PA152: Efficient Use of DB

3. Representing Data Elements

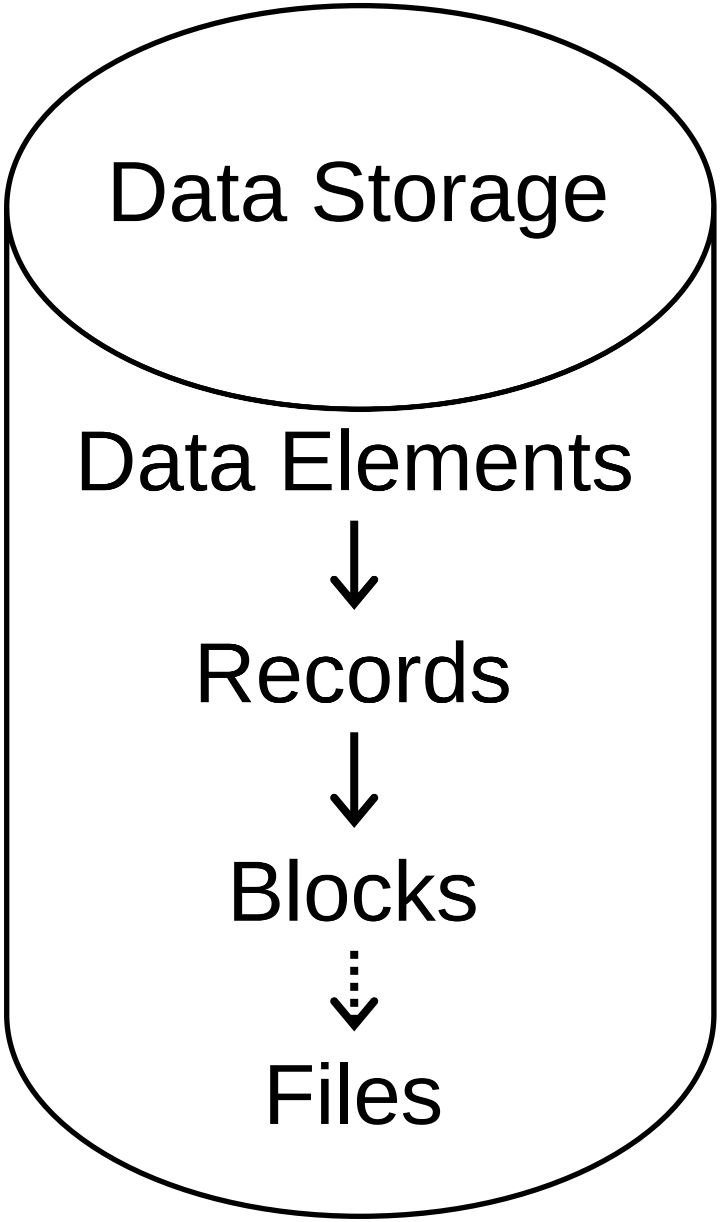
Vlastislav Dohnal

Outline

- *Data elements and fields*
- Records
- Block organization
- Properties and examples

Representing Data

- What to store?
  - name
  - salary
  - date
  - picture
- How to store?
  - values → byte sequence



Data Element Types

- Integers
  - By range: 2, 4, 8 bytes
  - E.g., 35 in 16 bits

00000000	00100011

  - Typically *sign bit* or *ones complement*
- Real numbers
  - Floating-point numbers
    - *n* bits split to mantissa and exponent (IEEE 754)
  - Fixed decimal point (*number(p,s)*)
    - Encoding a group of 9 digits (in base 10) into 4 bytes
    - Store as a string in base 10

Data Element Types

- Boolean
  - Usually as an integer
  - True | 1111 1111 |
  - False | 0000 0000 |
  - No reason to use less than 1 byte
- Bit array
  - Length + bits
    - Typically rounded up to next multiple of 4/8 bytes

Data Element Types

- Date
  - number of days since “epoch” (e.g., Jan 1, 1970)
    - or as a packed 3-byte integer DD + MM*32 + YYYY*16*32
  - string YYYYMMDD (8 bytes)
    - YYYYDDD (7 bytes)
    - Why not YYMMDD?
- Time
  - number of seconds since midnight
    - number of milliseconds or microseconds
    - or as a packed 3-byte integer
      DD*24*3600 + HH*3600 + MM*60 + SS
  - fractions of second
    - As string HHMMSSFF or as above with fractional part separately (up to 3 bytes for 6 digits)
  - time zones – time converted and stored in UTC
    - so converted from given/local time zone to UTC

Data Element Types

- Datetime
  - Combining date and time
    - Year*13+month; day, hour, min, sec + fraction
      - 5 bytes + fractional part
- Timestamp
  - Seconds since epoch with μsec
    - midnight Jan 1, 1970 UTC; Jan 1, 2000 in Pg.
- Enumerated type
  - Assign integers (ordinal numbers)
  - red → 0, green → 1, blue → 2, yellow → 3, ...
  - Pg stores the value as string

Data Element Types

- Characters & character sets
  - In ASCII encoding – 1 byte
  - Multi-byte characters
    - UCS-2 (UTF-16) – UTF-8 encoding in 16 bits
      - Characters with ordinal numbers from 0 to 65535
    - UTF-8 – variable-length encoding
      - Character may occupy 1-4 bytes
        - Originally up to 6
        - Now it is limited to the same range as UTF-16.
      - Representation:

```
0xxxxxxx
110xxxxx 10xxxxxx
11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
```

total number of bytes

Data Element Types

- Strings
  - Fixed length
    - Size limited, so
      - shorter strings filled with space
      - longer strings cut off
  - Variable length
    - Length plus content
    - Null-terminated
      - must be read completely
      - cannot use zero character (ord == 0) in the string
  - Character set issues (encoding)

Storing Data Elements: Summary

- Each element has a “type”
  - bit interpretation
  - size
  - special “unknown” value (NULL)
- Usually, fixed length
  - predefined bit representation
- Variable length
  - length plus content/value

Outline

- Data elements and fields
- *Records*
- Block organization
- Properties and examples

Record

- List of related data elements
  - i.e., their values
  - Fields, Attributes
- E.g.
  - Employee
    - name – Novák
    - salary – 1234
    - start_date – Jan 1, 2000

Record Types by Length

- Fixed length
  - Each record of same size (in bytes)
- Variable length
  - Saving space
  - More complex implementation
  - Can store large data (images, …)

Record Schema

- Describes record structure
- Information contained
  - Number of attributes
  - Order of attributes
  - Data type and name of each attribute

Record Types by Schema

- Fixed schema
  - Same schema for all records
    - Stored out of record (in data dictionary)
- Variable schema
  - Record itself contains schema
  - Useful for:
    - “sparse“ records (many NULLs)
    - repeating attributes
    - evolving formats
      - schema changes during DB lifetime

Example: Fixed Length and Schema

- Employee
  1) id – 2 byte integer
  2) name – 10 chars
  3) department – 2 bytes

} schema

55	n o v á k	02
83	d l o u h ý	01

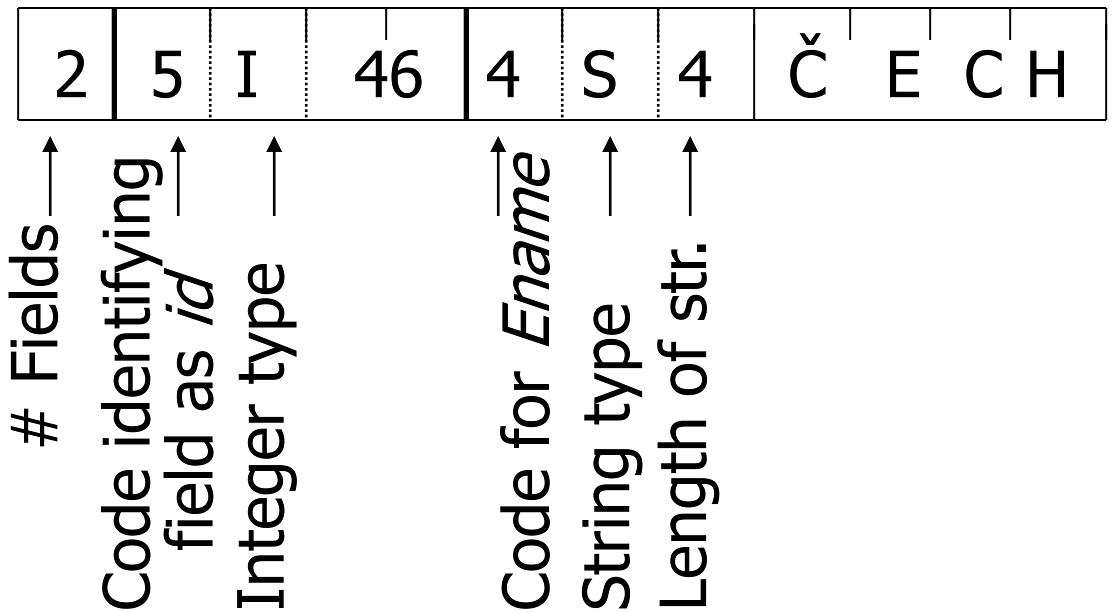
} records

- Padding to “convenient“ size
  - Faster memory access when address is round to 4 (8) bytes

55	n o v á k	02	- -

Example: Variable Length and Schema

- Employee:



Codes identify attribute names stored elsewhere; could be strings directly, i.e., tags.

