CS525: Advanced Database Organization

Notes 3: File and System Structure

Yousef M. Elmehdwi

Department of Computer Science

Illinois Institute of Technology

yelmehdwi@iit.edu

August 28th, 2018

Slides: adapted from a courses taught by Hector Garcia-Molina, Stanford, Elke A. Rundensteiner, Worcester Polytechnic Institute, Shun Yan Cheung, Emory University, &Marc H. Scholl,University of Konstanz

Announcement: Assignment 1 - Storage Manager

- Course website: http://cs.iit.edu/~cs525/yousef
- Implement a storage manager that allows read/writing of blocks to/from a file on disk
- Assignment 1 - Storage Manager
- Due on 09/18/2018 by 11:59pm

Data Storage: Overview

- How does a DBMS store and manage large amounts of data? (last lecture)
- What representations and data structures best support efficient manipulations of this data? (today)

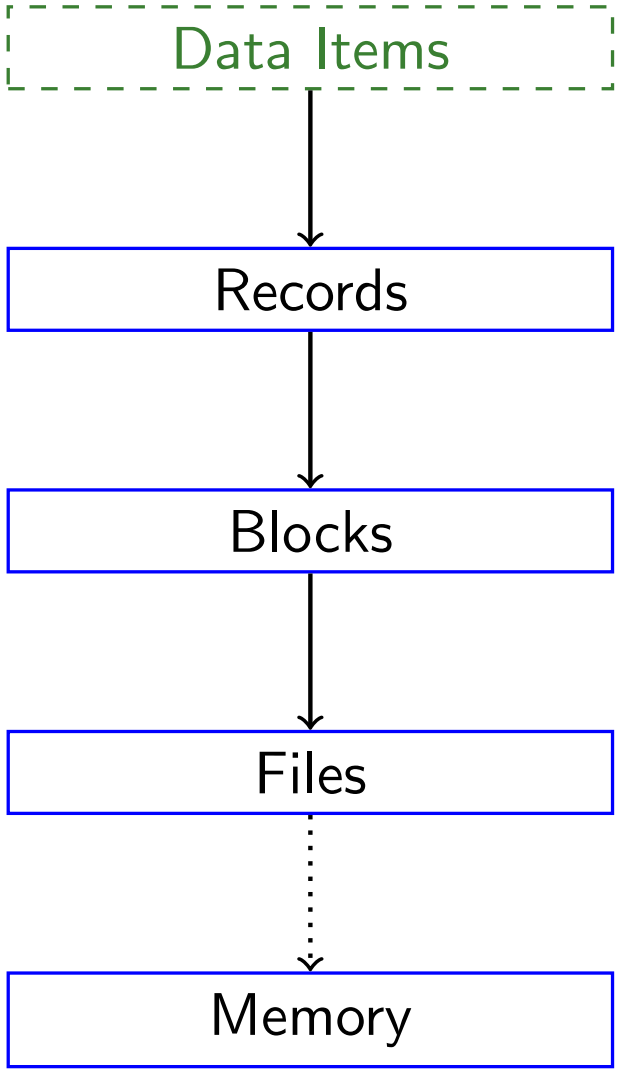
Today

- How to lay out data on disk

Principles of Data Layout/Physical data organization

- Attributes of relational tuples represented by sequences of bytes called fields
- Fields grouped together into records (Record: sequence of fields)
  - representation of tuples
- Record schema (or type): sequence of field names and their corresponding data types
- Records stored in blocks
- Blocks contain typically more than one record. If the records are too big, they span more than one block
- File: collection of blocks that forms a relation
  - i.e., File: collection of records with the same schema (typically), usually spanning a number of blocks
- Blocking speeds up data access by eliminating some seeks and rotational delays
- Block access is a cost unit for file operation

Overview

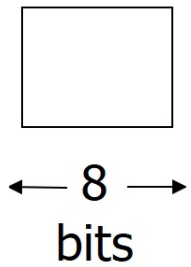


How the principal SQL datatypes are represented as fields of a record?

What are the data items we want to store?

- a salary
- a name
- a date
- a picture

$\Rightarrow$ What we have available: Bytes



To represent

- Integer (short): 2 bytes
  - e.g., 35 is

00000000	00100011

- Integer (long): 4 bytes
- Real, floating point
  - n bits for mantissa, m for exponent

To represent

- Characters
  - various coding schemes suggested (ASCII, UTF8, ...)
- Example: (8 bits ASCII)

A:	1000001
a:	1100001
5:	0110101
LF:	0001010

To represent

- Boolean

e.g., TRUE	1111 1111
FALSE	0000 0000

- Application specific
  - represented by integer codes (e.g., by two/eight bits)
  - e.g., RED $\rightarrow$ 1 BLUE $\rightarrow$ 2 GREEN $\rightarrow$ 3 YELLOW $\rightarrow$ 4 ...
- Can we use less than 1 byte/code? Yes, but only if desperate

To represent

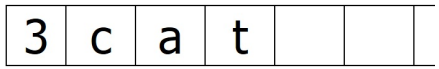
- Dates
  e.g.,
  - Integer, # days since Jan 1, 1900
  - 8 characters, `YYYYMMDD`
  - 7 characters, `YYYYDDD`
    - Where DDD are digits between 001 and 366 denoting a day of that year
  - 10 chars: `YYYY-MM-DD`
- Time
  e.g.,
  - Integer, seconds since midnight
  - Characters, `HHMMSSFF`

To represent: String of characters

- **Variable-Length Character Strings**
  e.g, `VARCHAR(n)`: $n + 1$ bytes max
  - Two common representation:
    - Null terminated
      e.g.,

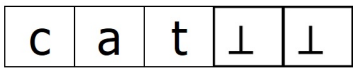
    - Length given (length + content)
      e.g.,

To represent: String of characters

- **Fixed-Length Character Strings**
  e.g., `CHAR(n)`
  - *n* bytes
  - If the value is shorter, fill the array with a pad character, whose 8-bit code is not one of the legal characters for SQL strings

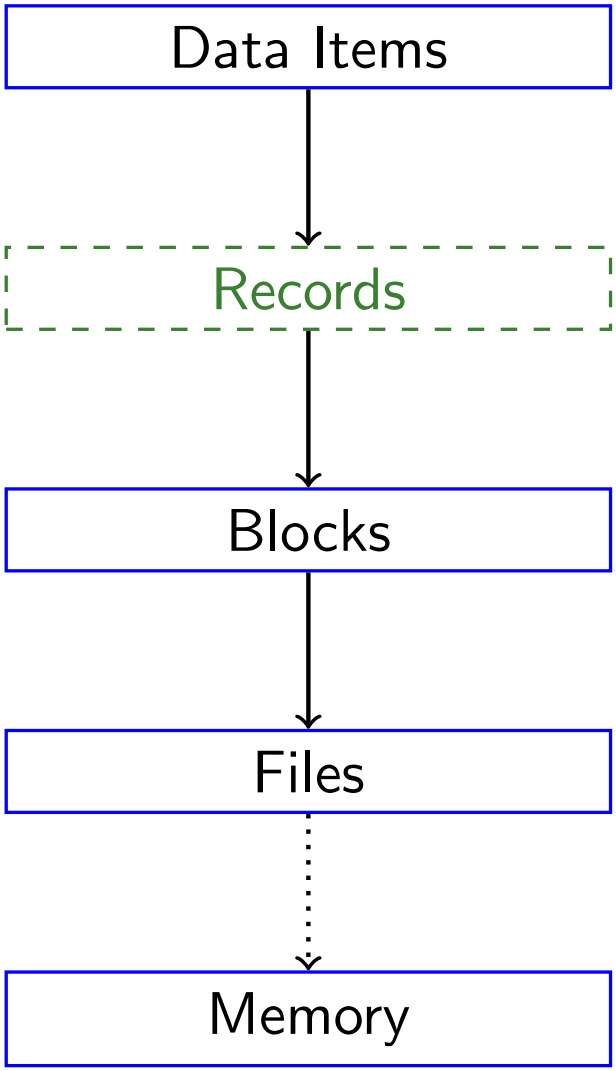
  e.g., `CHAR(5)`



Key Points

- Fixed length items
- Variable length items
  - usually length given at beginning
- Type of an item: Tells us how to interpret (plus size if fixed)

Overview



How fields are grouped together into records?

Record

- How fields are grouped together into records
- Collection of related data items (called FIELDS)
- Typically used to store one tuple
- E.g.: `Employee` record consisting of:
  - name field, `CHAR(20)`,
  - salary field, `Number`,
  - date-of-hire field, `Date`,
  - ...

Types of records

- Main choices:
  - FIXED vs VARIABLE FORMAT
  - FIXED vs VARIABLE LENGTH

Fixed format

- A `schema` contains information such as:
  - Number of fields (attributes)
  - type of each field (length)
  - order of attributes in record
  - meaning of each field (domain)
- The `schema` is consulted when it is necessary to access components of the record
- Not associated with each record.

fixed format and length

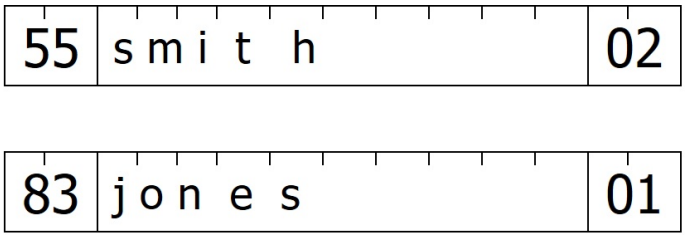
- All records have the same length and same number of fields (all the fields of the record have a fixed length)
- The address of any field can be computed from info in the system `schema`
- To form the record, we can simply concatenate the fields

Example: fixed format and length

Example: Employee record

1. E#, 2 byte integer
2. E.name, 10 char.
3. Dept, 2 byte code

Schema

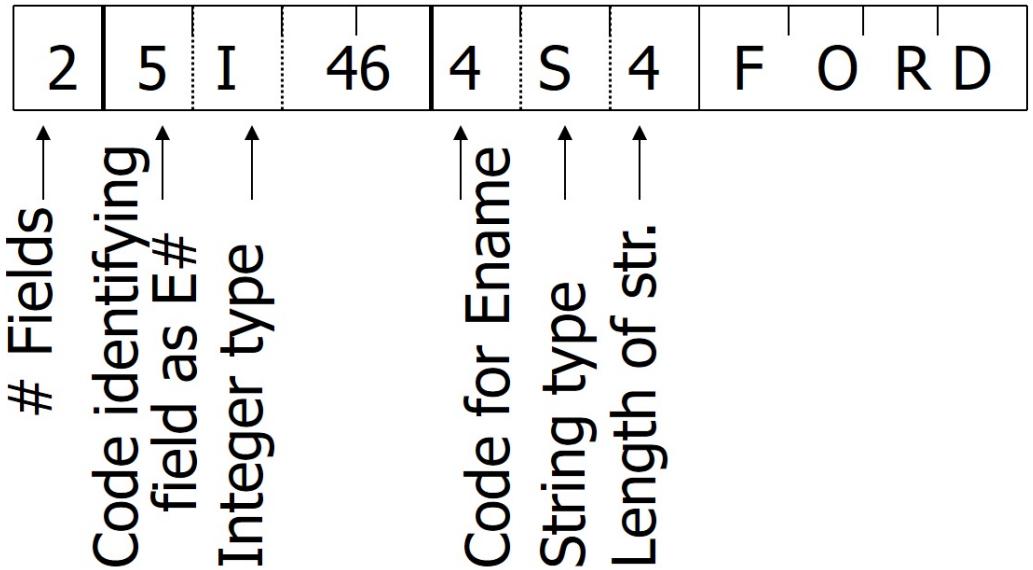


Records

Variable format

- Not all fields are included in the record, and/or possibly in different orders.
- Record itself contains format "Self Describing"
  - every record contains (# fields, type of each field, order in record, ...) information in its header

Example: variable format and length



Why Variable Format?/Variable format useful for

- “sparse” records, eg. medical records
- repeating fields
- information integration

