Example: Associative Arrays

- An environment can be expressed as an associative array, e.g.:

```
$myEnv = array(
  ”phptype”   => ”pgsql”,
  ”hostspec” => ”localhost”,
  ”port”      => ”5432”,
  ”database” => ”petersk09”,
  ”username” => ”petersk09”,
  ”password” => ”geheim”);
```

Making a Connection

- With the DB library imported and the array $myEnv available:

```
$myCon = DB::connect($myEnv);
```

Function connect in the DB library

Class is Connection because it is returned by DB::connect()

Executing SQL Statements

- Method query applies to a Connection object
- It takes a string argument and returns a result
  - Could be an error code or the relation returned by a query

Example: Executing a Query

- Find all the bars that sell a beer given by the variable $beer

```
$beer = 'Od.Cl.';
$result = $myCon->query(
  "SELECT bar FROM Sells" .
  "WHERE beer = '$beer';");
```

Method application

Concatenation in PHP

Remember this variable is replaced by its value.

Cursors in PHP

- The result of a query *is* the tuples returned
- Method fetchRow applies to the result and returns the next tuple, or FALSE if there is none

Example: Cursors

```
while ($bar = $result->fetchRow())
{
  // do something with $bar
}
```

Example: Tuple Cursors

```
$bar = "C.Ch.";
$menu = $myCon->query(
  "SELECT beer, price FROM Sells
  WHERE bar = '$bar';");
while ($bp = $result->fetchRow())
{
  print $bp[0] . " for " . $bp[1];
}
```

An Aside: SQL Injection

- SQL queries are often constructed by programs
- These queries may take constants from user input
- Careless code can allow rather unexpected queries to be constructed and executed

Example: SQL Injection

- Relation Accounts(name, passwd, acct)
- Web interface: get name and password from user, store in strings *n* and *p*, issue query, display account number

```
$result = $myCon->query(
"SELECT acct FROM Accounts WHERE
  name = '$n' AND passwd = '$p';");
```

User (Who Is Not Bill Gates) Types

Comment in PostgreSQL

Name: `gates' --`

Password: `who cares?`

Your account number is 1234-567

The Query Executed

```
SELECT acct FROM Accounts
WHERE name = 'gates'  --' AND
 passwd = 'who cares?'
```

All treated as a comment

Summary 8

More things you should know:

- Stored Procedures, PL/pgsql
- Declarations, Statements, Loops,
- Cursors, Tuple Variables
- Three-Tier Approach, JDBC, PHP/DB

Database Implementation

Database Implementation

Isn‘t implementing a database system easy?

- Store relations
- Parse statements
- Print results
- Change relations

Introducing the

DanDB 3000

Database Management System

- The latest from DanLabs
- Incorporates latest relational technology
- Linux compatible

DanDB 3000 Implementation Details

- Relations stored in files (ASCII)
- Relation R is in /var/db/R
- Example:

```
Peter # Erd.We.
Lars  # Od.Cl.
  :
```

DanDB 3000 Implementation Details

- Directory file (ASCII) in /var/db/directory
- For relation R(A,B) with A of type VARCHAR(n) and B of type integer:
  R # A # STR # B # INT
- Example:

```
Favorite # drinker # STR # beer # STR
Sells # bar # STR # beer # STR # ...
   :
```

DanDB 3000
Sample Sessions

```
% dandbsql
   Welcome to DanDB 3000!
>
  :
  :

> quit
%
```

DanDB 3000
Sample Sessions

```
> SELECT *
  FROM Favorite;

  drinker # beer
  ##################
  Peter   # Erd.We.
  Lars    # Od.Cl.
  (2 rows)

>
```

DanDB 3000
Sample Sessions

```
> SELECT drinker AS snob
  FROM Favorite, Sells
  WHERE Favorite.beer = Sells.beer
    AND price > 25;

  snob
  ######
  Peter
  (1 rows)

>
```

DanDB 3000
Sample Sessions

```
> CREATE TABLE expensive (bar TEXT);
> INSERT INTO expensive (SELECT bar
  FROM Sells
  WHERE price > 25);
>
```

Create table with expensive bars

DanDB 3000 Implementation Details

- To execute "**`SELECT * FROM`** ***`R`*** **`WHERE`** ***`condition`***":

1. Read /var/db/dictionary, find line starting with "R #"
2. Display rest of line
3. Read /var/db/R file, for each line:
   a. Check condition
   b. If OK, display line

DanDB 3000 Implementation Details

- To execute "**`CREATE TABLE`** ***`S`*** **`(A1 t1, A2 t2);`**":
  1. Map t1 and t2 to internal types T1 and T2
  2. Append new line "**`S # A1 # T1 # A2 # T2`**" to /var/db/directory

- To execute "**`INSERT INTO`** ***`S`*** **`(SELECT * FROM R WHERE`** ***`condition`*** **`);`**":
  1. Process select as before
  2. Instead of displaying, append lines to file /var/db/S

DanDB 3000 Implementation Details

- To execute "**SELECT *A*,*B* FROM *R*,*S* WHERE *condition;***":
  1. Read /var/db/dictionary to get schema for R and S
  2. Read /var/db/R file, for each line:
     a. Read /var/db/S file, for each line:
        i. Create join tuple
        ii. Check condition
        iii. Display if OK

DanDB 3000 Problems

- Tuple layout on disk
  - Change string from 'Od.Cl.' to 'Odense Classic' and we have to rewrite file
  - ASCII storage is expensive
  - Deletions are expensive
- Search expensive – no indexes!
  - Cannot find tuple with given key quickly
  - Always have to read full relation