- Called „Tagged fields“

Example (cont.): Repeating Attribute

- Employee's children

3	Name: Jan Novák	Child: Tomáš	Child: Pavel

  - Useful in case of arrays, etc.

- Repeating attribute may not mean variable length either schema
  - Can set maximum number of values
    - Unused space filled with NULLs

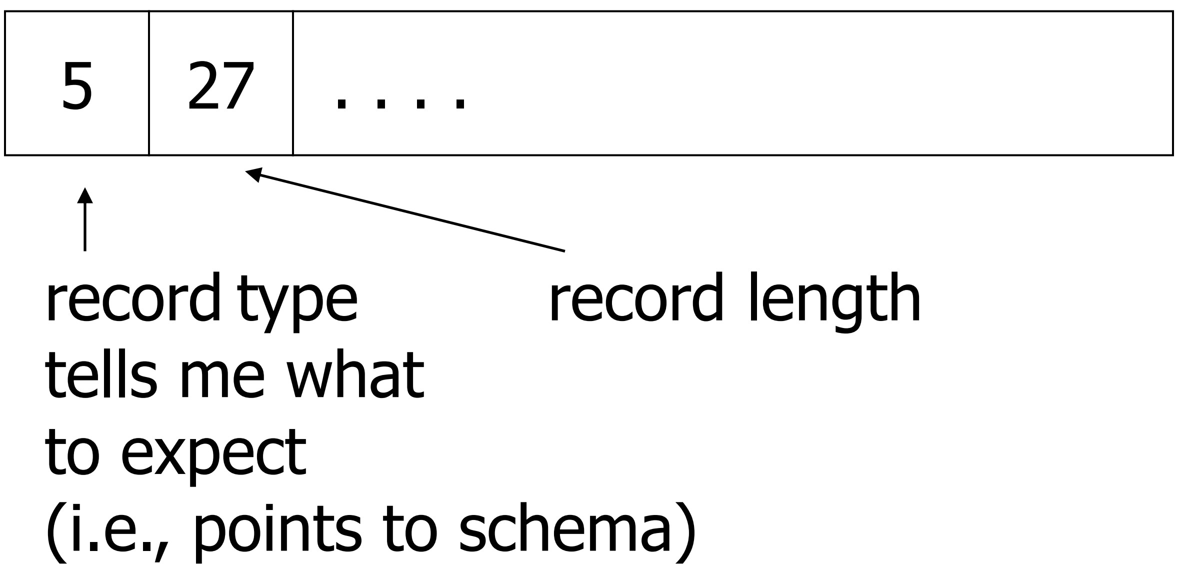
Novák	Potápění	Šachy	--

Schema Changes

- Fixed schema
  - every record must be updated
- Variable schema
  - only changed records get updated

„Intermediate“ Schema

- Compromise between fixed and variable schema
- Record schema “version” in record header



Record Header

- = information about the record (fixed length; no relation to attribute values)
  - Record schema "version" (pointer)
  - Record Length
  - Creation / update / access timestamp
  - OID (Object Identifier) – "record ID", "tuple ID"
  - Bit array of NULL value flags
    - One bit for each attribute
  - …

Other Issues

- Compression
  - Increase speed of accessing/updating (fewer bytes)
  - Within record (values independently)
  - Collection of records
    - More effective (can build a dictionary, find common patters)
    - More complex to implement
- In Pg, LZ4 and PGLZ
  - Lempel, Ziv based lossless algorithm
    - high compression performance (>0.5GMB/s per core)
    - extreme decompression perf. (>6 GB/s per core)

Other Issues

- Encryption
  - Consequence to indexing…
  - How to do range queries?
  - …
  - Solution:
    - Encrypt buffer data during file system I/O
    - WAL records stored in WAL buffers that get encrypted when writing to the file system

Storing Objects

- Current commercial DBMS support objects
  - Extension of relational DBMS
  - OODBMS
- Objects have attributes
  - Primitive types $\rightarrow$ store as a record
  - Collections $\rightarrow$ create a new relation
    - Referencing using OIDs

Storing Relations

- Row-oriented
  - Tackled up to now…
    - Example of row-oriented storage:
      - Order(id, cust, prod, store, price, date, qty)

id1	cust1	prod1	store1	price1	date1	qty1
id2	cust2	prod2	store2	price2	date2	qty2
id3	cust3	prod3	store3	price3	date3	qty3

- Column-oriented
  - Values of the same attribute stored together

Column-oriented Storage

- Relation
  - Order(id, cust, prod, store, price, date, qty)

id1	cust1
id2	cust2
id3	cust3
id4	cust4
...	...

id1	prod1
id2	prod2
id3	prod3
id4	prod4
...	...

...

Id may or may not be stored.
Could exploit record ordering

Comparison

- Advantage of column-oriented storage
  - More compact (no padding to 4/8 bytes, compression, …)
  - Efficient access (e.g., data mining)
    - Process few attributes but all their values
- Advantage of row-oriented storage
  - Record update / insertion more efficient
  - Whole record access more efficient

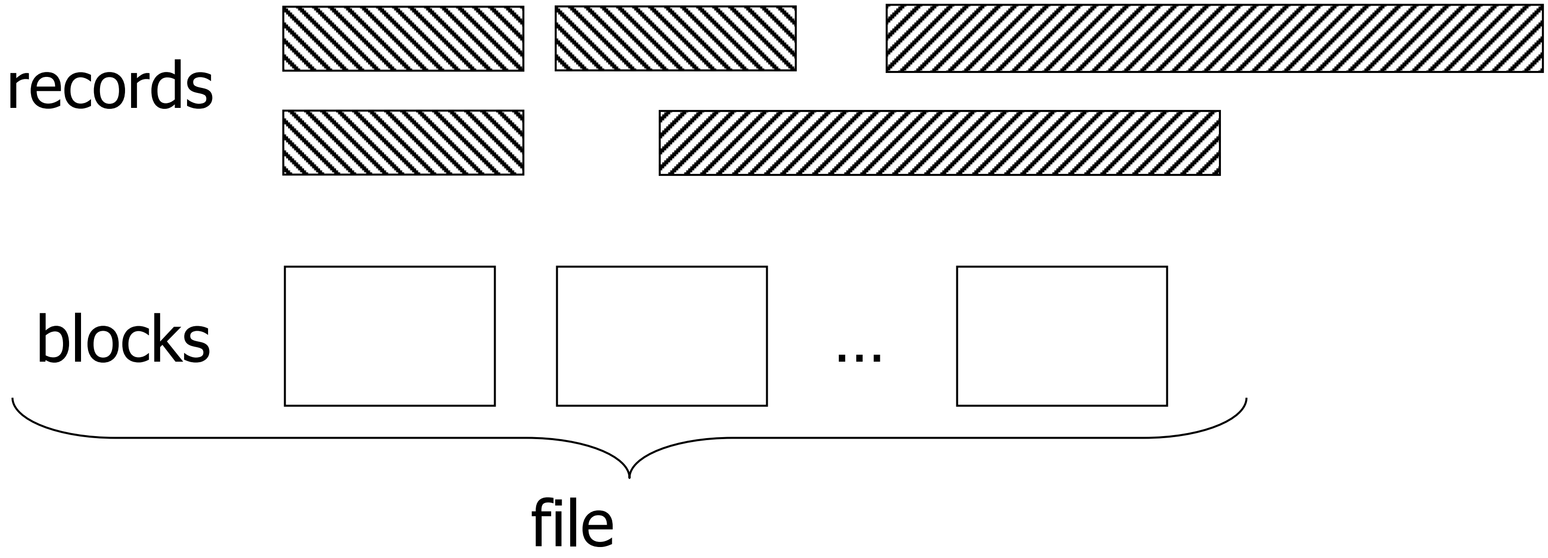
Mike Stonebraker, Elizabeth O'Neil, Pat O'Neil, Xuedong Chen, et al.: *C-Store: A Column-oriented DBMS*, VLDB Conference, 2005.
http://www.cs.umb.edu/~poneil/vldb05_cstore.pdf

Outline

- Data elements and fields
- Records
- *Block organization*
- Properties and examples

Block Organization

- Records
  - Fixed length
  - Variable length
- Block of fixed length



Block Organization

- Issues for storing records in blocks:
  1. Separating records
  2. Spanned vs. unspanned records
  3. Sequencing
  4. Interlacing more relations
  5. Indirection

