
UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN
CS411 - Data Storage/Representation
illinois.edu

Exam

- Grades posted on Compass
  - Still grading online students
  - 15 point curve
  - mean: 85.4
  - median: 88.6

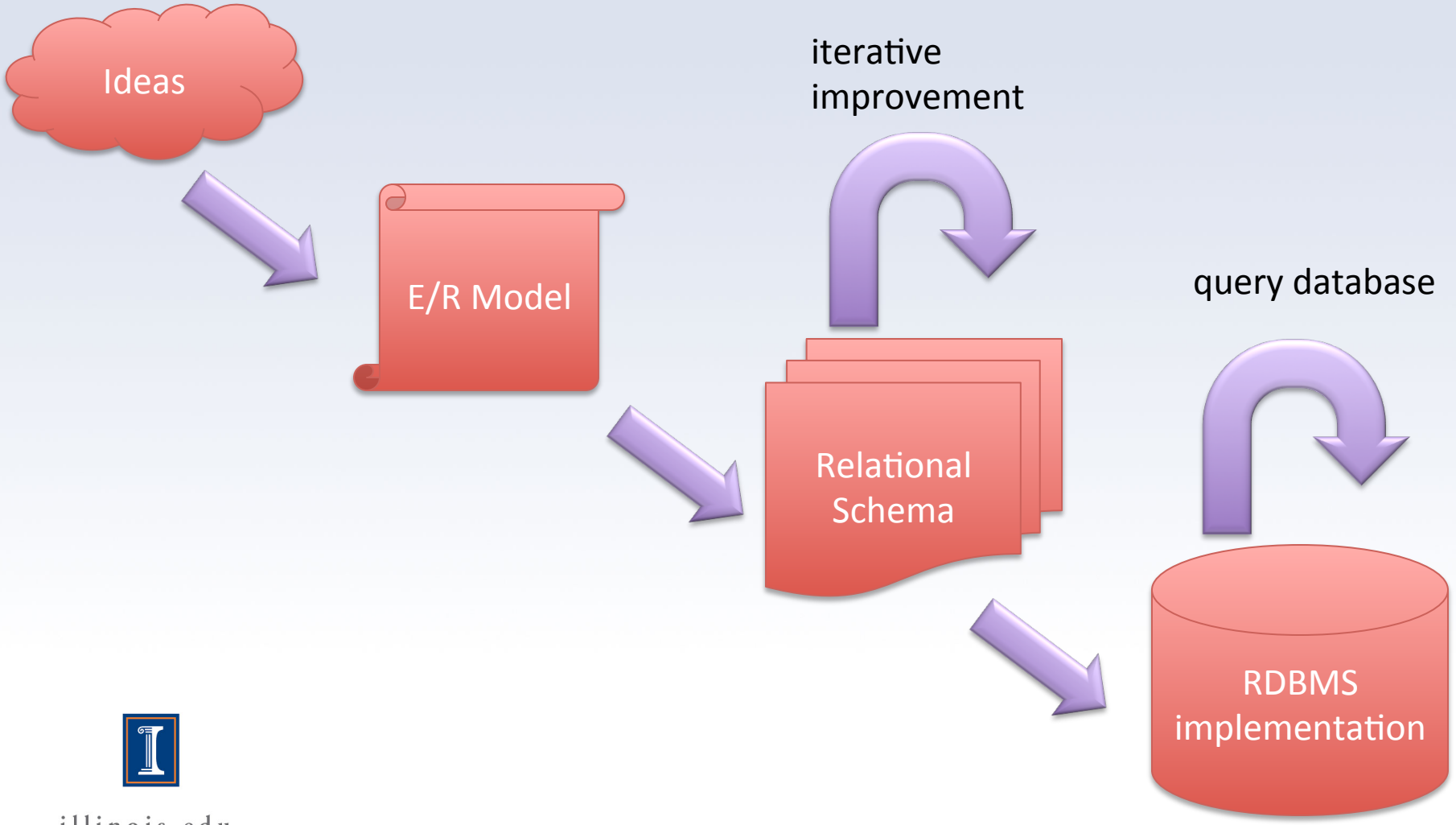
Exam

- Solutions have been posted
- Multiple choice:
  - in svn directory, you'll find multiple_choice.txt

Announcements

- MP2 will be posted Friday
  - About 10 SQL statements to write

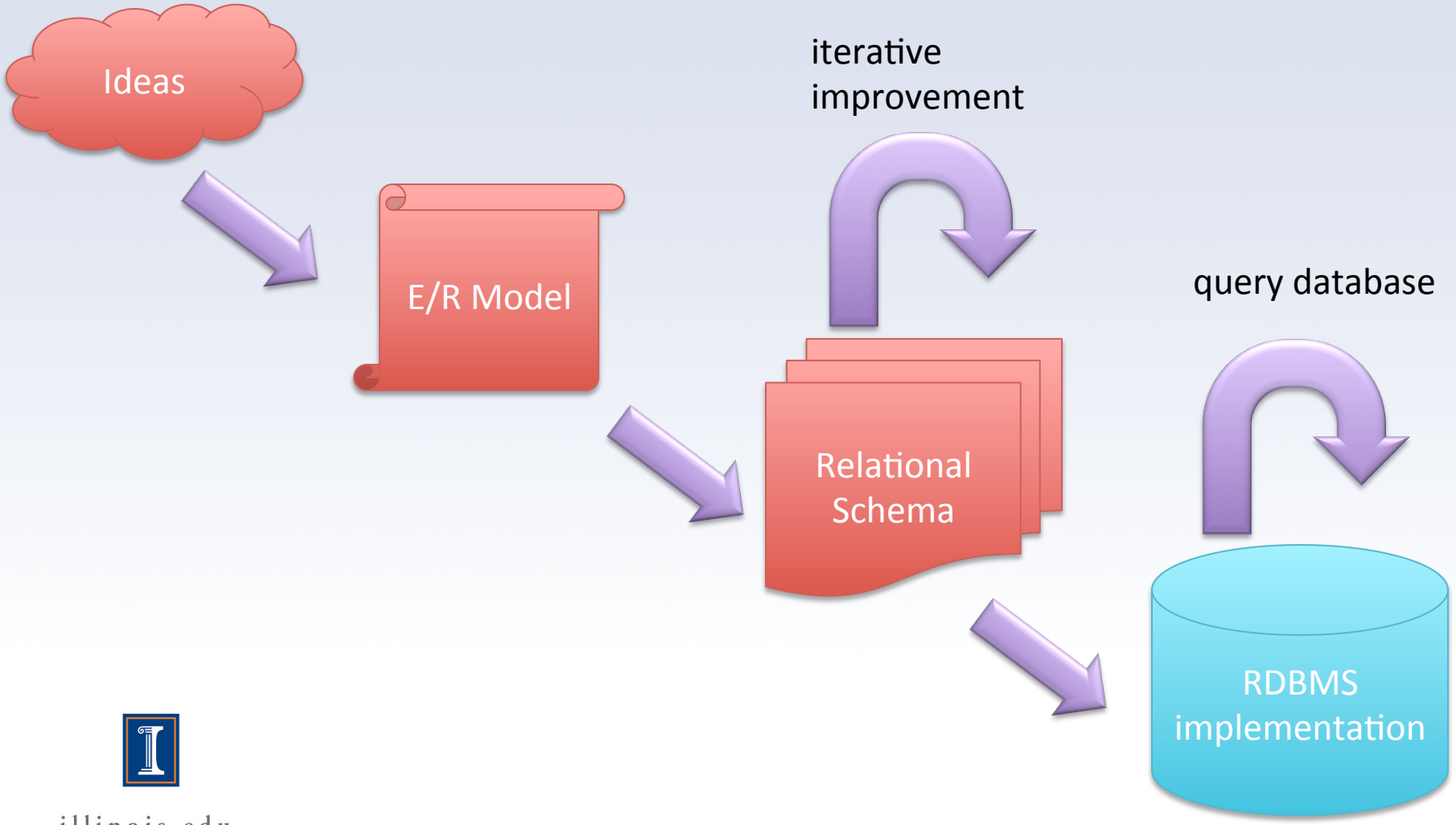
Our view of databases



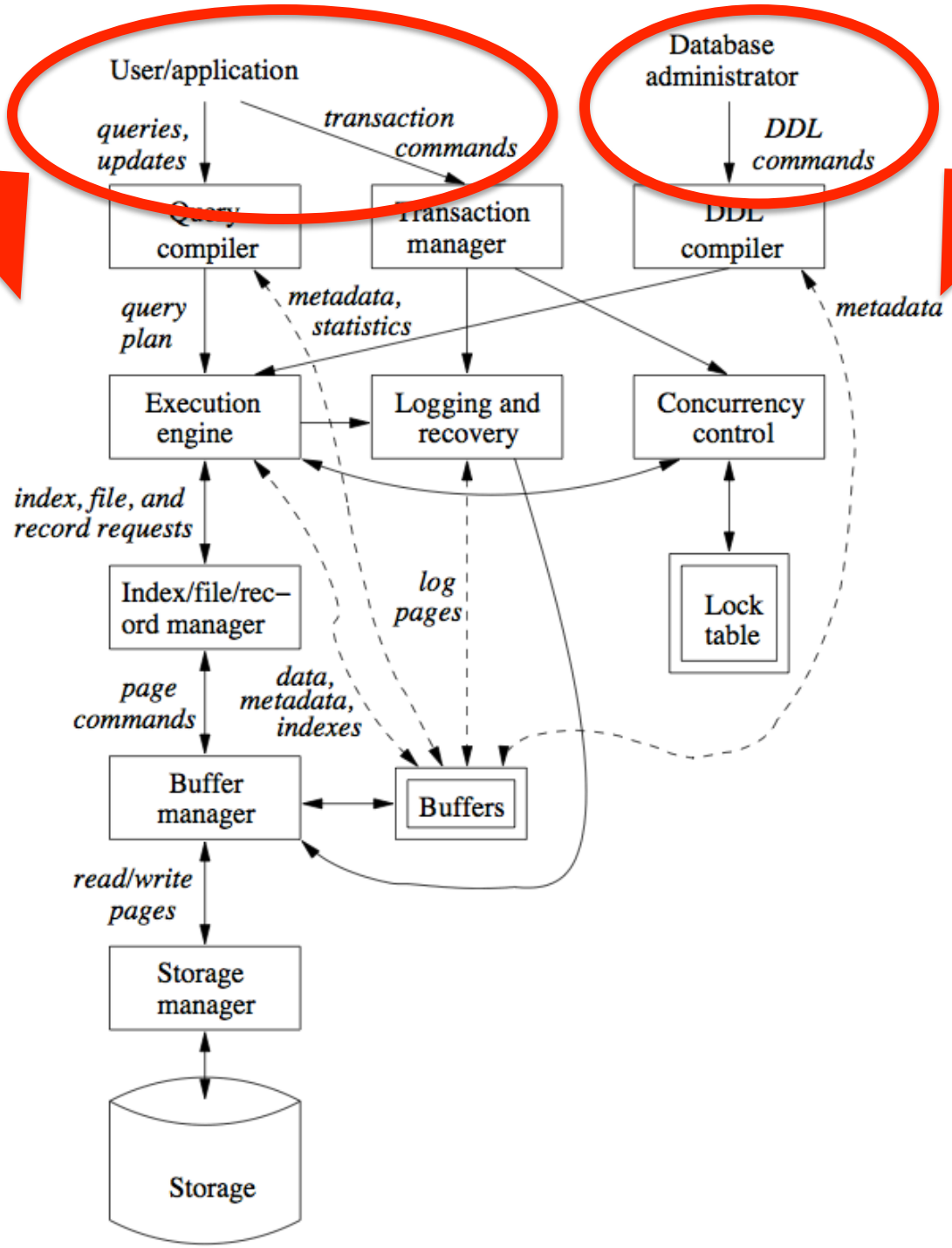
What's next?

- Anybody can USE a DBMs
- This is a 400 level course in a top CS program
- We need to go further!
- We need to know how DBMSs ***ACTUALLY WORK!***

Our view of databases

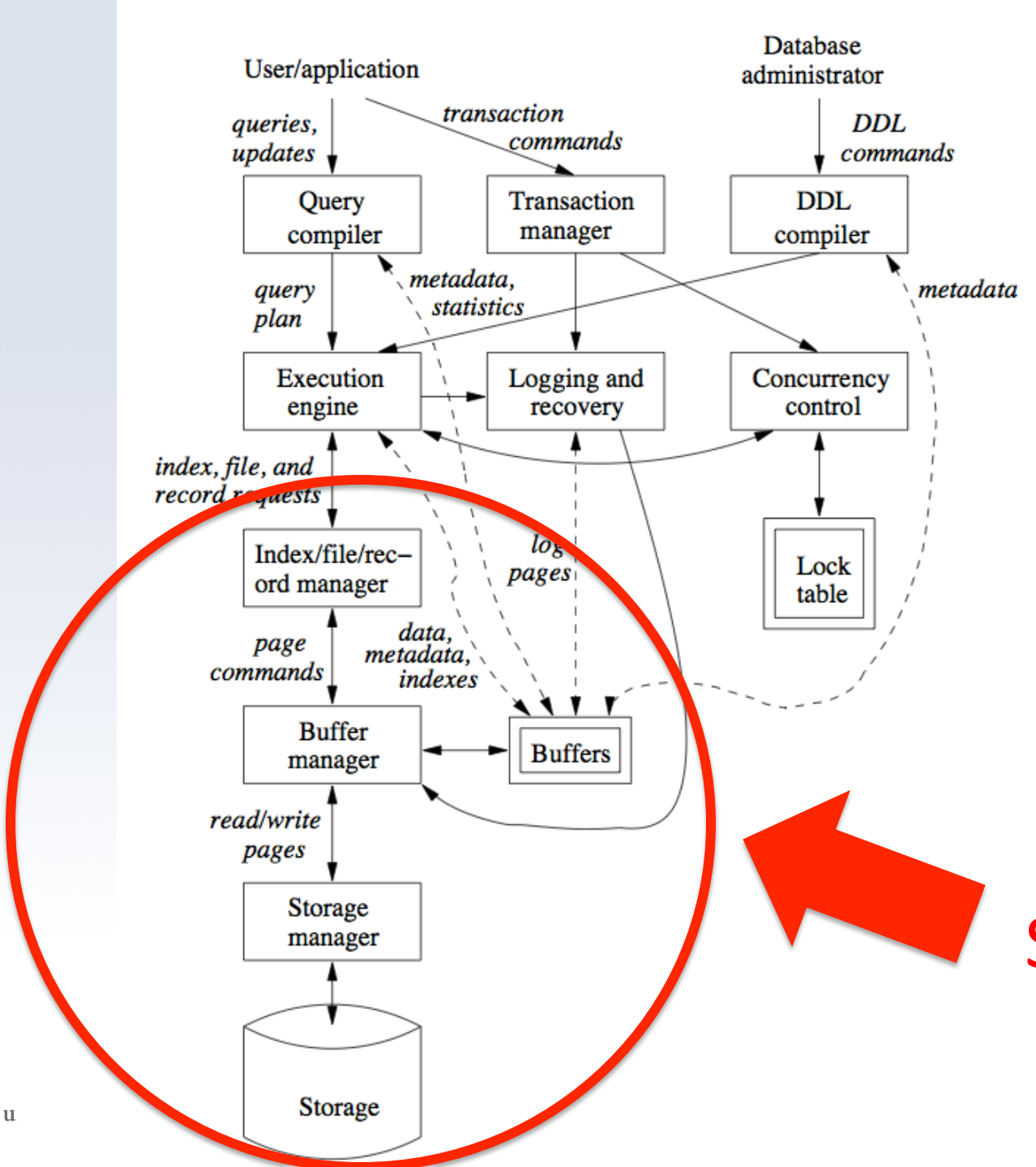


YOU ARE HERE



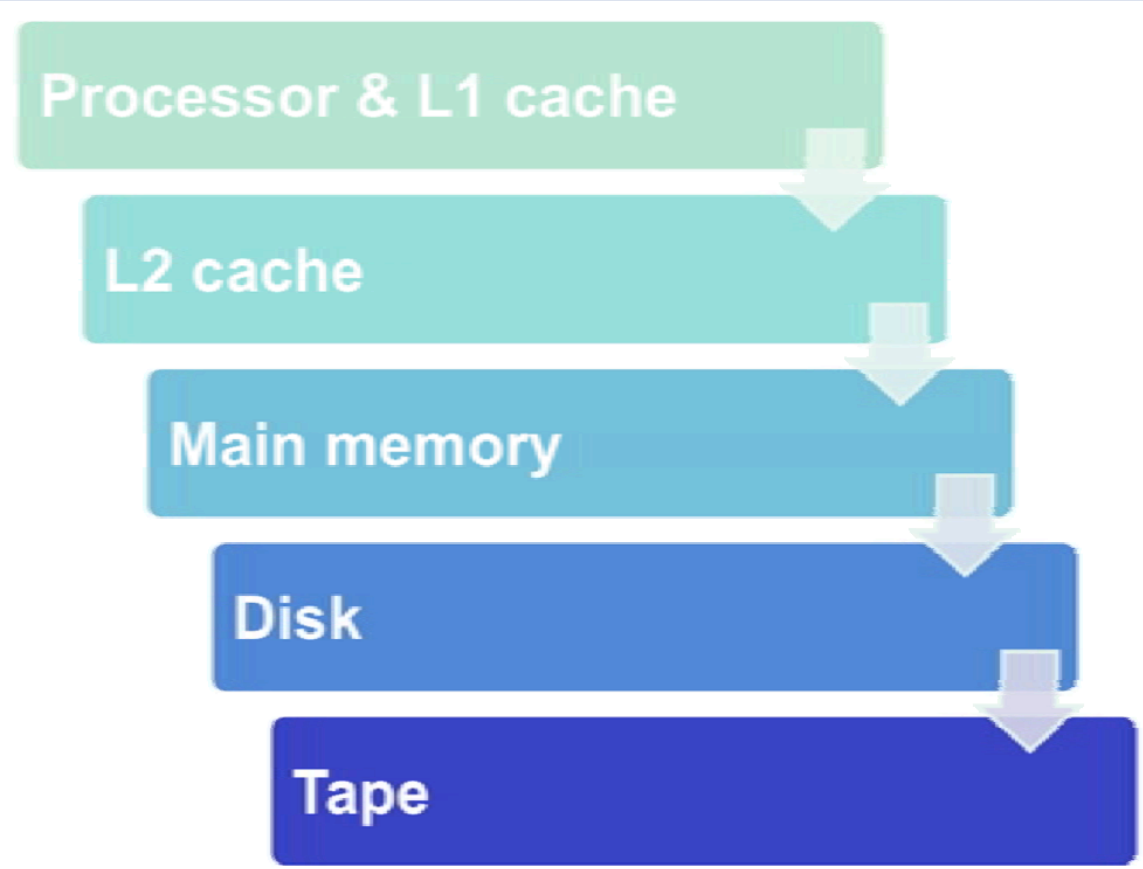
ALSO HERE

illinois.edu


User/application
Database administrator
queries, updates
transaction commands
DDL commands
Query compiler
Transaction manager
DDL compiler
query plan
metadata, statistics
metadata
Execution engine
Logging and recovery
Concurrency control
index, file, and record requests
log pages
Index/file/rec–ord manager
Lock table
page commands
data, metadata, indexes
Buffer manager
Buffers
read/write pages
Storage manager
Storage

