Representing Data Elements

Contains slides by
Hector Garcia-Molina, Vera Goebel

Overview

- ✓ Basic data representation – fields
- ✓ Records
- ✓ Data layout on disk
- ✓ Pointer management moving records
- ✓ Comparison

Representation of Basic Data Types

Data Representation

- ✓ Attributes need to be represented by fixed- or variable-length sequences of bytes called fields

- ✓ Fields are put in fixed- or variable-length collections called records

- ✓ Records are stored in physical blocks where design is dependent on access pattern, modification policy, having sorted records, ….

- ✓ Records belonging together (relation or extent) are stored together and form a file.

Basic Data Elements

- ✓ What do we want to store?
names, addresses, salaries, dates, times, pictures, sounds, videos, ….

- ✓ What is available: bits and bytes (0's and 1's)

- ✓ We must define a bit sequence within a byte (or a consecutive collection of bytes) that has a certain meaning

- ✓ Ultimately, all data is represented by sequence of bytes (operations on single bits is more expensive, makes storage more complex, …)

- ✓ A data element may be of *fixed length* or *variable length* (first, we assume only fixed length)

Numbers

- ✓ Numbers are easy – just a binary representation which allows the machine's hardware to perform arithmetic operations
- ✓ Integers:
  - ➢ short: 2 bytes
    e.g., 35 can be represented as $\boxed{00000000}$ $\boxed{00100011}$
    $(0 \times 2^{15} + \ldots + 1 \times 2^{6} + 0 \times 2^{5} + 0 \times 2^{4} + 0 \times 2^{3} + 0 \times 2^{2} + 1 \times 2^{1} + 1 \times 2^{0})$
  - ➢ long: 4 bytes
  - ➢ signed: first bit tells whether or not it is a negative number
  - ➢ unsigned: all bits are used for one positive number
- ✓ Reals:
  - ➢ *n* bytes for mantissa, *m* for exponent (e.g., 1 + 1)
    e.g., 35.4 can be represented as $\boxed{00100011}$ $\boxed{00000100}$

Characters – I

- ✓ Single characters - char:

  various coding schemes suggested, most popular is *ascii*

  e.g.:

A:	01000001
a:	01100001
5:	00110101
LF:	00001010

- ✓ Fixed-length character strings – char(n):

  array of characters, each coded as above, use *padding characters* to fill out all fields if string is shorter than n

  e.g.:

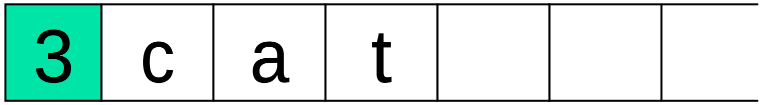
```
CHAR(5) x;
x = “cat”;
```

c	a	t	☒	☒

Characters – II

✓ “Variable”-length character strings – varchar(n): actually a fixed length string of n characters, but the text value has a length that varies

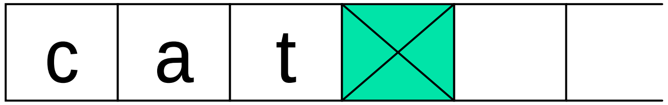
- *length + content* (n + x bytes):



3	c	a	t			

first x bytes indicate the length of the text value

- *null-terminated* (n + 1 bytes):



c	a	t	☒		

first bytes is used for the text value, string is ended with NULL character

Date and Time

- ✓ Date, e.g.:
  - ➢ integer, # days since Jan 1, 1900
  - ➢ 8 characters – YYYYMMDD
  - ➢ why not YYMMDD?
- ✓ Time, e.g.:
  - ➢ integer – seconds since midnight
  - ➢ characters – HHMMSS

Boolean and Enumerated Types

- ✓ Boolean
  - ➢ TRUE `1111 1111`
  - ➢ FALSE `0000 0000`
- ✓ Enumerated types – give a finite set of valid values, e.g., `enum {Mon, Tur, Wed,..., Sun} days;` can be represented by 1, 2, 3, ..., 7
- ✓ Can we use less than one byte per value (e.g., boolean as 1 bit, days as 3 bits)?

⇨ YES, but it is usually inconvenient (complex and error-prone operations – use only if storage shortage)

Records

Records

- ✓ Records are collections of related values – fields grouped together (typical values tuples or objects), e.g.:
  `Employee: name, gender, department, ...`

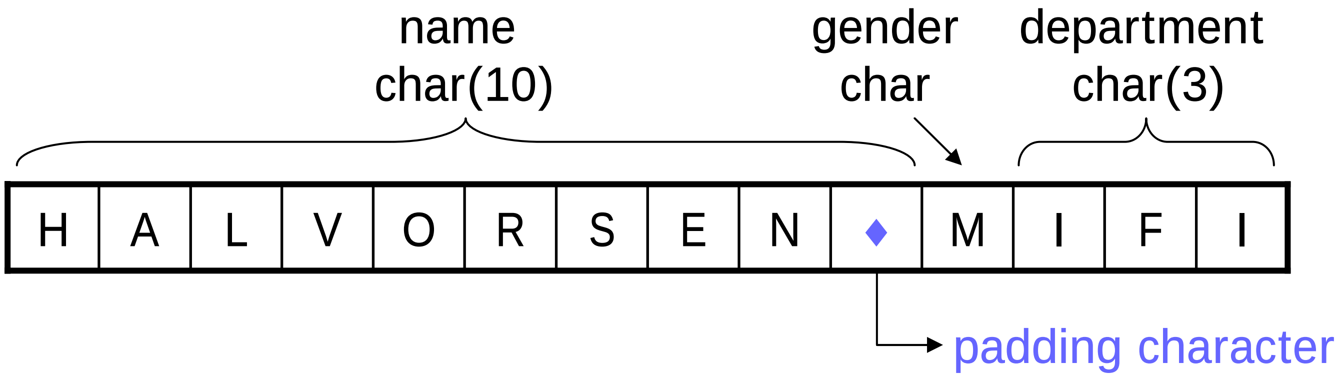
- ✓ A record type consists of *names* and *types* of the fields,
  e.g.:

```
name char(10),
gender character,
department char(3), ...
```

- ✓ Records may be of
  - ➢ fixed format or variable format
  - ➢ fixed length or variable length

Fixed Format and Length Records

- ✓ Easiest approach is to store each field (in its defined length) sequentially, e.g.;



name
char(10)

gender
char

department
char(3)

H	A	L	V	O	R	S	E	N	♦	M	I	F	I

padding character

Schema

- ✓ The record schema includes the record types and each field’s offset within the record

- ✓ The DBS maintains schema information which is essentially what appears for example in CREATE TABLE for a relation
  - ➢ attributes and their types (record schema)
  - ➢ order of the attributes (fields)
  - ➢ constraints such as keys, …