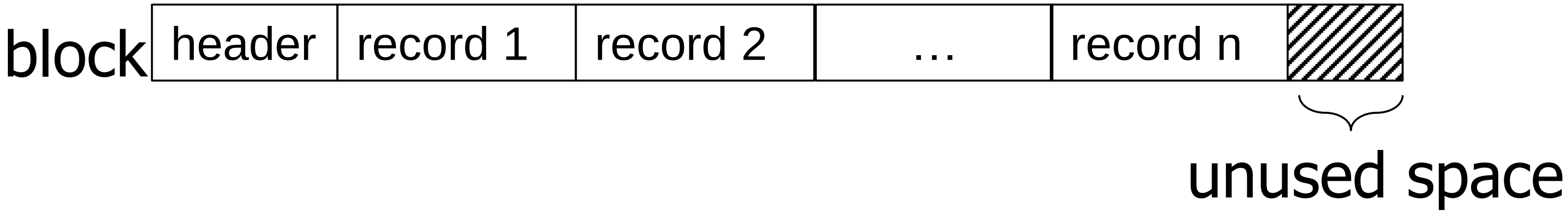
Separating Records

- Fixed-length records
  - No delimiter
  - Store record count and point to 1st record
- Variable-length records
  - Delimiter / special marker
  - Store record lengths (or offsets)
    - Within each record
    - <u>In block header</u>

Separating Records

- Variable-length records

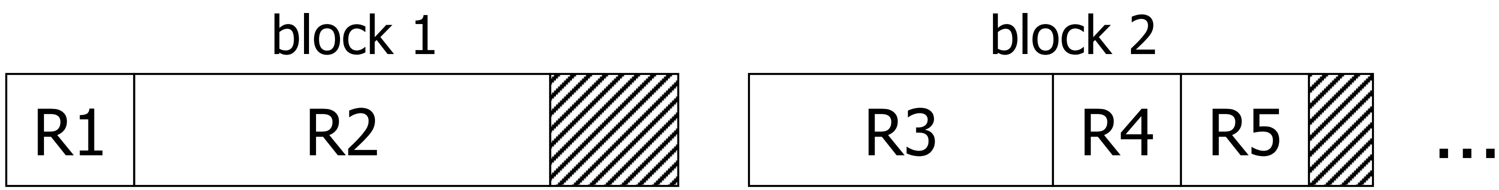
- Organization: block header, records



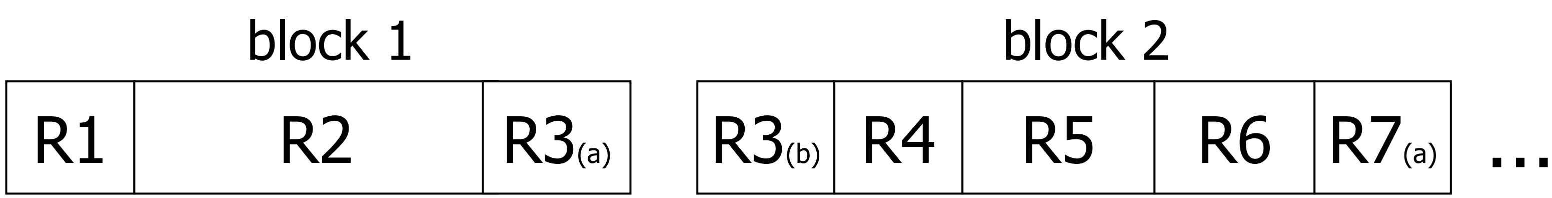
- Header
  - Pointers to other blocks (overflow, index, …)
  - Block type (relation, overflow, index, …)
  - Relation ID
  - (Directory of record offsets)
  - Timestamps (creation, modification, access)

Spanned vs. unspanned

- Unspanned
  - each record in a block
  - simple, but not space efficient

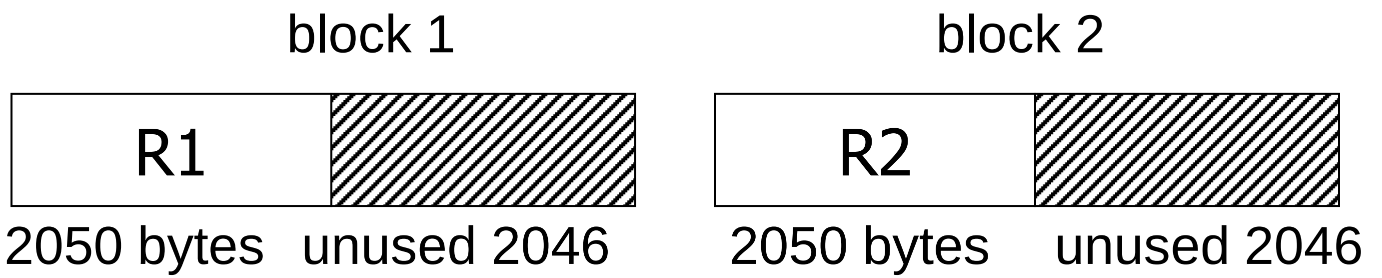


- Spanned
  - record split across blocks
  - required when <u>a record exceeds block size</u>!



Unspanned: Example

- Records cannot cross block boundary
  - 10^6 records, each 2 050 bytes (fixed length)
  - Block size 4 096 bytes



  - Space allocated: 10^6 * 4096 B
  - Space utilized: 10^6 * 2050 B
  - Utilization ratio: 50.05%

Unspanned

- Options for *oversized* attribute values
  - The Oversized-Attribute Storage Technique
    - TOAST or *"the best thing since sliced bread"***
    - Principle
      - A TOAST table is created (chunk_id, chunk_seq, value)
      - Value is split into “chunks”
        - Chunks form records in TOAST table
        - Chunk identified by (chunk_id, chunk_seq)
      - Original space is used to store length of the value, toast table id and chunk id.
  - Compression
  - Split into multiple records within the table (internally)

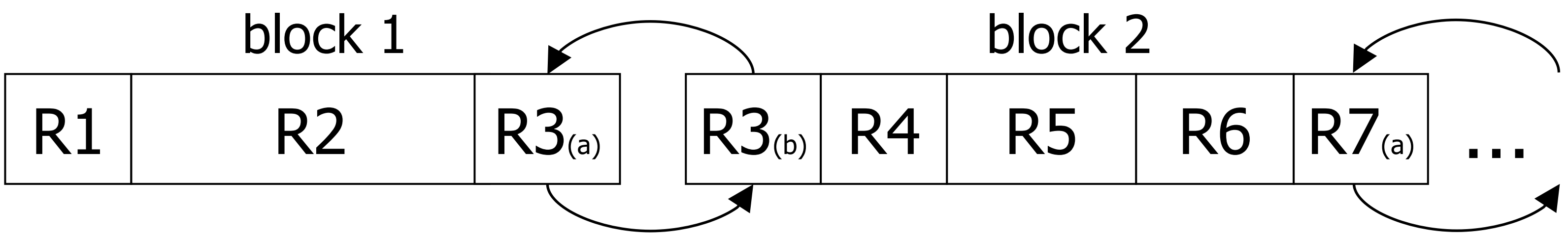
** [cit. dokumentace PostgreSQL]

Unspanned

- Large Objects (LOBs)
  - Two types: binary / text
  - Stored off the table
    - in consecutive blocks (in a separate file)
  - Typically, not indexed by DBMS
    - i.e., cannot search in the value

Spanned

- Record split across blocks
  - Blocks must be ordered or
  - Use pointers



- Record split into "fragments"
  - Bit flag "fragmented" in header
  - Pointers to next / previous fragments

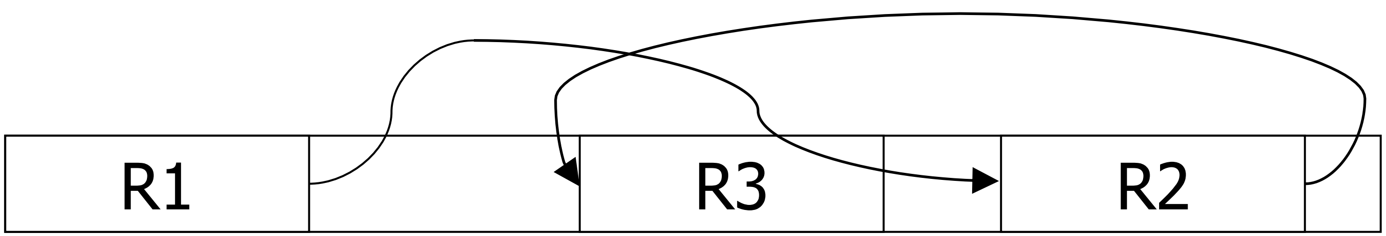
Sequencing

- Ordering records in file (and blocks)
  - by some key value
  - => <u>sequential file</u>
- Reason:
  - Efficient record access in the key ordering
  - E.g., good for merge-join, order by, …
- Solution in DBMS
  - <u>Clustered index</u>