DanDB 3000 Problems

- Brute force query processing
  - Example:

```
SELECT * FROM R,S WHERE R.A=S.A
AND S.B > 1000;
```

  - Do select first?
  - Natural join more efficient?
- No concurrency control

DanDB 3000 Problems

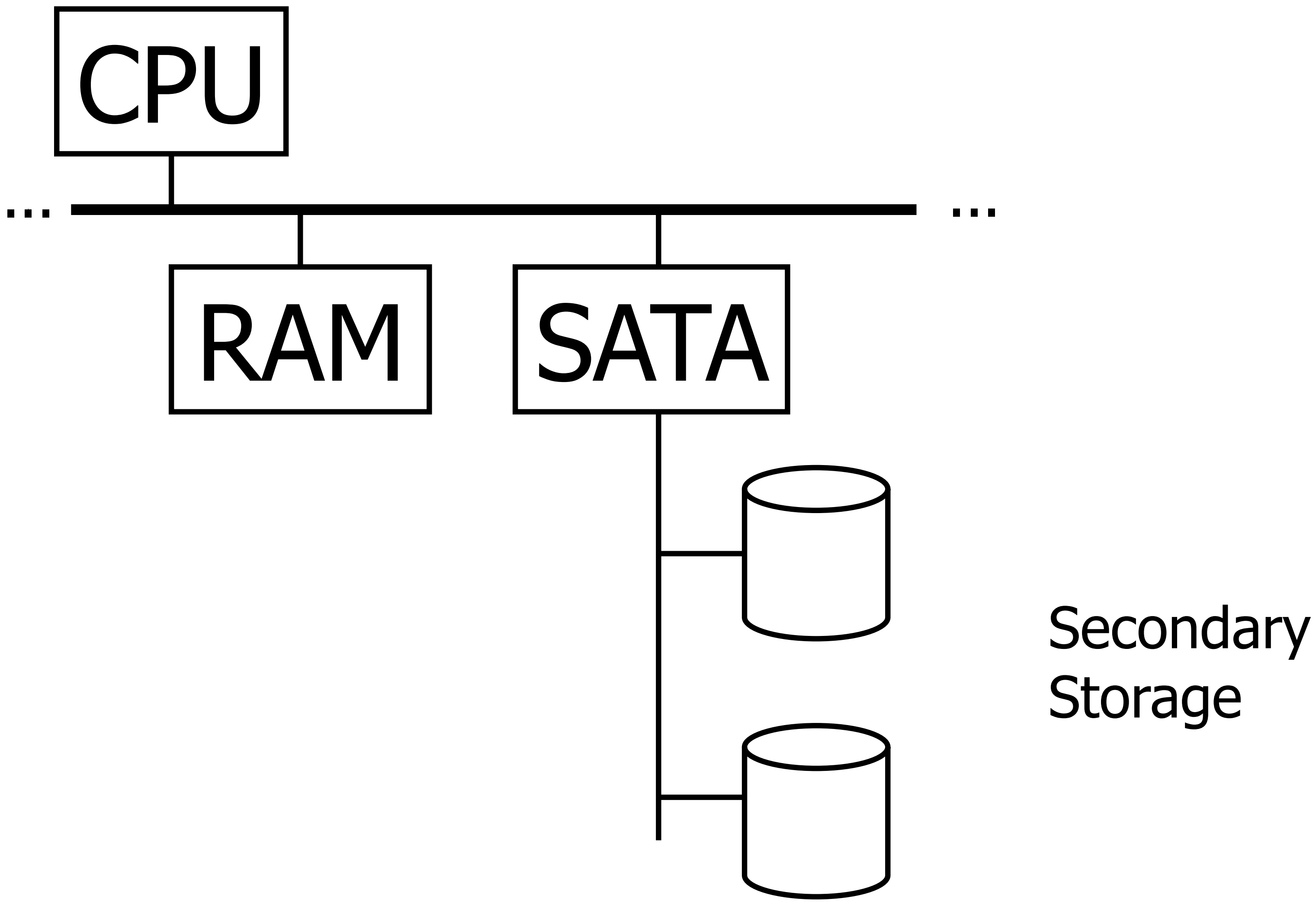
- No reliability
  - Can lose data
  - Can leave operations half done
- No security
  - File system insecure
  - File system security is too coarse
- No application program interface (API)
  - How to access the data from a real program?