- ✓ The scheme is consulted when accessing a field

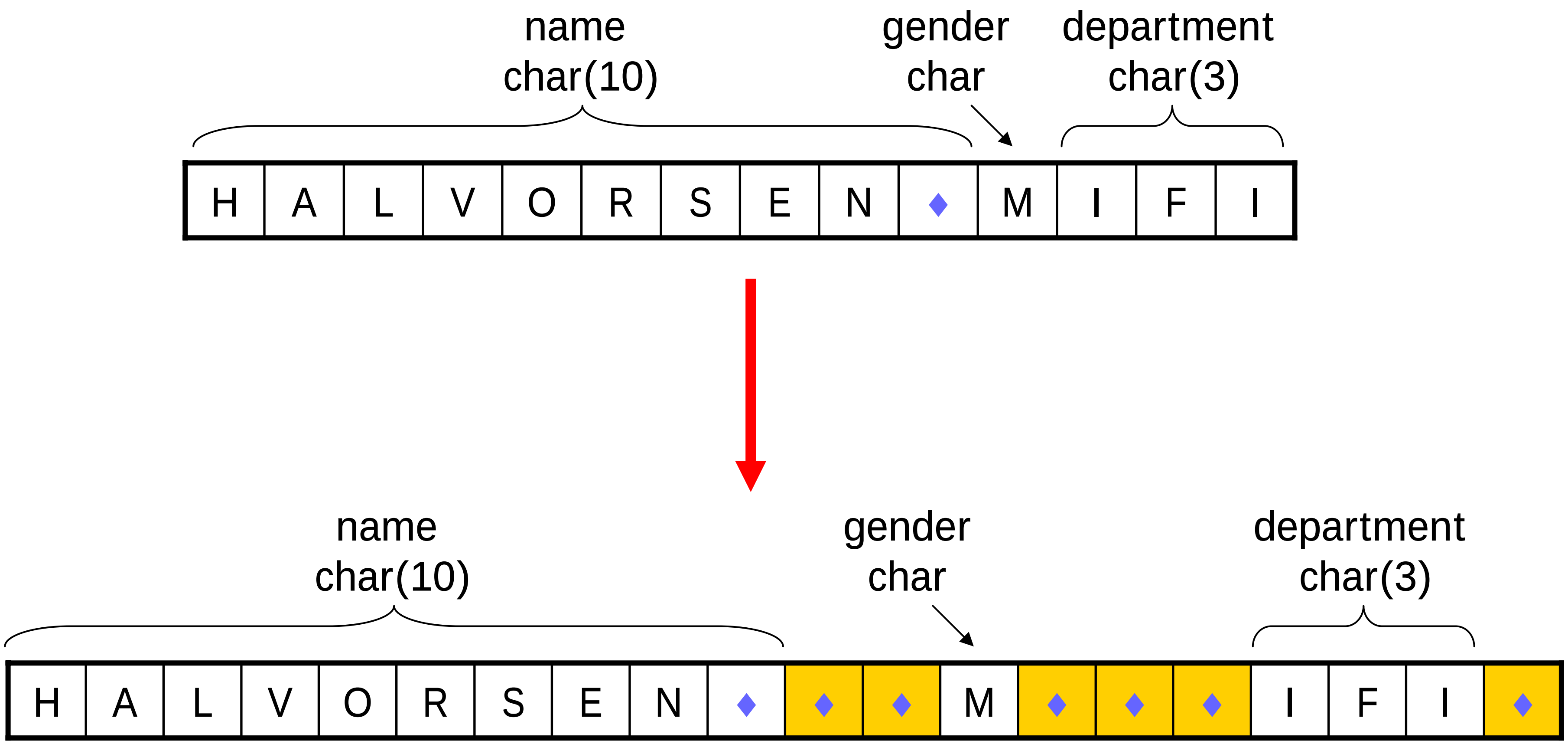
Data Alignment - I

- ✓ 4- and 8-byte alignment:
  some machines require (or are more efficient) if a field starts at a memory address that is 4- or 8-byte aligned
  - ➢ store data as on previous slide and align data when copying it to main memory
  - ➢ store data in an aligned form, i.e., each field is a multiple of the align-number, and just copy block to memory

- ✓ The last solution is usually preferred, i.e., all field sizes are rounded up to the next multiple of the align size

Data Alignment - II

- ✓ Fixed-length example (4-byte alignment):



name
char(10)

gender
char

department
char(3)

H	A	L	V	O	R	S	E	N	♦	M	I	F	I

name
char(10)

gender
char

department
char(3)

H	A	L	V	O	R	S	E	N	♦	♦	♦	M	♦	♦	♦	I	F	I	♦

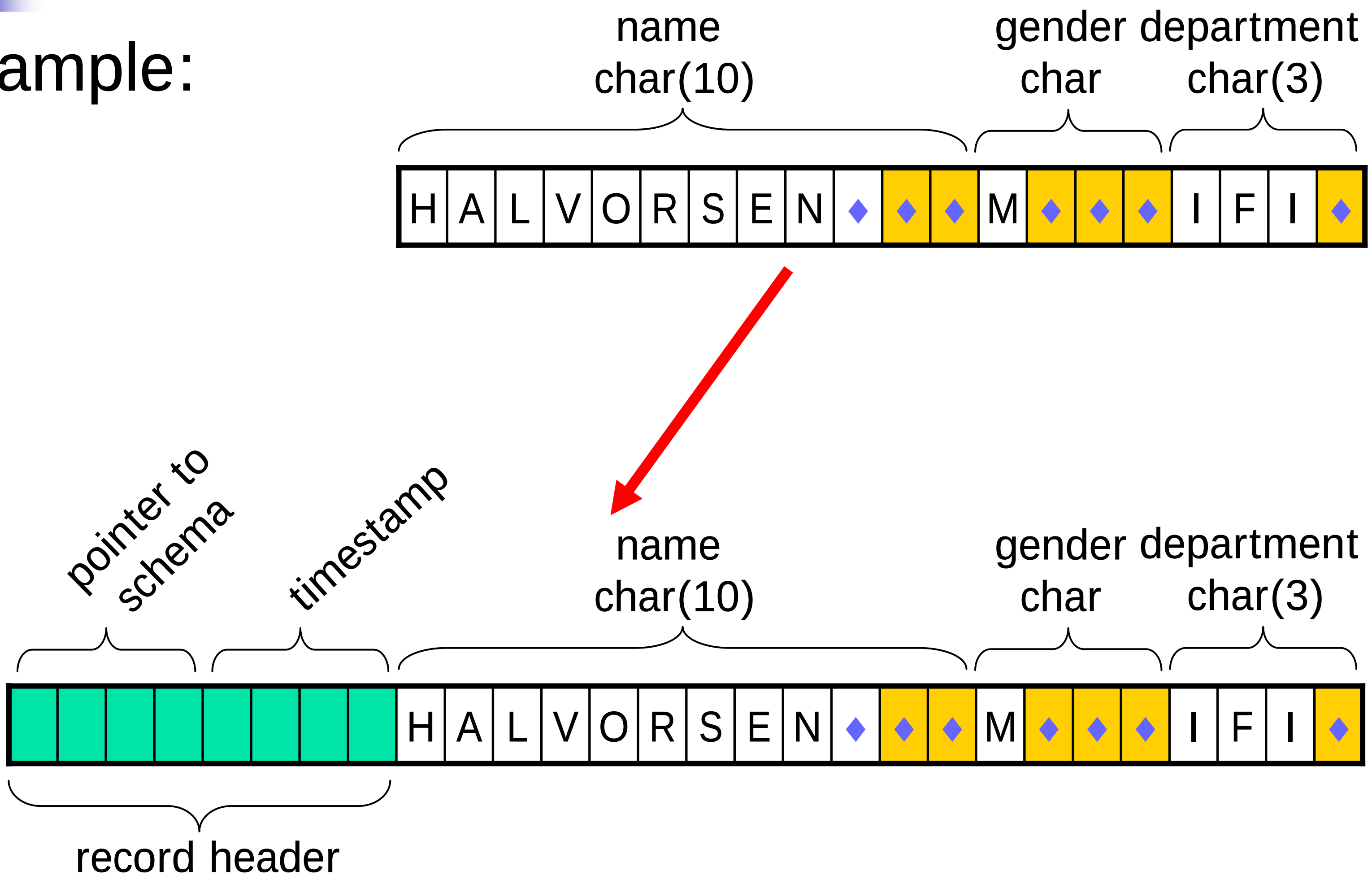
Record Headers – I

- ✓ There is often information about a record that is not a field, e.g.:
  - ➢ record schema
  - ➢ length of record
  - ➢ timestamps
  - ➢ ....
- ✓ This information is stored in a record header requiring additional bytes for this additional information
- ✓ Some of this information is equal for all records of this type, i.e., provide only a pointer
- ✓ Still, some information may be equal for each record (and deducible from schema), but still we might put it into the record header – why?

  → for example reducing accesses to slower storage – e.g.:
  - ➢ length of records if using clustering (described later)
  - ➢ ...

Record Headers – II

- ✓ Example:

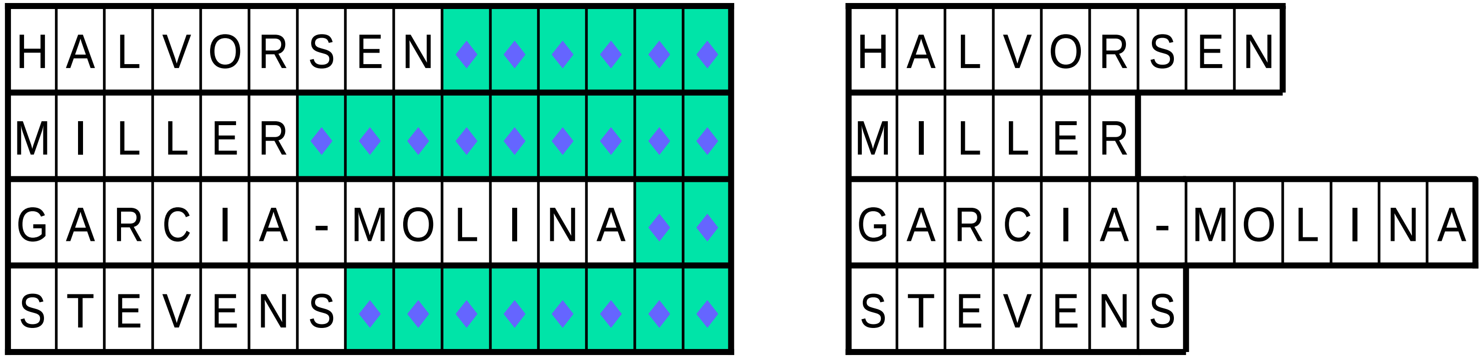


Note:
if we need x-byte alignment, each of the record header fields also must be a multiple of x