Sequencing – sequential file

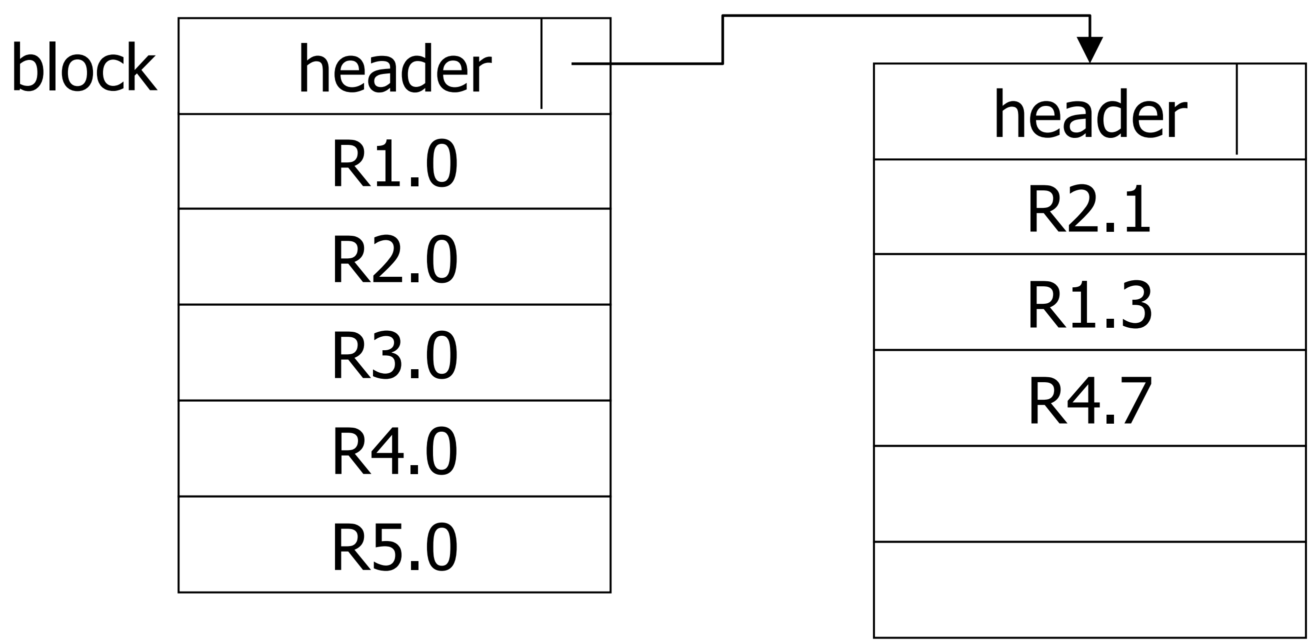
- Stored consecutively
  - physically contiguous

- Linked
  - preserve order!



Sequencing – sequential file

- Overflow area
  - Records in sequence
    - reorganization needed after record modifications
  - Pointer to an overflow area / block

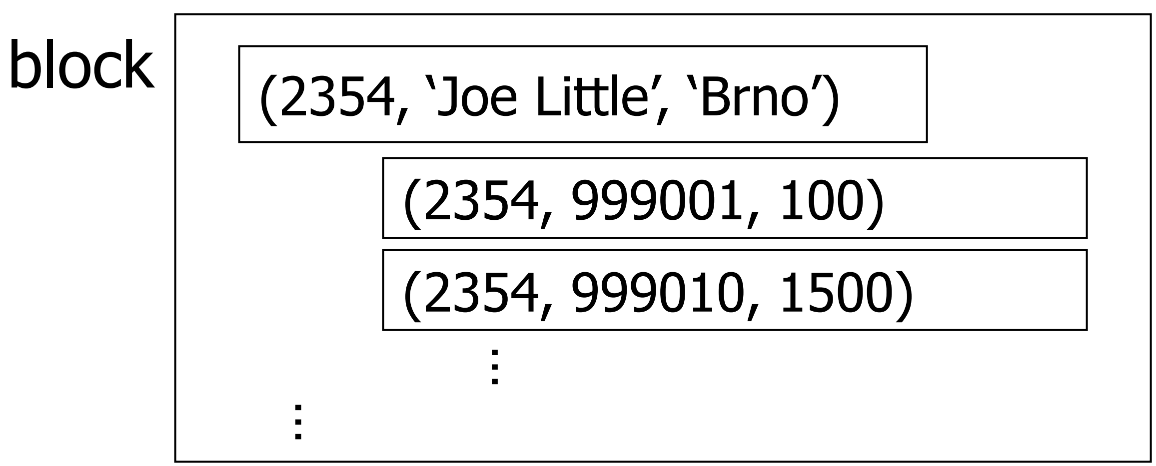


Relation Interlacing

- Records of multiple tables in one block
  - Records of more relations accessed simultaneously
  - Store together $\rightarrow$ Access faster
  - More complex implementation

Relation Interlacing: Example

- Relations: employee (eid, name, address)
  deposit (eid, did, amount)
- Good for query Q1:
  - SELECT name, address, amount
    FROM deposit, employee
    WHERE deposit.eid = employee.eid AND employee.eid = 2354

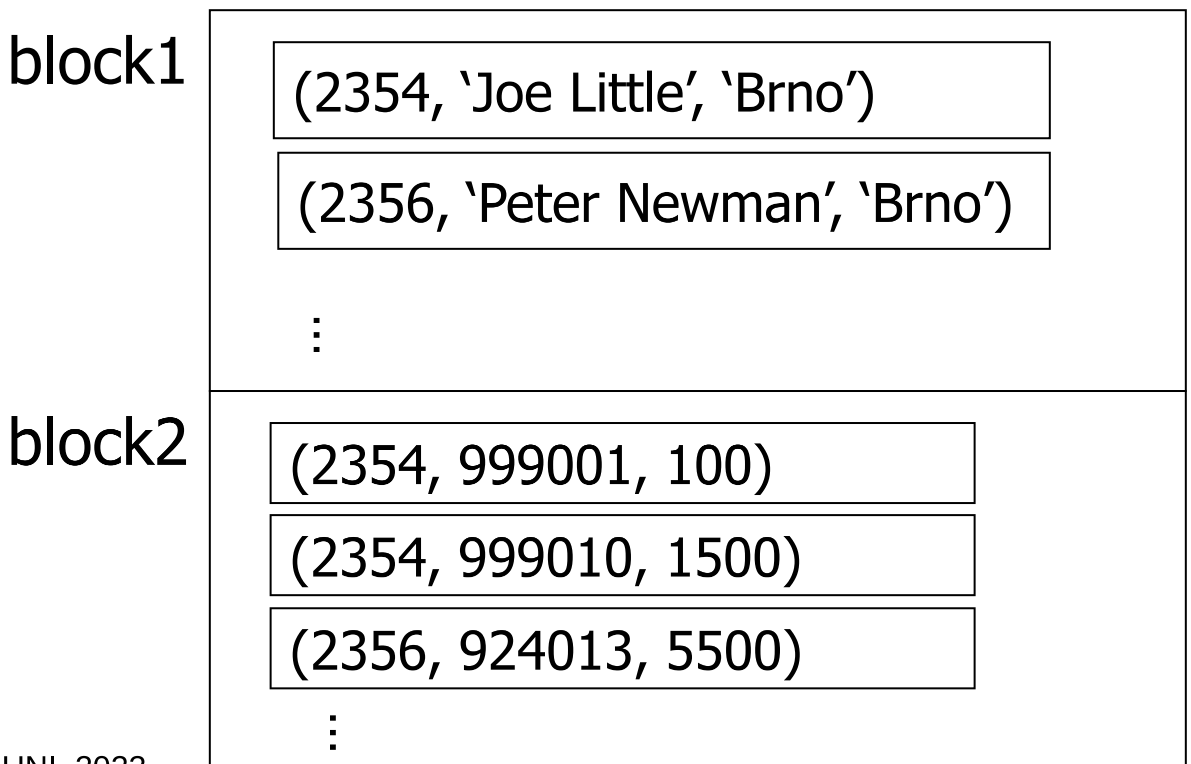


Relation Interlacing: Example

- Query Q2:
  - SELECT * FROM employee
- Interlacing not convenient for Q2
  - Depends on frequency of individual queries