START HERE

The Memory Heirarchy



The Memory Heirarchy

1. Processor Cache
   - accessed in a few nanoseconds
2. Main Memory
   - accessed in tens of nanoseconds
3. Secondary Storage
   - accessed in tens of miliseconds
4. Tertiary Storage
   - accessed in seconds

The Memory Heirarchy

1. Processor Cache
   - can store tens of MB
2. Main Memory
   - can store tens of GB
3. Secondary Storage
   - can store a few TB
4. Tertiary Storage
   - can potentially store petabytes

The Memory Heirarchy

1. Processor Cache
   - volatile
2. Main Memory
   - volatile
3. Secondary Storage
   - nonvolatile
4. Tertiary Storage
   - nonvolatile

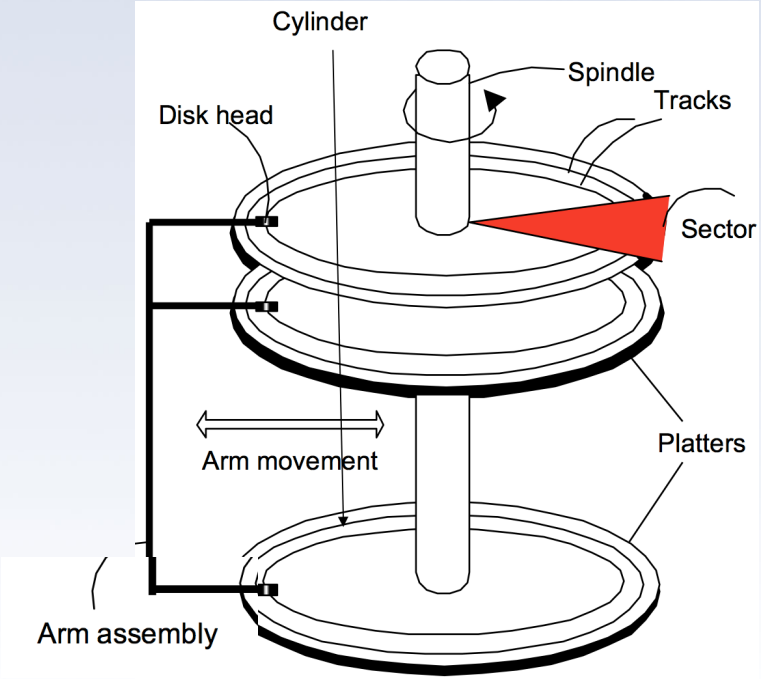
The Memory Heirarchy

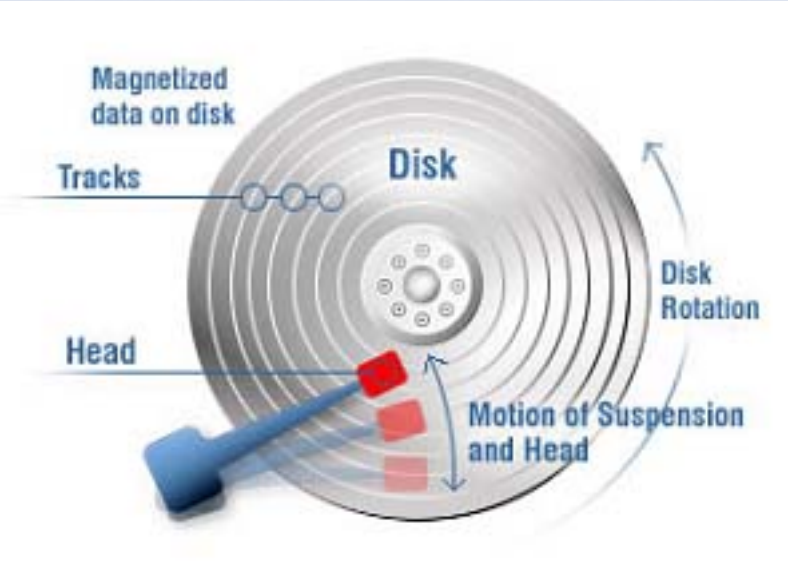
1. Processor Cache
   - fast, small, volatile, expensive
2. Main Memory
   - medium, medium, volatile
3. Secondary Storage
   - slow, huge, nonvolatile, cheap
4. Tertiary Storage
   - lethargic, gargantuan, nonvolatile, cheap

Mechanics of Disks

- DBMSs were designed to manage access ***secondary storage***
- Secondary storage is made of hard disks
- To understand DBMS design, we will start with the hard disks

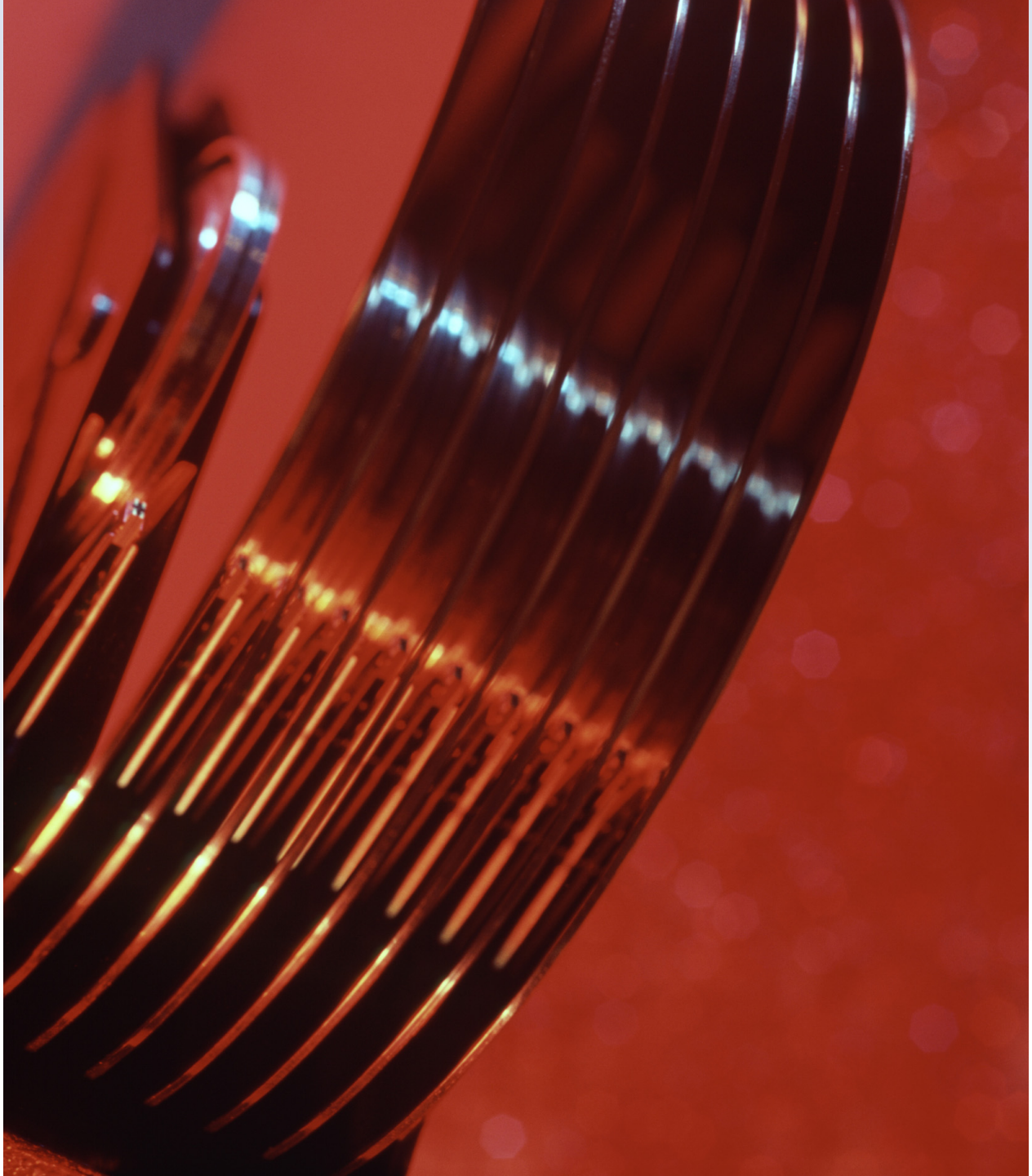
Mechanics of Disks





Mechanics of Disks

- Hard disk spinning
- Fun with hard drives
- Not so fun with hard drives



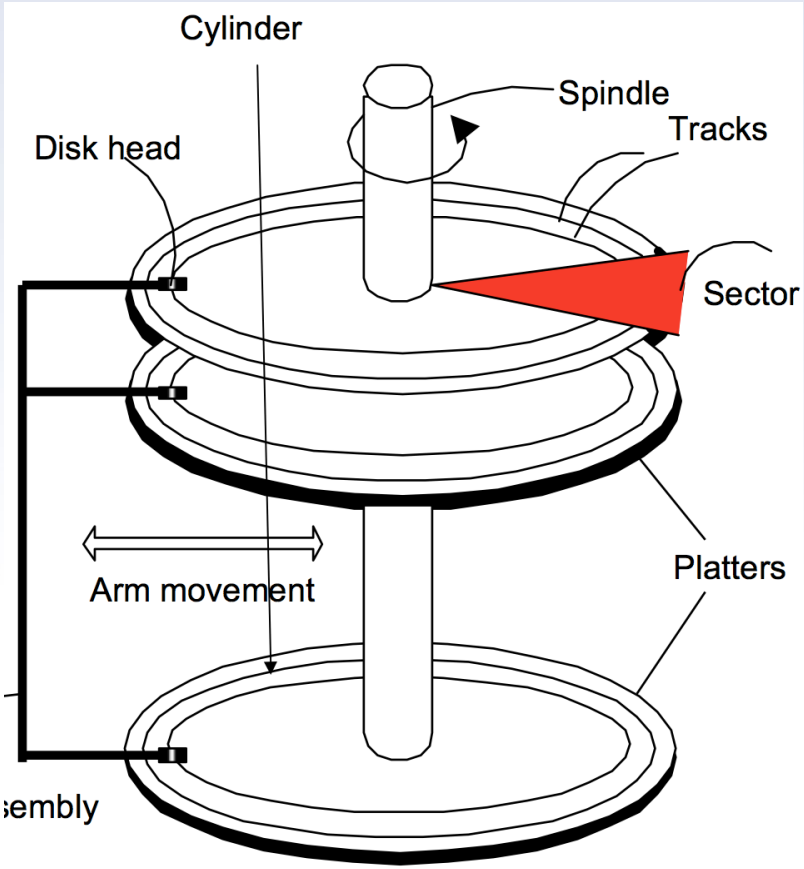
illinois.edu

Disk access

- Fundamental unit of transfer is a *block*
- Blocks are typically 4-64kb
- When in main memory, blocks are sometimes called *pages*

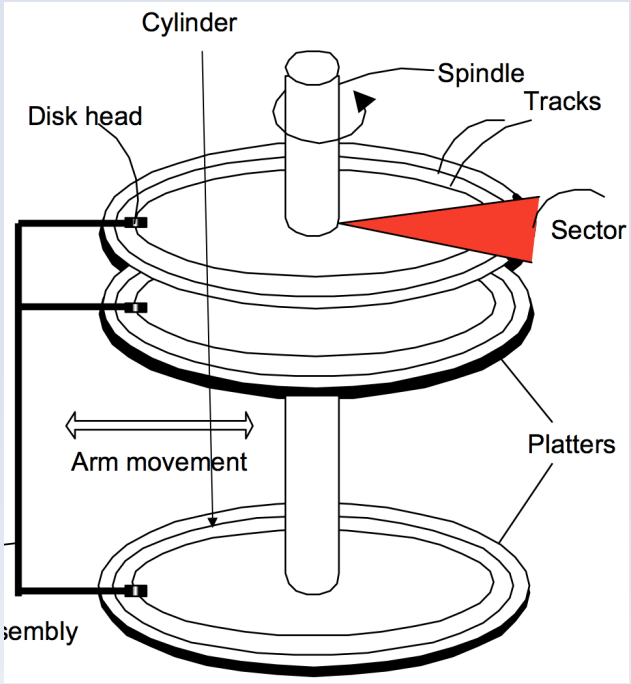
Disk access

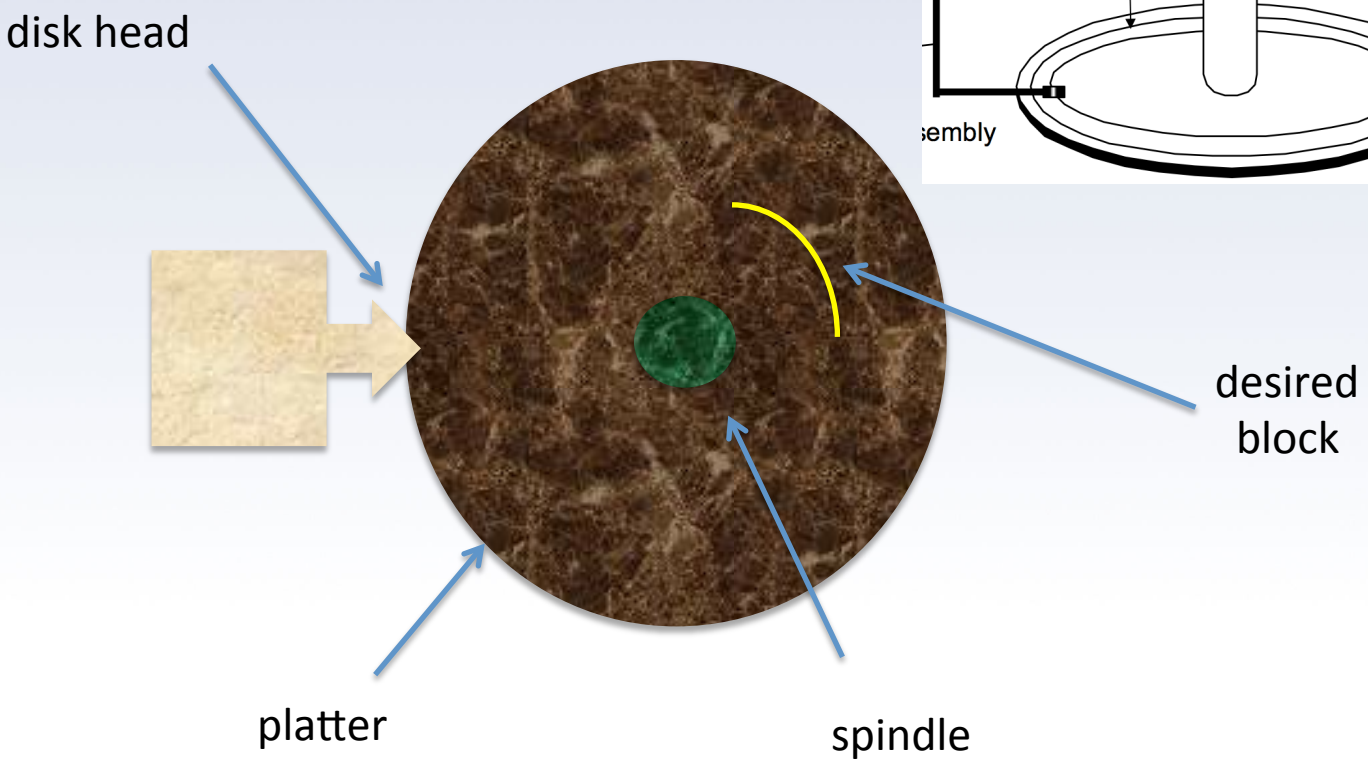
- Each block has a ***physical address***
  1. host machine (if multiple machines involved)
  2. disk number
  3. cylinder number
  4. track number
  5. block number



Disk access

- Requires three steps:
  1. ***Seek time*** - Move head to the right cylinder
  2. ***Rotational latency*** - Wait for disk to spin to sector beginning
  3. ***Transfer time*** - Time for head to read/write to sector as it passes by

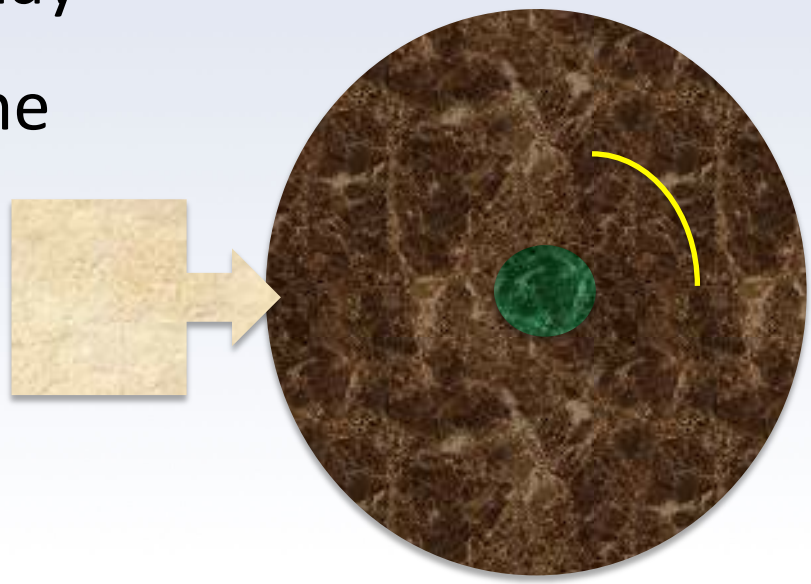

Cylinder
Spindle
Tracks
Disk head
Sector
Arm movement
Platters
sembly


disk head
desired block
platter
spindle

1) seek time

2) rotation delay

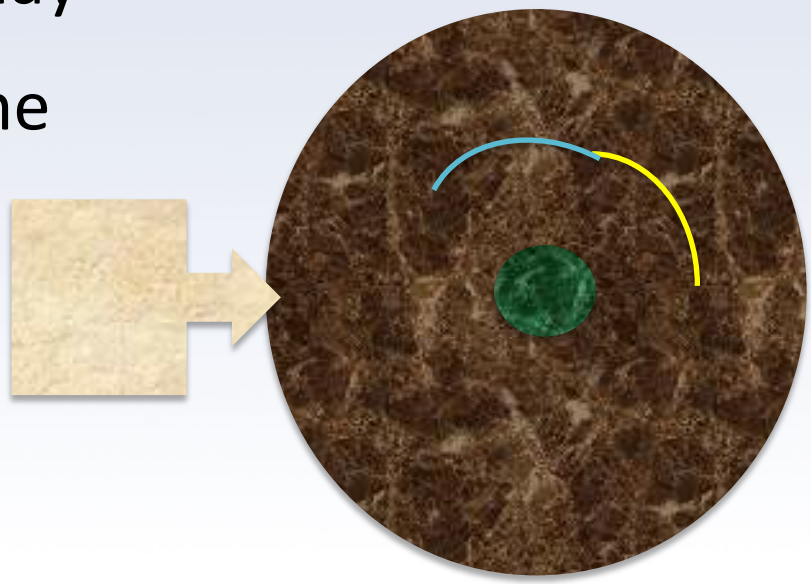
3) transfer time



1) seek time

2) rotation delay

3) transfer time



Disk access

KEY POINT:

If we have to read multiple blocks in a row, we only have to pay the seek and rotation delay costs ***once***

Disk access

KEY POINT:

It's better to do sequential reads

Arranging DBMS data

- How can we organize relations and their tuples on the disk?
- Let's start with a fixed-length record
- Before we begin, we need to learn new terminology

New Terminology

Rel Model	attribute	tuple	relation
SQL	column/field	row	table
Disk	field	record	file

Fixed Length Records

```
CREATE TABLE Person(
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    gender CHAR(1),
    birthdate DATE
);
```

FIRST GUESS: 295 bytes + 1 bit

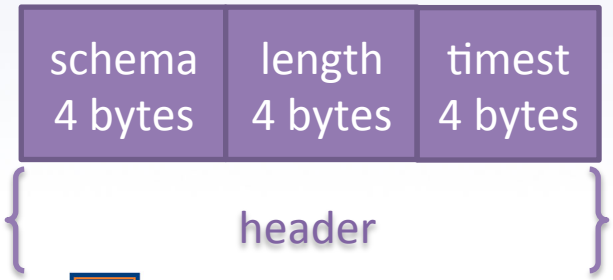
Fixed Length Records

- Reality is slightly more complicated
  - Align fields along ***word boundaries***
    - minimum unit addressable in memory
    - usually 4 or 8 bytes
  - Need to include a header
    - pointer to the schema for the record
    - length of the record
    - last modified timestamp/access metadata
    - pointers to the fields

Fixed Length Records

```
CREATE TABLE Person(
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    gender CHAR(1),
    birthdate DATE
);
```

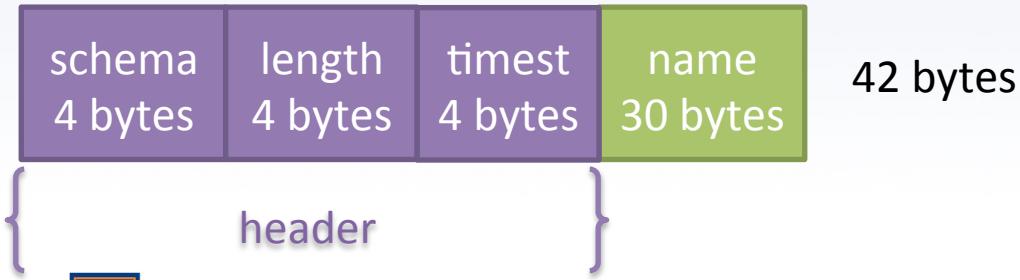
Add a pointer to the schema, the length, and the timestamp data



12 bytes

Fixed Length Records

```
CREATE TABLE Person(
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    gender CHAR(1),
    birthdate DATE
);
```

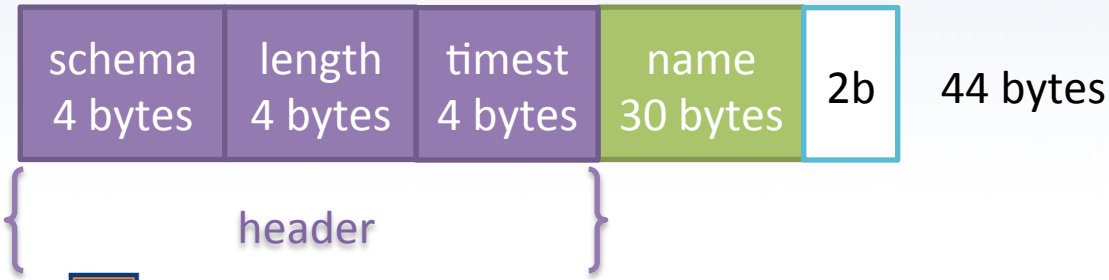


42 bytes

Fixed Length Records

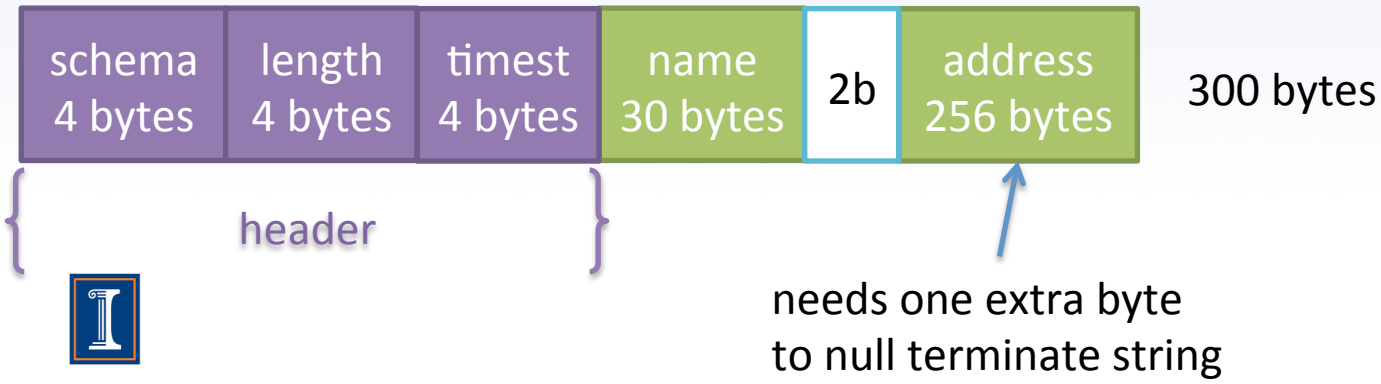
```
CREATE TABLE Person(
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    gender CHAR(1),
    birthdate DATE
);
```

Need to pad out name so divisible by 4



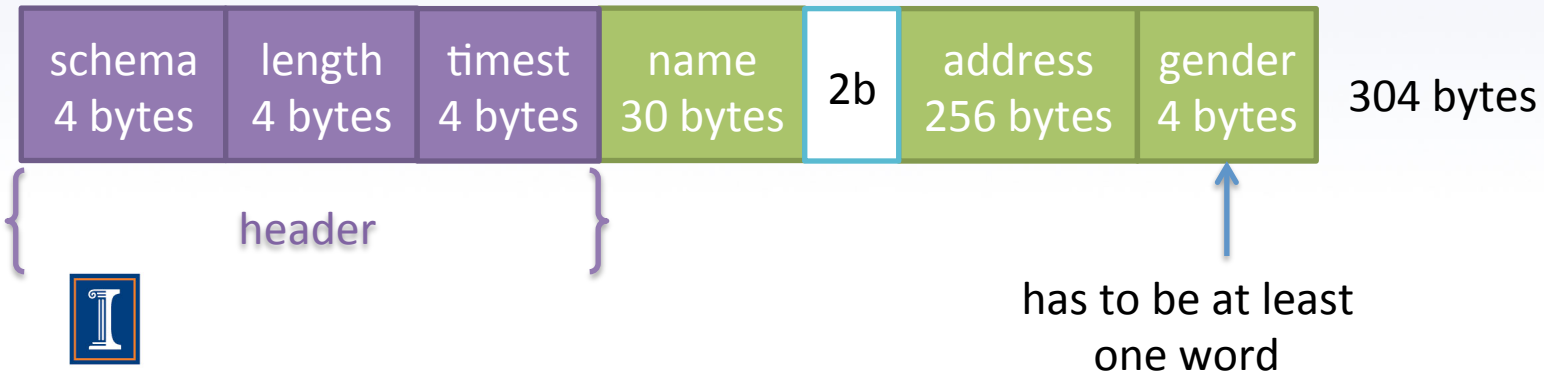
Fixed Length Records

```
CREATE TABLE Person(
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    gender CHAR(1),
    birthdate DATE
);
```



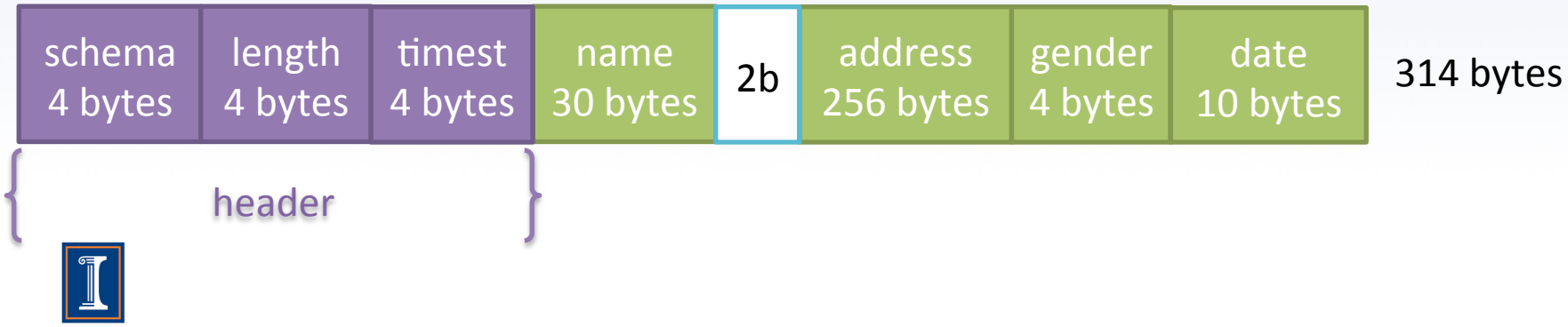
Fixed Length Records

```
CREATE TABLE Person(
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    gender CHAR(1),
    birthdate DATE
);
```



Fixed Length Records

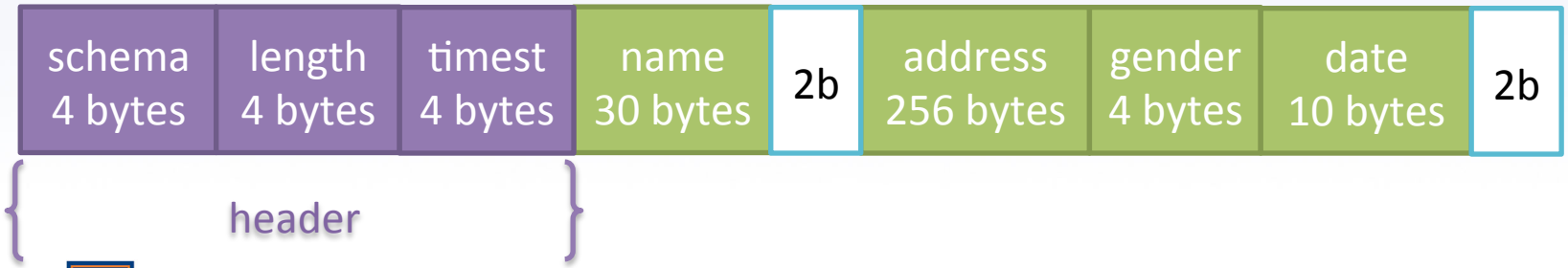
```
CREATE TABLE Person(
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    gender CHAR(1),
    birthdate DATE
);
```



Fixed Length Records

```
CREATE TABLE Person(
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    gender CHAR(1),
    birthdate DATE
);
```

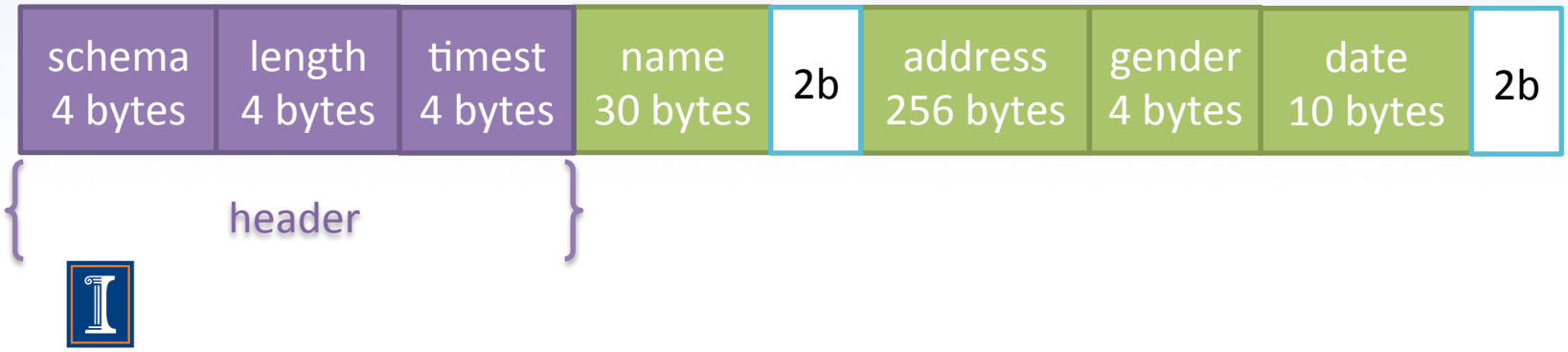
Need more padding…



Fixed Length Records

```
CREATE TABLE Person(
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    gender CHAR(1),
    birthdate DATE
);
```