DanDB 3000 Problems

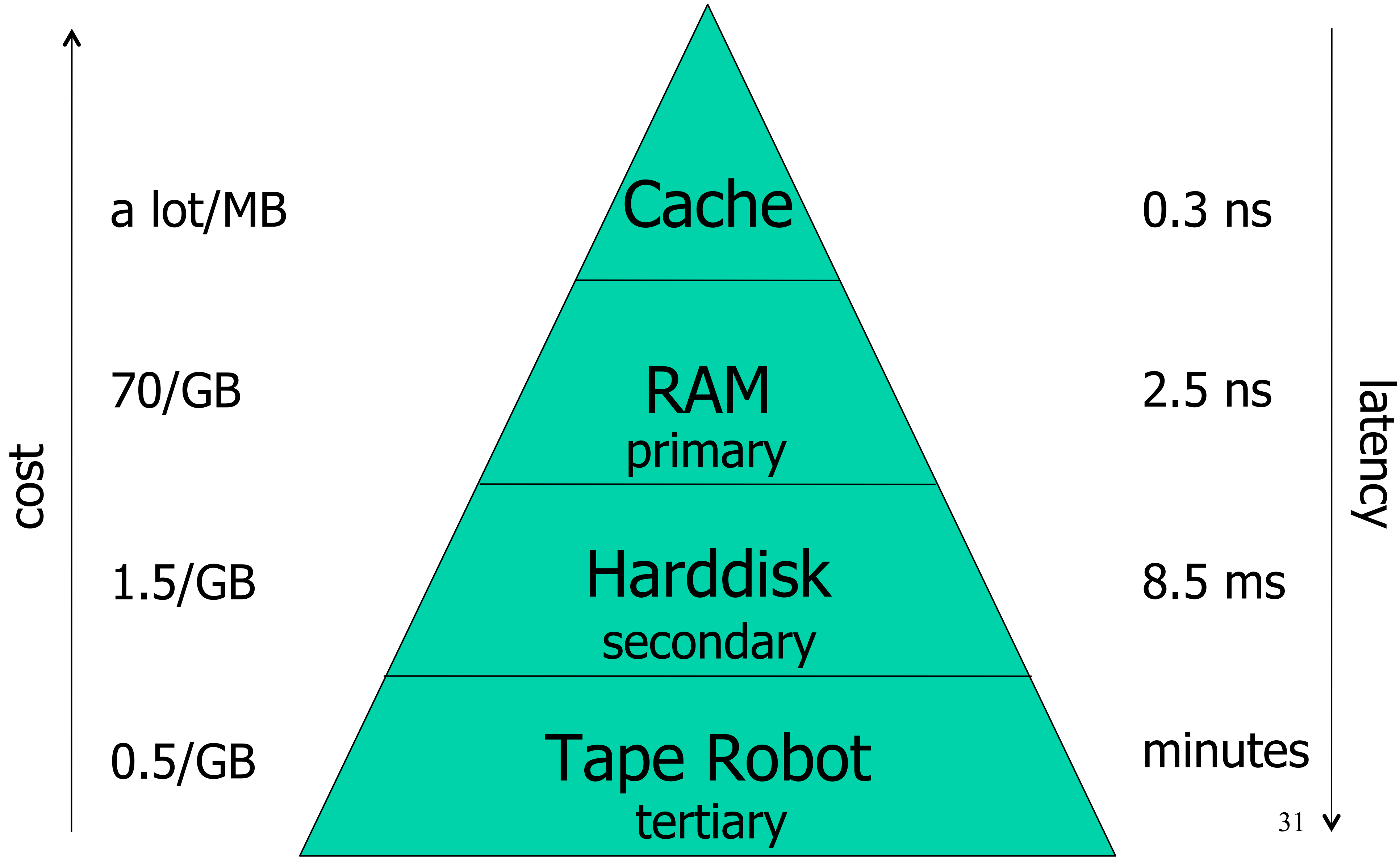
- Cannot interact with other DBMSs
  - Very limited support of SQL
- No constraint handling etc.
- No administration utilities, no web frontend, no graphical user interface, ...
- Lousy salesmen!

Data Storage

Computer System



The Memory Hierarchy

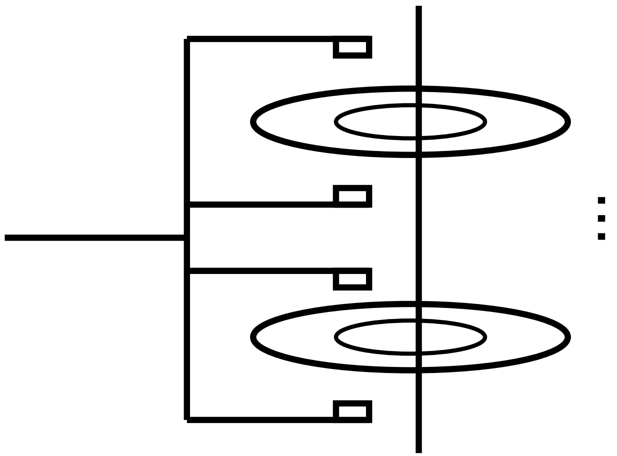


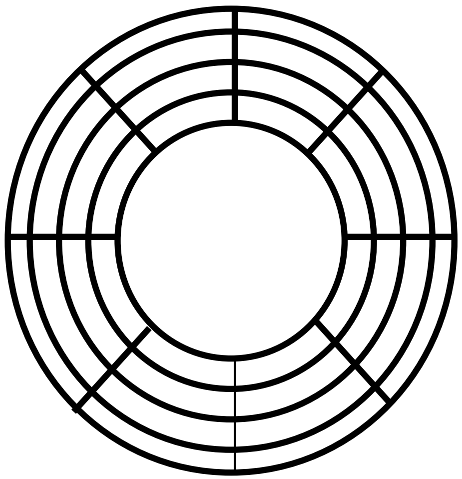
DBMS and Storage

- Databases typically too large to keep in primary storage
- Tables typically kept in secondary storage
- Large amounts of data that are only accessed infrequently are stored in tertiary storage
- Indexes and current tables *cached* in primary storage

Harddisk

- N rotating magenetic platters
- 2xN heads for reading and writing
- track, cylinder, sector, gap