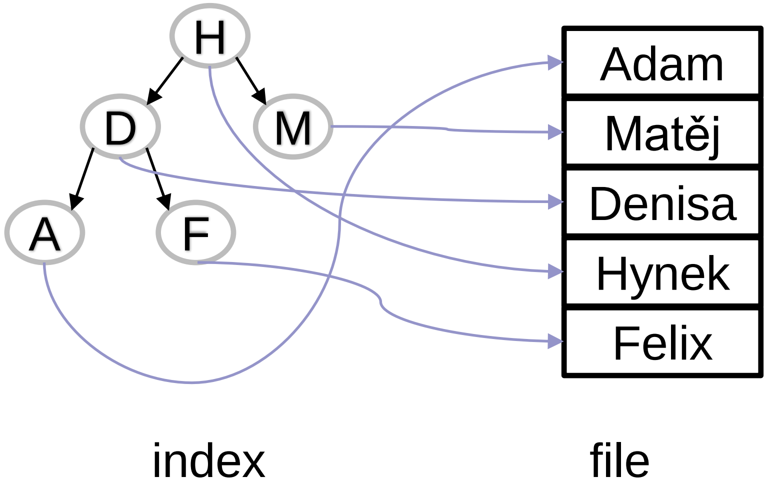
Relation Interlacing

- Solution:
  - Do not mix within one block
  - Store block in near proximity (same disk cylinder)



Indirection (Pointers to Records)

- Applications:
  - Spanned records
  - Referencing blocks / record (e.g., in indices)
  - Linked blocks (e.g., in indices)
  - OODBMS: objects referencing other objects



Indirection

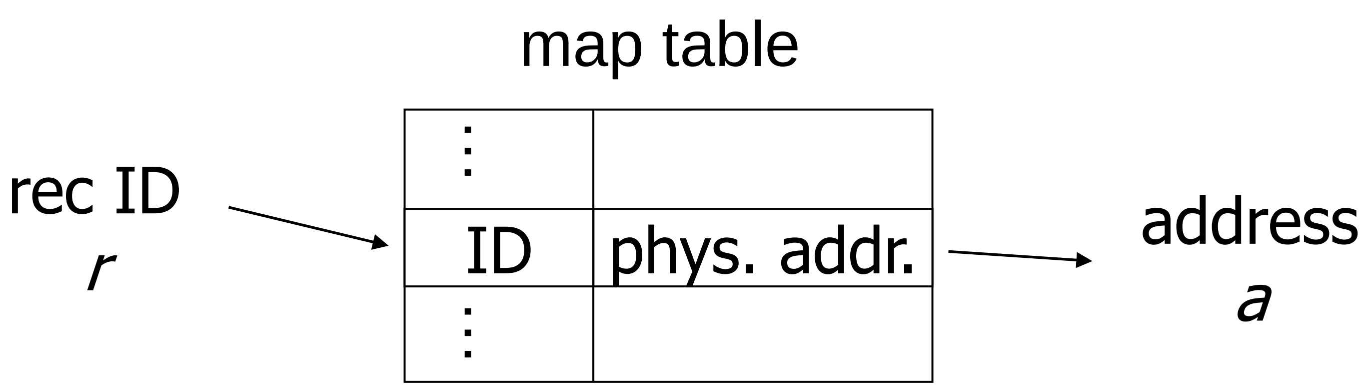
- Record address
  - Memory address
    - direct addressing
    - 8-byte pointer in the virtual memory of a process
  - DB address
    - sequence of bytes describing record location in external memory
    - direct vs. indirect addressing

Indirection in DB (DB Address)

- Direct addressing
  - Physical record address
    - Purely physical address in storage
      - Device ID, track, platter, block, byte-offset in block
  - Not flexible
    - E.g., block or records reallocation

Indirection in DB (DB Address)

- Indirect addressing
  - Record / block identified by its ID
  - ID = logical address
    - any sequence of bits
  - Map table: ID $\rightarrow$ physical address



Indirection in DB (DB Address)

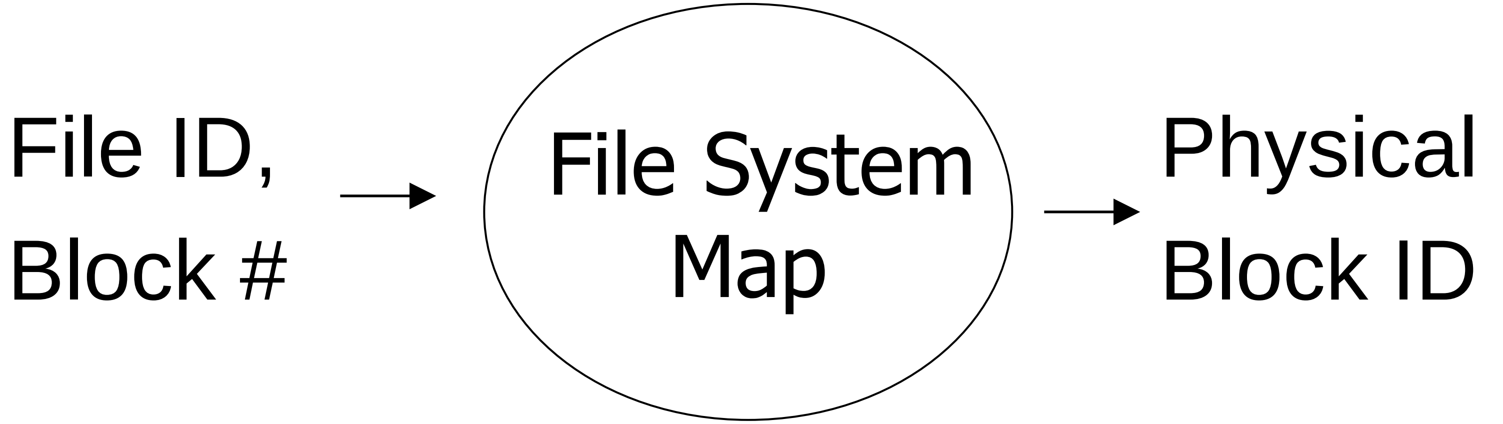
- Indirect addressing
  - Disadvantage
    - Increased costs
      - Accessing map table
      - Storing map table
  - Advantage
    - Very flexible
      - Deletion/insertion of records
      - Optimization of block storage

Indirection in DB (DB Address)

- Combination = suitable option
  - Phys. record address =
    phys. block address + position
    - The position in a list of records within block
      - At this position, byte-offset to the record is stored
  - Advantages
    - Can move records within block
      - No changes to phys. address
    - Map table is not necessary
  - Disadvantage
    - Minor: Moving a record to another block
      - Replace it with a pointer to new location (block + position)
    - Major: Not flexible in moving blocks (defragmentation)

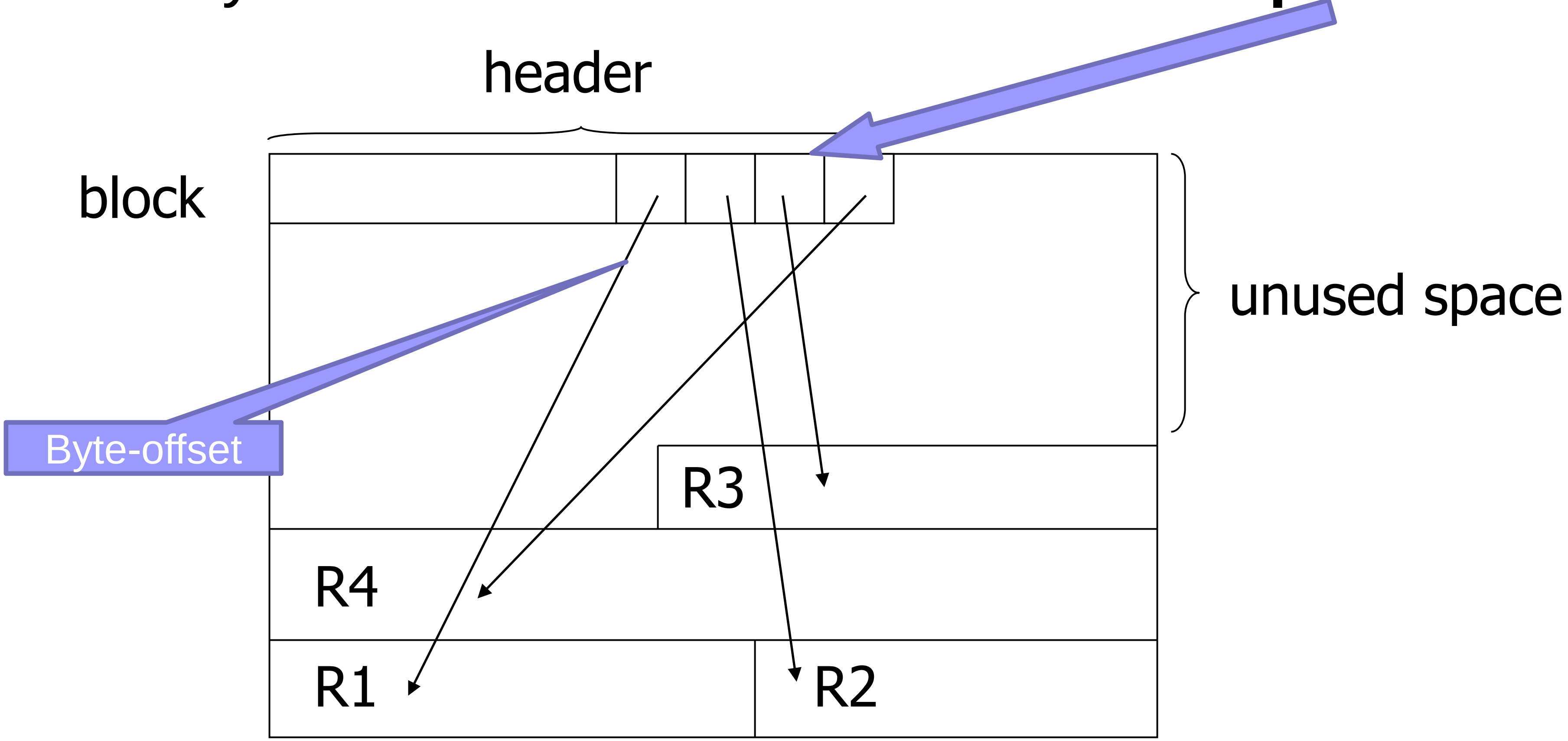
Indirection in DB (DB Address)