Variable
Fields and Records

Fixed Records and Variable-Length Fields – I

- ✓ Data items (stored in a field) where the size varies, e.g.:
  - ➢ text strings like name, address, …
    e.g., fixed vs variable declaration of names:



H	A	L	V	O	R	S	E	N	♦	♦	♦	♦	♦	♦
M	I	L	L	E	R	♦	♦	♦	♦	♦	♦	♦	♦	♦
G	A	R	C	I	A	-	M	O	L	I	N	A	♦	♦
S	T	E	V	E	N	S	♦	♦	♦	♦	♦	♦	♦	♦

H	A	L	V	O	R	S	E	N				
M	I	L	L	E	R							
G	A	R	C	I	A	-	M	O	L	I	N	A
S	T	E	V	E	N	S						

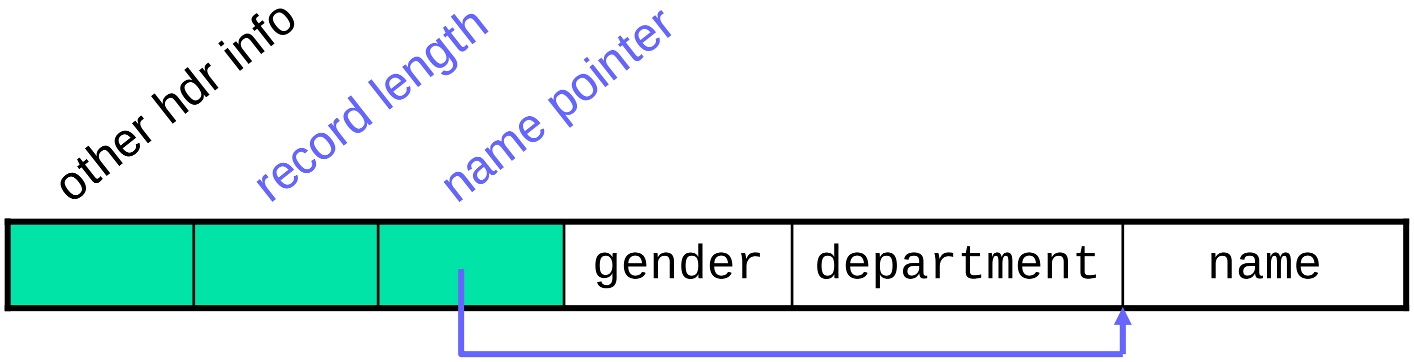
  - ➢ large data items like pictures, audio clips, video clips, …
    e.g.: the size of a video may vary according to length, encoding format, frame rate, color depth, resolution, …

→ waste of space to make the field large and fixed size to hold the largest element, if the average is much less

Fixed Records and Variable-Length Fields – II

- ✓ If one or more fields in a record have variable size, the record header must contain enough information to find any field
  - ➢ add record length in record header
  - ➢ put all fixed length fields first
  - ➢ add pointers (offsets of first byte) to variable-length fields in record header

- ✓ Example
  `Employee: name, gender, department_code`



Repeating Record Fields – I

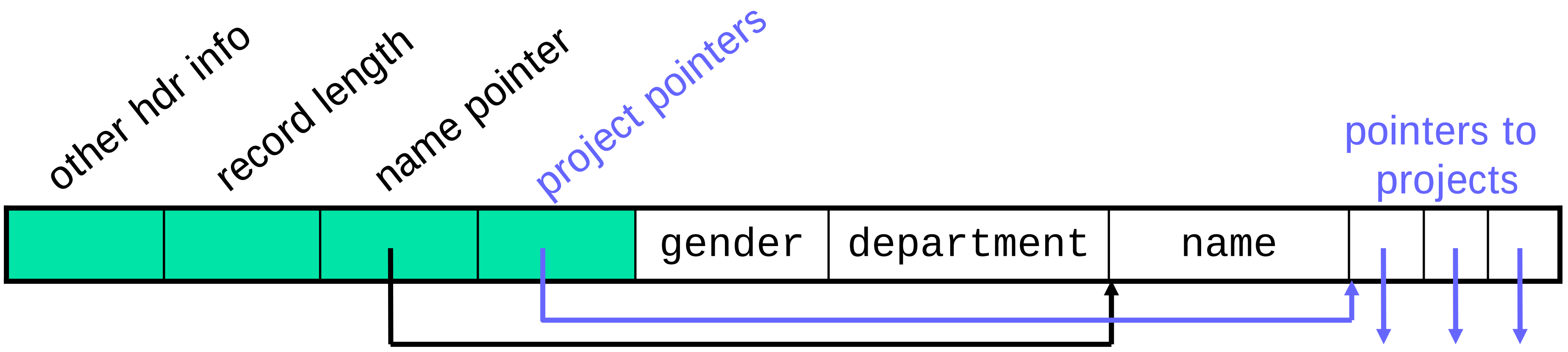
- ✓ Records may contain a variable number of a field F, e.g.:
  - ➢ representing a one-many-relationship in ODL objects, e.g., set of children
    (in the relational model we would have a connecting relation)
  - ➢ having a collection type as attribute type, e.g., set of phones

- ✓ "Solution" 1:
  Group all occurrences of F and treat as a variable length field
  - ➢ add pointer (offset to first byte) to first element
  - ➢ if each field F is L bytes long, element `i` is accessed by
    `offset + ((i – 1) x L)`
  - ➢ the final element is found by comparing with offset of next field or record length

Repeating Record Fields – II

- ✓ Example

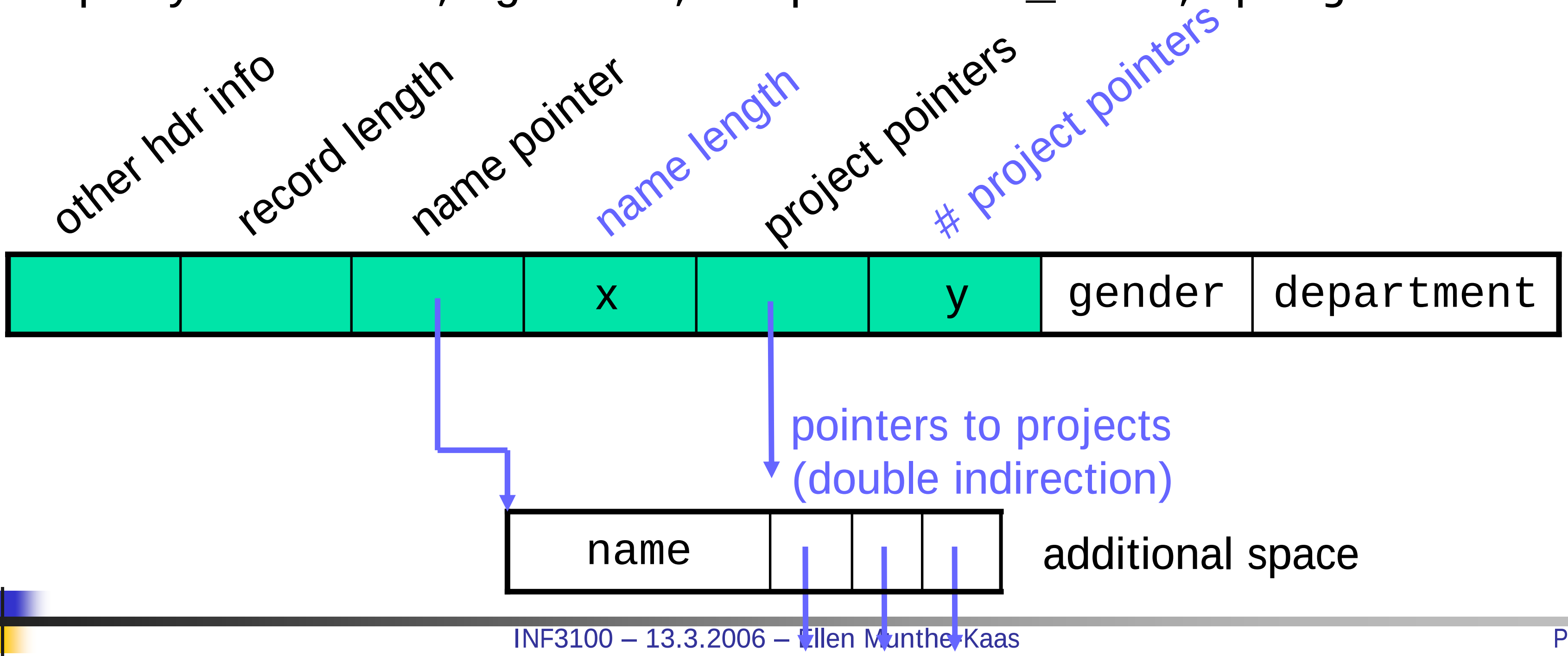
```
Employee: name, gender, department_code, projects
```