Example: variable format record with repeating fields

- e.g., `Employee` has one or more children

3	E_name: Fred	Child: Sally	Child: Tom

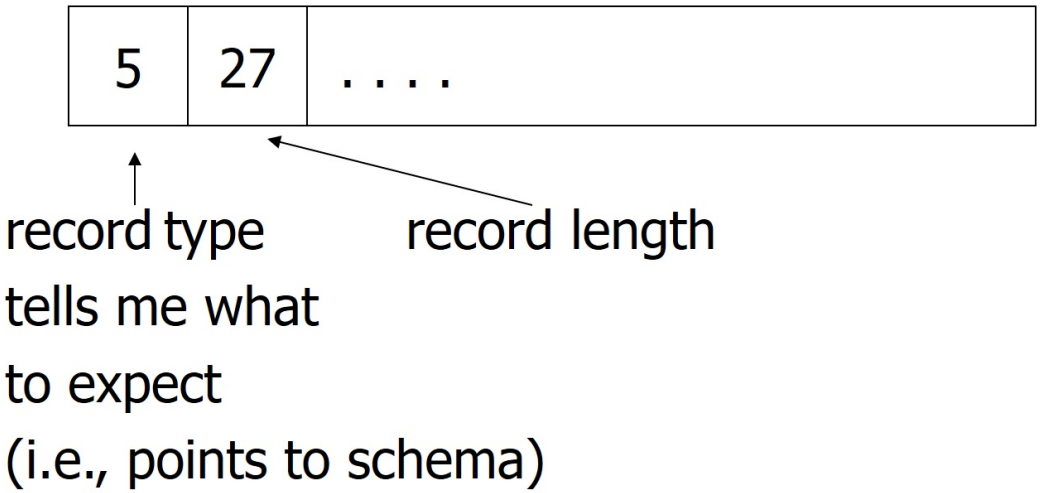
Note

- Repeating fields does not imply variable format, nor variable size
- key is to allocate maximum number of repeating fields (If not used, set to null)
- e.g., a person and her hobbies.

Mary	Sailing	Chess	--

Many variants between fixed - variable format

- Example 1: Include *record type* in record

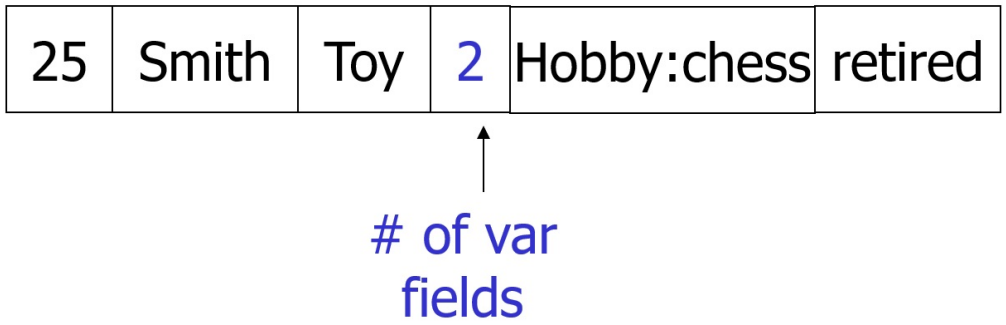


Record header

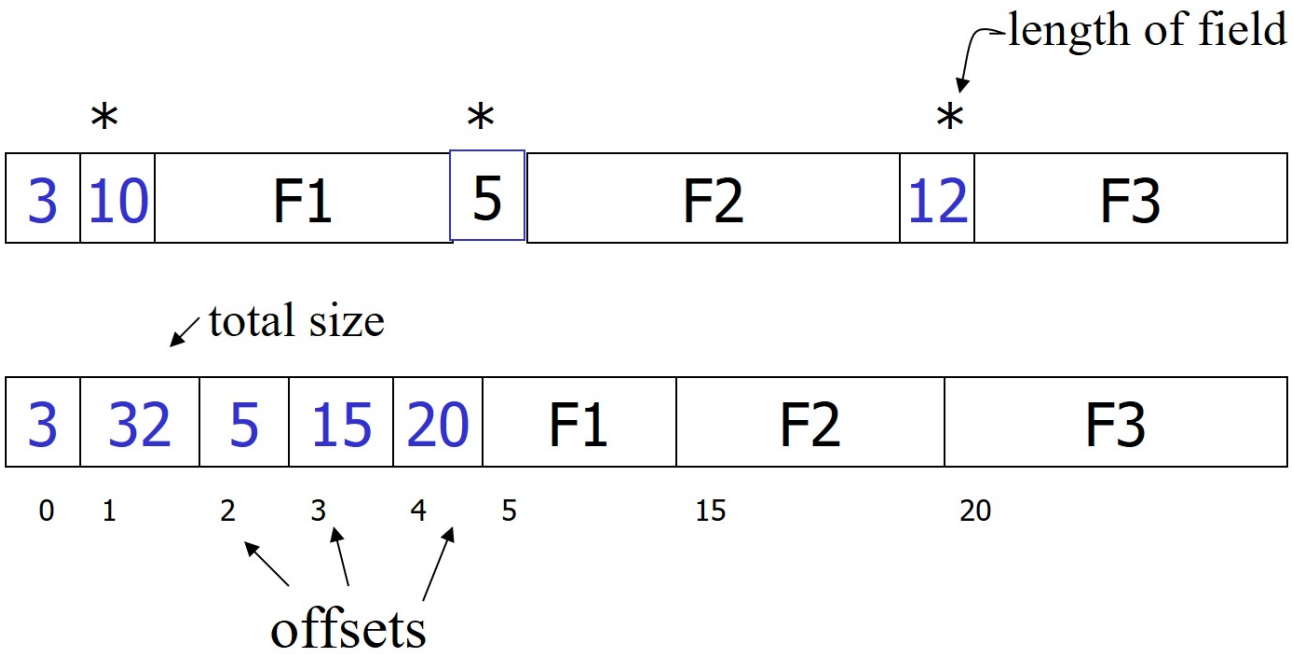
- Reserved part at the beginning of a record
  - Data at beginning that describes record
- Typically contains:
  - pointer to schema (record type)
  - length of record (for skipping)
  - time stamp (create time, modification time, last access)
  - other stuff

Many variants between fixed - variable format

- Example 2: Hybrid format: one part is fixed, other is variable
- E.g.: All employees have `E#`, `name`, `dept`; and other fields vary.



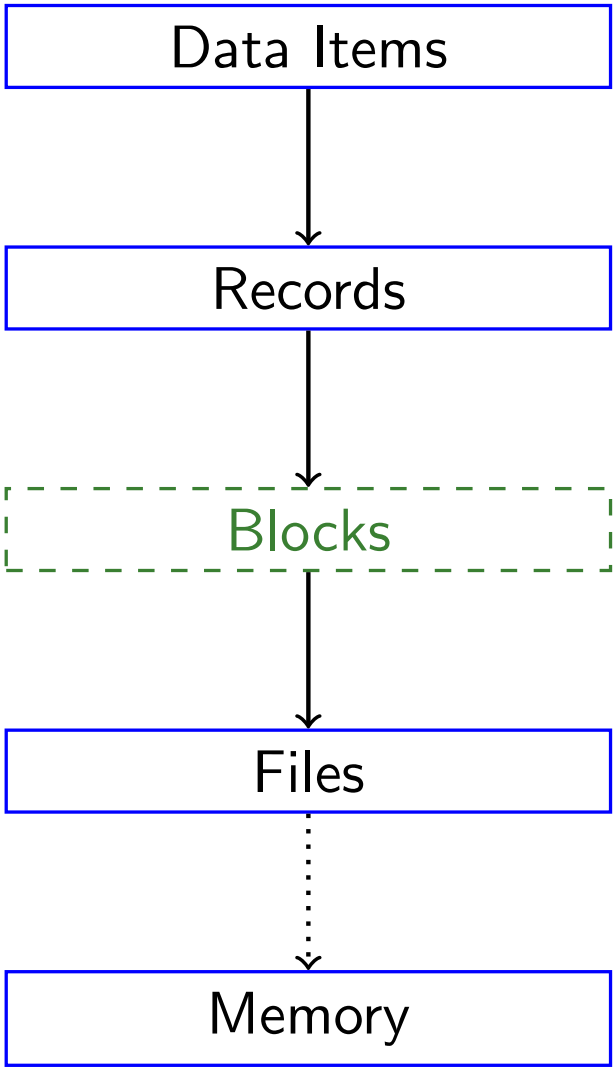
Also, many variations in internal organization of record



Other interesting issues

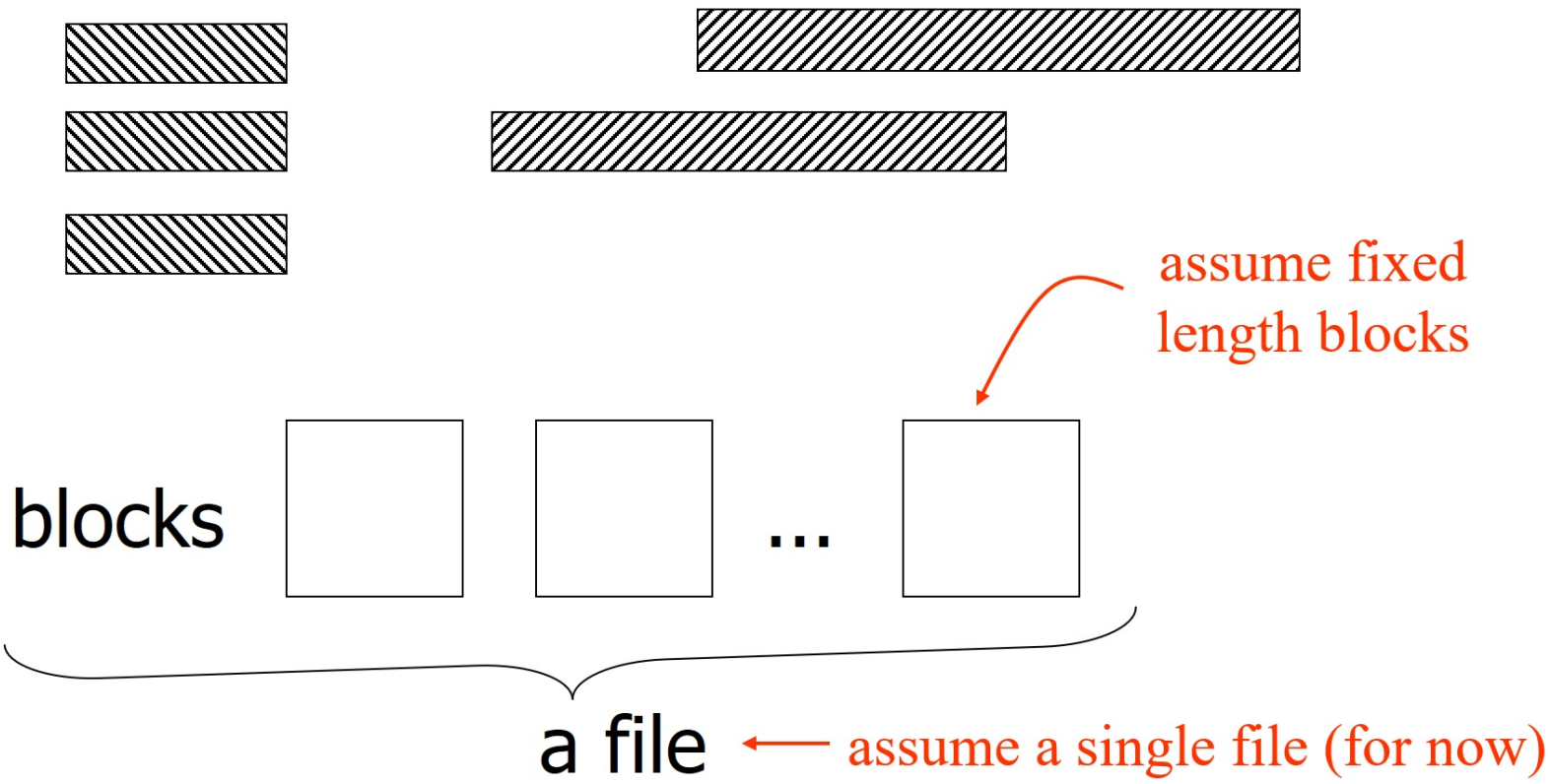
- Compression
  - within record - e.g. code selection
  - collection of records - e.g. find common patterns
- Encryption

Next



Next: placing records into blocks

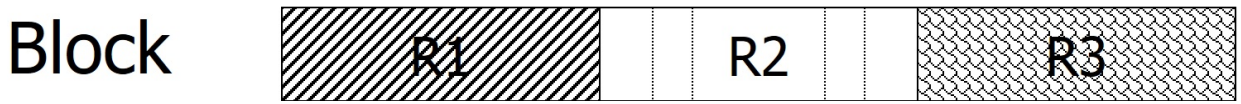
- Files consist of blocks containing records
- How to place records into blocks?