- Widely used option
  - Record address =
    File ID + block number + position
  - Blocks are organized by a file system
    - blocks are numbered from zero within each file



Indirection in DB (DB Address)

- Indirection in block (Slotted Page Structure)
  - Phys. record addr. = file id + blck no. + **position**



Block Header

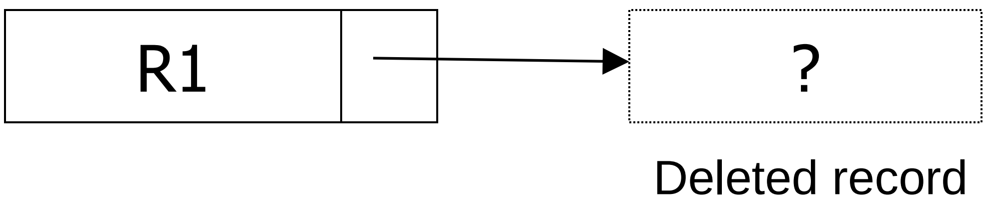
- Present in each block
  - File ID (or RELATION ID or DB ID)
  - Block type
    - e.g., record of type, overflow area, TOAST table, …
  - Block ID (this one)
  - Record directory (points to record data)
  - Pointer to free space (beginning, end)
  - Pointer to other blocks (e.g., in indices)
  - Modification timestamp/version number

Record Modifications

- Insertion
  - Typically, “no problem”
- Deletion
  - Unused space management
- Update
  - Same size
    - Ok
  - Enlarging/shrinking
    - Same issues as for insertion/deletion

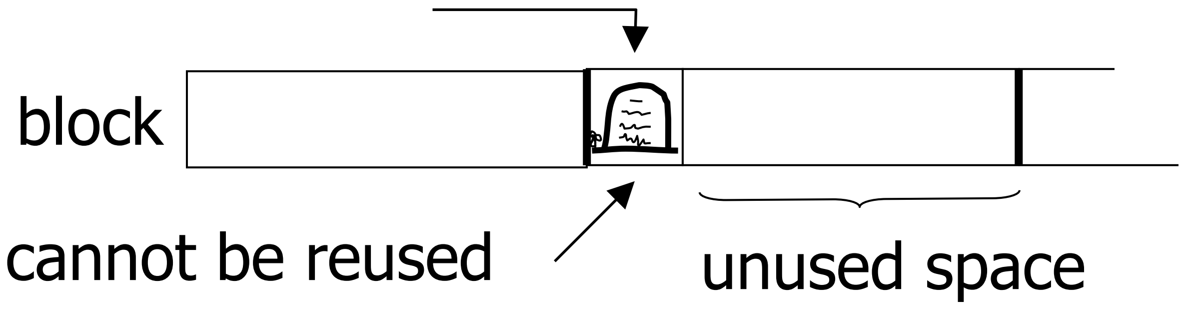
Deletion

- Pointer to deleted records
  - Must be invalidated
  - Cannot point to new data
  - *Dangling pointers*



Deletion

- Direct record addressing (phys. addr.)



1. Mark as deleted
   - With a marker (tombstone)
   - One bit
     - Reality: several bytes due to memory padding
2. Advertise the free space
   - Linked list of unused areas

Deletion

- Indirect addressing
  - *Map table*
  - Deleted record is freed in the block
  - Tombstone in map table

Map Table

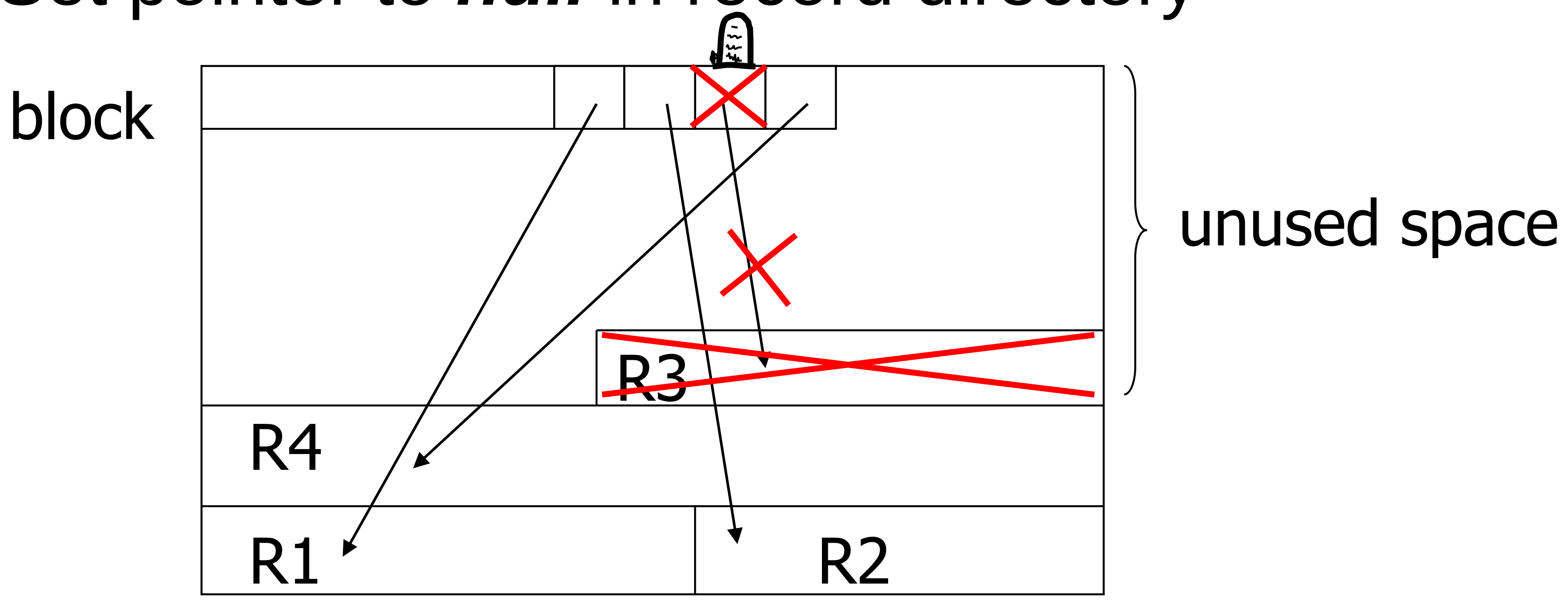
ID	LOC
⋮	
7788	
⋮	

← Cannot be reused

- or mapping is deleted, but no ID reuse!

Deletion

- Rec. addr. = block addr. + rec. position
  - Free space occupied by the record
  - Defragment space
    - to make it contiguous
  - Set pointer to ***null*** in record directory



Deletion

- Store rec. ID in the record
  - Check ID during record access
  - No overhead than extending the record with RecID
  - If RecID is the pointer itself, some other identification, e.g., xmin is necessary to differentiate the records.