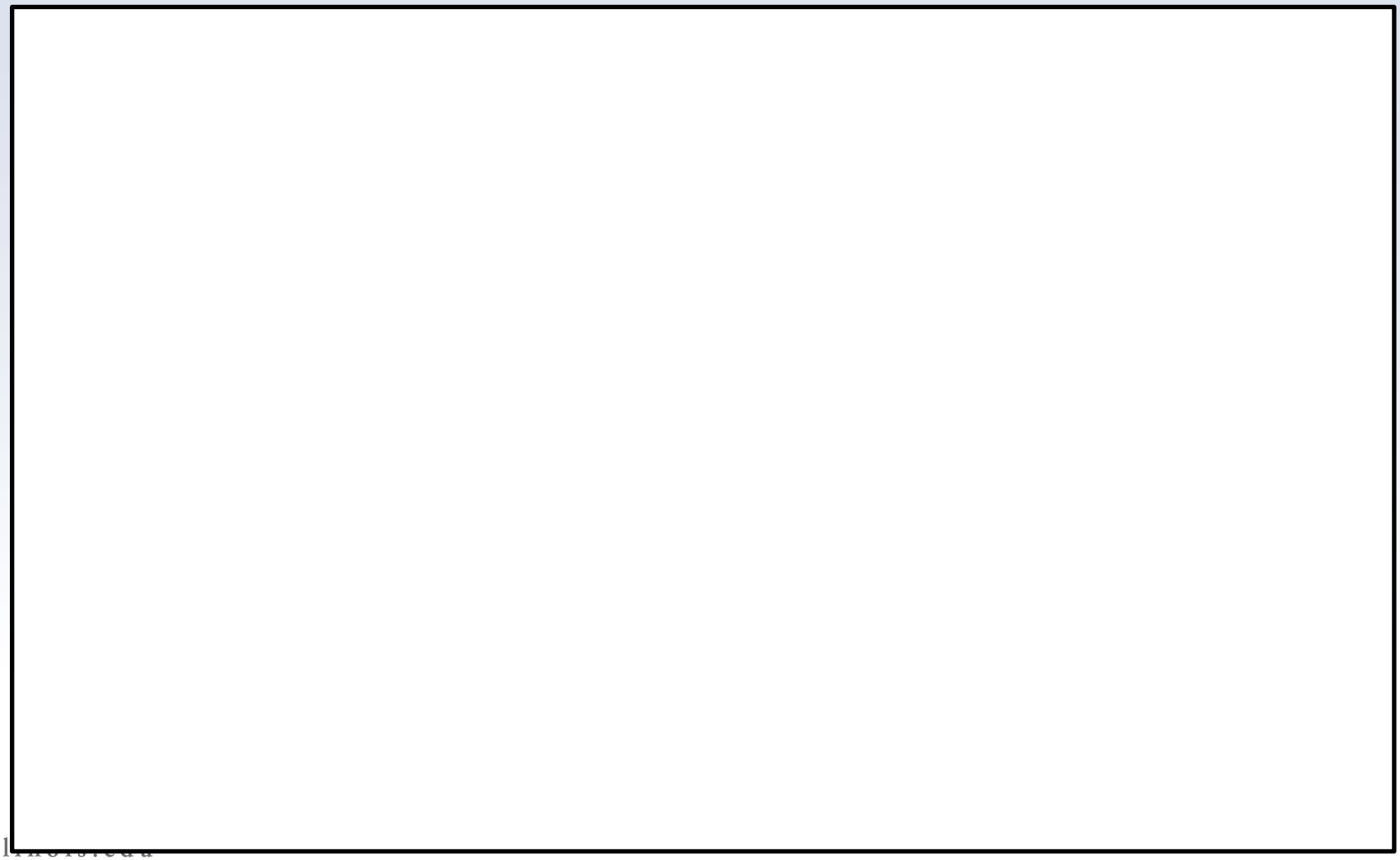
Actually takes 316 bytes



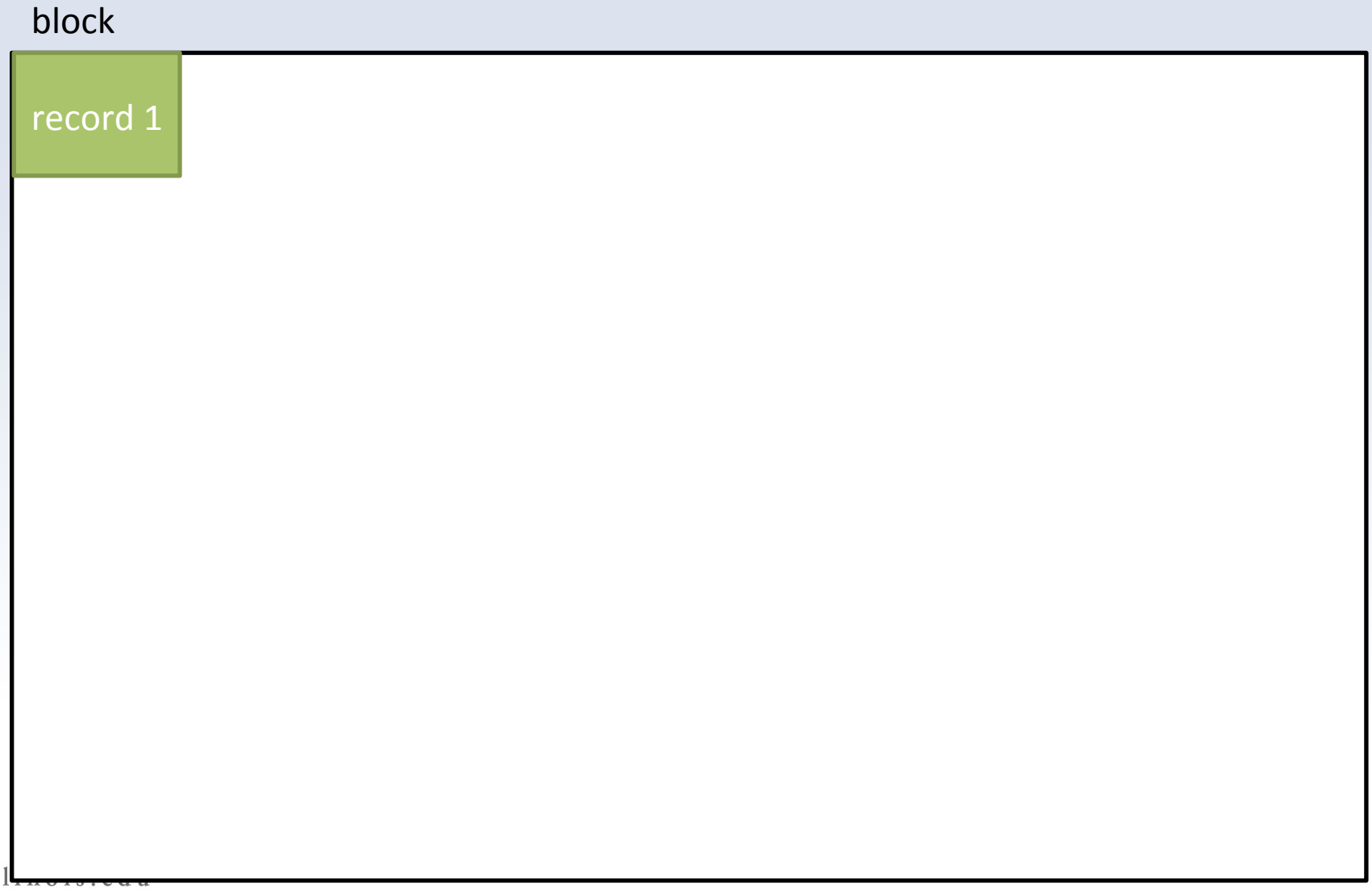
Putting records into blocks

- Now we know how to represent fixed length records (tuples)
- How do we pack a lot them into blocks?

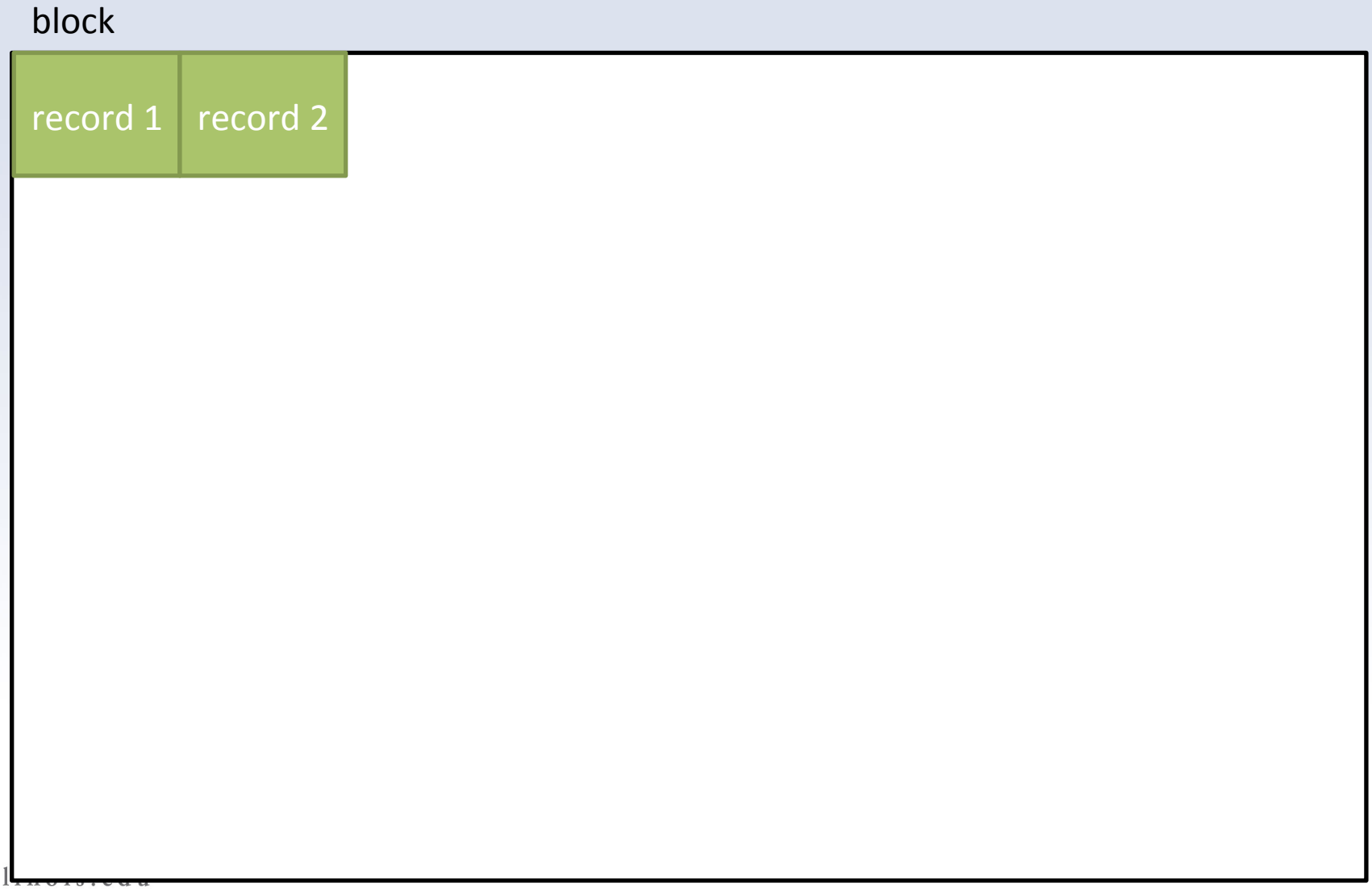
Putting records into blocks



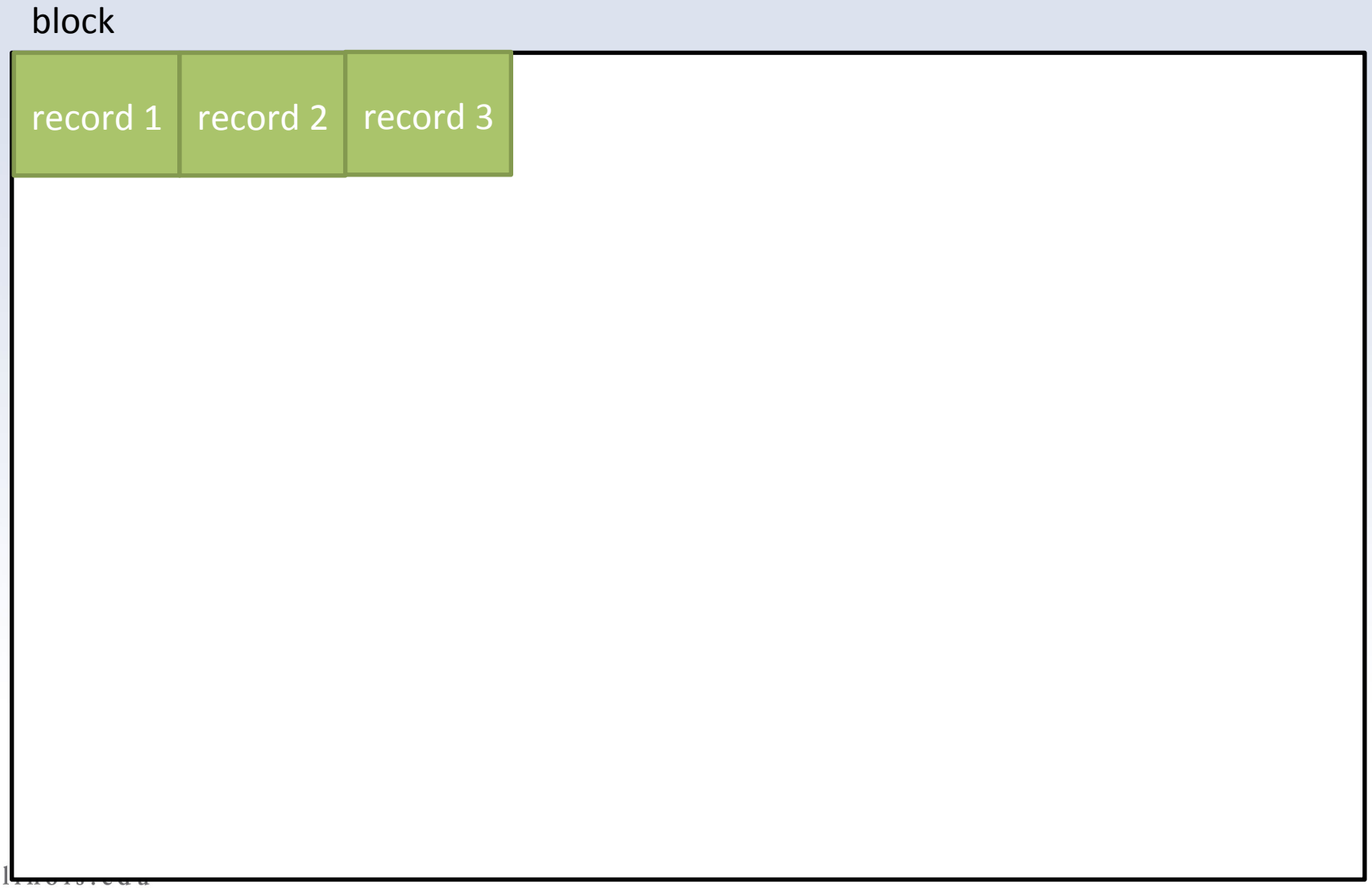
Putting records into blocks



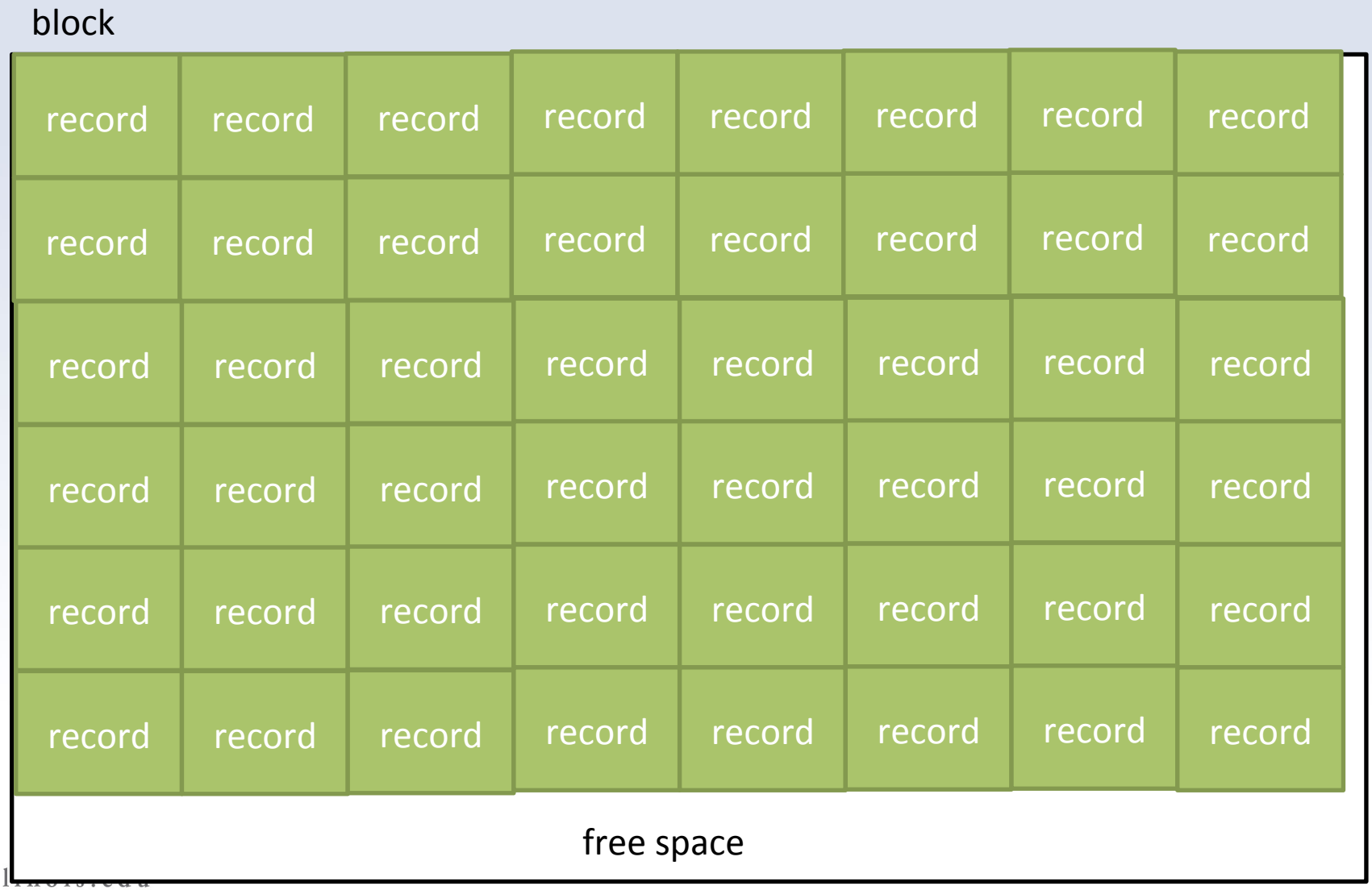
Putting records into blocks



Putting records into blocks



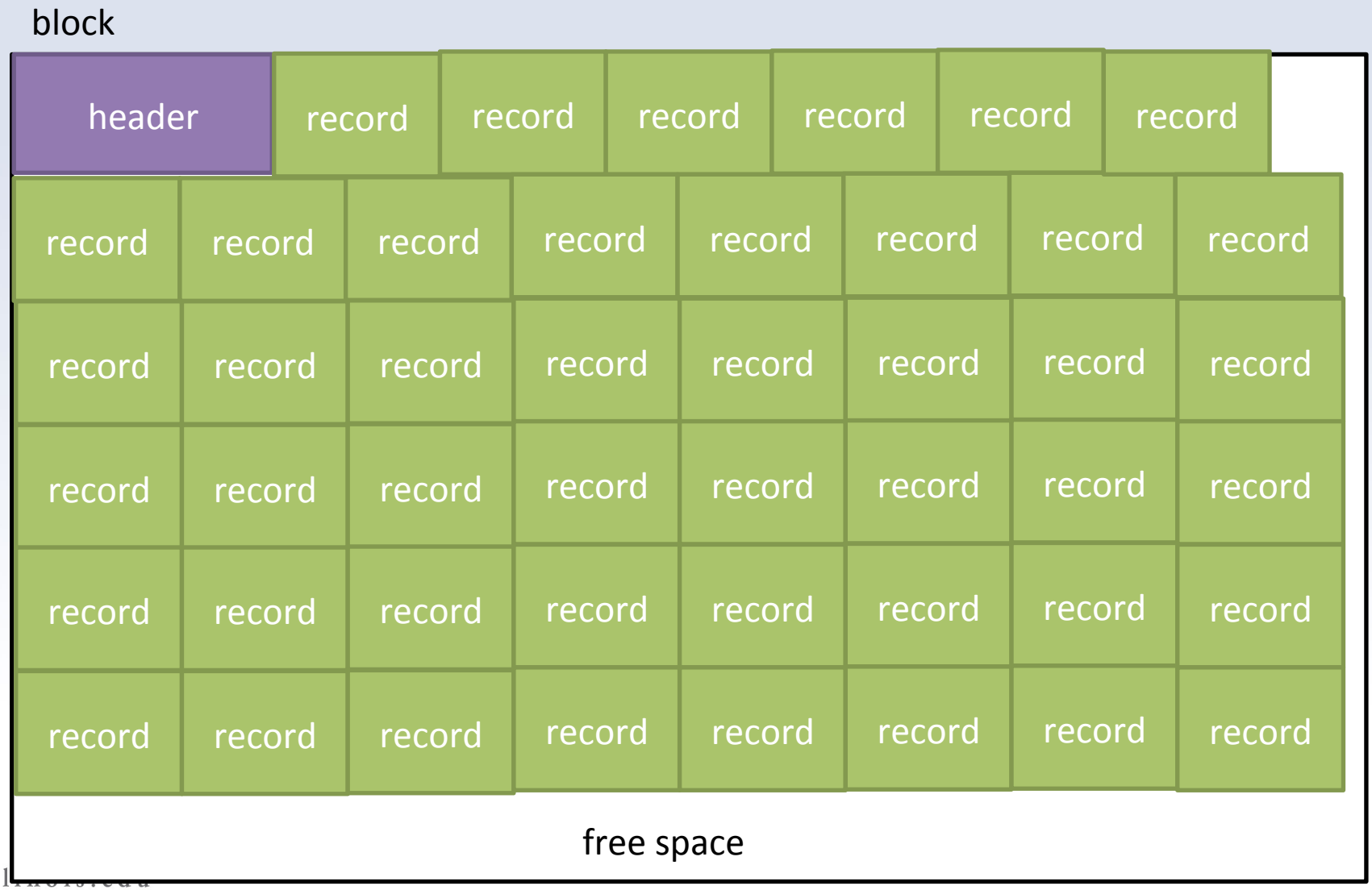
Putting records into blocks



Putting records into blocks

- In reality, we want to have a header:
  - Information about the relation
  - "Directory" of records in the block
  - Links to the "previous" and "next" blocks
  - Last modified timestamp/access metadata