(not exactly double indirection, but close)

Repeating Record Fields – III

- ✓ "Solution" 2:
  Keep fixed length record and put the variable-length portion on separate block, add for each variable-length field
  - a pointer in record header to place where the field starts
  - either a counter, total length, or end address
- ✓ Example
  `Employee: name, gender, department_code, projects`



Repeating Record Fields – IV

- ✓ *"Solution" 1* (variable-length record):
  - ☺ less block accesses (possible disk I/O's) examining all fields
  - ☹ random record access requires reading all headers
  - ☹ more complicated to move records around
- ✓ *"Solution" 2* (fixed-length record + indirection)
  - ☺ eases searching as record *i* is accessed by (*i-1) x record size*
  - ☺ easy to move records around
  - ☹ several blocks must be accessed to get whole record
- ✓ A *compromise* is to have a fixed-length record holding
  - ➢ some repeating fields
  - ➢ pointer to where additional occurrences can be found
  - ➢ count of how many additional occurrences there are

Variable-Format Records – I

- ✓ Records do not necessarily have fixed formats, i.e., fields and their orders may vary during run-time

- ✓ Record itself contains format (“self describing”) – using tags
  - ➢ name
  - ➢ type
  - ➢ length
  - ➢ value

role of field

Variable-Format Records – II

- ✓ For what is variable-format records useful?
  - ➢ information-integration applications, for example using XML and semi-structured data models, like data warehousing and mediation
  - ➢ records with a flexible schema, e.g., an attribute may not appear at all (allowing NULL values)
  - ➢ ...
- ✓ Example: author record with tagged fields

code for name	code for string type	length	value	code for book title	code for string type	length	value
N	S	6	Ullman	T	S	35	Database Systems: The Complete Book

Question

- ✓ We have seen examples for
  - ➢ fixed format and length records
  - ➢ variable format and length records

- ✓ Does *fixed format* and *variable length* make sense?

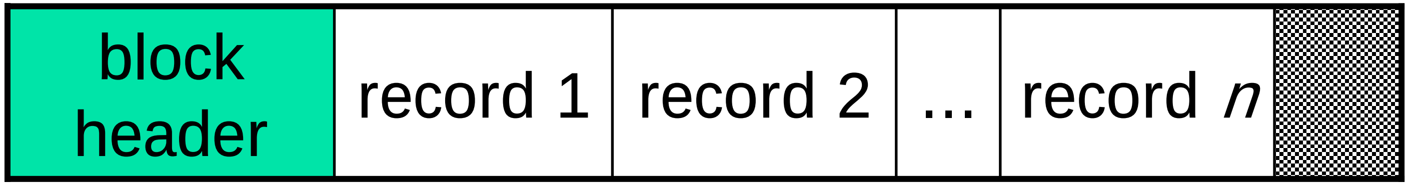
- ✓ Does *variable format* and *fixed length* make sense?

- ✓ Does it make sense to have *hybrid* formats?

Placing Records into Blocks

Disk Blocks

- ✓ Records representing tuples of a relation or objects of an extent of a class are stored on disk



block header | record 1 | record 2 | … | record *n* | ▨

- ✓ The disk block header is optional and *may* contain
  - block ID
  - directory with offsets of each record
  - modification and access timestamps
  - information about which relation(s) the tuples belong to
  - links to other blocks
  - ...

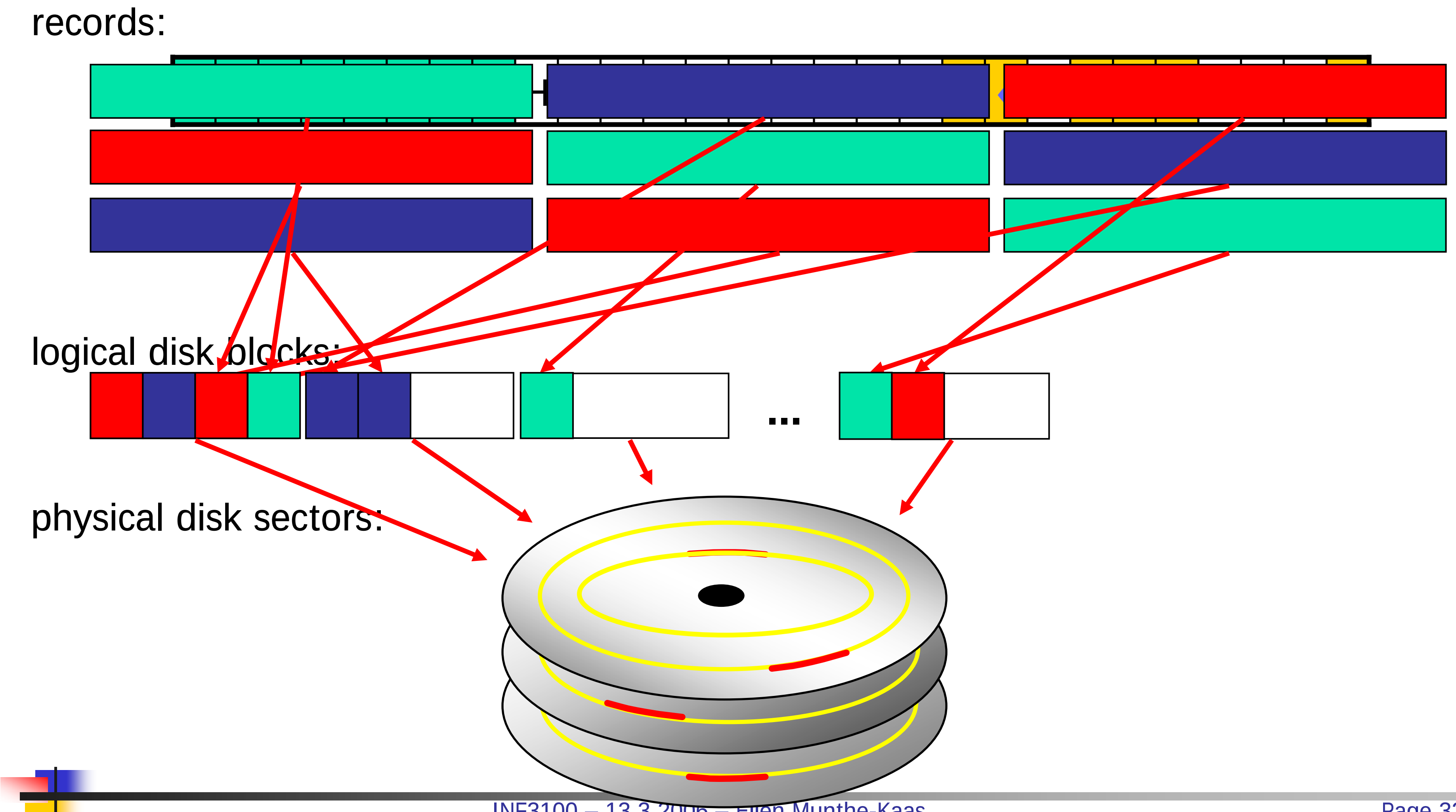
Allocating Disk Blocks

- ✓ Contiguous allocation:
  store file in contiguous blocks on the disk
  - ☺ fast to read whole file
  - ☹ update file difficult
- ✓ Linked allocation:
  each block has a pointer to next block
  - ☺ easy to expand file
  - ☹ slow to read whole file or a random block
- ✓ Cluster allocation:
  several contiguous blocks (segments) and linking several segments with pointers
- ✓ Indexed allocation:
  having index blocks pointing to actual file blocks (e.g., I-node)

- ✓ Different combinations of the above schemes …

Placing Records into Blocks – I

- ✓ We have seen different kinds of records
- ✓ How do we put the records on disk blocks?



records:

logical disk blocks:

...

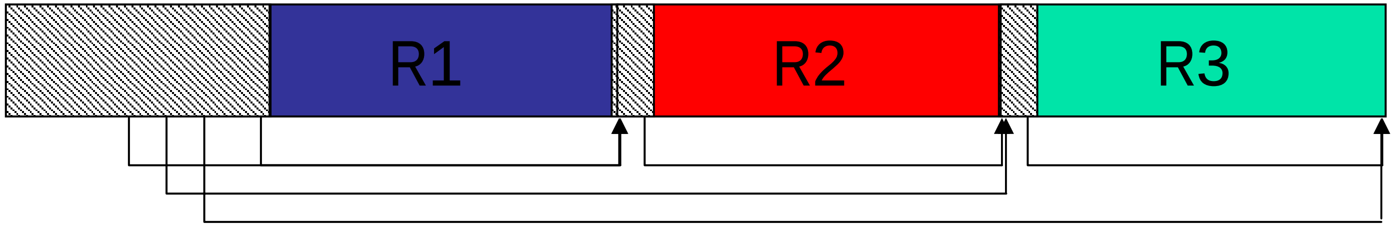
physical disk sectors:

Placing Records into Blocks – II

- ✓ Some options when storing records in blocks:
  - separating records within a block
  - spanned vs unspanned storage
  - mixed record types – clustering
  - sequencing
  - indirection
  - ...