Options for storing records in blocks

1. separating records
2. spanned vs. unspanned
3. mixed record types - clustering
4. split records
5. sequencing
6. indirection

(1) Separating records



(a) no need to separate - fixed size records

(b) special marker

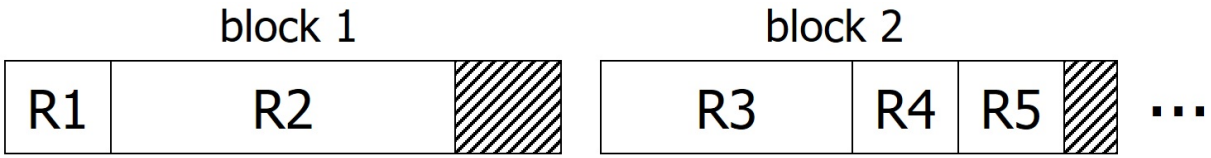
(c) give record lengths (or offsets)

  i) within each record

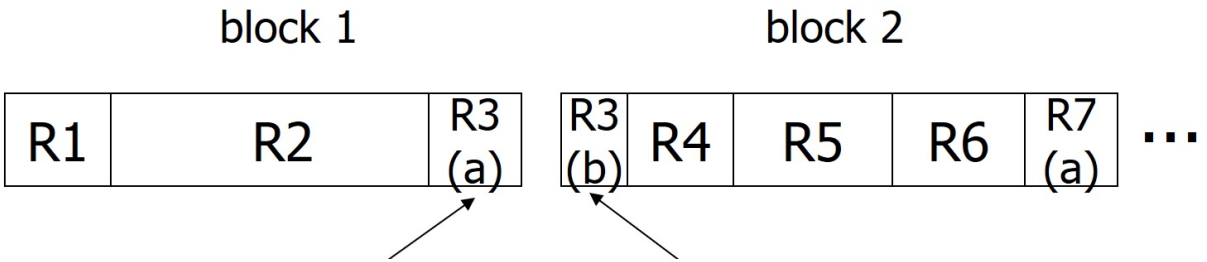
  ii) in block header

(2) Spanned vs. Unspanned

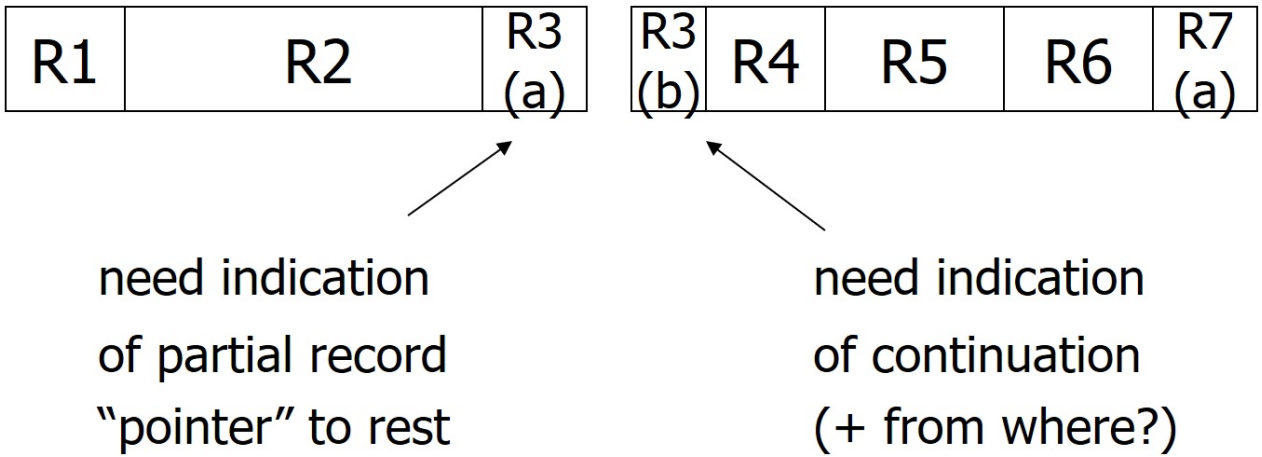
- **Unspanned**: Every records are stored within one block (i.e.: a record does not span over multiple blocks)



- **Spanned**: Some records are stored using multiple blocks (i.e., a record can span over multiple blocks)



With spanned records: How to store spanned record

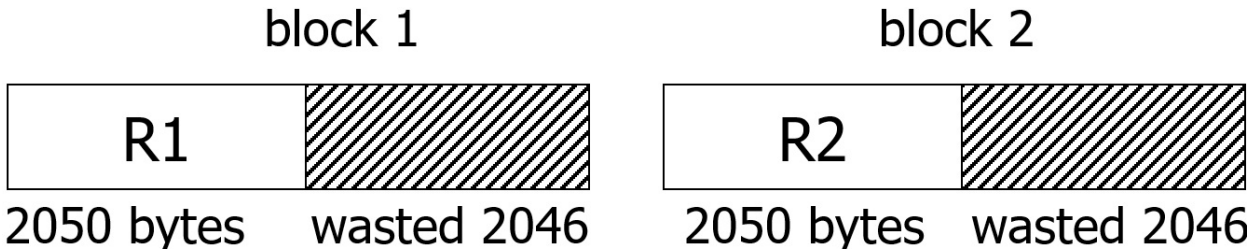


Spanned vs. unspanned

- Unspanned is much simpler, but may waste space
- Spanned essential if `record size` $>$ `block size`

Example

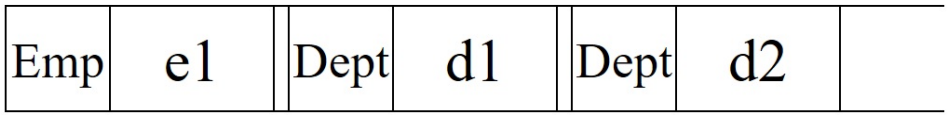
- 10^6 records
- each of size 2,050 bytes (fixed)
- block size = 4096 bytes



- if records are just slightly larger than half a block, the wastage can approach 50%
- Utilization = 50% $\Rightarrow$ $\frac{1}{2}$ of space is wasted

(3) Mixed record types

- **Mixed** - records of different types (e.g. `Employee`, `Dept`) allowed in same block
- e.g., a block



Why do we want to mix?

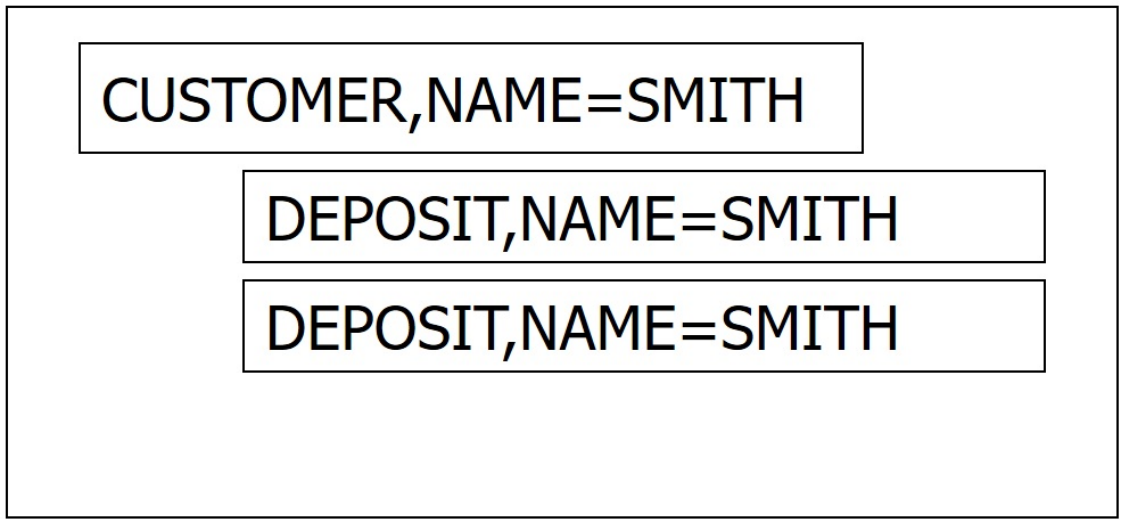
- **Answer:** Clustering
  - Records that are frequently accessed together should be in the same block
- **Problems**
  - Creates variable length records in block
  - Must avoid duplicates (how to cluster?)
  - Insert/delete are harder

Example Clustering

- Q_1)

```
SELECT A#, C_NAME, C_CITY,
FROM DEPOSIT, CUSTOMER
WHERE DEPOSIT.C_NAME = CUSTOMER.C.NAME;
```

a block



Example Clustering

file organization

department

dept_name	*building*	*budget*
Comp. Sci.	Taylor	100000
Physics	Watson	70000

instructor

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

multitable clustering of *department* and *instructor*

Comp. Sci.	Taylor	100000
45564	Katz	75000
10101	Srinivasan	65000
83821	Brandt	92000
Physics	Watson	70000
33456	Gold	87000