Putting records into blocks



Putting records into blocks

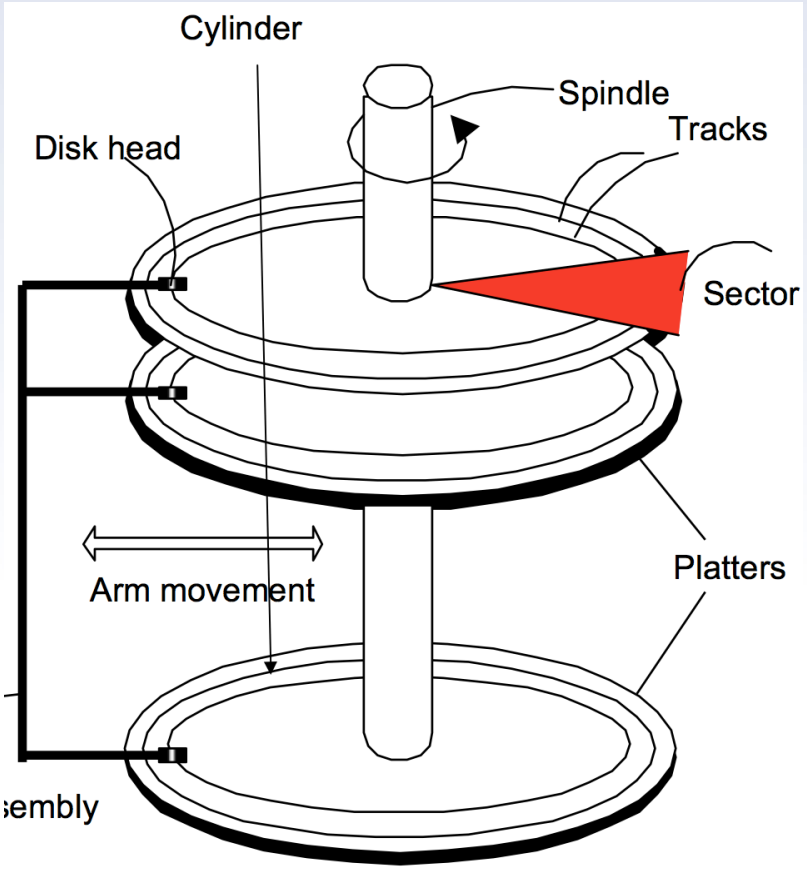
- How many records fit?
  - $\lfloor (blockSize - headerSize)/recordSize \rfloor$
- Example:
  - header=12 bytes
  - block=4,096 bytes
  - record=316 bytes
  - records per block=12

Putting records into blocks

- How many records fit?
  - $\lfloor (blockSize - headerSize)/recordSize \rfloor$
- Example:
  - header=12 bytes
  - block=4,096 bytes
  - record=316 bytes
  - records per block=12
  - free space per block=292 bytes

Addressing records

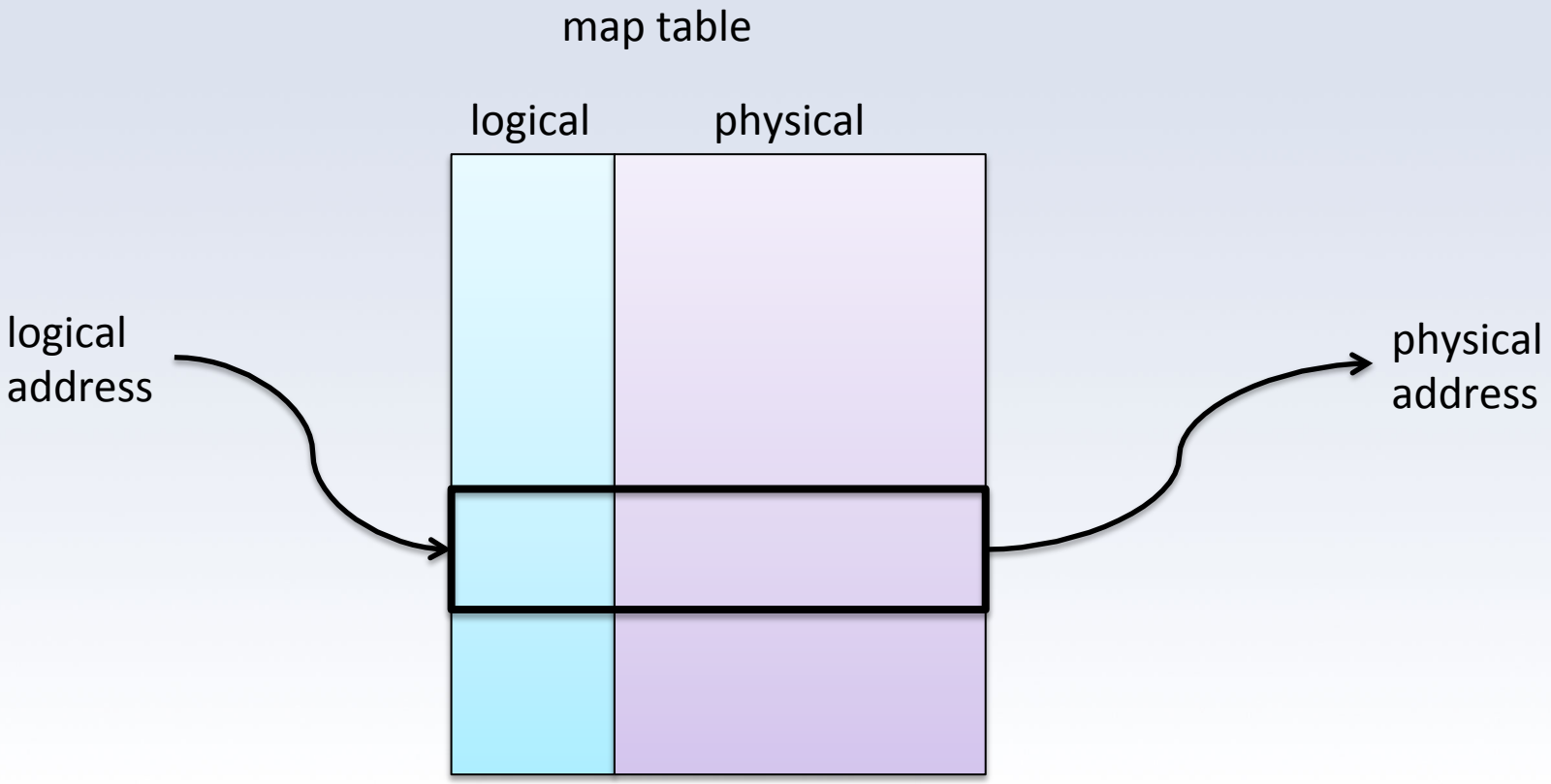
- Each record has a ***physical address***
  1. host machine (if multiple machines involved)
  2. disk number
  3. cylinder number
  4. track number
  5. block number
  6. offset of the record



Addressing records

- Each record has a ***logical address***
  - arbitrary string of bytes
  - fixed length
- A ***map table*** translates the physical address to the logical address

Addressing records



Addressing records

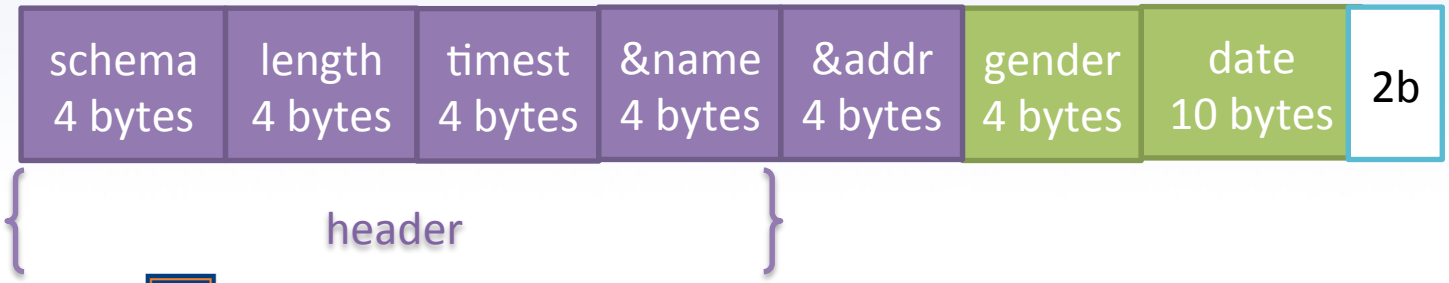
- Map table advantages
  - provides a level of indirection
  - we can move records on the disk, but maintain logical address
  - can have heterogeneous physical addresses
  - if offsets are consistent, we can use ***structured addressing***

Variable length records

- Sometimes, fields have varying lengths
  - e.g. name, address
- Solution:
  - store fixed length fields first
  - store pointers to beginnings of variable length fields in header

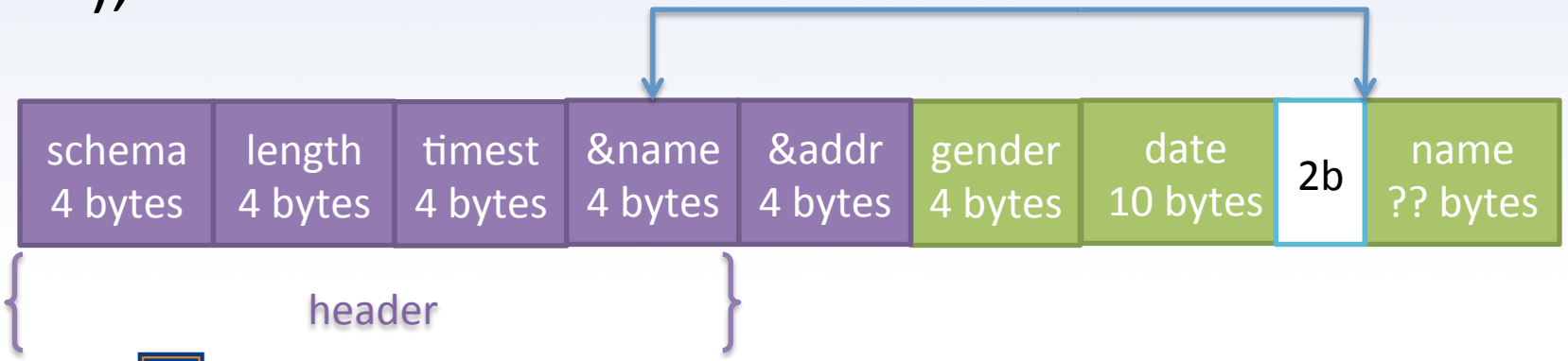
Variable Length Records

```
CREATE TABLE Person(
    name VARCHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    gender CHAR(1),
    birthdate DATE
);
```



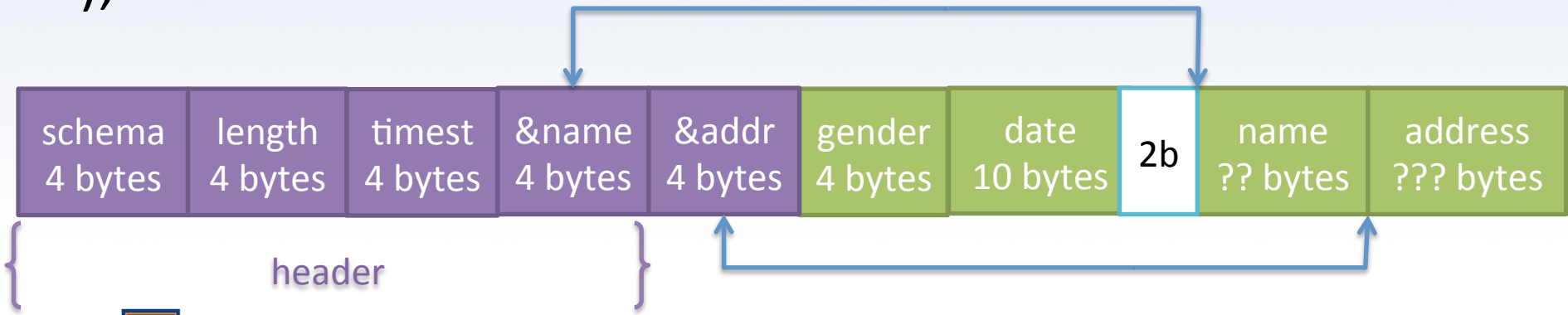
Variable Length Records

```
CREATE TABLE Person(
    name VARCHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    gender CHAR(1),
    birthdate DATE
);
```

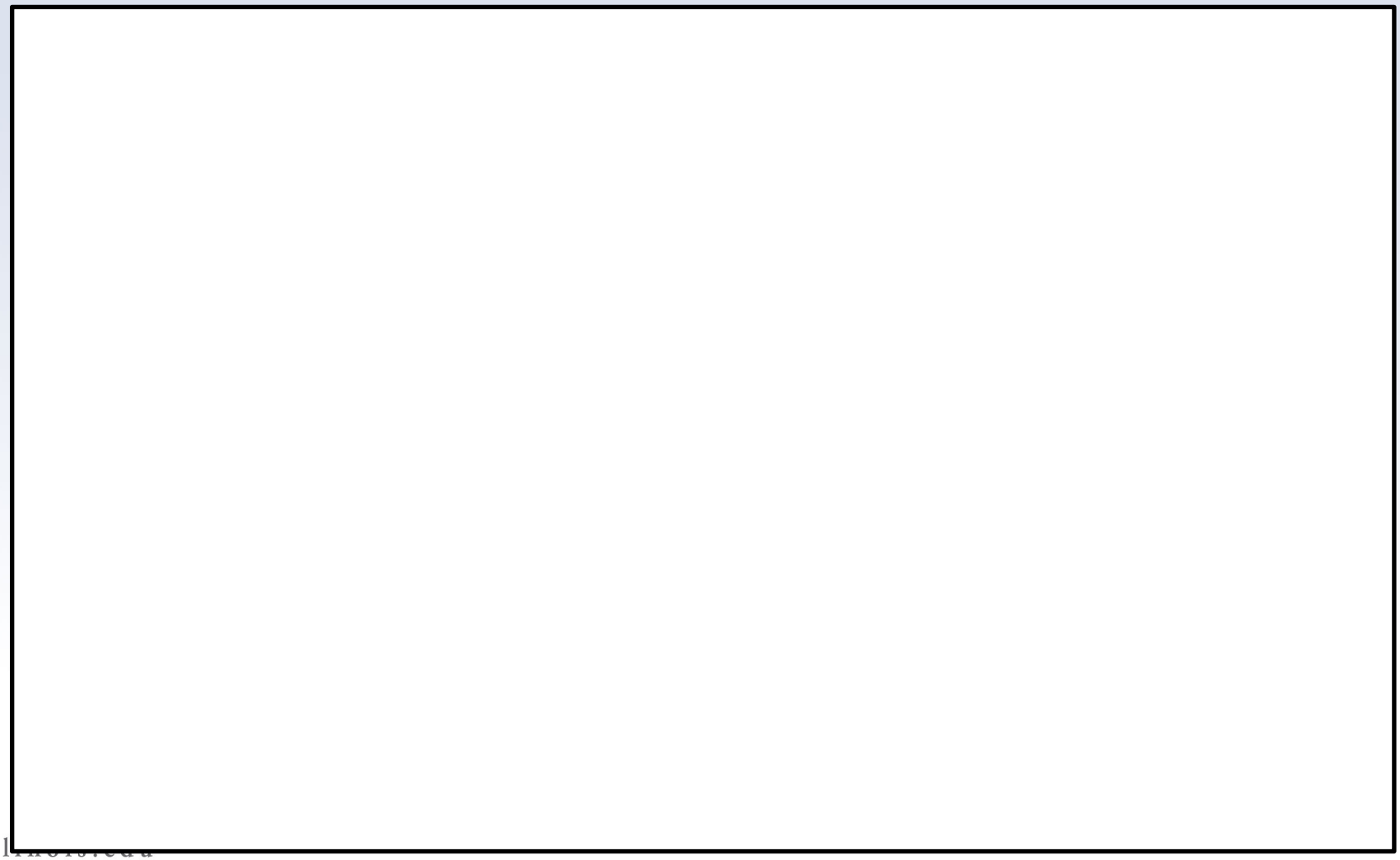


Variable Length Records

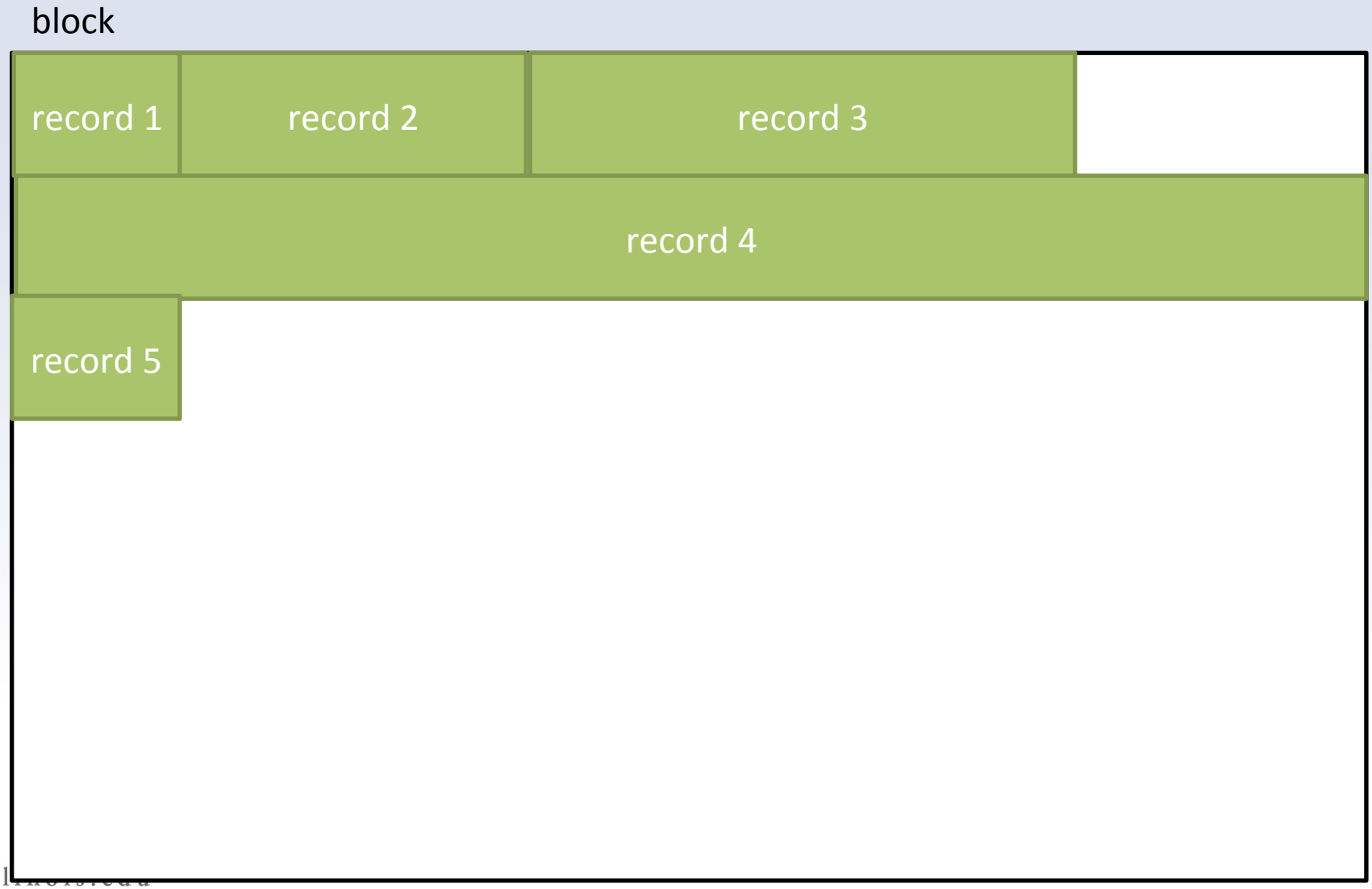
```
CREATE TABLE Person(
    name VARCHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    gender CHAR(1),
    birthdate DATE
);
```