Insertion

- Unordered file
  - Append to end of file
    - Last block, or allocate new
  - Insert into unused space of existing block
    - Need to handle variable length of records

Insertion

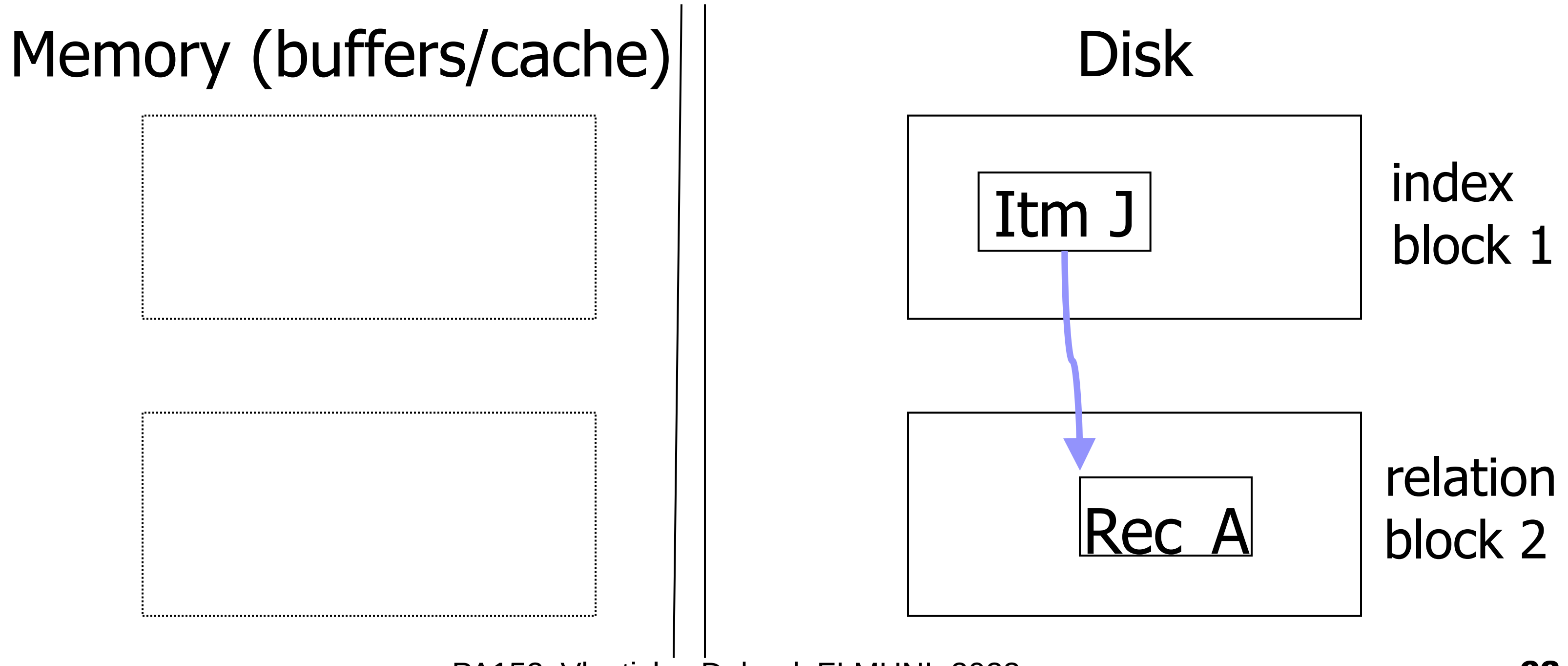
- Ordered file (sequential)
  - Unfeasible without indirect addressing nor record positions (offsets)
  - Find free space in a “neighboring” block → reorganize
    - Move last record in the block to the next block
      - Put a marker in the original place to point to the new location
  - Use overflow block
    - Pointer to an overflow block is in the block header

Update

- Record enlarged
  - Within a block
    - No need for tombstones
    - Move following records
  - Create an overflow block
  - …
- Record shrunk
  - by analogy…
  - May free overflow blocks

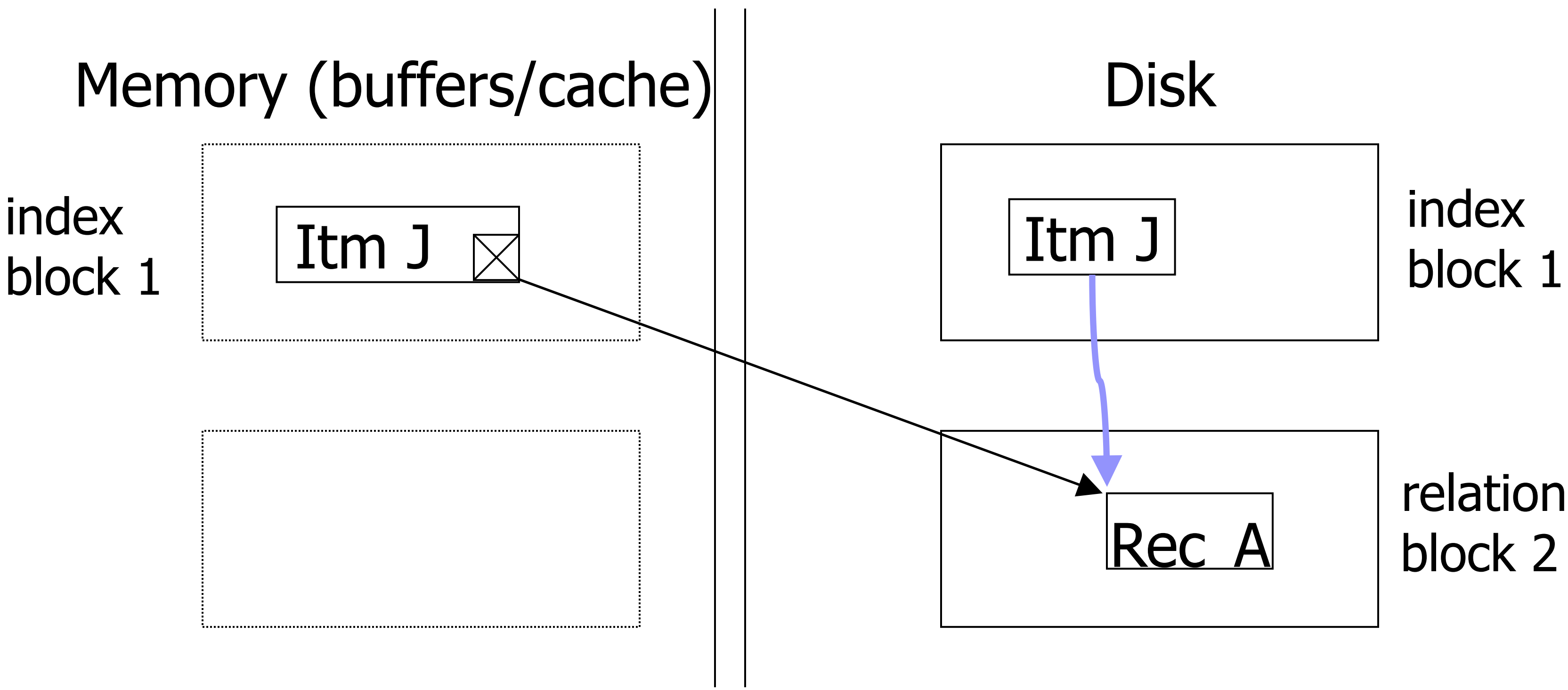
Memory Buffers and Pointers

- DB pointer in memory are inefficient
- Pointer swizzling
  - Change of DB pointer to memory pointer and back