Example Clustering

- If Q_1 frequent, clustering good
- But if Q_2 frequent

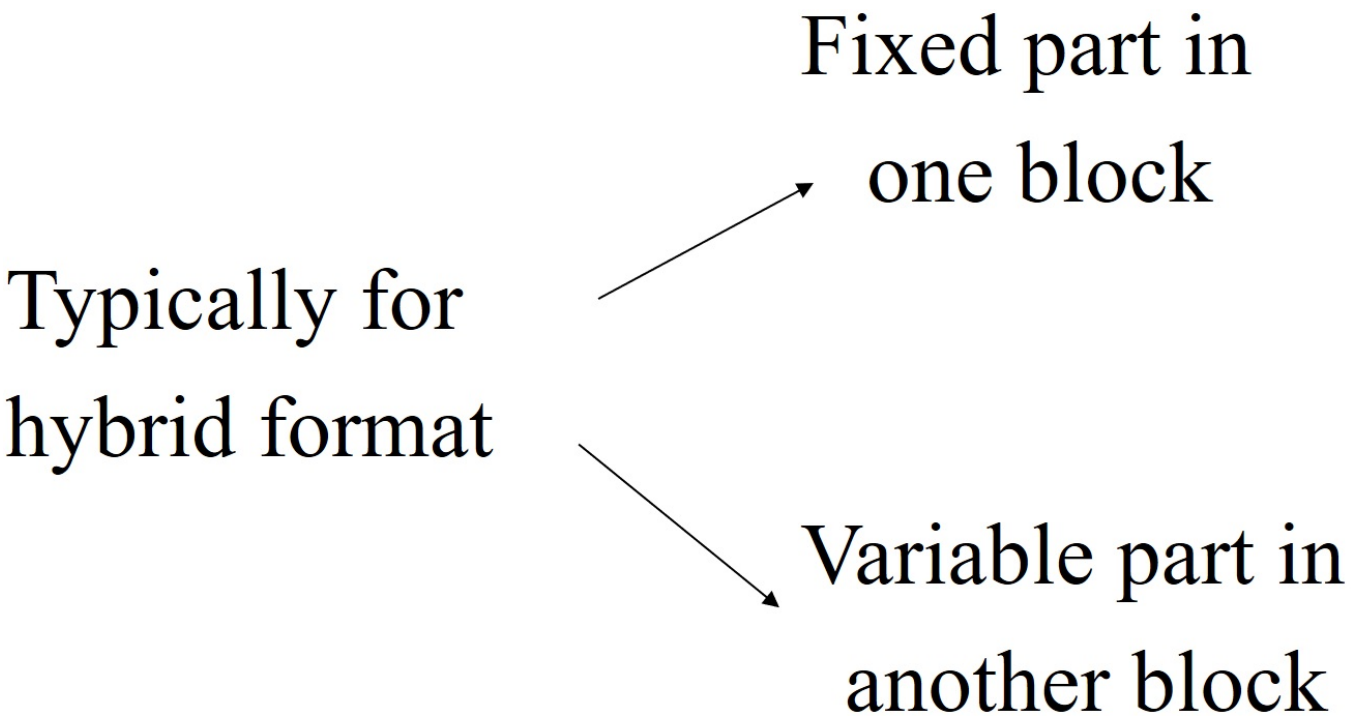
```
SELECT *
FROM CUSTOMER;
```

- Clustering is counter productive

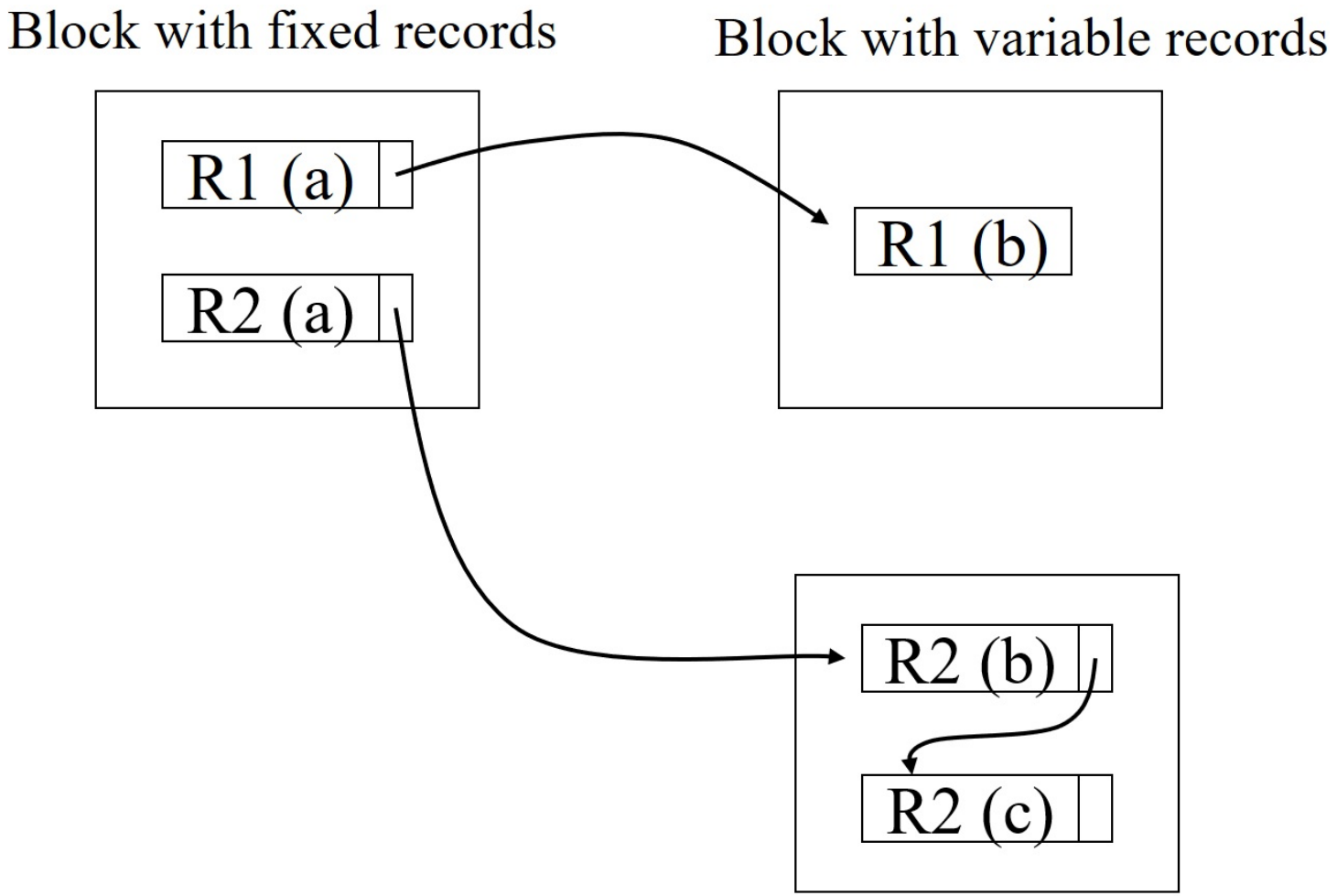
Compromise

- No mixing, but keep related records in same cylinder ...

(4) Split records



Example


Block with fixed records
Block with variable records
R1 (a)
R2 (a)
R1 (b)
R2 (b)
R2 (c)

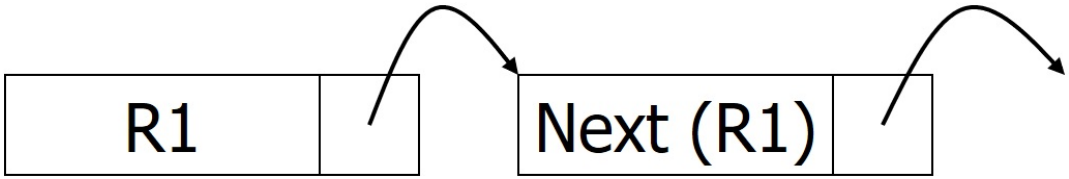
(5) Sequencing

- Ordering records in file (and block) by some key value
  - Sequential file ($\rightarrow$ sequenced file)
- Why sequencing?
  - Typically to make it possible to efficiently read records in order
    - e.g., to do a merge-join - discussed later
    - Can be used for binary search

Sequencing Options

(a) Next record physically contiguous

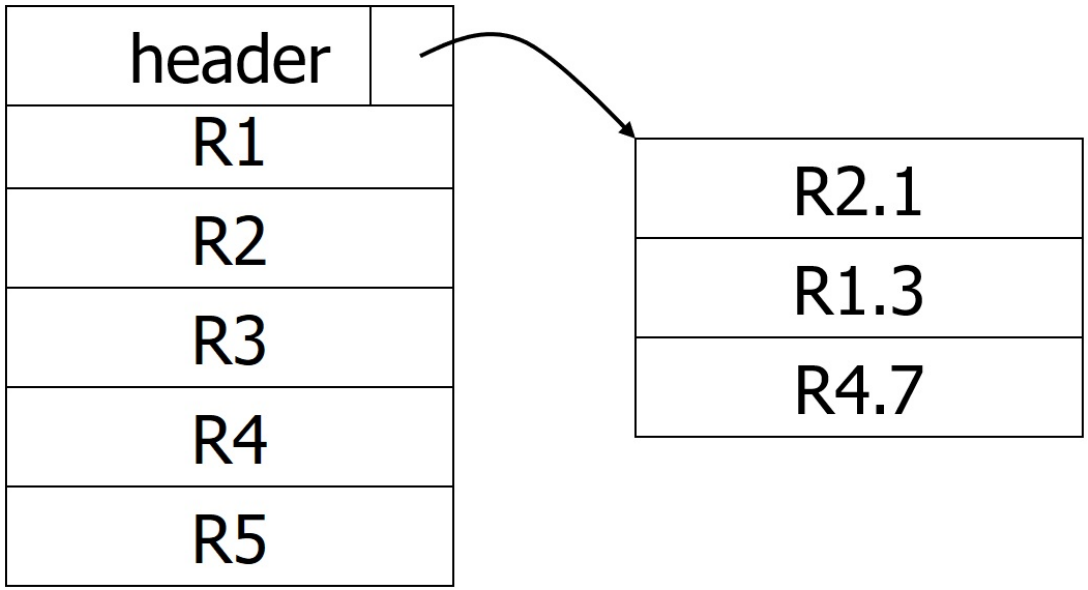
(b) Records are linked



What about `Insert/Delete`?

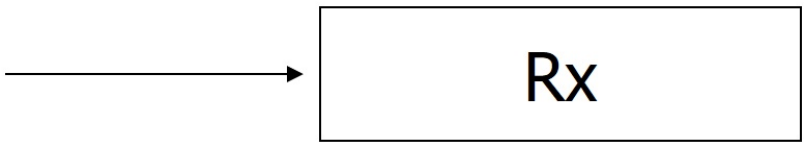
Sequencing Options

(c) Overflow area
Records in sequence



(6) Indirection Addressing

- How does one refer to records? Identifying a block/record on disk



- **Problem**: Records can be on disk or in (virtual) memory.

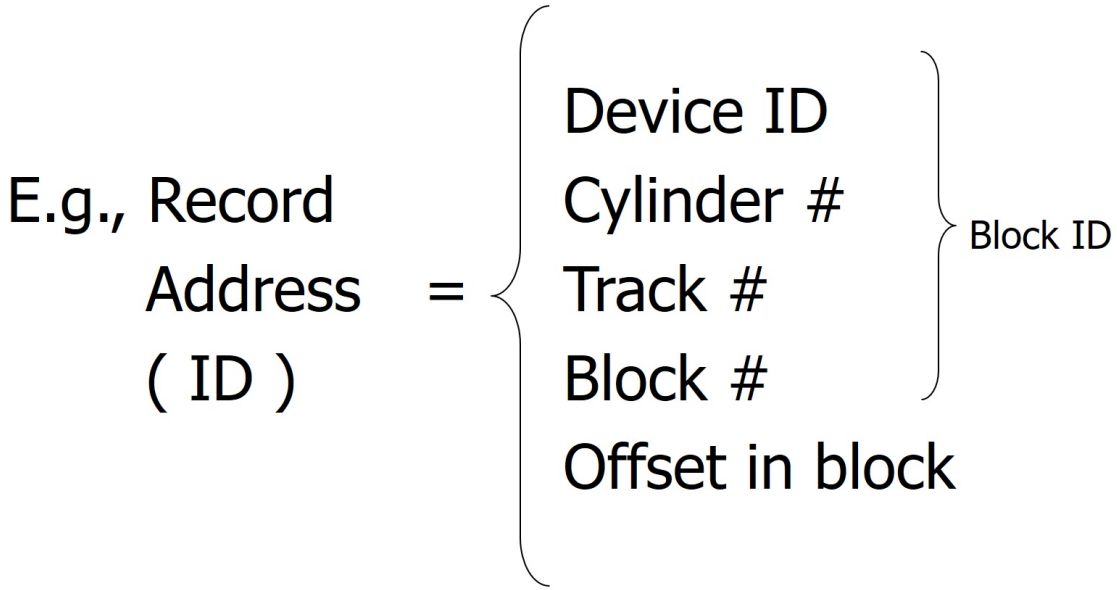
Types of addresses to identify blocks/records

There are 2 types of address to identify a block/record in use:

1. Database Address:
   - Used to identifies data (block or record) stored on disk
   - There are 2 kinds of database addresses:
     - Physical address
     - Logical address
2. Virtual memory address(paging): used to identify data (block or record) stored in (virtual) memory

Purely Physical Addressing

- direct addressing format for identify block/record on a disk



☺ gives exact position of record

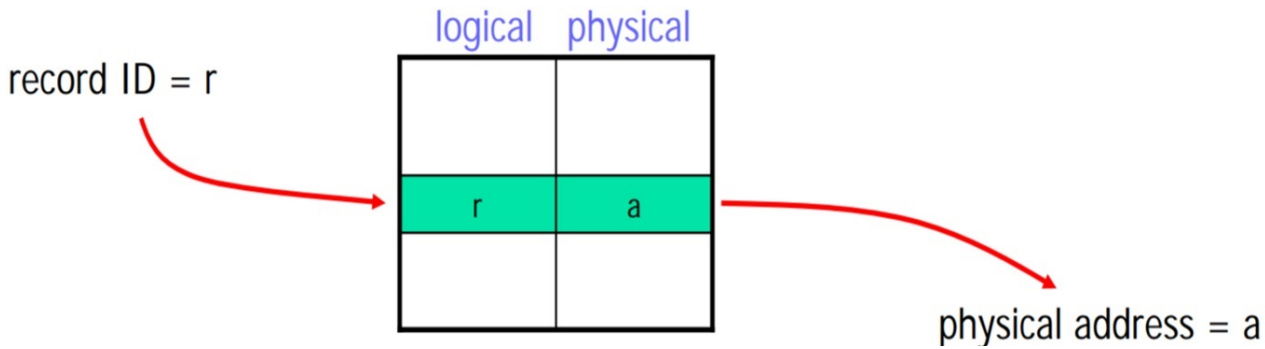
☺ no indirection - direct access

☹ long addresses

☹ must update all occurrences of pointers if record moves

Logical Address

- an indirect addressing format for identify block/record on a disk
- Logical block address:
  - Each block/address is assigned a unique logical address
  - Logical address = an arbitrary string of fixed length bits
    - (Can be generated automatically using some sequence generator or keep adding 1 to a counter)
- DBMS uses a `map table` to translate:



- To speed up access, the `map table` is organized as a `hash table`

☺ update only entry in map table in case of modification

Trade-off

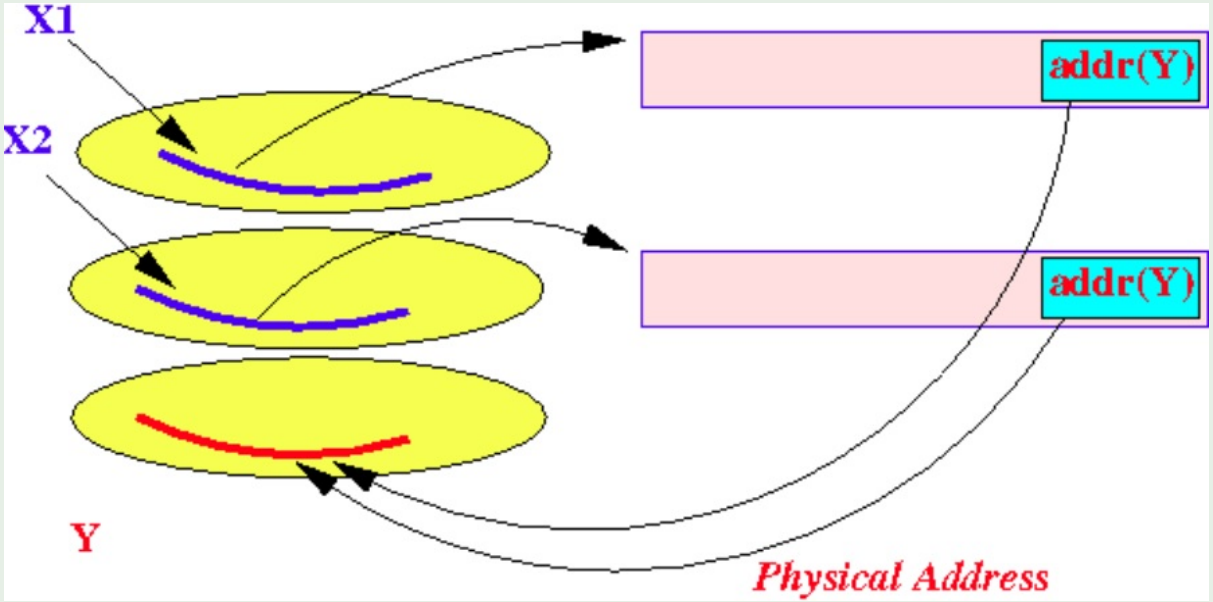
- Flexibility to move records (for deletions, insertions) $\leftrightarrow$ Cost of indirection (lookup)
- What to do: Options in between?

Motivation for using logical (database) address

- Problem with referencing another record using a physical address

Example

- 2 records reference the record Y

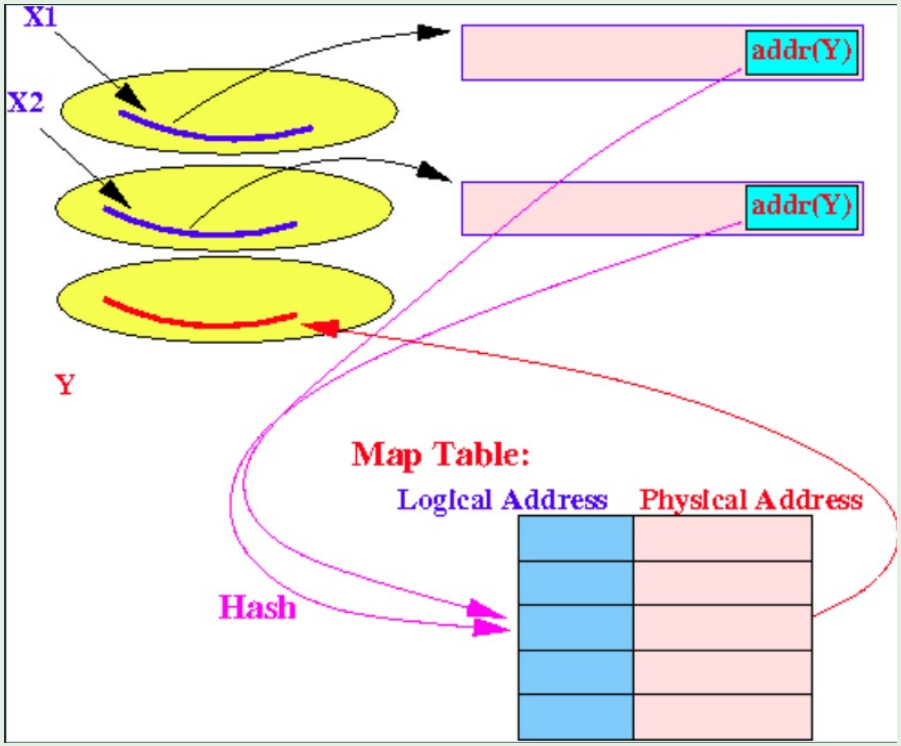


- Problem: If record Y is moved to a different part of the disk.
  - We must update many addresses

Motivation for using logical (database) address (Cont.)

Example (Solution)

- Using a logical address as reference



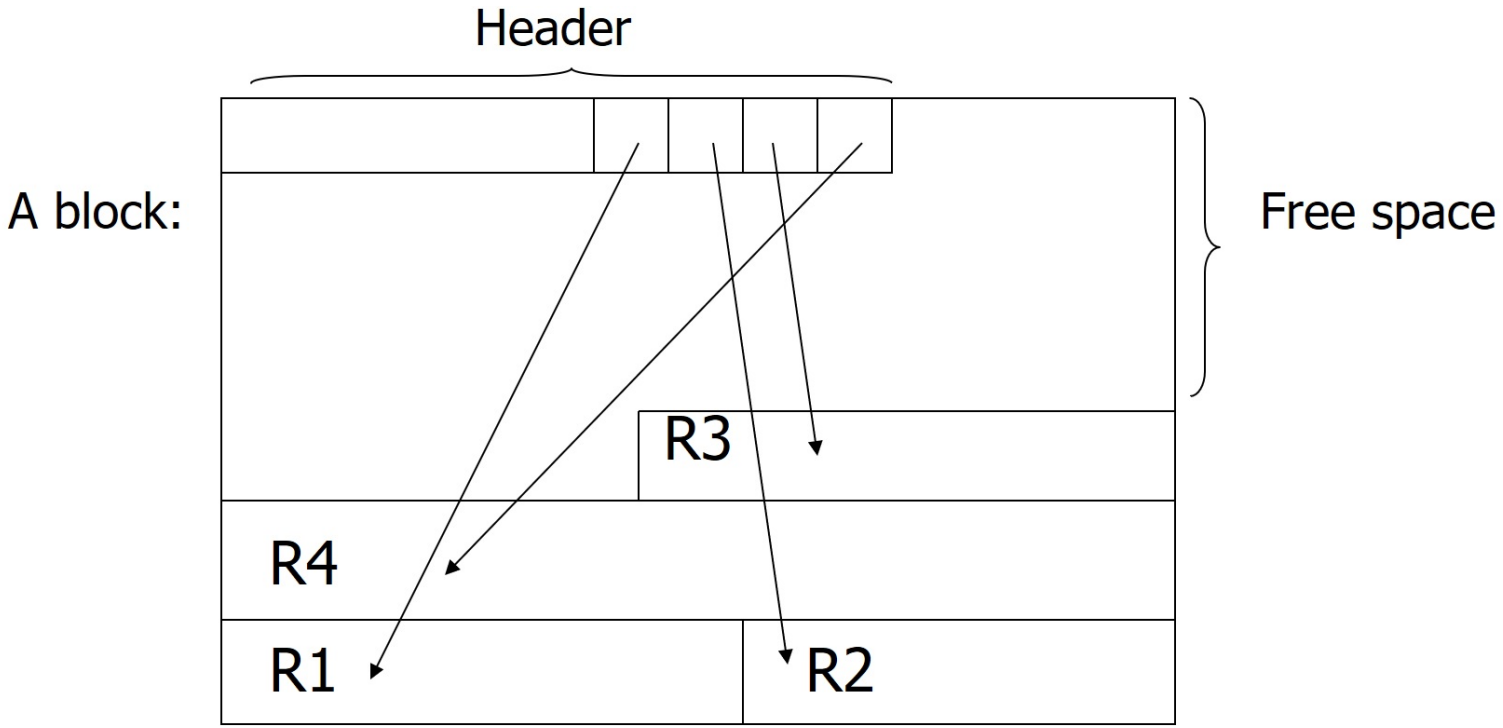
- If we move the record Y, we only need to
  - Update the physical address in the map table

Block header

- Data at beginning that describes block
- May contain:
  - File ID (or RELATION or DB ID)
  - This block ID
  - Record directory
  - Pointer to free space
  - Type of block (e.g. contains recs type 4; is overflow, . . .)
  - Pointer to other blocks “like it”
  - Timestamps . . .

Example: Indirection in block

- Consider the records stored in a block:



- `Address of a record = address of the block that contains the record + some offset information`

Pointer Swizzling

- When the block is read in main memory, it receives a main memory address
- Need another translation table
- Optimization: Pointer Swizzling
  - The process of replacing a physical/logical pointer with a main memory pointer
  - Still need translation table, but subsequent references are faster

Other Topics

1. Modification of Records
2. Buffer Management
3. Comparison of Schemes

Modification of Records

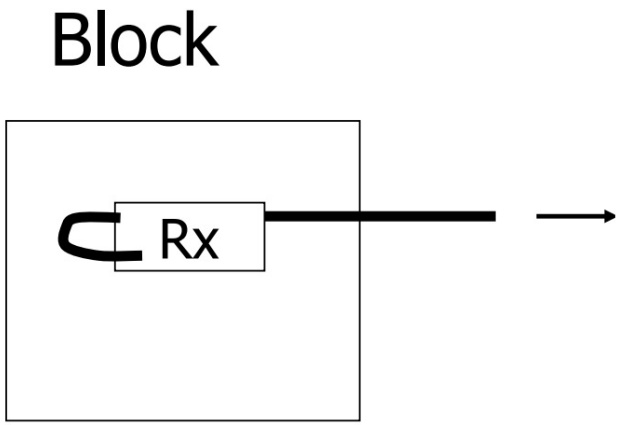
How to handle the following operations on the record level?