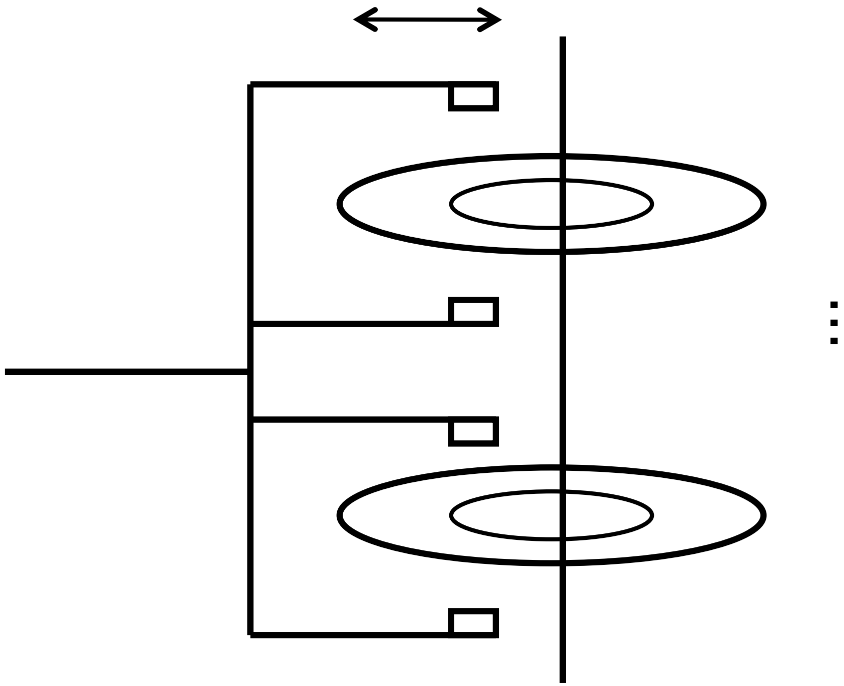


Harddisk Access

- access time: how long does it take to load a block from the harddisk?
- seek time: how long does it take to move the heads to the right cylinder?
- rotational delay: how long does it take until the head gets to the right sectors?
- transfer time: how long does it take to read the block?
- access = seek + rotational + transfer