Putting records into blocks



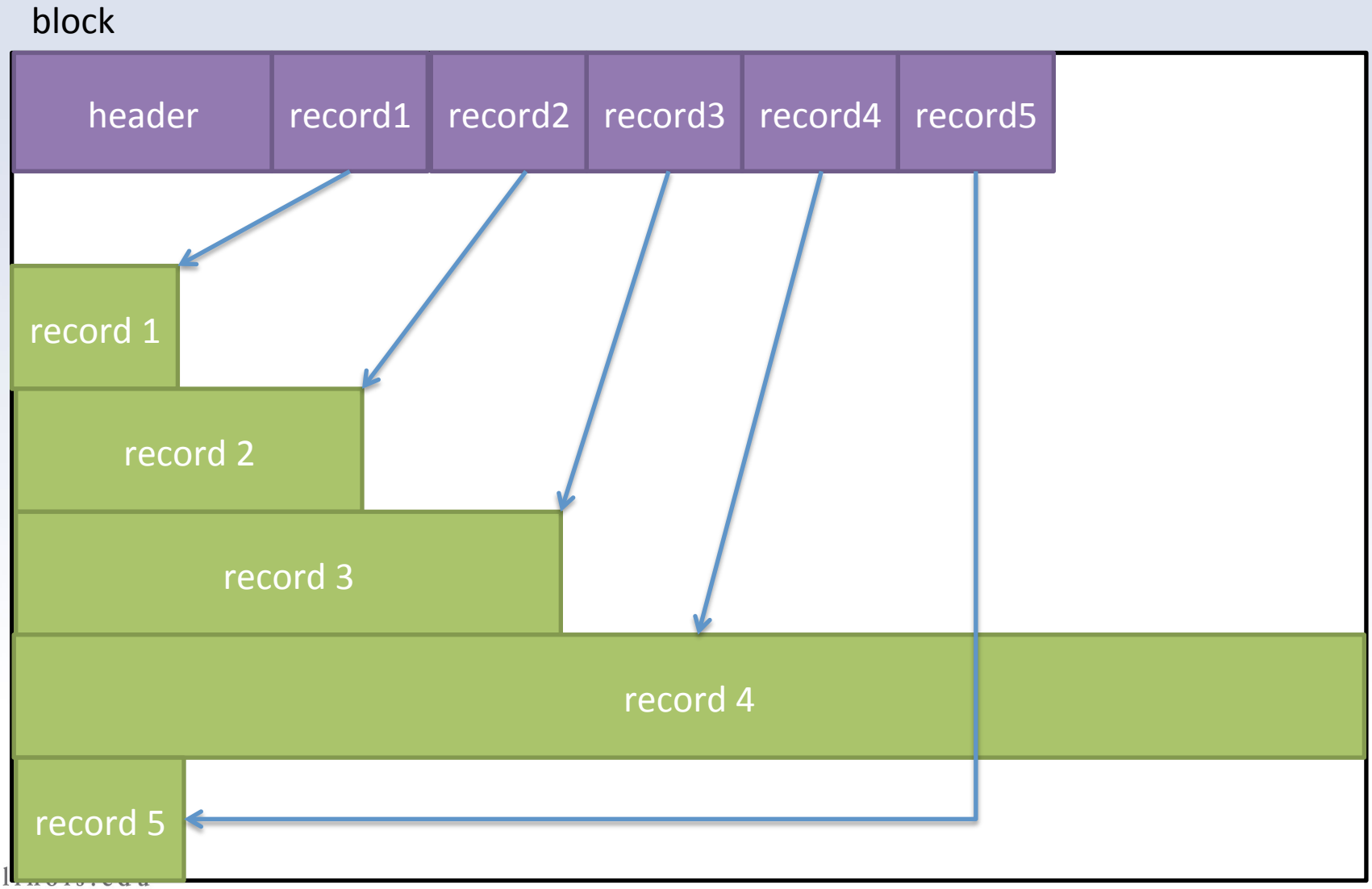
Putting records into blocks



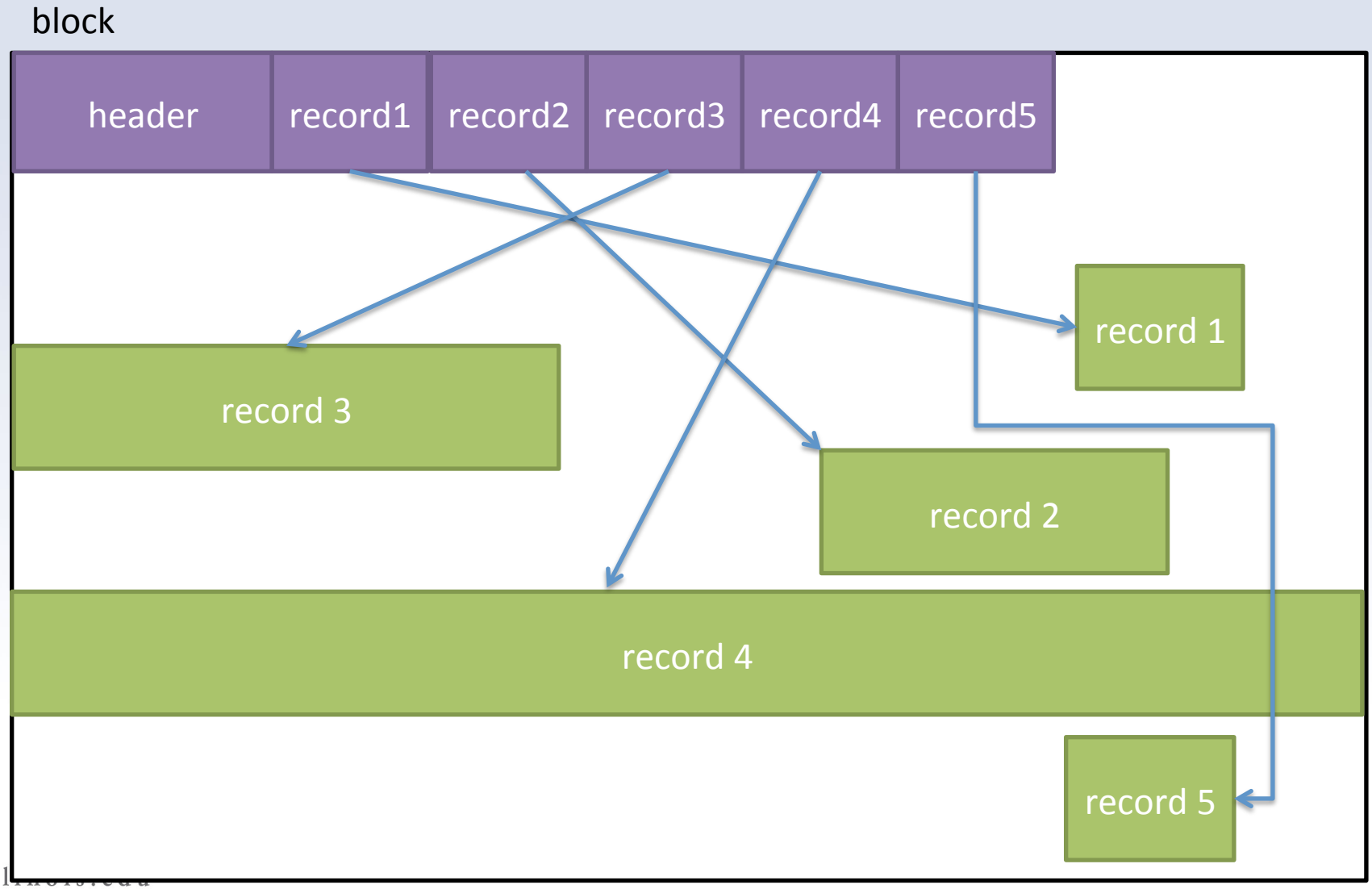
Putting records into blocks

- In reality need
  - header
  - find and skip records quickly
  - add/delete/move records quickly
- Add offset and length index to header

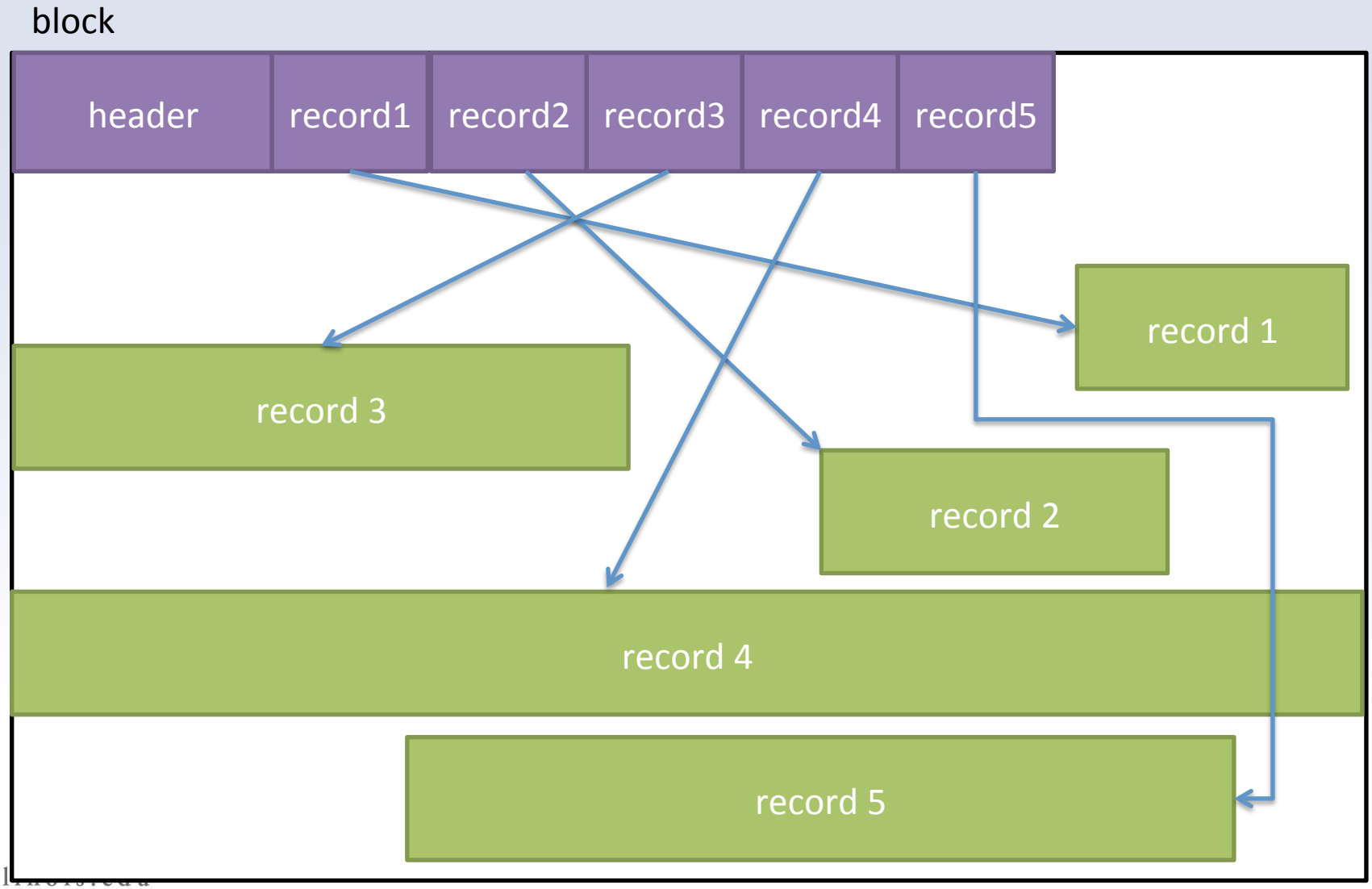
Putting records into blocks



Putting records into blocks



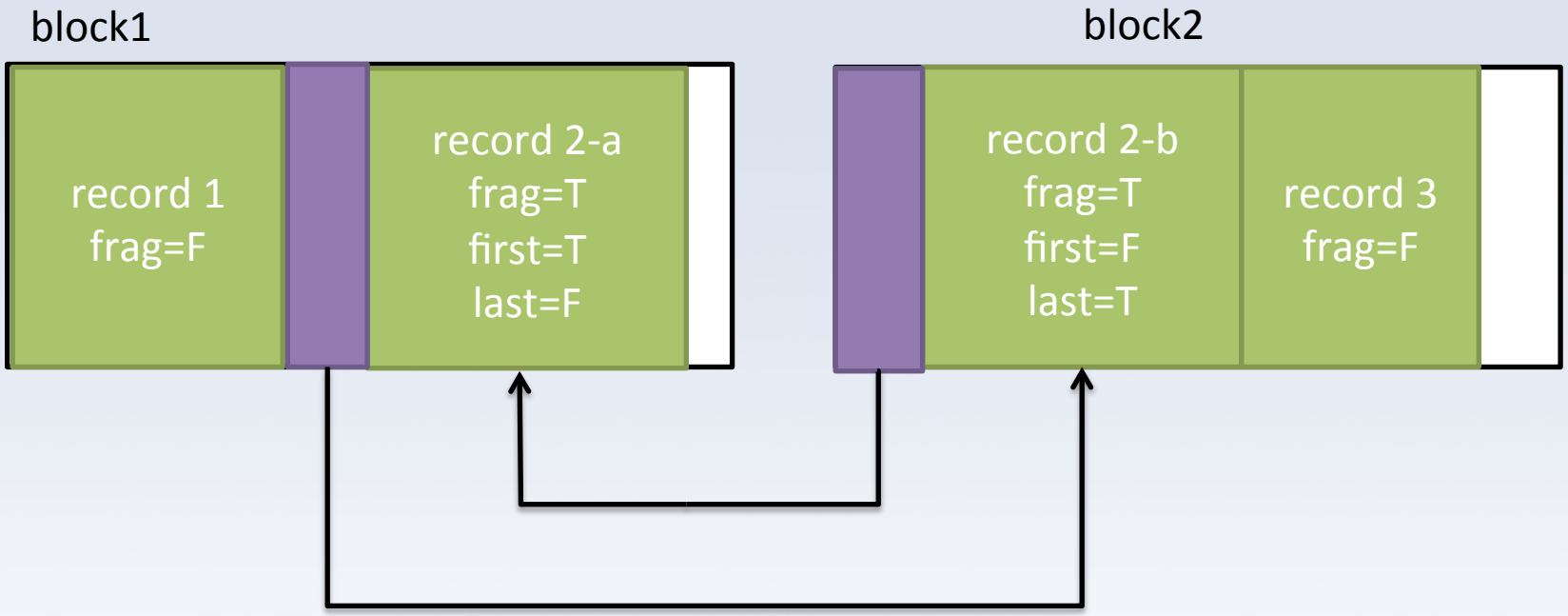
Putting records into blocks



Record fragments

- If a record is too large to fit in a block
  - it ***spans*** multiple blocks
  - we keep a ***record fragment*** in each block
- To handle this:
  - header needs flag indicating fragment status
  - need flags for fragment beginning and end of record
  - fragments need next/previous pointers

Record fragments



(technically, frag, first, and last should be part of **header**)

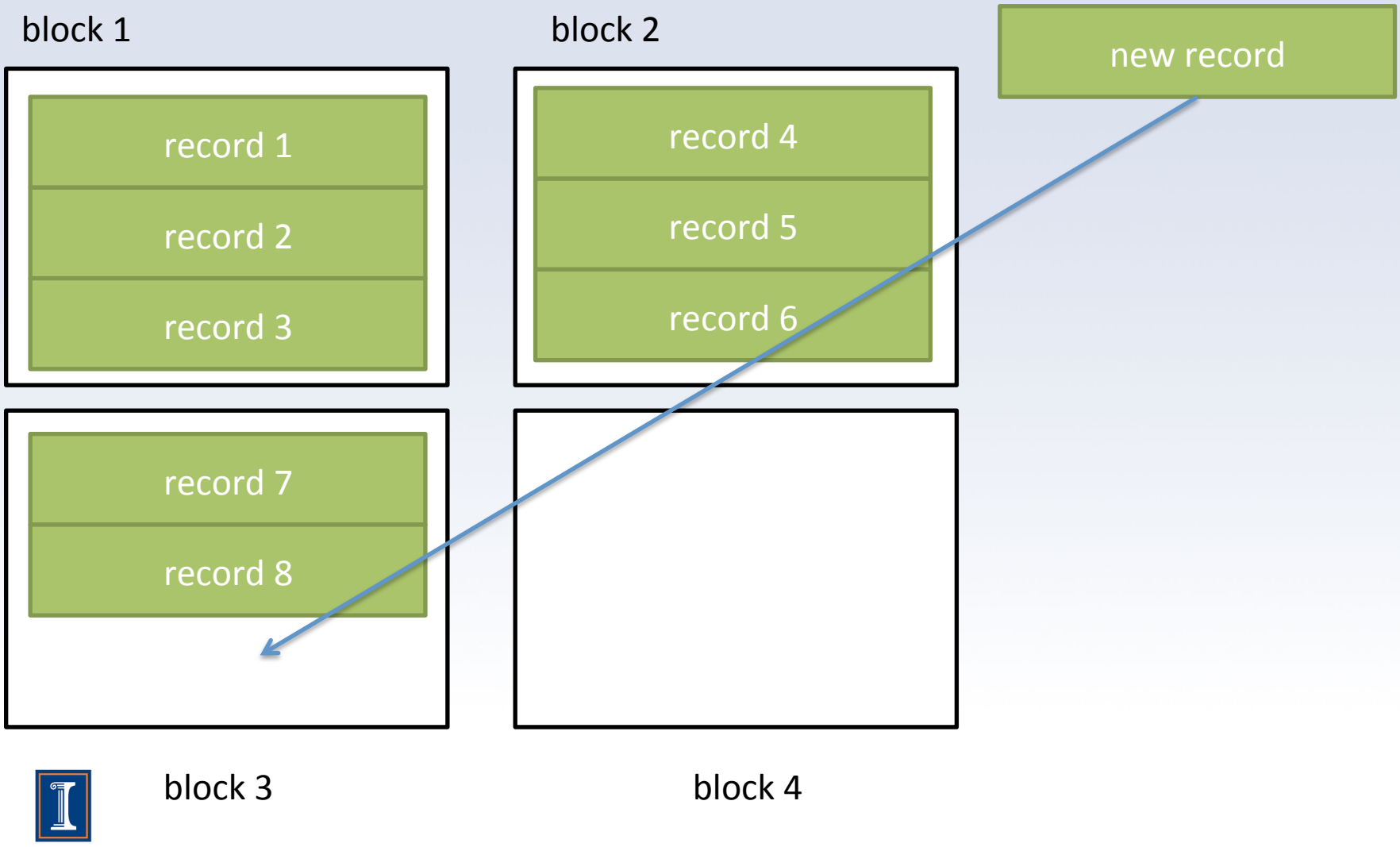
BLOBS

- Binary Large Objects (BLOBS)
  - huge data that isn't really part of record
  - e.g. MPEGs, MP3s, PDFs
  - stored in separate blocks
  - only retrieved when we need them

Insertion

- If the file isn't sorted (e.g. by primary key), just insert the tuple at the end of the last block.