1. Insertion
2. Deletion
3. Update

1) Insertion

- **Easy case** Records fixed length/not in sequence(unordered)
  - Insert new record at end of file
  - or, in deleted slot
- **A little harder**
  - If records are variable size, not as easy
  - may not be able to reuse space - fragmentation
- **A Difficult case**: records in sequence (ordered)
  - Find position and slide following records
  - If records are sequenced by linking, insert overflow blocks

2) Deletion



Options

(a) Deleted and immediately reclaim space by shifting other records or removing overflows

(b) Mark deleted and list as free for re-use

Trade-offs

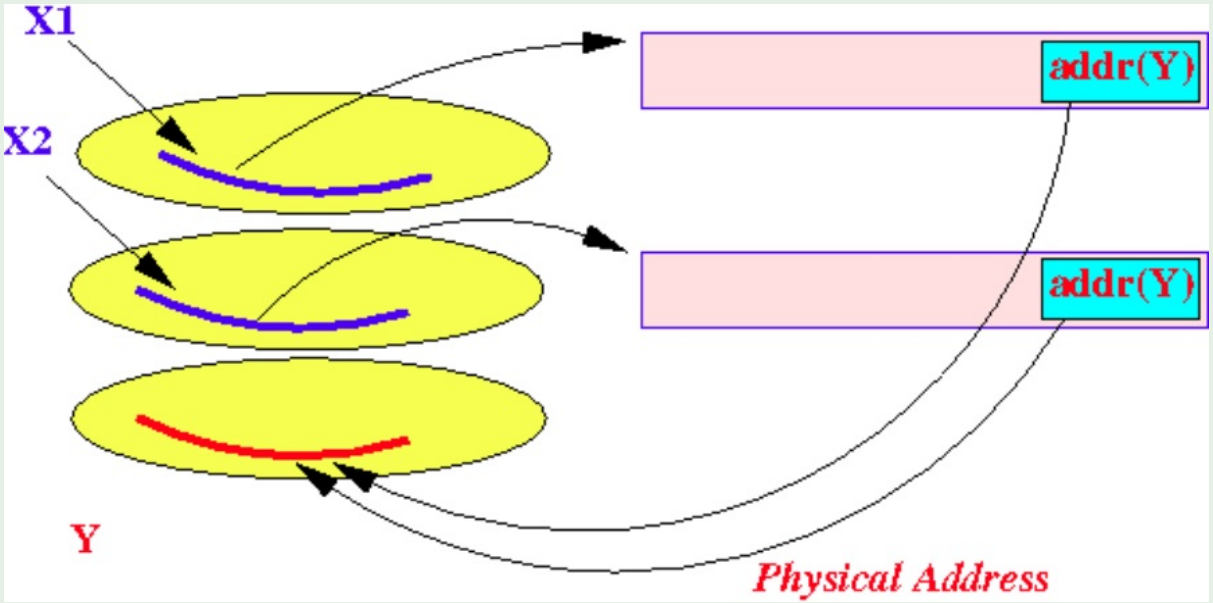
- How expensive is immediate reclaim?
  - How expensive is to move valid record to free space for immediate reclaim
- How much space is wasted?

Concern with deletions

A caveat when using physical addresses to reference a block/record

Example

- Record Y can be referenced by other records (e.g., records $X1$ & $X2$)

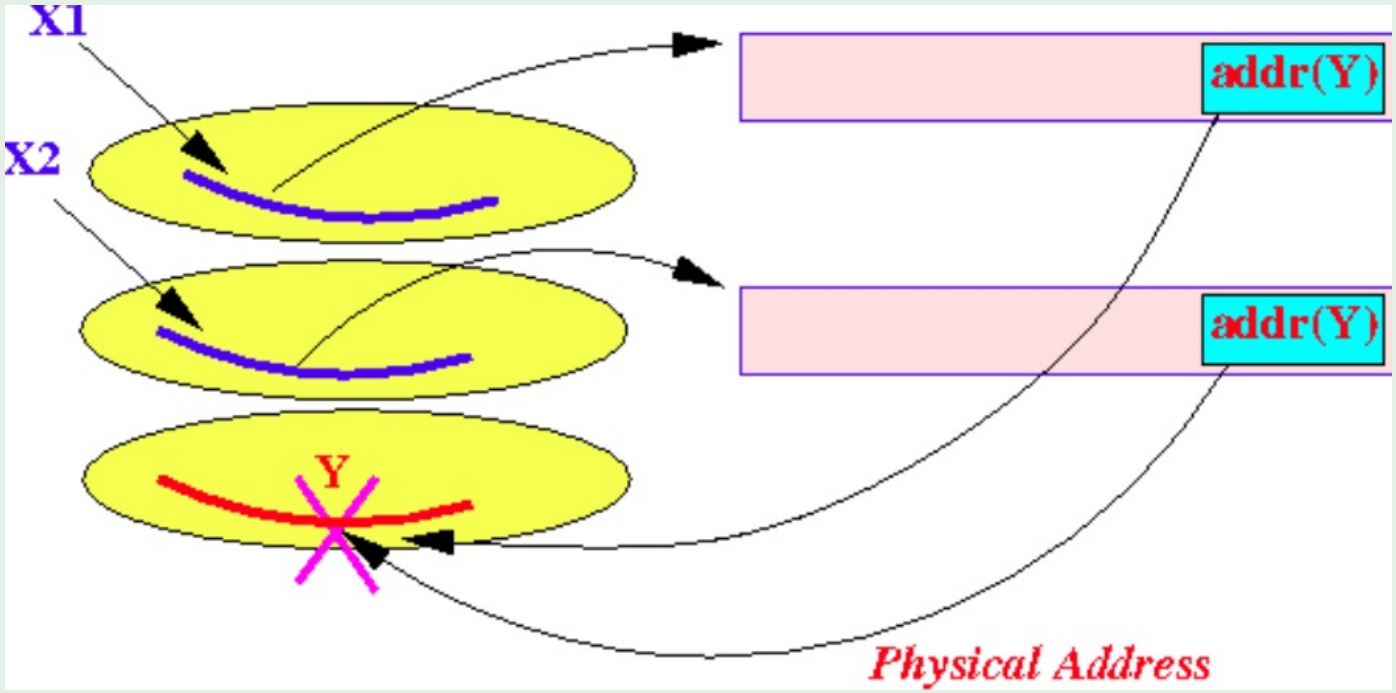


Concern with deletions

A caveat when using physical addresses to reference a block/record

Example

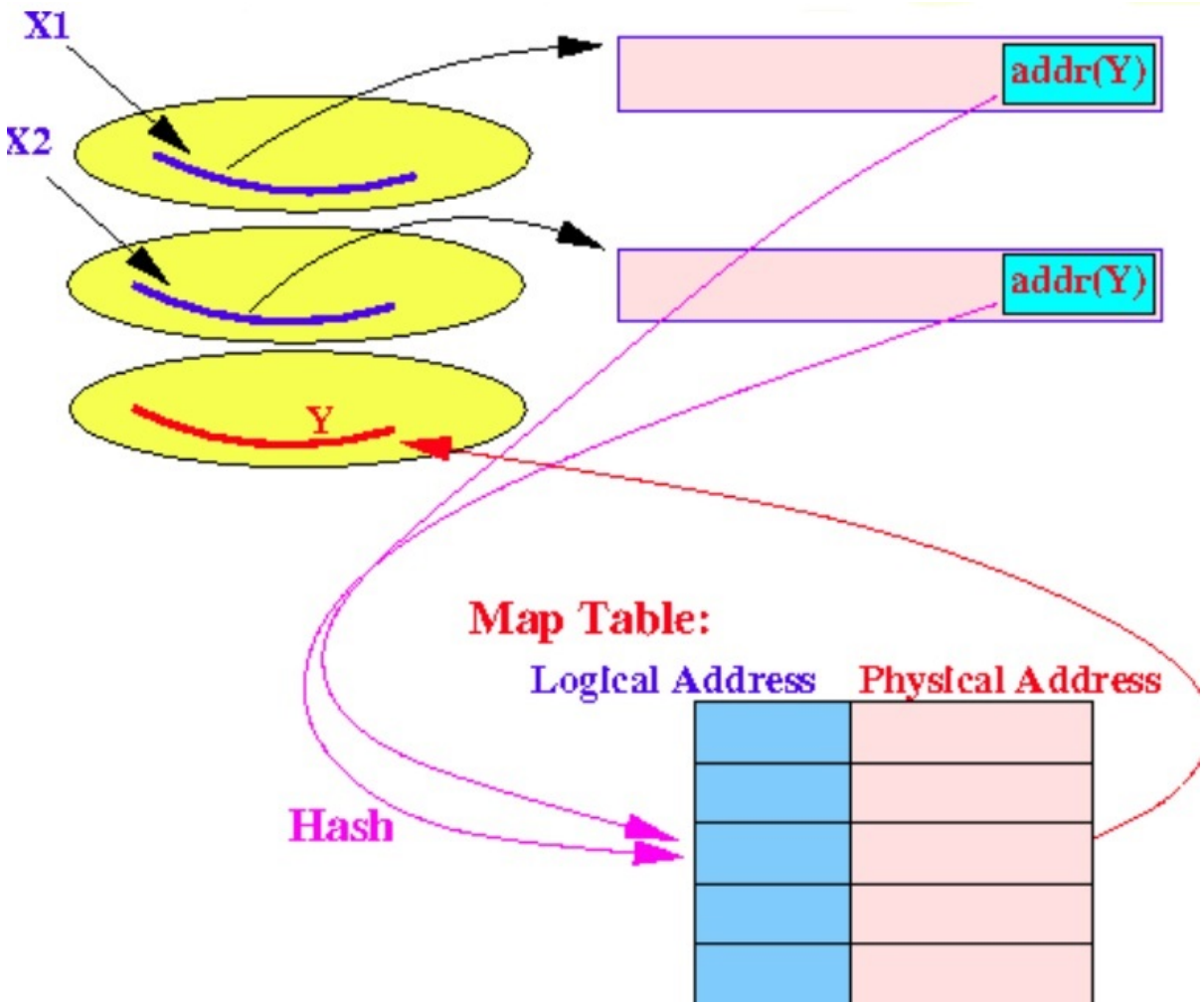
- When the record Y is deleted



- the physical addresses will reference an incorrect record

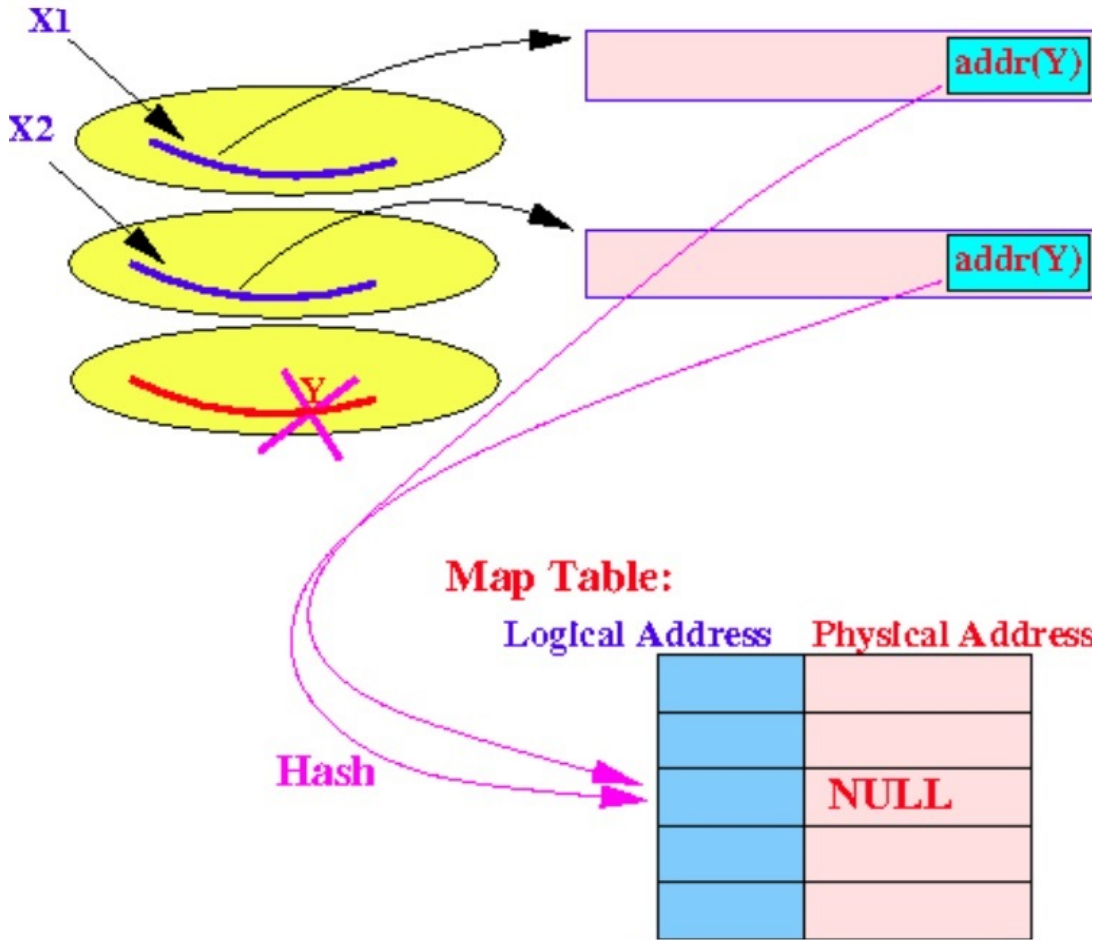
Techniques to handle record deletion

- Using logical addresses is easy
- Before deleting record Y that is referenced by records $X1$ and $X2$