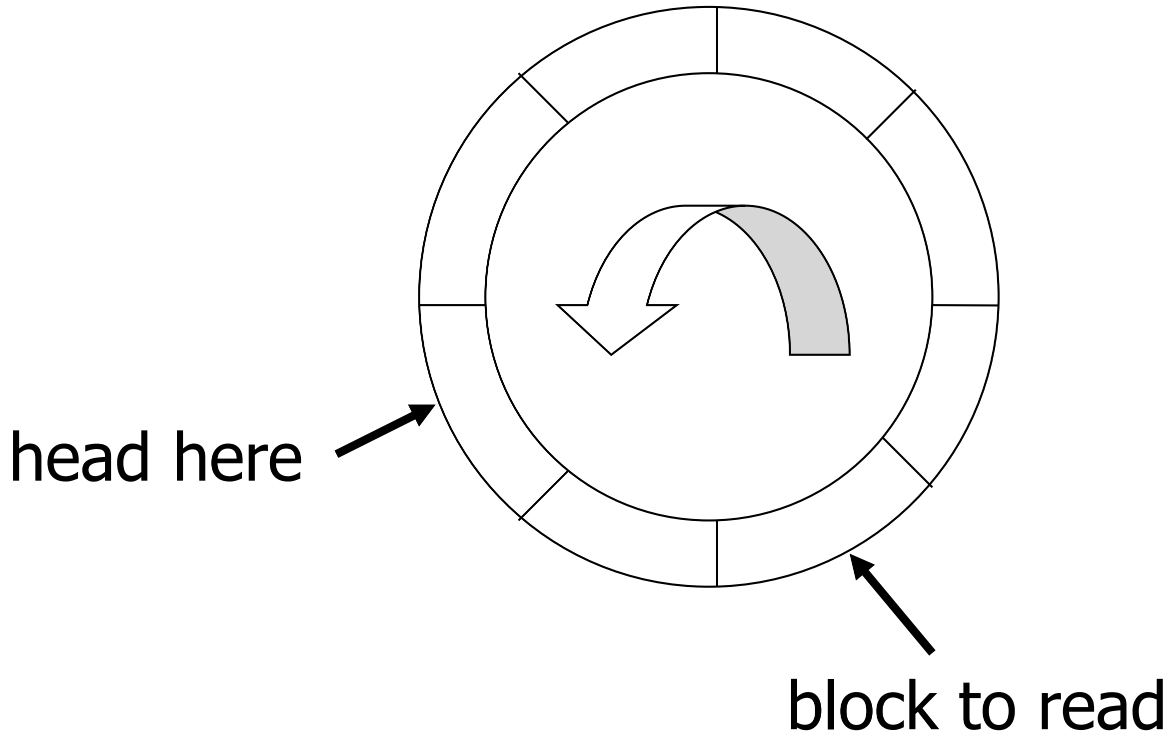
Seek Time

- average seek time = ½ time to move head from outermost to innermost cylinder



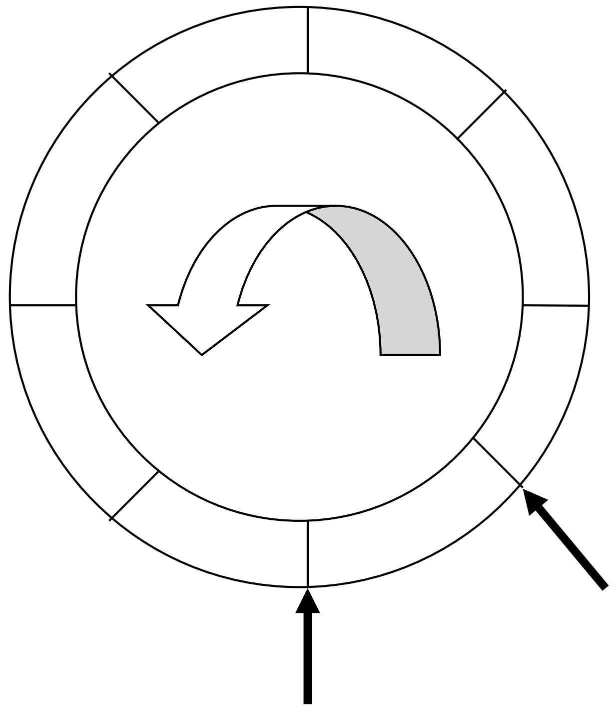
Rotational Delay

- average rotational delay = ½ rotation



Transfer Time

- Transfer time = 1/n rotation when there are n blocks on one track



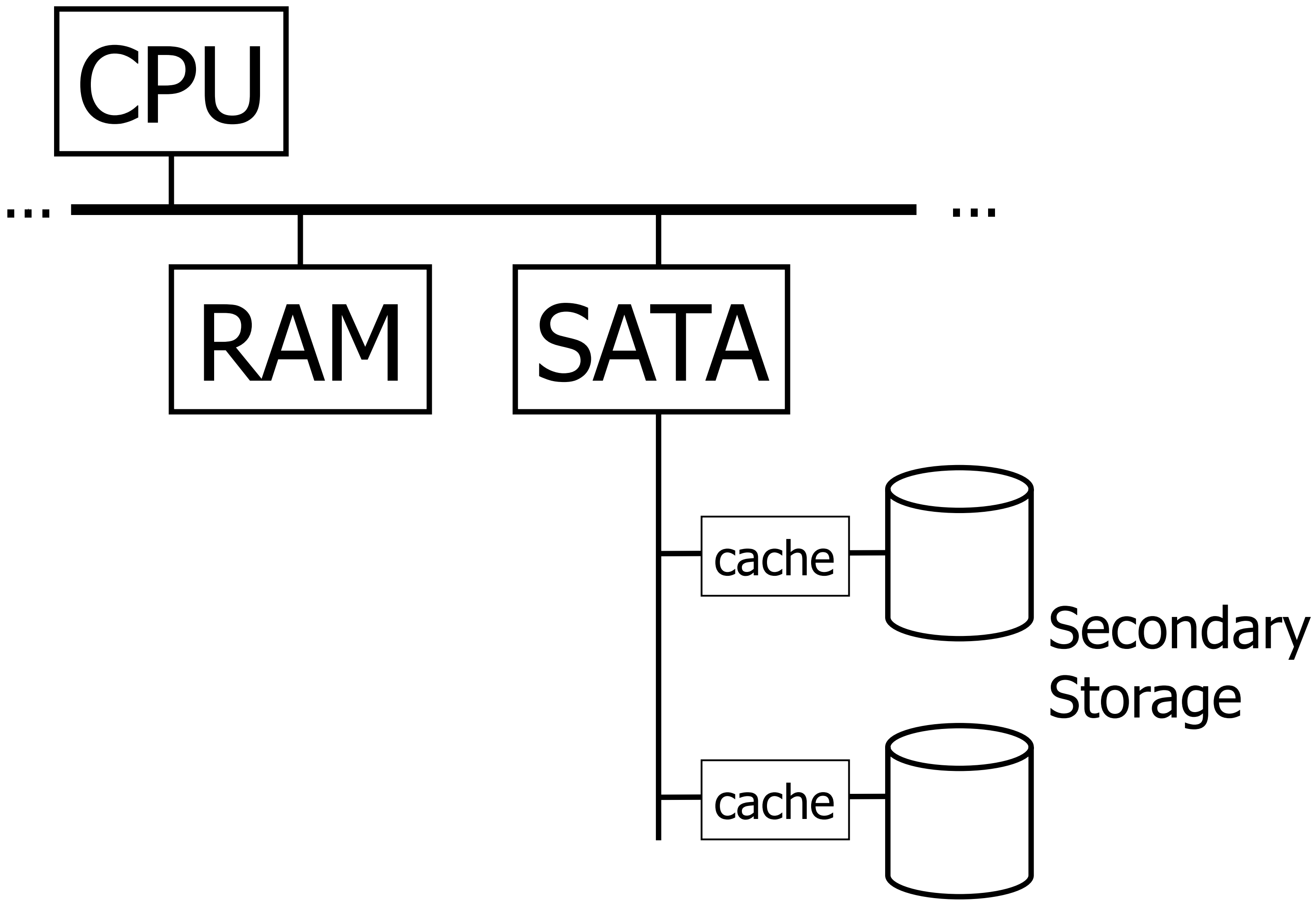
Access Time

- Typical harddisk:
  - Maximal seek time: 10 ms
  - Rotational speed: 7200 rpm
  - Block size: 4096 bytes
  - Sectors (512 bytes) per track: 1600 (average)
- Average access time: 9.21 ms
  - Average seek time: 5 ms
  - Average rotational delay: 60/7200/2 = 4.17 ms
  - Average transfer time: 0.04 ms

Random vs Sequential Access

- Random access of blocks:
  1/0.00921s * 4096 byte = 0.42 Mbyte/s
- Sequential access of blocks:
  120/s * 200 * 4096 byte = 94 Mbyte/s
- Performance of the DBMS dominated by number of random accesses