Separating Blocks

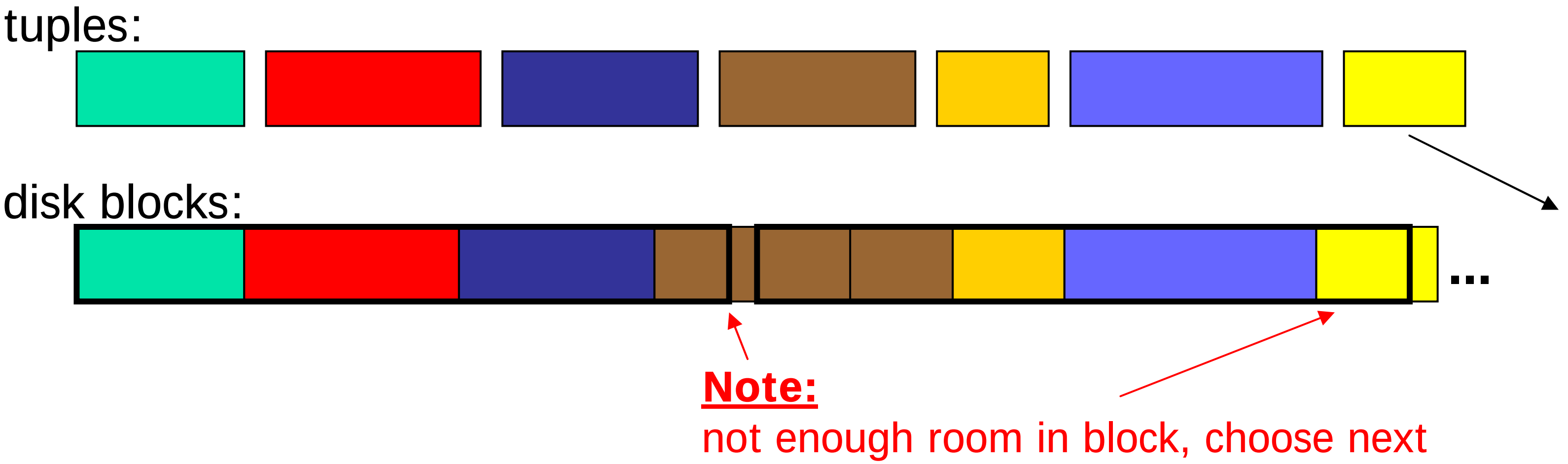
Block



- ✓ fixed size records – no need to separate
- ✓ special separation marker
- ✓ give record lengths (or offsets)
  - within each *record header*
  - in disk *block header*

Spanned vs Unspanned Records – I

✓ Unspanned records must be within one disk block

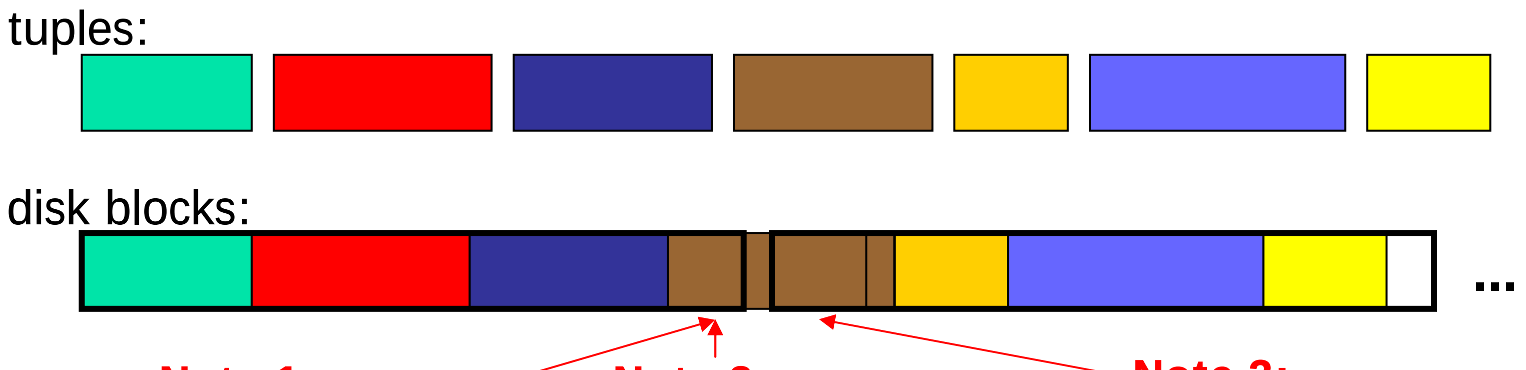


Note:
not enough room in block, choose next

- ☺ easy to find a record
- ☺ need only to access one block to find a record
- ☹ introduce fragmentation – wasted space in a disk block
- ☹ access many blocks to retrieve many records

Spanned vs Unspanned Records – II

✓ Spanned records can be split between two (or more) disk blocks



Note 1: not enough room in one block, split and store on two blocks

Note 2: need indication that it is a partitial record and a “pointer” to the rest

Note 3: need indication that it is a continuation of a record and a pointer back

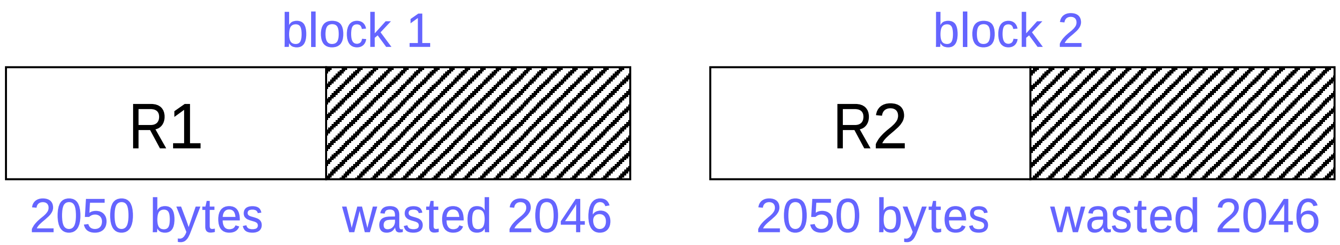
- ➢ spanned essential if record size larger than block size
- ☺ no fragmentation
- ☺ saves total disk blocks
- ☺ saves total disk accesses retrieving many records
- ☹ may need to access two or more blocks to find a record
- ☹ more complex headers, i.e., fragmentation field, fragment number, pointers to other fragments

Spanned vs Unspanned Records – III

- ✓ *Unspanned records*:
  - fixed size records:
    - records per block: blocking factor $bfr = \lfloor B/R \rfloor$ where B is block size and R is record size
    - unused space u per block: $u = B - (bfr \times R)$
    - blocks $b_{unspanned}$ needed for a file with r records: $b_{unspanned} = \lceil r/bfr \rceil$
  - variable sized records
    - each block may store a different number of records
    - bfr, u, b above gives an average if R is average record size
- ✓ *Spanned records*:
  - number of blocks needed for a file, whose size is f, is given by $b_{spanned} = f/B$
  - average record size (and file size) is somewhat larger compared to unspanned records, because we need some elements in the headers for pointers, etc.