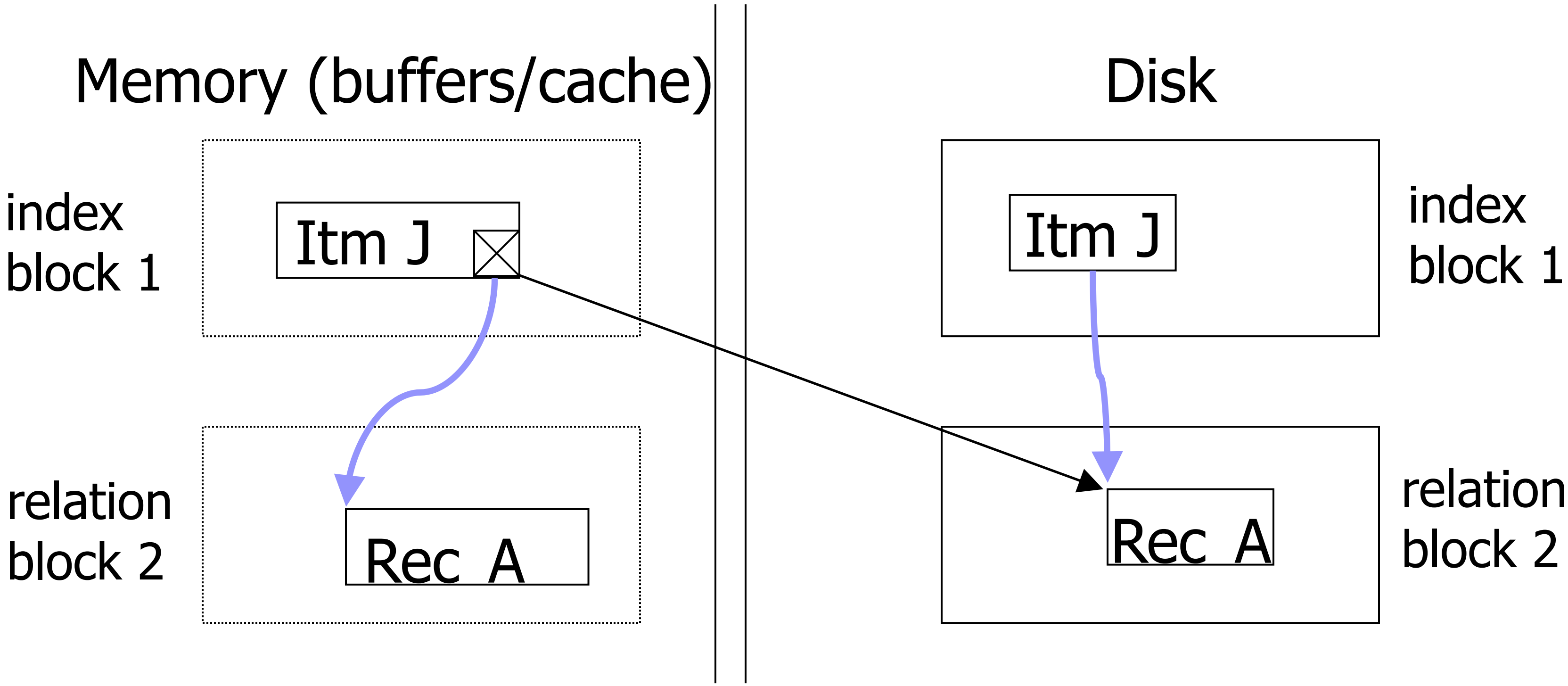
Memory Buffers and Pointers

- After loading block 1 in memory
  - no update is necessary



Pointer Swizzling

- After reading block 2, pointer updated



Pointer Swizzling

- When:
  - Automatically – immediately after reading
  - On request – on first use/access
  - Never – use map table instead
- Implementation:
  - DB address updated to memory address
    - Build a *Translation table*
      - store a pair (disk addr., memory addr.) for each record
  - Flag (swizzled/unswizzled) in the pointer

Buffer Management

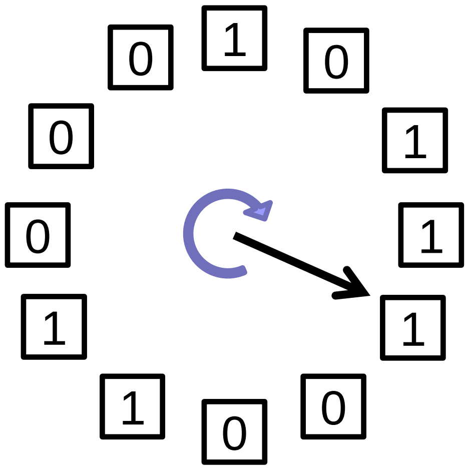
- DB features needed
  - Keep some blocks in memory/cache
    - Indices, join of relations, …
- Different strategies
  - LRU, FIFO, pinned blocks, toss-immediate, …

Buffer Management Strategies

- LRU
  - Update timestamp on access to block
  - → significant maintenance, but effective
- FIFO
  - Store time of loading, no update on access
  - → improper for highly accessed blocks
    - e.g., root of B^+ tree
- Pinned blocks
  - Blocks allocated in buffers forever

Buffer Management Strategies

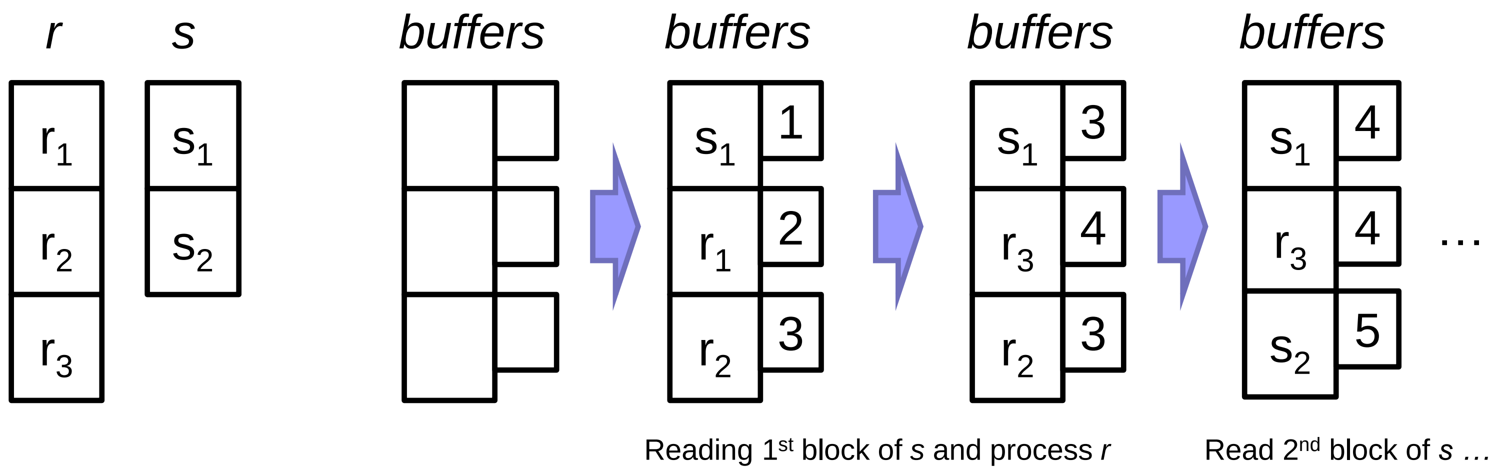
- "Clock" algorithm
  - Efficient approximation of LRU
  - Hand points to last read record
  - Rotates to find a block to be written back to disk and replaced (flag is 0).
  - On loading / accessing a block, set the flag to 1
  - Reset the flag on passing over the blocks
- Can implement *pinned blocks*. How?



Buffer Management: Example

- Join relations with LRU:
  - Blocked Nested loops:

For each b_s in *s* do
  For each b_r in *r* do
    For each t_s in b_s do
      For each t_r in b_r do
        Join tuples t_r and t_s

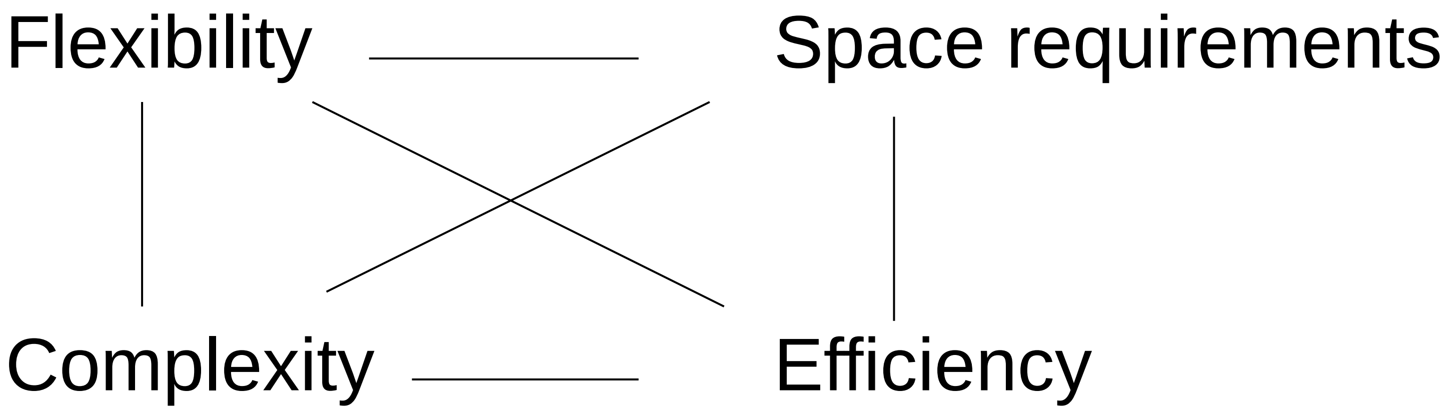


  - LRU ineffective: blocks to process removed
    - Need to *pin blocks* of relation *r*/*s*

Outline

- Data elements and fields
- Records
- Block organization
- *Properties and examples*

Own Implementation



- Access type (operations) and their efficiency:
  - Locating a record with given key
    - Getting consecutive records
  - Insert / update / delete of records
  - Table scan
  - File reorganization

Specialized Systems

- BigTable
  - Distributed storage for tuples, by Google
  - Scalable up to petabytes (1PB=1000TB)

    F. Chang, J. Dean, S. Ghemawat, et al.:
    Bigtable: A Distributed Storage System for Structured Data,
    Seventh Symposium on Operating System Design and Implementation (OSDI), 2006.
    http://labs.google.com/papers/bigtable-osdi06.pdf
- HBase
  - Distributed storage for tuples
  - Open-source Apache projekt Hadoop

    http://hadoop.apache.org/

Properties of BigTable and HBase

- Not traditional relational database systems
  - NoSQL databases
- Storage as a “key→value” map
  - row_id, column_id, time → value
  - Value is structured, but of variable schema
- Records are versioned
  - see time component in the key
- Ordered by row_id

Lecture's Takeaways

- Differences in storing values
  - Handling NULL values in attributes
- Organization of records in blocks
- Pointers in DBMS
  - Why and how; cooperation with memory pointers
- To revise
  - Sequential file
    - Record manipulation operations
  - Index files (sparse/dense indexes)