On Disk Cache



Problems with Harddisks

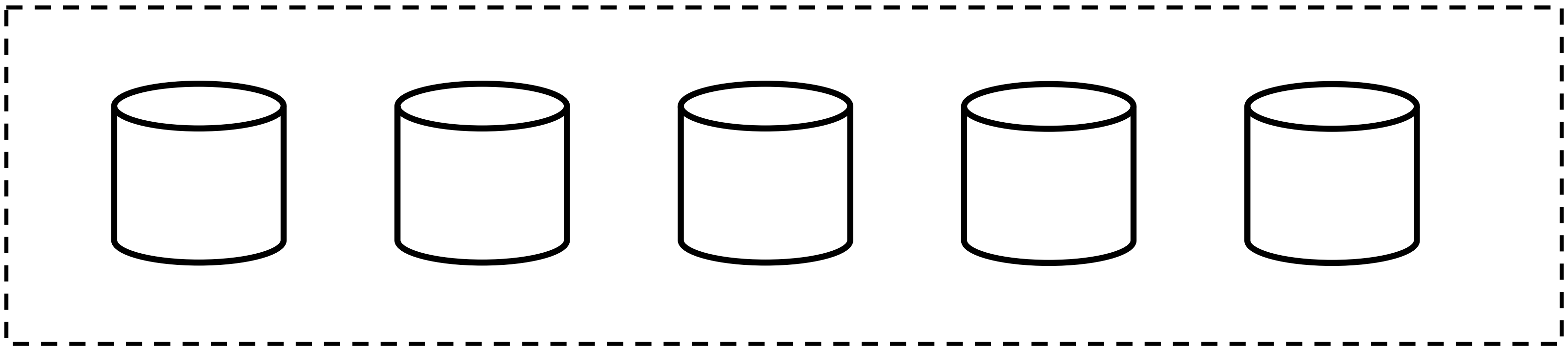
- Even with caches, harddisk remains bottleneck for DBMS performance
- Harddisks can fail:
  - Intermittent failure
  - Media decay
  - Write failure
  - Disk crash
- Handle intermittent failures by rereading the block in question

Detecting Read Failures

- Use checksums to detect failures
- Simplest form is parity bit:
  - 0 if number of ones in the block is even
  - 1 if number of ones in the block is odd
  - Detects all 1-bit failures
  - Detects 50% of many-bit failures
  - By using n bits, we can reduce the chance of missing an error to 1/2^n

Disk Arrays

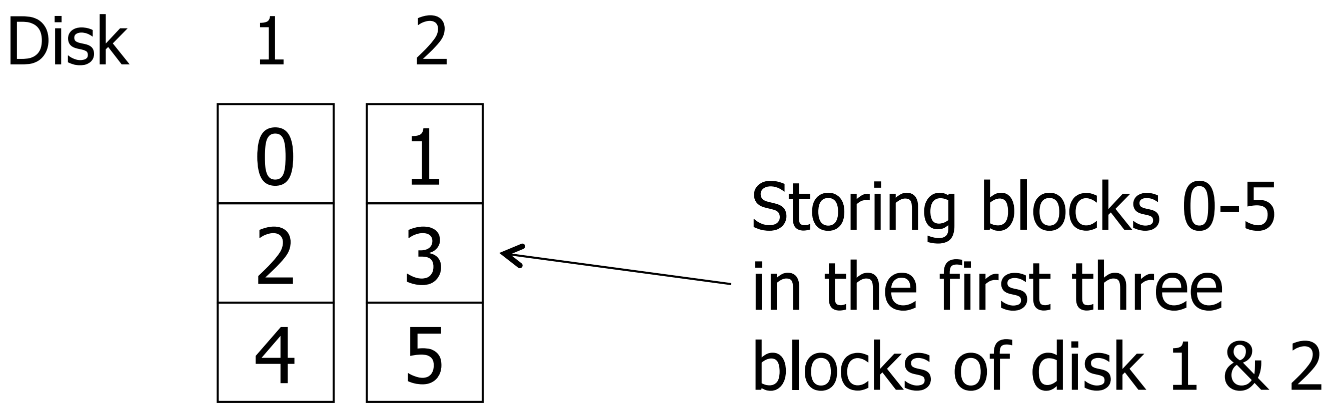
- Use more than one disk for higher reliability and/or performance
- RAID (Redundant Arrays of Independent Disks)



logically one disk

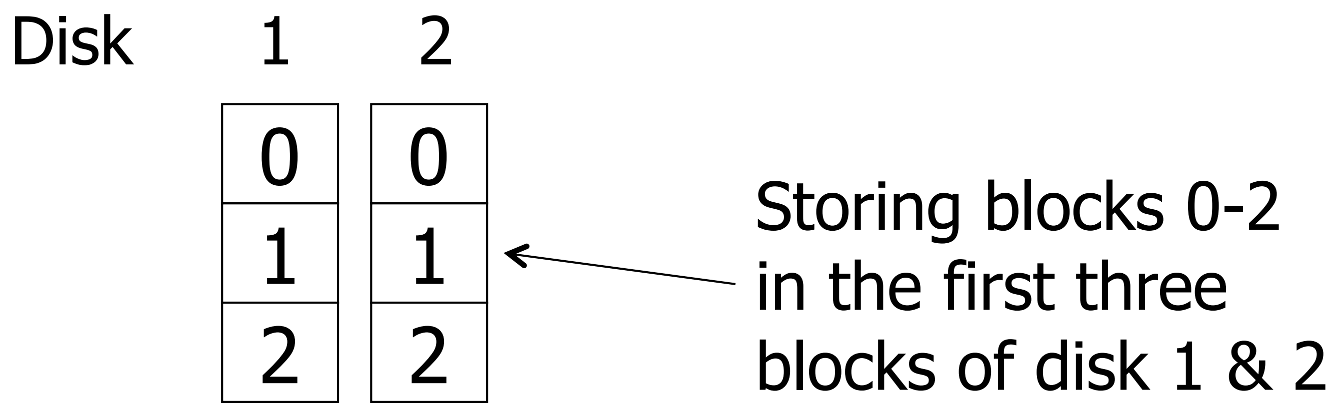
RAID 0

- Alternate blocks between two or more disks (“Striping“)
- Increases performance both for writing and reading
- No increase in reliability



