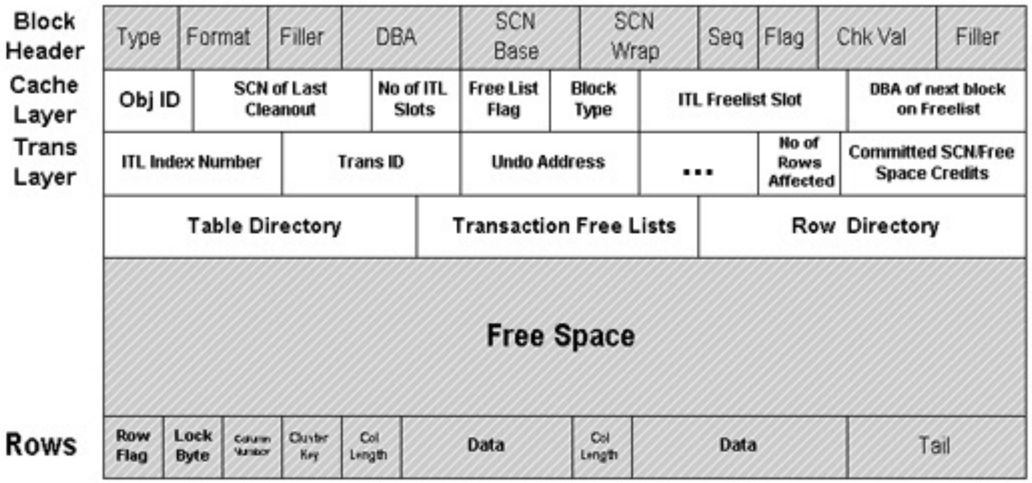


- Oracle procedure for finding free space
- Free space is administered at the level of segments
  - Logical database objects
- Explanation
  - TFL: transaction free list
  - PFL: process free list
  - MFL: master free list
  - HWM: High water mark

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
  - 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