Spanned vs Unspanned Records – IV

- ✓ Example: 10^6 records, fixed record size = 2050 bytes, block size = 4096 bytes
  - ➢ unspanned:



- bfr = $\lfloor 4096/2050 \rfloor = 1$
- total blocks = $\lceil 10^6/1 \rceil = 10^6$
- space used = 2050 x 10^6 = 1955 MB
- total space = 4096 x 10^6 = 3906 MB

} utilization ≈ 50%

- ➢ spanned:
  - each record is larger – say 3 bytes for indicating fragmentation, fragment number, and pointers
  - file size f = $10^{6\ x}$ (2050 + 3) = 1957 MB
  - total blocks = 1957 MB / 4 KB ≈ 0.501 x 10^6

Mixed Record Types (Clustering) – I

- ✓ Allow records of different types interleaved on same block – why?
- ⇨ Records that are frequently accessed together should be in the same block to minimize disk I/O
- ✓ Example:
  using clustering on customer and account records in a bank DBS

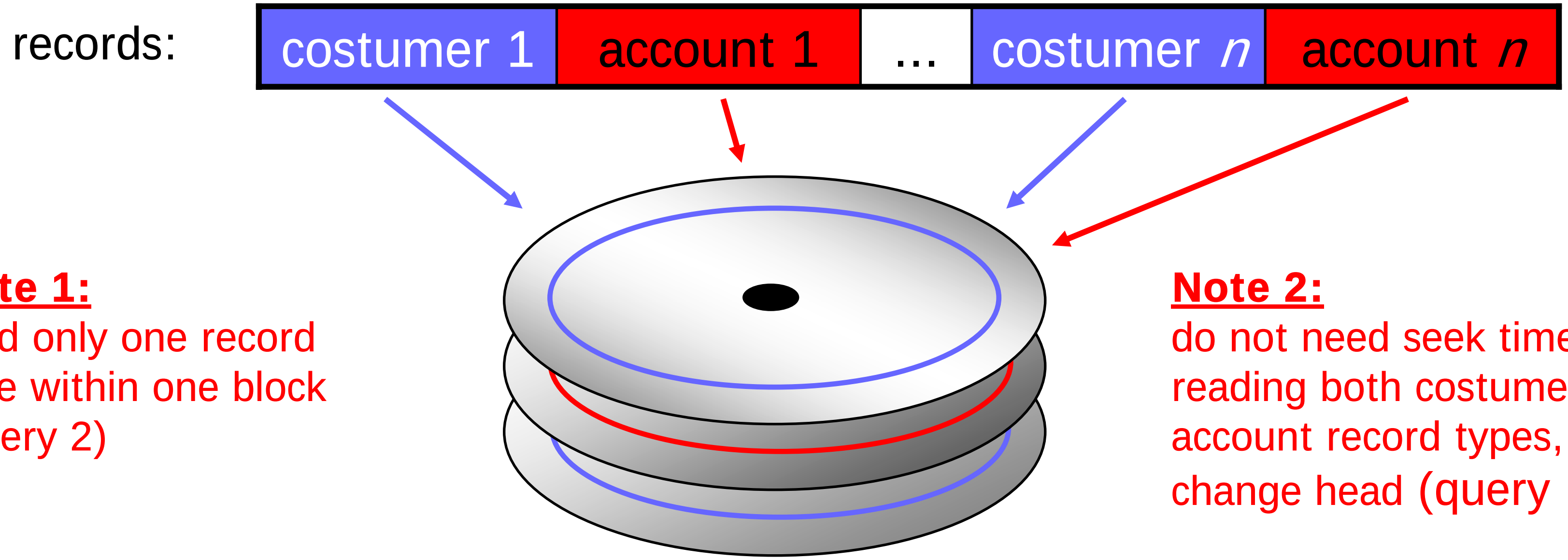
block header	customer 1	account 1	…	customer *n*	account *n*	▨

- ➢ query 1:
```
SELECT c.id, c.name, sum_credit = sum(a.credit)
FROM customer c, account a
WHERE c.id = a.owner
GROUP BY c.id
```
- ⇨ if query 1 is frequent, clustering can be *efficient*

- ➢ query 2:
```
SELECT c.name, c.address
FROM customer c
```
- ⇨ if query2 is frequent, clustering can be *counter productive*

Mixed Record Types (Clustering) – II

- ✓ As different queries will be used, one might need a way to optimize several different queries
- ✓ Compromise:
  no mixing, but keep related records in adjacent blocks (e.g., same cylinder, but different tracks)
- ✓ Example:
  using the compromise on costumer and account records in a bank DBS



Note 1:
read only one record type within one block (query 2)

Note 2:
do not need seek time when reading both costumer and account record types, only change head (query 1)

Sequencing

- ✓ Ordering records in file (and block) by some key value

- ✓ Why sequencing?

⇨ To make it possible to *efficiently read records in order*

  - ➢ merge-join
  - ➢ quick lookup using indexes
  - ➢ ...

- ✓ Keeping the records sorted makes insert and modification operations more complex

Indirection – I

- ✓ How do we represent addresses, pointers, or references?
  - ➢ data on disk
  - ➢ data in main memory

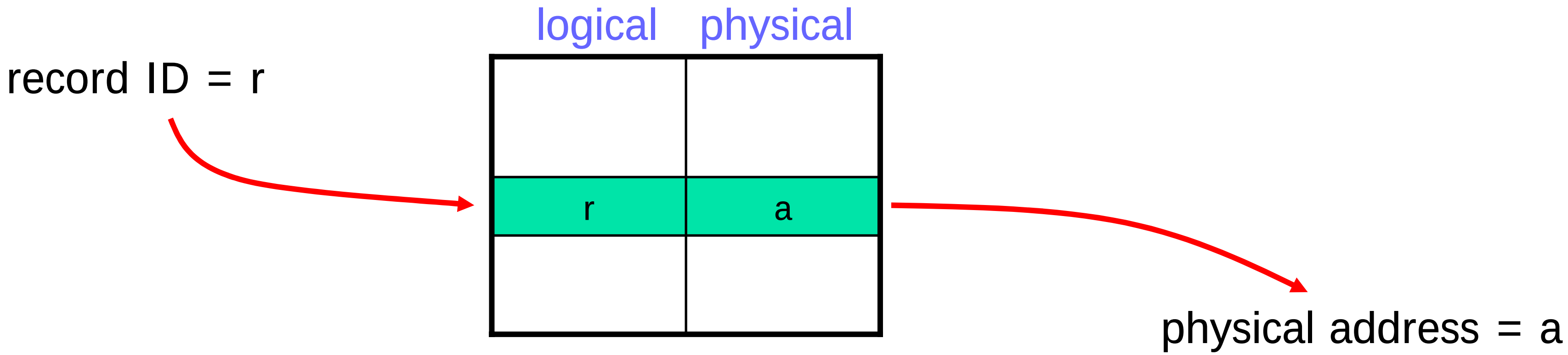
- ✓ Pure physical:
  e.g.: *record address* =

$$\left\{\begin{array}{l} \left.\begin{array}{l}\text{device ID}\\ \text{cylinder}\\ \text{track}\\ \text{block}\end{array}\right\}\text{block ID} \\ \text{offset}\end{array}\right.$$

- ☺ gives exact position of record
- ☺ no indirection – direct access
- ☹ long addresses
- ☹ must update all occurrences of pointers if record moves

Indirection – II

- ✓ Indirect:
  e.g., record ID arbitrary bit string and using a map table



record ID = r

logical	physical
r	a

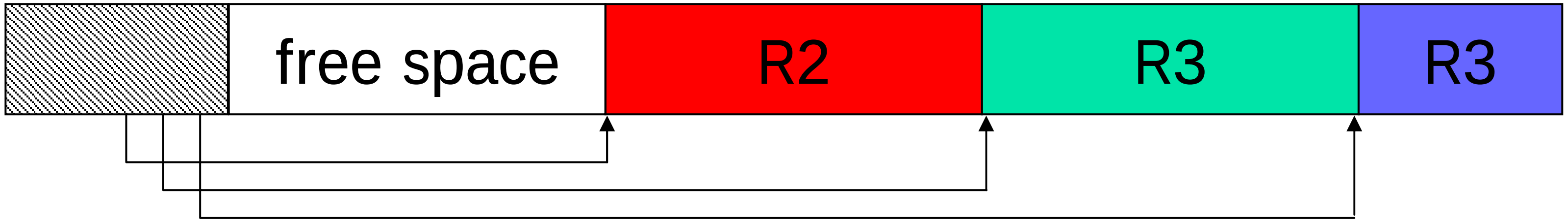
physical address = a

- ☺ update only entry in map table in case of modification
- ☹ one memory reference (or disk access) to read map table

Indirection – III

- ✓ Which one to choose is a tradeoff: flexibility vs cost

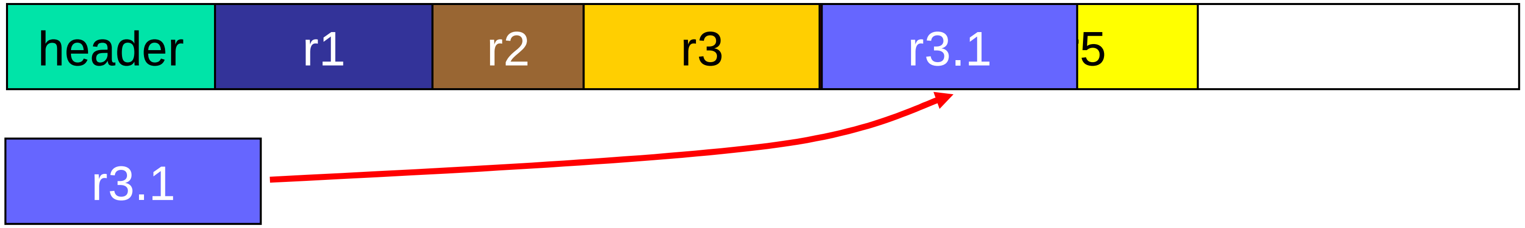
- ✓ However, many combinations possible
  - ➢ physical block number and record number (fixed size)
  - ➢ physical block number and offset table (variable size)



  - ➢ logical block (file system) and block offset
  - ➢ ...