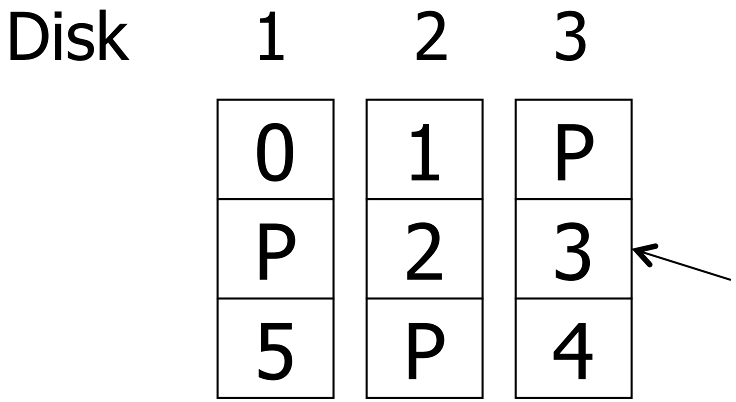
RAID 1

- Duplicate blocks on two or more disks ("Mirroring")
- Increases performance for reading
- Increases reliability significantly



RAID 5

- Stripe blocks on n+1 disks where for each block, one disk stores parity information
- More performant when writing than RAID 1
- Increased reliability compared to RAID 0



Storing blocks 0-5 in the first three blocks of disk 1, 2 & 3

RAID Capacity

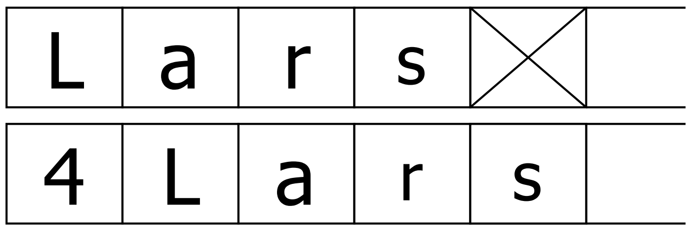
- Assume disks with capacity 1 TByte
- RAID 0: N disks = N TByte
- RAID 1: N disks = 1 TByte
- RAID 5: N disks = (N-1) TByte
- RAID 6: N disks = (N-M) TByte
- ...

Storage of Values

- Basic unit of storage: Byte [ ] ← 8 bits →
- Integer: 4 bytes
  Example: 42 is
  | 00000000 | 00000000 | 00000000 | 00101010 |
- Real: n bits for mantissa, m for exponent
- Characters: ASCII, UTF8, ...
- Boolean: | 00000000 | and | 11111111 |

Storage of Values

- Dates:
  - Days since January 1, 1900
  - DDMMYYYY (not DDMMYY)
- Time:
  - Seconds since midnight
  - HHMMSS
- Strings:
  - Null terminated
  - Length given



DBMS Storage Overview

Values

↓

Records

↓

Blocks

↓

Files

⋮

Memory

Record

- Collection of related data items (called Fields)
- Typically used to store one tuple
- Example: Sells record consisting of
  - bar field
  - beer field
  - price field