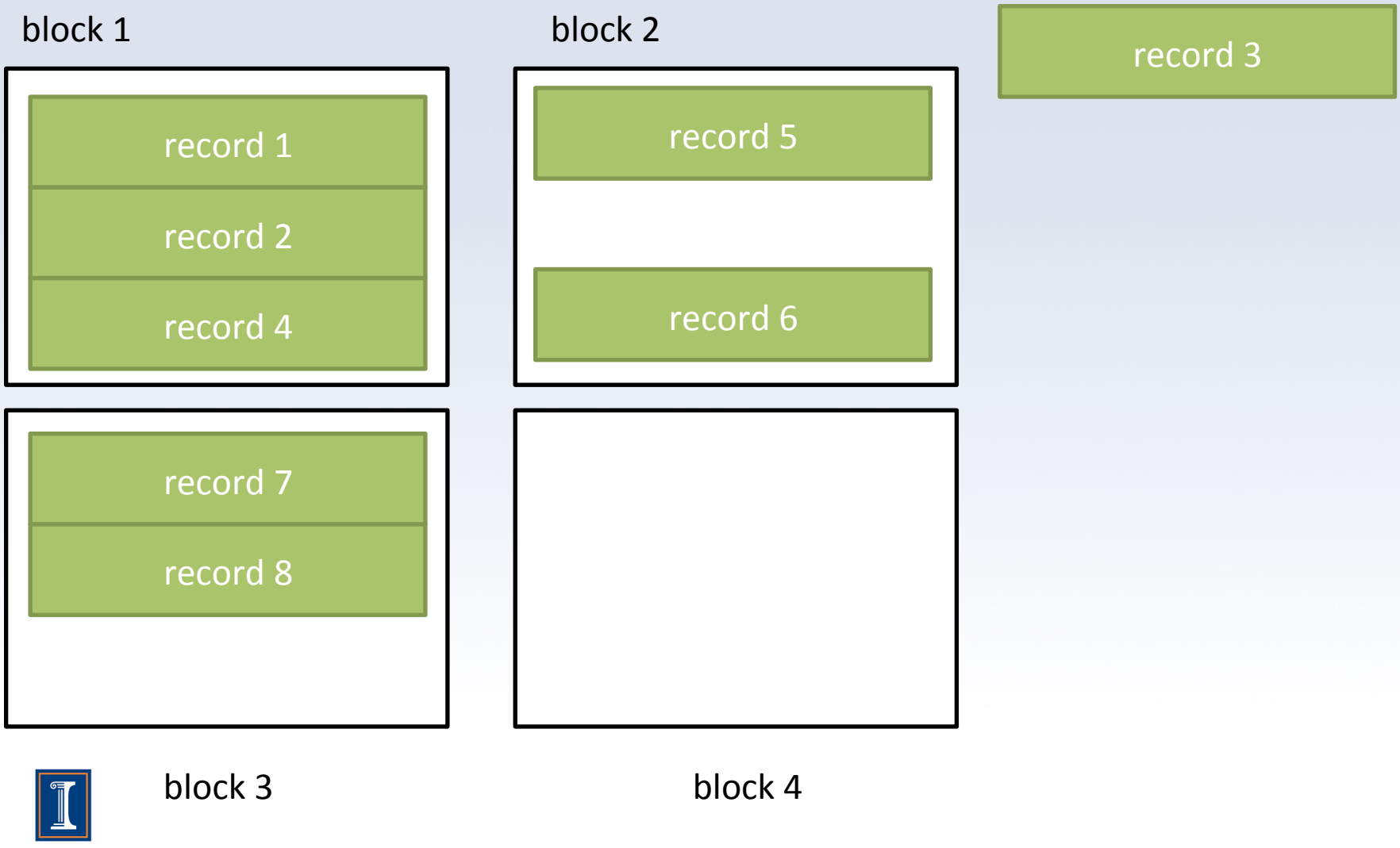
Insertion


block 1
block 2
new record
record 1
record 2
record 3
record 4
record 5
record 6
record 7
record 8
block 3
block 4

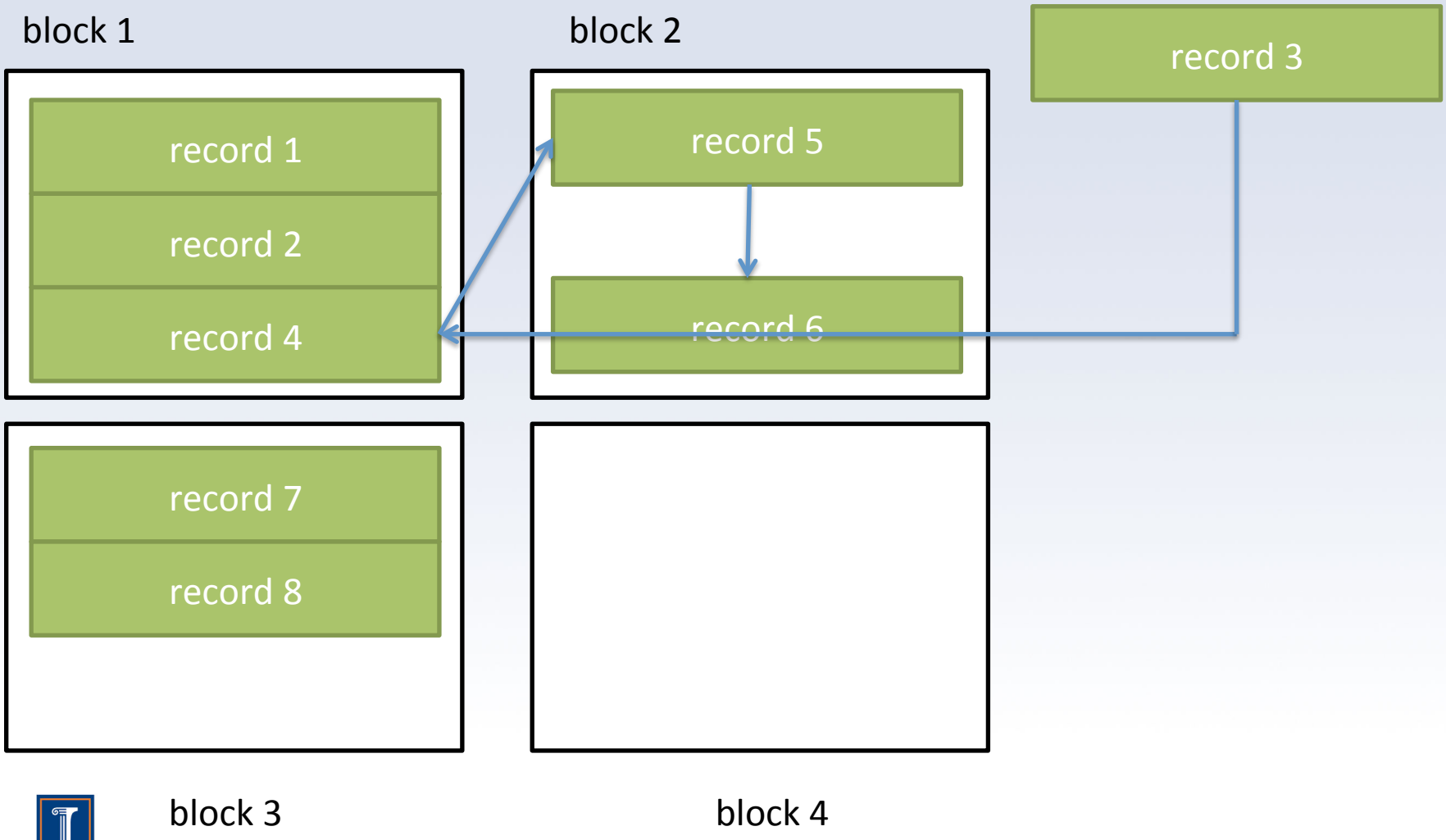
Insertion

- If it is sorted, tuple has to go in a *specific place* in a *specific block*
- What if there is no room in that block?
  - try to rearrange blocks in nearby pages to make room
  - create an overflow page

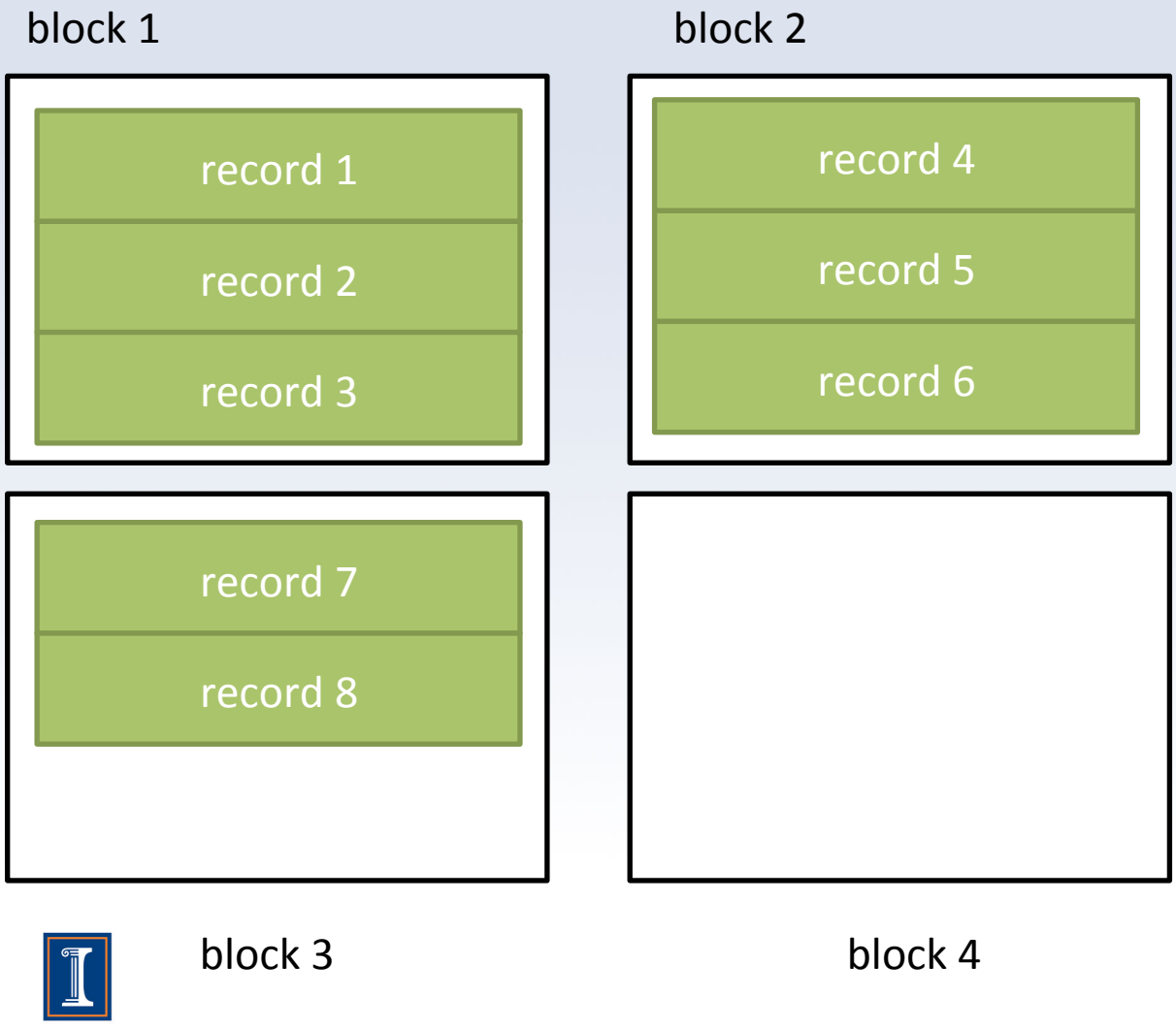
Rearrange


block 1
record 1
record 2
record 4
block 2
record 5
record 6
record 3
record 7
record 8
block 3
block 4

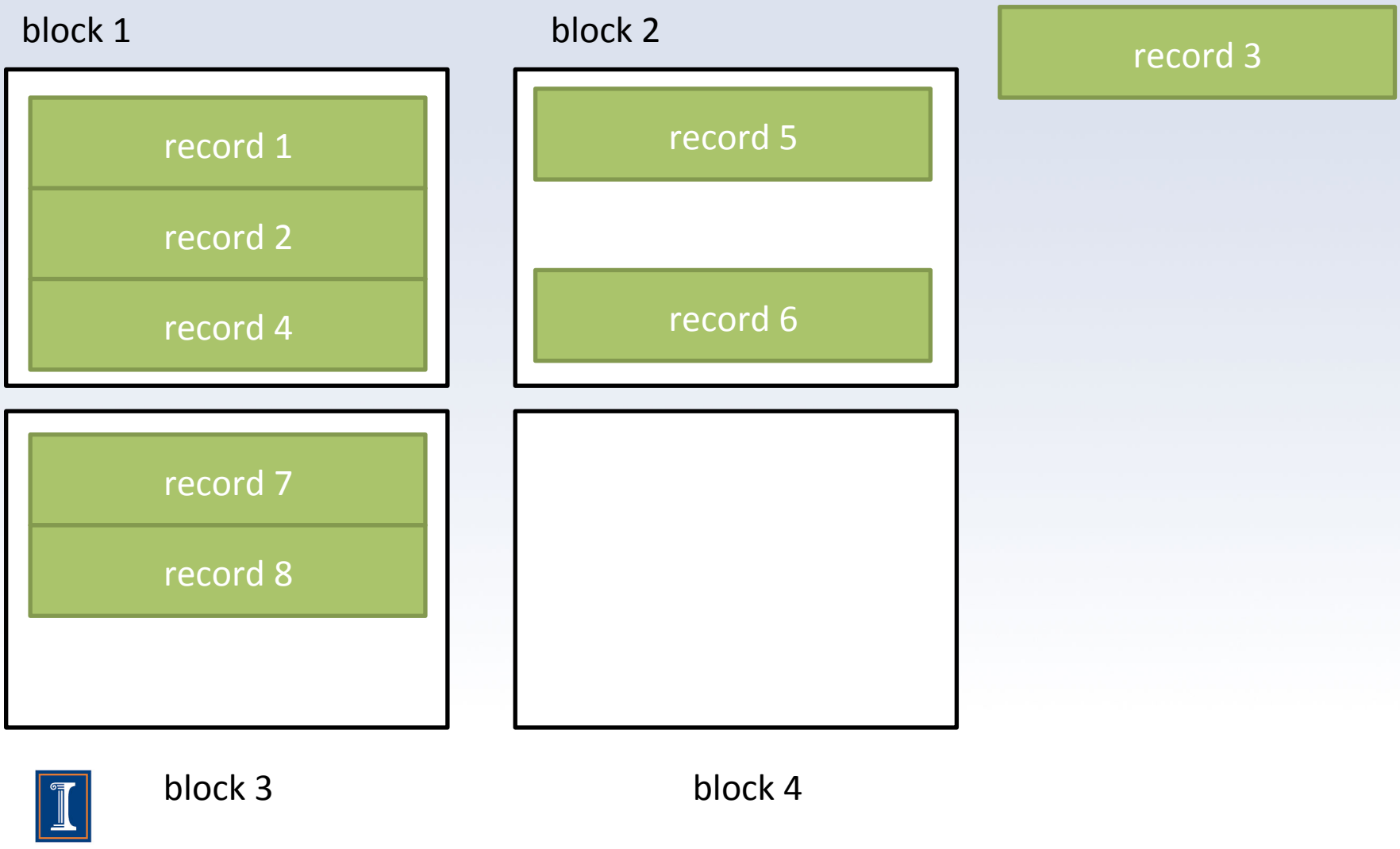
Rearrange


block 1
block 2
record 3
record 1
record 2
record 4
record 5
record 6
record 7
record 8
block 3
block 4