File and Record Operations – I

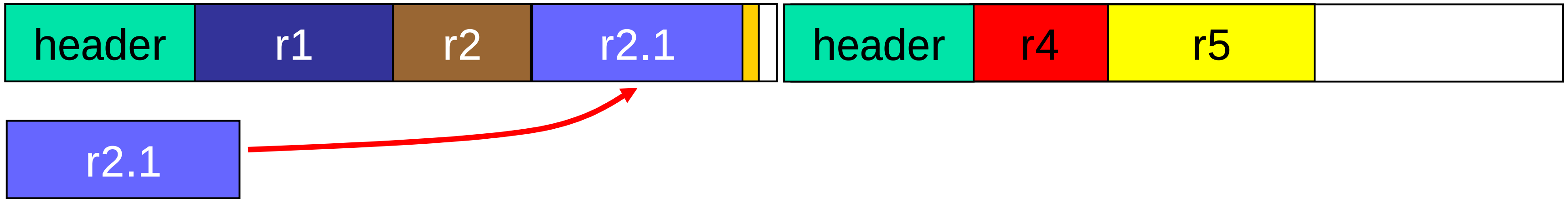
- ✓ Insertion:
  - *no order* : just insert new record in any available space, or get a new block
  - *sorted* : find appropriate block
    - space in block – slide blocks to the side and insert new, e.g.: insert record r3.1 between r3 and r4:



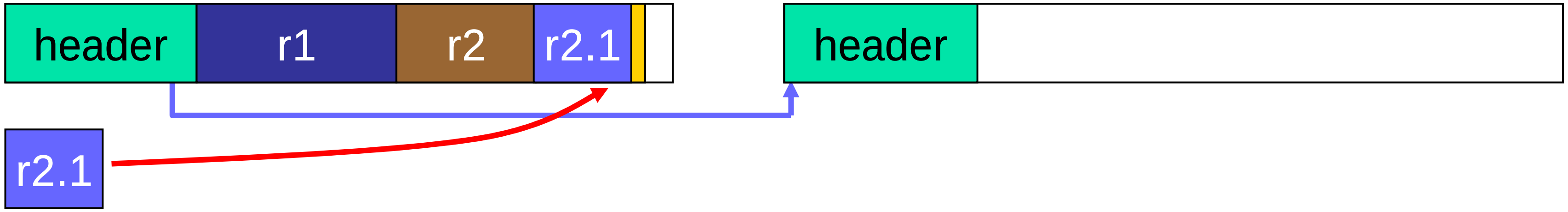
Note:
references and pointers to a record which is moved must be updated – depending on how we manage addresses and pointers

File and Record Operations – II

- no room in block – find space in a *"near-by" block*, slide last block(s) to next block, and insert new
  e.g.: insert record r2.1 between r2 and r3

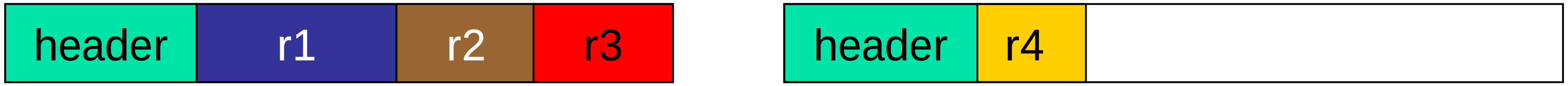


- no room in block – create *overflow block*, add pointer to block header, if necessary use block sliding as above,
  e.g.: insert record r2.1 between r2 and r3



File and Record Operations – III

✓ Deletion:



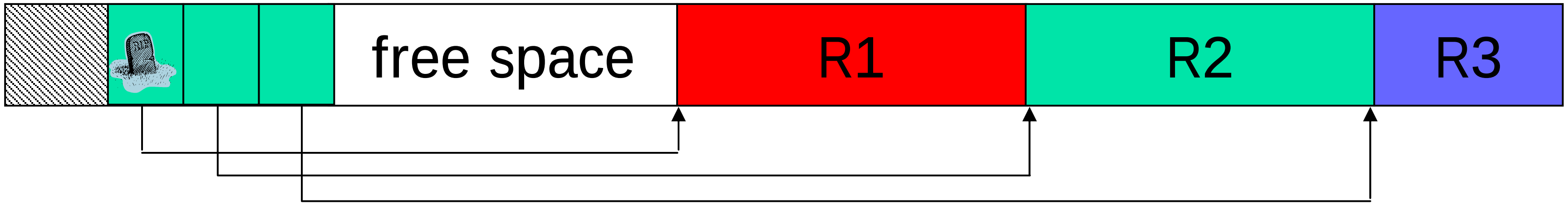
header	r1	r2	r3

header	r4	

- remove record r3
- slide other records to have one large available space in block
- may be able to do away with some blocks – save space

File and Record Operations – IV

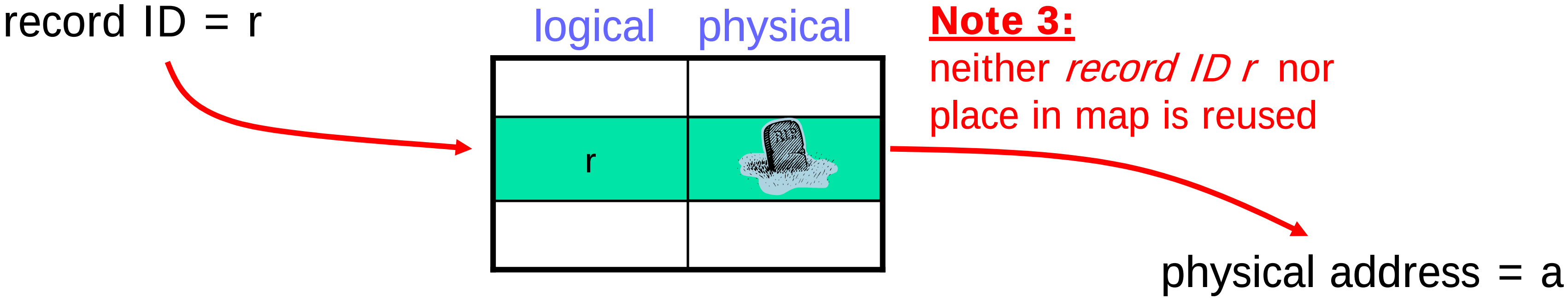
- complications – references:
  - update all references in various records
  - leave “invalid mark” (tombstones) in old location
    - physical addresses



Note 1:
space for tombstone is never re-used

Note 2:
tombstone may also go into record header

- logical addresses



Note 3:
neither *record ID r* nor place in map is reused

File and Record Operations – IV

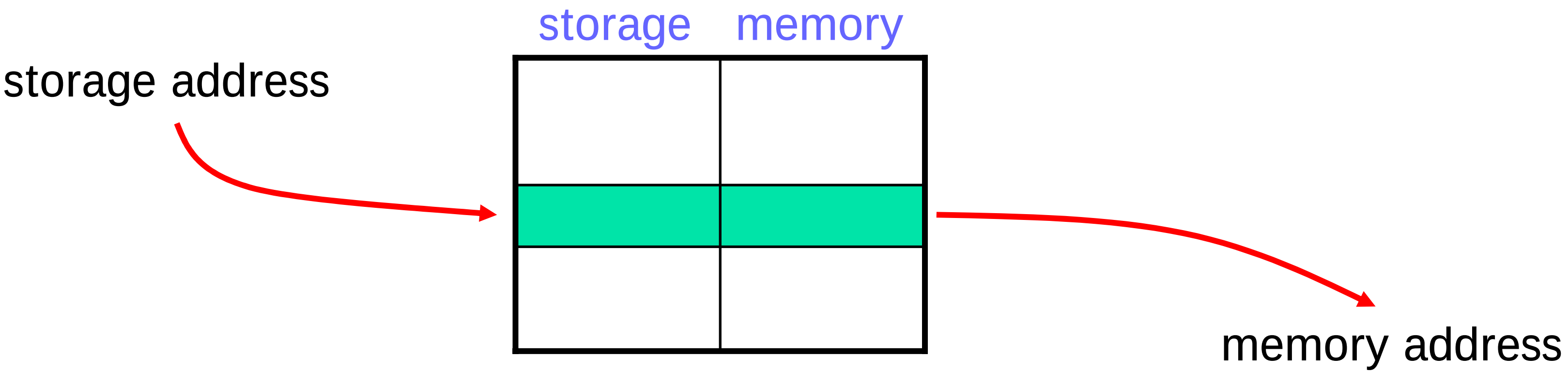
✓ Update:

- fixed length records are easy –
  just replace old value with new value
- if updated record is longer, we need additional space
  - slide records
  - overflow block
- if updated record is shorter, we can “compress” data

Pointer Management

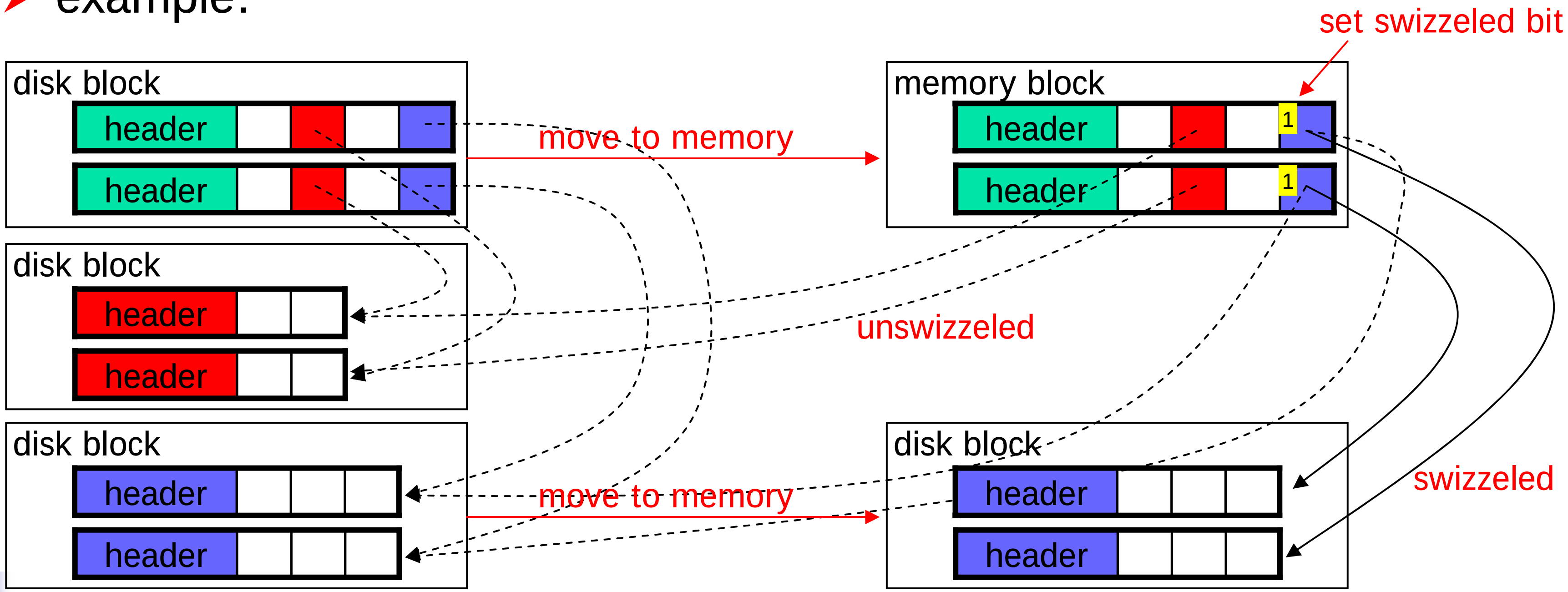
Managing Pointers – I

- ✓ Pointers are often part of a record, i.e., a field is a reference to another record
- ✓ If the data block is in memory, it is *far more efficient to use the memory address* of the record than the physical storage address
- ✓ Translation table:



Managing Pointers – II

- Pointer swizzling avoid repeated translations
  - when we move a record from secondary storage to main memory, the pointers to this record are swizzled (translated)
  - a pointer then consists of
    - a *swizzled bit*
    - the *pointer value*, i.e., either a secondary storage address or memory address as appropriate
  - example:



Managing Pointers – III

- ✓ Automatic swizzling:
  - ➢ locate all pointers *to* records in newly loaded block and swizzle the pointers to the new memory address
  - ➢ locate all pointers *in* records in newly loaded block and swizzle the pointers to records that are currently in memory
  - ☺ quick accesses to the record's references
  - ☹ much wasted work if the swizzled references are not used
- ✓ On-Demand swizzling:
  - ➢ leave all pointers unzwizzled when moving disk block into memory
  - ➢ if a record is accessed and we follow a reference, we swizzle the pointer when used
  - ☺ does not waste time swizzling pointers that will not be used
  - ☹ slower first access to referenced record due to swizzling

Managing Pointers – IV

- ✓ No swizzling: always use translation table
  - ☺ does not waste time swizzling pointers that will not be used
  - ☺ less complex design – no swizzling decisions needed
  - ☹ slower access to referenced records due to lookup in translation table each time

- ✓ Programmer-controlled swizzling:
  - ➢ at implementation time, the programmer knows some records that will be frequently used – swizzle these
  - ➢ use *no* or *on-Demand* swizzling on rest
  - ☺ speeds up accesses to frequently used records

Managing Pointers – V

- ✓ Pointers must be unswizzled when a block is returned to disk

- ✓ One might pin certain memory blocks, i.e., it cannot be moved back to secondary storage
  - ➢ frequently used pages
  - ➢ swizzled pointers to records contained in the block

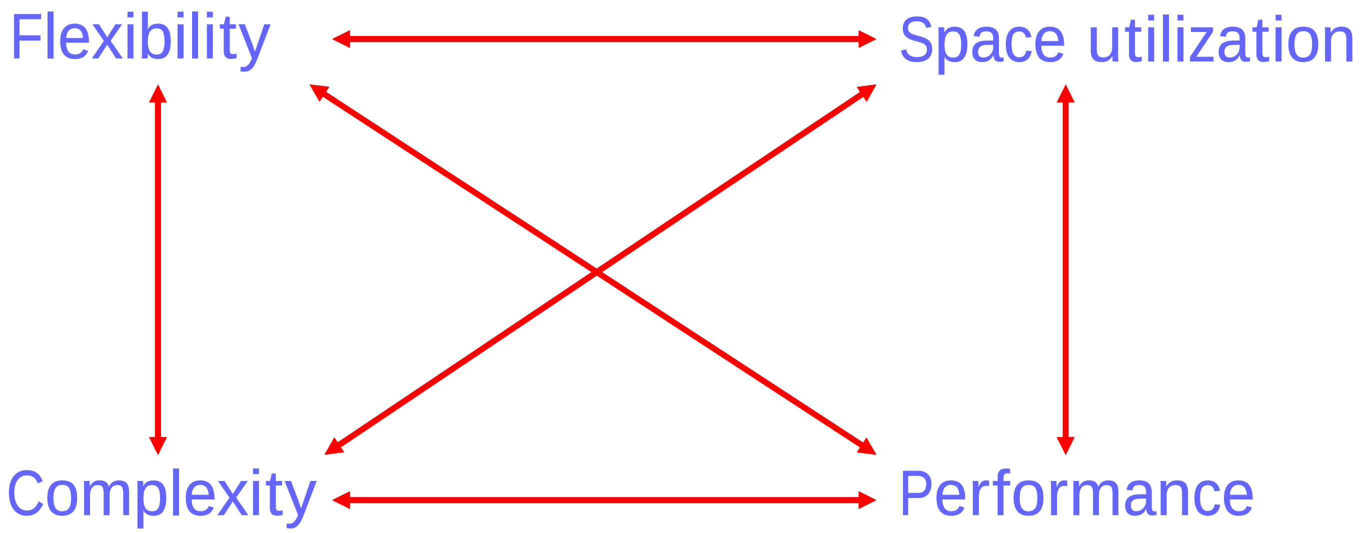
Comparison

Many Options – I

- ✓ There are numerous ways to organize data on disk:
  - ➢ fixed-length vs variable-length fields
  - ➢ fixed-length vs variable-length records
  - ➢ fixed-format vs variable-format records
  - ➢ byte-alignment
  - ➢ which "meta-data" to put in record header, block header, ...
  - ➢ separating records within a block
  - ➢ spanned vs unspanned storage
  - ➢ mixed record types – clustering
  - ➢ sequencing
  - ➢ indirection
  - ➢ different block allocation schemes
  - ➢ ...
- ✓ Which one is best for me?

Many Options – II

- ✓ To choose the “best”, there are several issues:



- ✓ Thus, the “best” design *depends on various parameters* like common operations, access patterns, amount of data, data types, …

Summary

- ✓ Basic data representation in fields:
  fixed vs variable length
- ✓ Records:
  fixed vs variable length and format
- ✓ Data layout on disk:
  block allocation, record placement, sequencing, clustering, …
- ✓ Pointer management moving records: swizzling
- ✓ Comparison:
  the “best” design dependent on various factors