Techniques to handle record deletion

- Using logical addresses is easy
- After deleting record Y



- Deleted record is identified by a NULL physical address in the Map table

Very important

- The logical address used by record Y must remain in the map table
- Furthermore:
  - The logical address used by record Y cannot be re-used

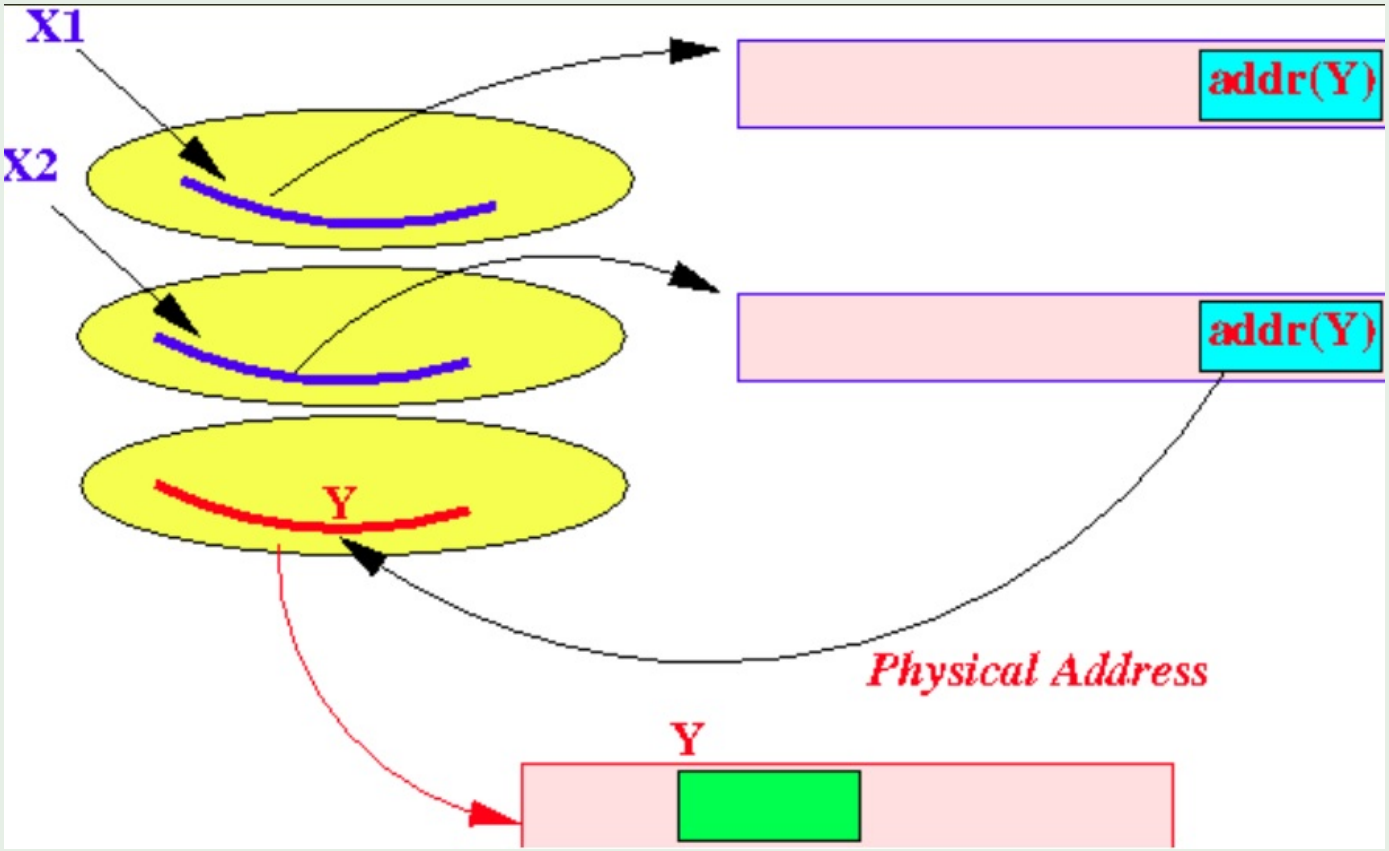
Techniques to handle record deletion

- Deleting a record using physical address: use a tombstone record
- Tombstone record: a (very small) special purpose record used to indicate a deleted record
- When a record is delete, it is replaced by the tombstone record
- This tombstone is permanent, it must exist until the entire database is reconstructed

Tombstones

Example

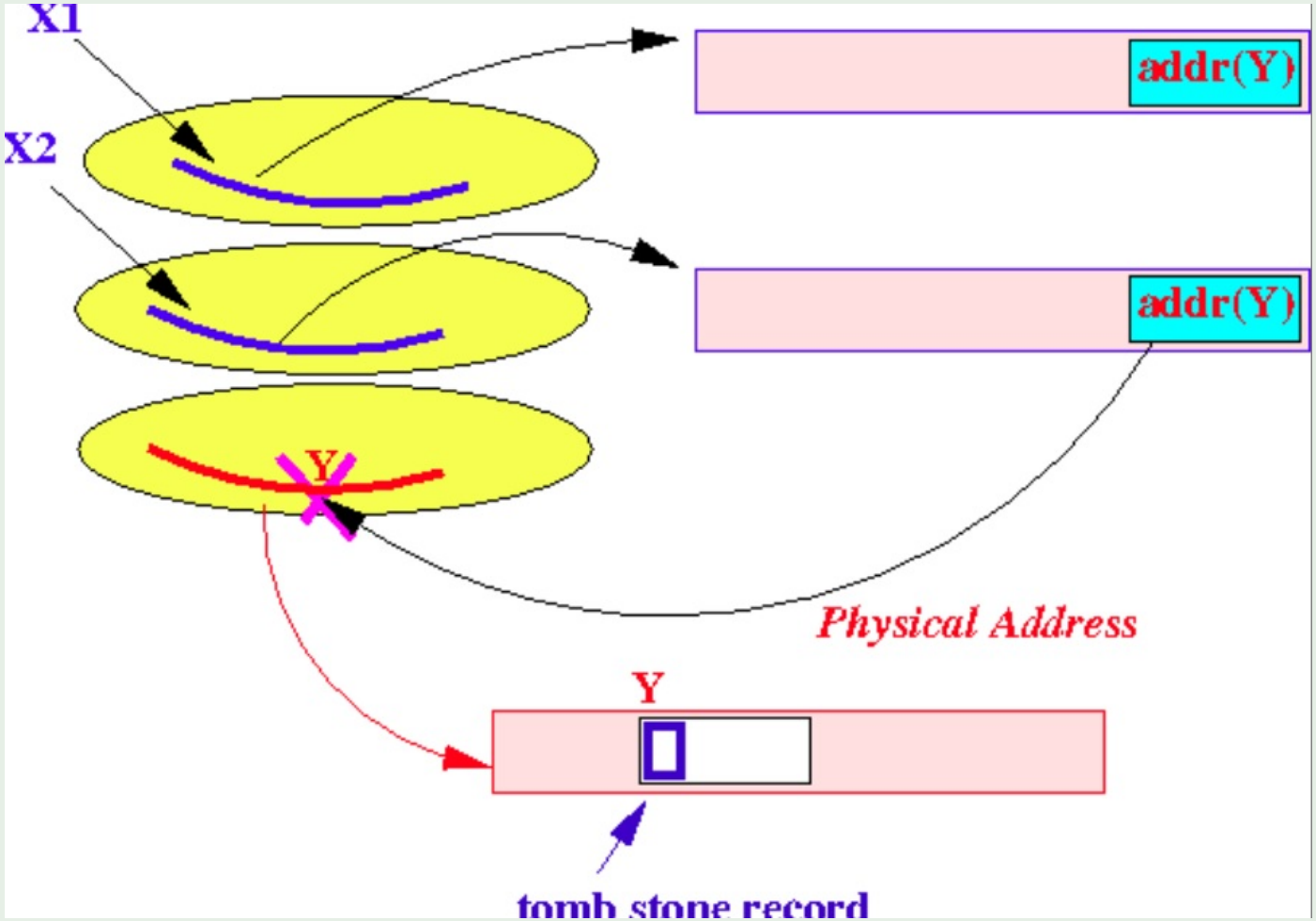
- Before deleting record Y



Tombstones

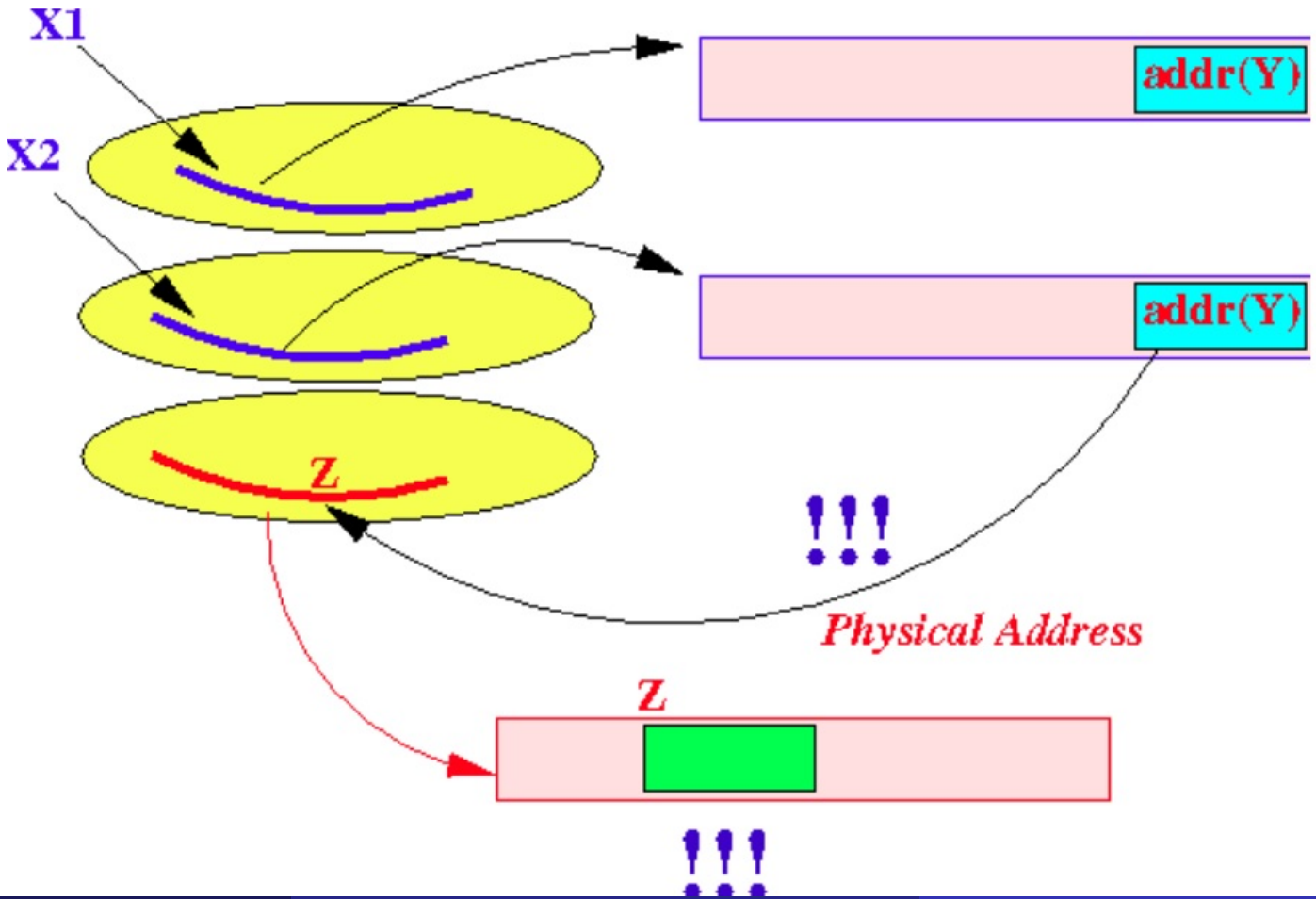
Example

- After deleting record Y



Tombstones

- When you insert a new record, you cannot use the space of a tombstone record (tombstone record must be preserved)
- Because: Existing record references to the deleted record will then references to the newly inserted record:



Update

- If new record is shorter than previous, easy
- If it is longer, need to shift records, create overflow blocks
- Note: We will never create a tombstone record in an update operation

Other Topics

1. Modification of Records
2. Buffer Management
3. Comparison of Schemes

Buffer Management

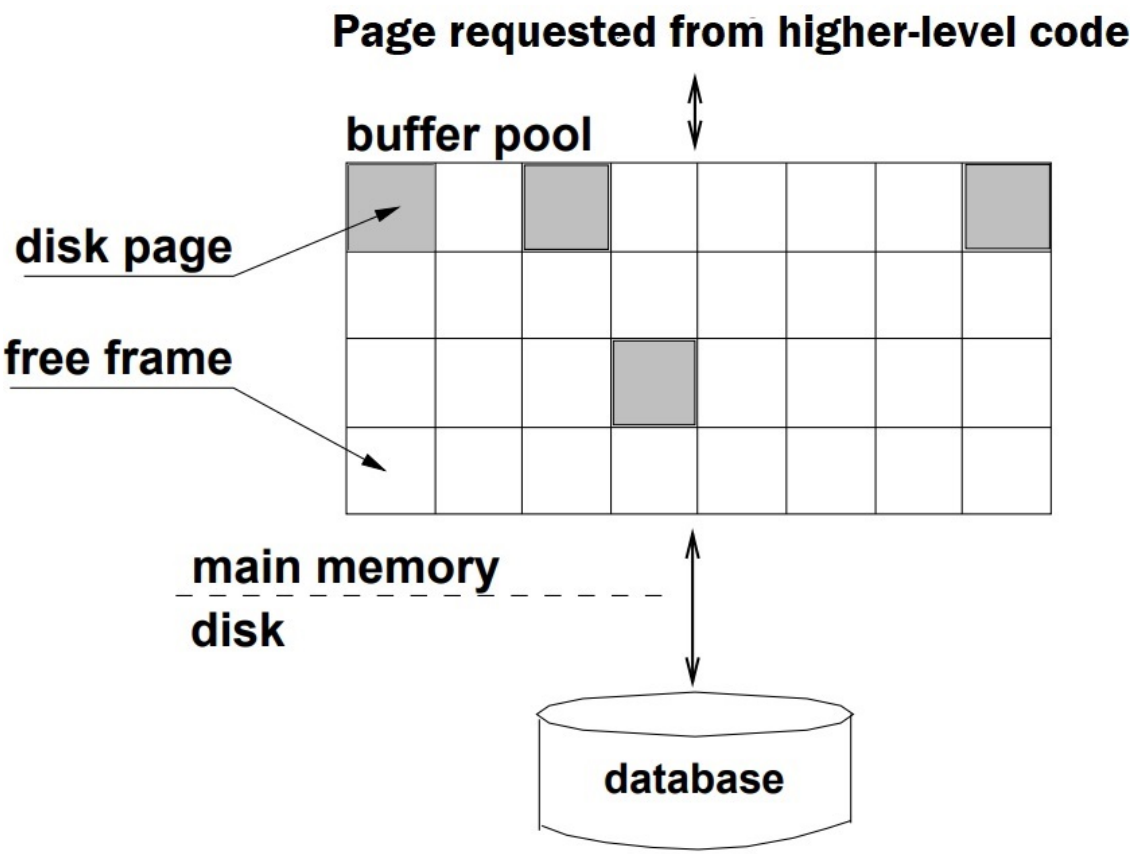
- For Caching of Disk Blocks
- Buffer Replacement Strategies
  - E.g., LRU, clock
- Pinned blocks
- Forced output
- Double buffering (Notes02)

Buffer Manager

`Size of the database on secondary storage` $\gg$ `size of available primary memory to hold user data.`

- To scan the entire pages of a 20 GB table (`SELECT * FROM ...`), the DBMS needs to
  1. **bring in** pages as they are needed for in memory processing,
  2. **overwrite** (replace) such pages when they become obsolete for query processing and new pages require in-memory space.
- The buffer manager manages a collection of pages in a designated main memory area, the **buffer pool**.
  - Manages blocks cached from disk in main memory
- once all slots, **frames**, in this pool have been occupied, the buffer manager uses a **replacement policy** to decide which frame to overwrite when a new page needs to be brought in.

Buffer Manager



- Two variable for each frame:
  - `pin_count`: indicates how many "users" (e.g., transactions) are working with that page,
  - boolean variable `dirty`: indicates whether the page has been modified since it was brought into the buffer pool from disk

Buffer replacement policies/strategies

The choice of victim frame selection (or buffer replacement) policy can considerably affect DBMS performance:

- LRU
- Clock
- ⋮

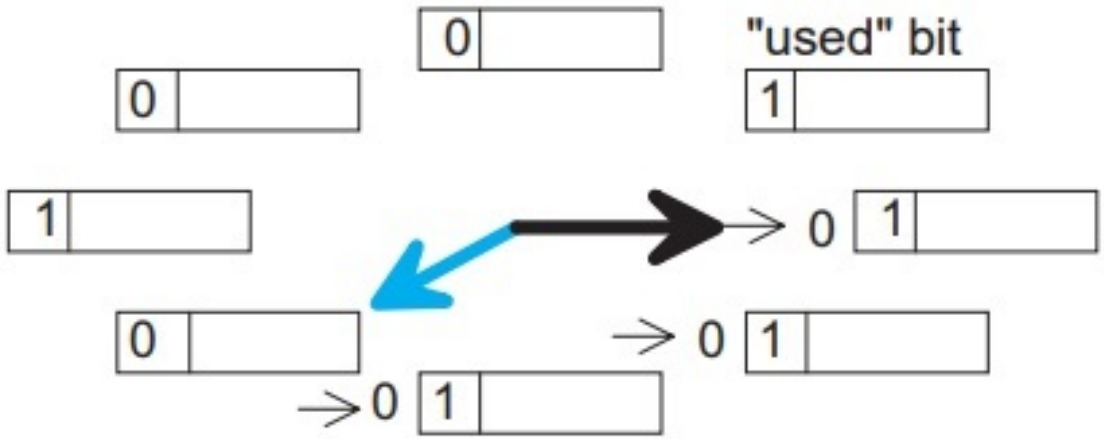
Least Recently Used (LRU)

- Replace page that has not been accessed for the longest time
- Implementation:
  - Keep a queue of pointers to frames with `pin_count` 0
  - A frame is added to the tail of queue, when `pin_count` is decremented to 0
  - To find the next victim, the page in the frame at the head of the queue

Clock: "second chance"

- Frames are organized clock-wise
- Number the N frames in buffer pool $0 \ldots N-1$, initialize counter `current` $\leftarrow 0$, and maintain a bit array `referenced`$[0 \ldots N-1]$, initialized to all 0
  - Page P is loaded or accessed `referenced`$[P] \rightarrow 1$
- To find the next victim, consider page `current`:
  - If `pin_count(current) = 0` and `referenced[current] = 0`, `current` is the victim.
  - Otherwise, `referenced[current]` $\leftarrow 0$, `current` $\leftarrow$ `(current +` $1)$ mod N, repeat.

Clock: “second chance”



Other Replacement Strategies

Other well-known replacement policies are, e.g.:

- LRU-K
- GCLOCK
- Clock-Pro
- ARC
- LFU

Why not use the Operating System for the task?

- DBMS may be able to anticipate access patterns
  - Hence, may also be able to perform prefetching
- DBMS needs the ability to force pages to disk, for recovery purposes

Row vs Column Store

1. So far we assumed that fields of a record are stored contiguously (**row store**)
2. Another option is to store like fields together (**column store**)

Row Store

- Order consists of
  `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

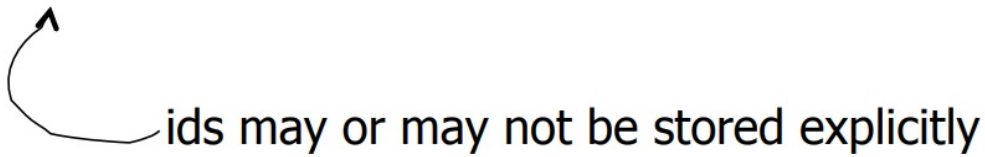
Column Store

- Order consists of
  `id, cust, prod, store, price, date, qty`

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

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

id1	price1	qty1
id2	price2	qty2
id3	price3	qty3
id4	price4	qty4
...	...	...



Row vs Column Store

- Advantages of Column Store
  - more compact storage (fields need not start at byte boundaries)
  - efficient reads on data mining operations
- Advantages of Row Store
  - writes (multiple fields of one record) more efficient
  - efficient reads for record access
- More information: "Column-Stores vs. Row-Stores: How Different Are They Really?"

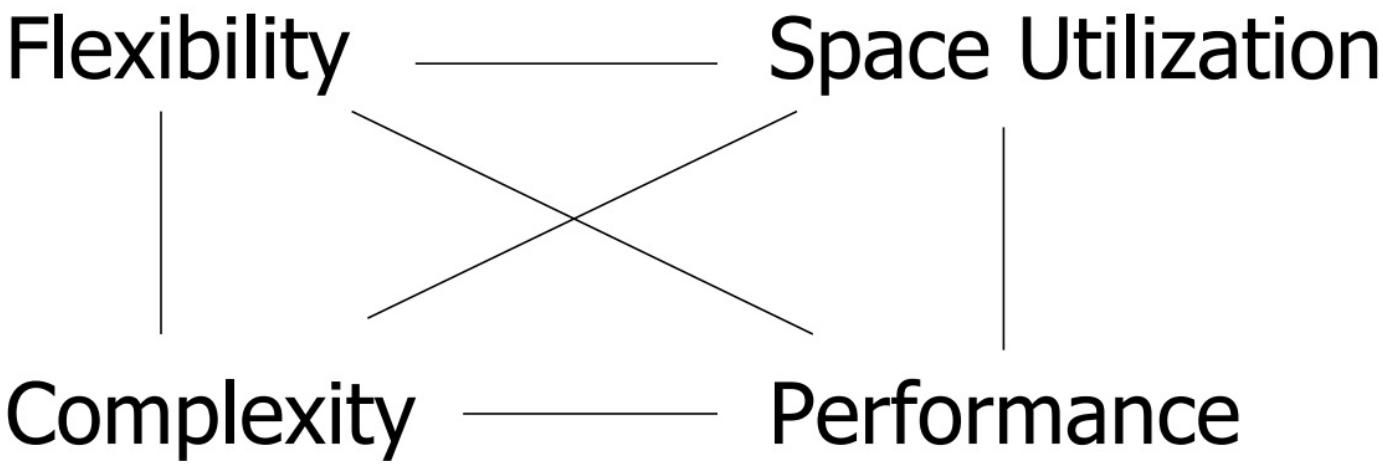
Other Topics

1. Modification of Records
2. Buffer Management
3. Comparison of Schemes

Comparison

- There are $10,000,000$ ways to organize my data on disk ...
- Which is right for me?

Issues



To evaluate a given strategy, compute following parameters

- space used for expected data
- expected time to
  - fetch record given key
  - fetch record with next key
  - insert record
  - append record
  - delete record
  - update record
  - read all file
  - reorganize file

Example

- How would you design MEGATRON 3000 storage system? (for a relational DB)
  - Variable length records?
  - Spanned?
  - What data types?
  - Fixed format?
  - Record IDs ?
  - Sequencing?
  - How to handle deletions?

Next

- How to find a record quickly, given a key