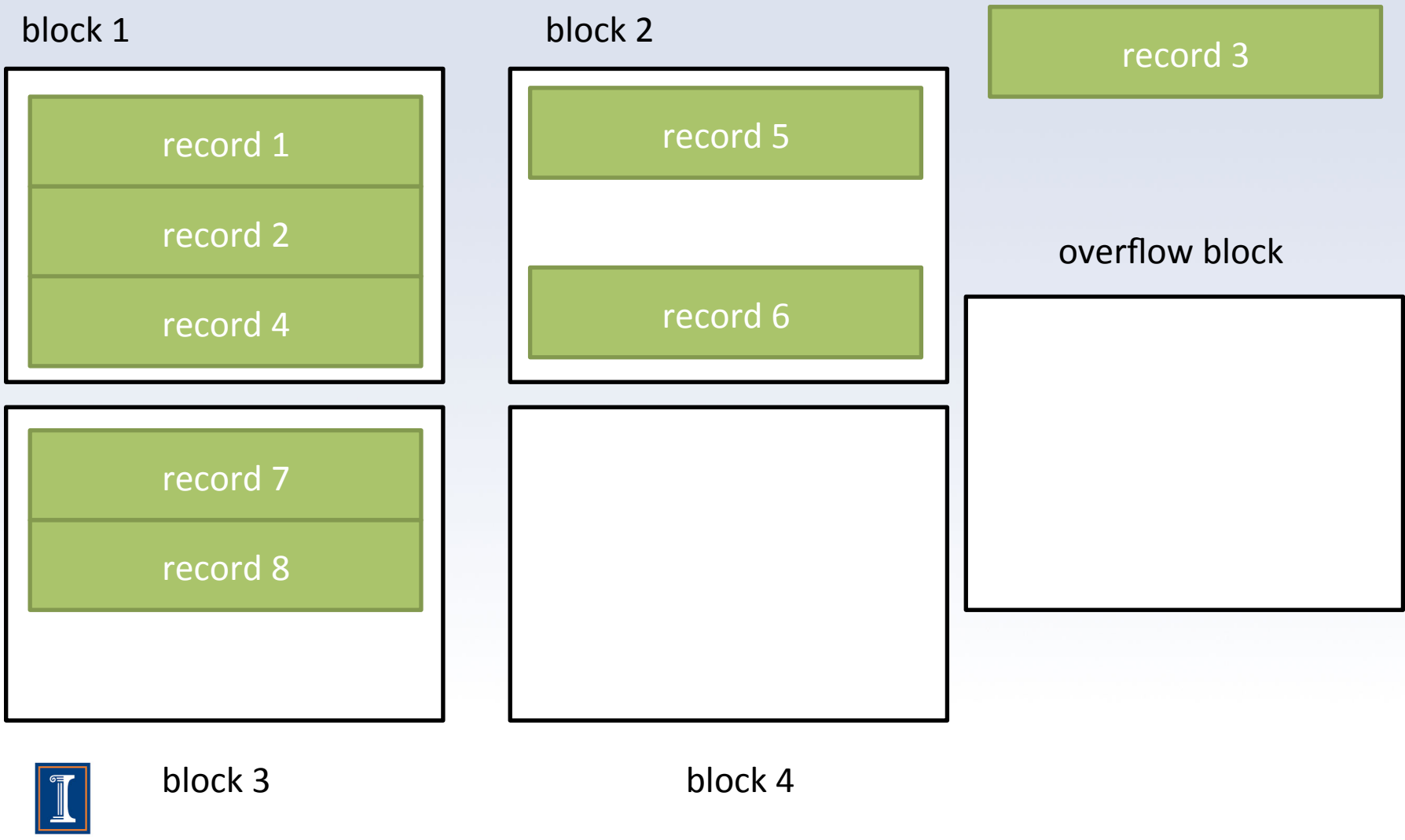
Rearrange


block 1
record 1
record 2
record 3
block 2
record 4
record 5
record 6
record 7
record 8
block 3
block 4

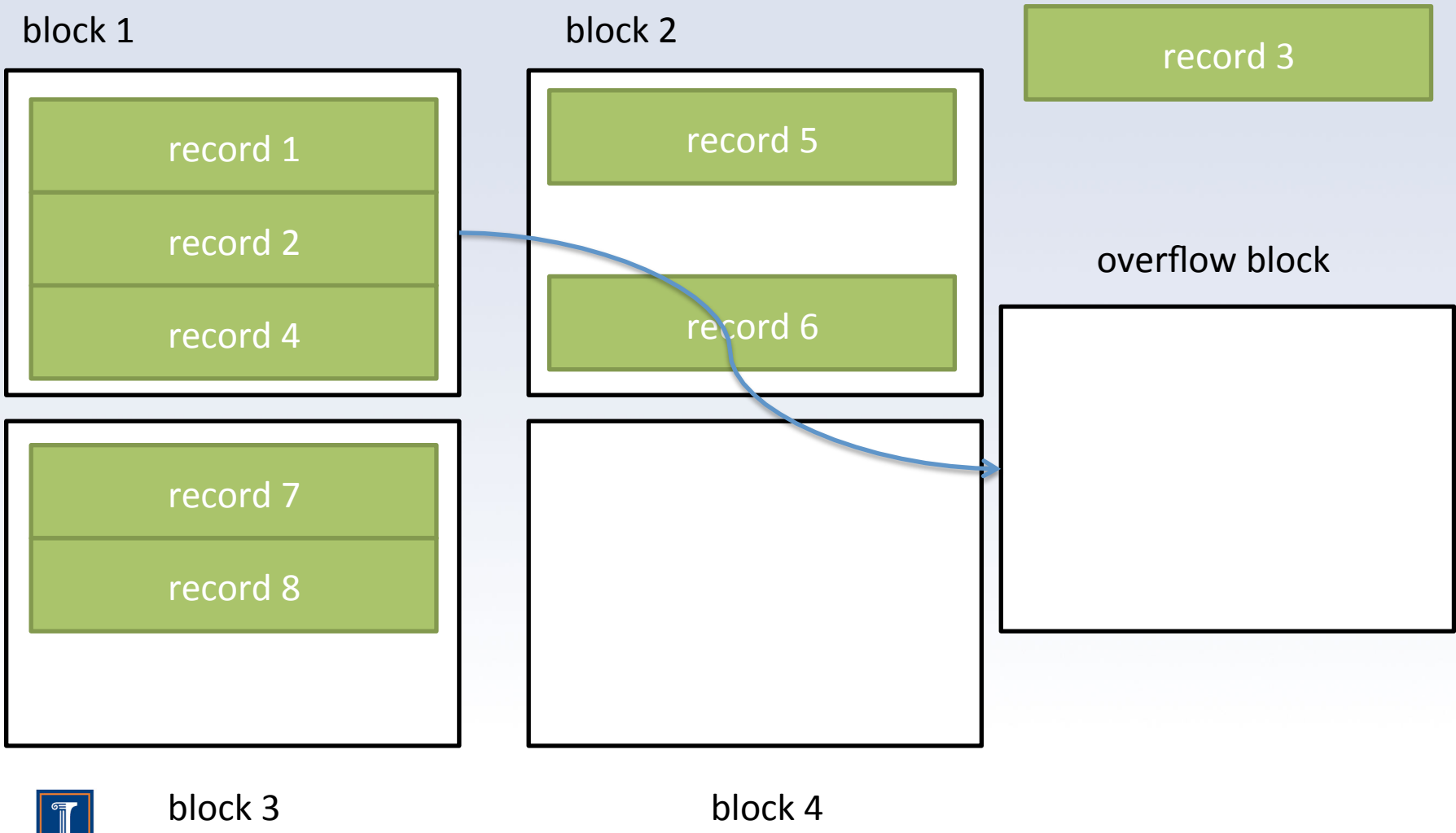
Overflow block



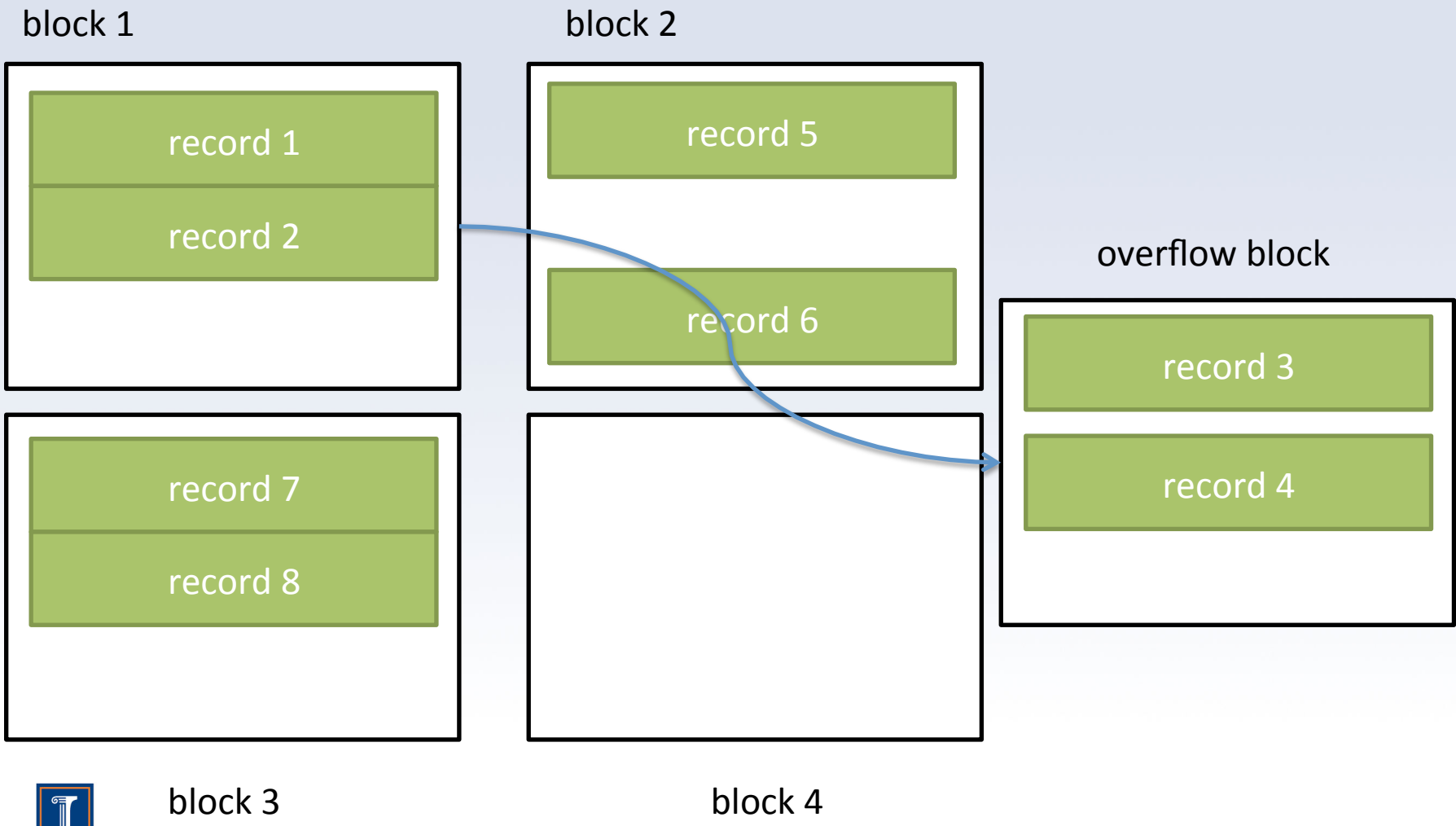
Overflow block



Overflow block



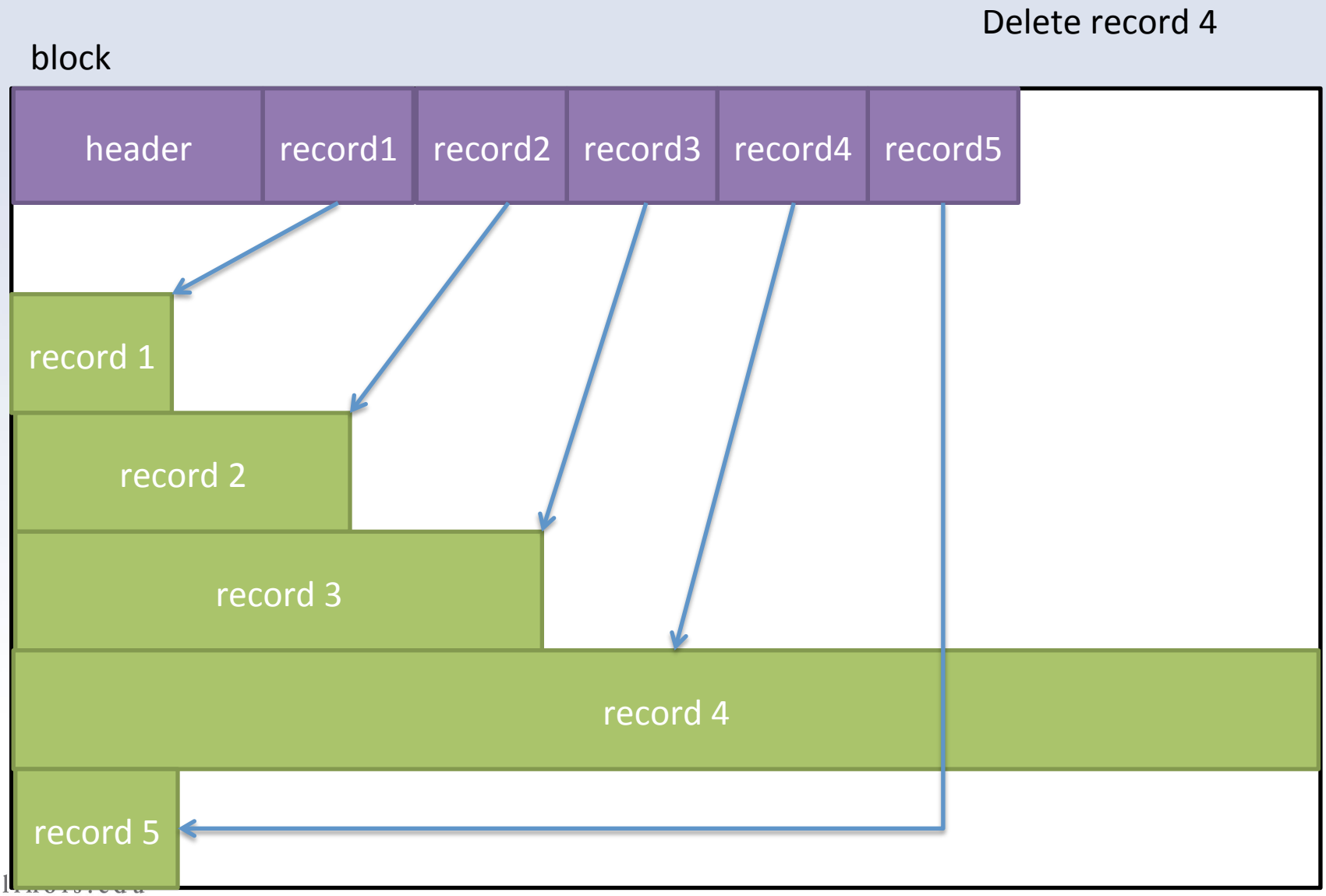
Overflow block



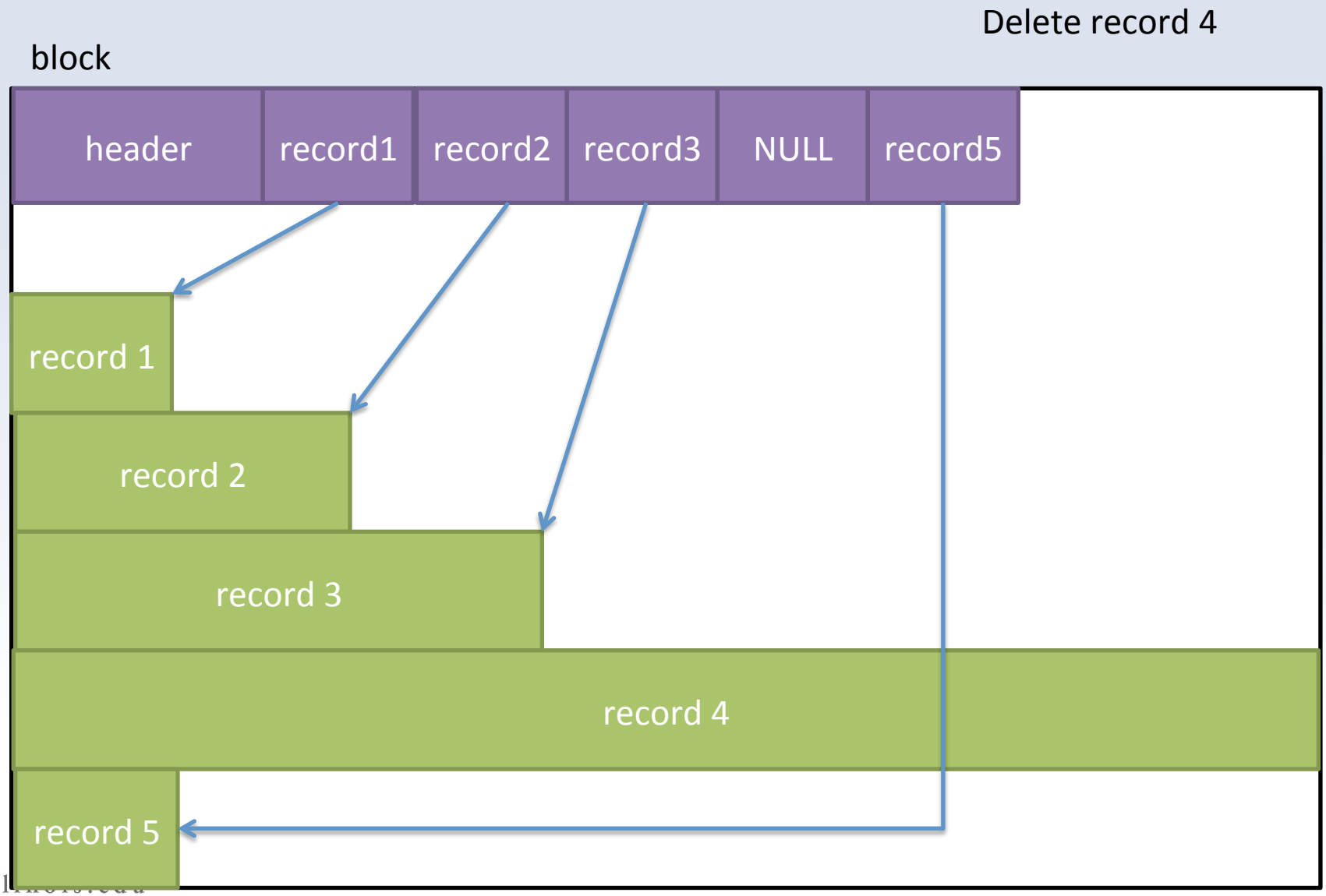
Deletions

- Deletions can be performed by modifying the header
  - update to a NULL pointer
  - can also update the map table to a NULL pointer
- NULL pointers for deleted records are called “tombstones”

Deletion


Delete record 4
block
header
record1
record2
record3
record4
record5
record 1
record 2
record 3
record 4
record 5

Deletion


Delete record 4
block
header
record1
record2
record3
NULL
record5
record 1
record 2
record 3
record 4
record 5