Record Metadata

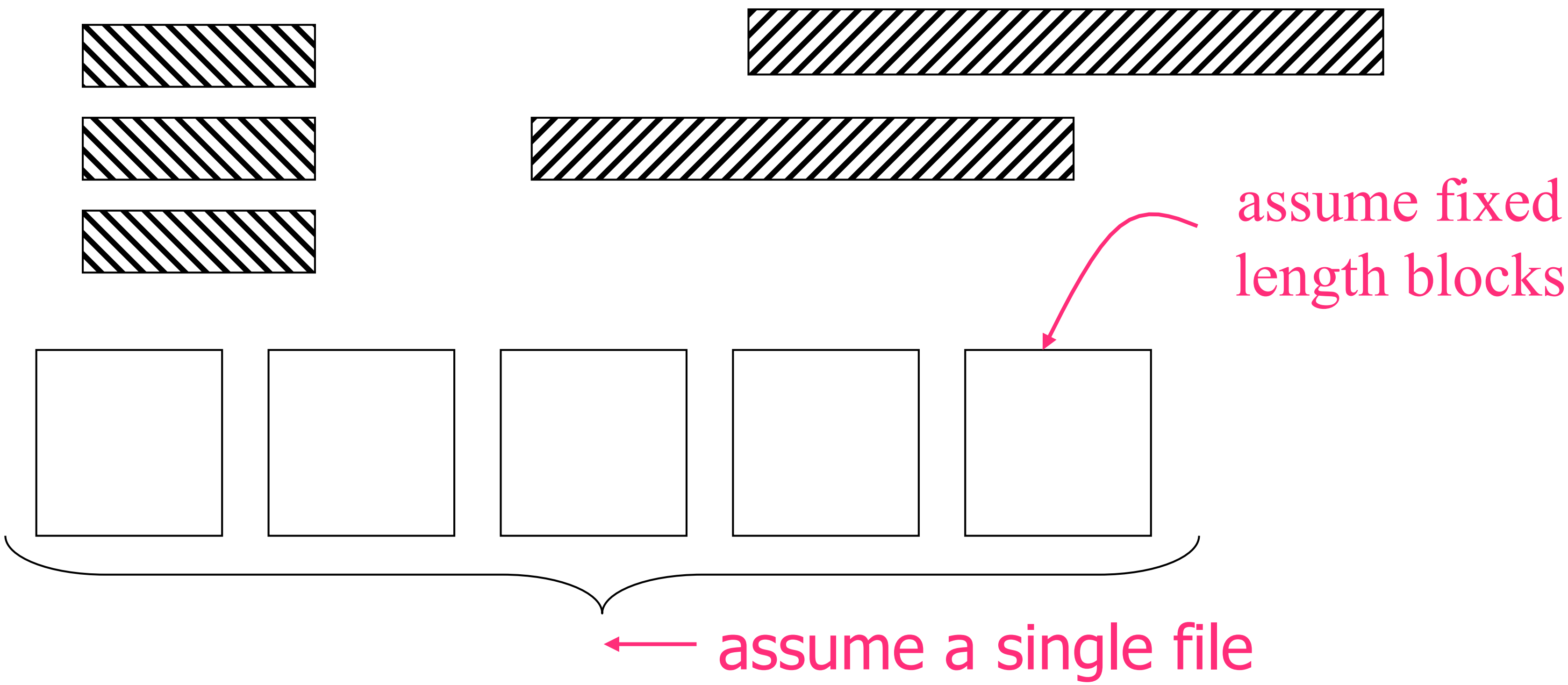
- For fixed-length records, schema contains the following information:
  - Number of fields
  - Type of each field
  - Order in record
- For variable-length records, every record contains this information in its header

Record Header

- Reserved part at the beginning of a record
- Typically contains:
  - Record type (which Schema?)
  - Record length (for skipping)
  - Time stamp (last access)

Files

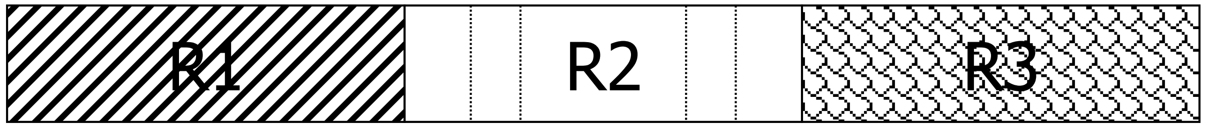
- Files consist of blocks containing records
- How to place records into blocks?



Files

- Options for storing records in blocks:
  1. Separating records
  2. Spanned vs. unspanned
  3. Sequencing
  4. Indirection

1. Separating Records

Block 

a. no need to separate - fixed size recs.

b. special marker

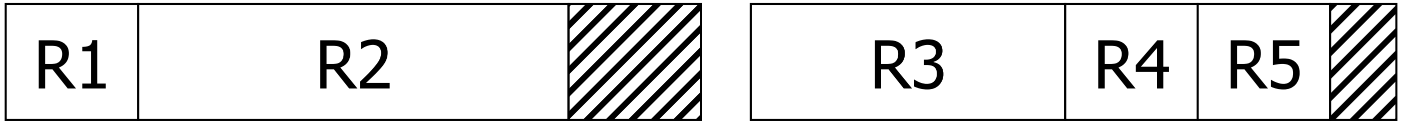
c. give record lengths (or offsets)

   i. within each record

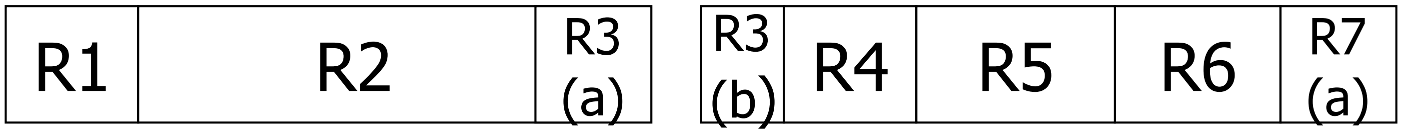
   ii. in block header

2. Spanned vs Unspanned

- Unspanned: records must be in one block



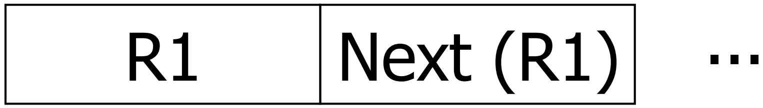
- Spanned: one record in two or more blocks



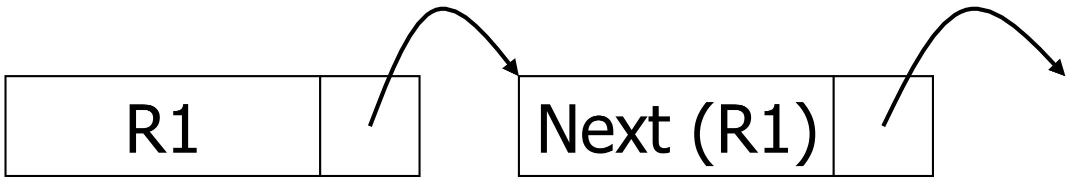
- Unspanned much simpler, but wastes space
- Spanned essential if record size > block size

3. Sequencing

- Ordering records in a file (and in the blocks) by some key value
- Can be used for binary search
- Options:
  a. Next record is physically contiguous

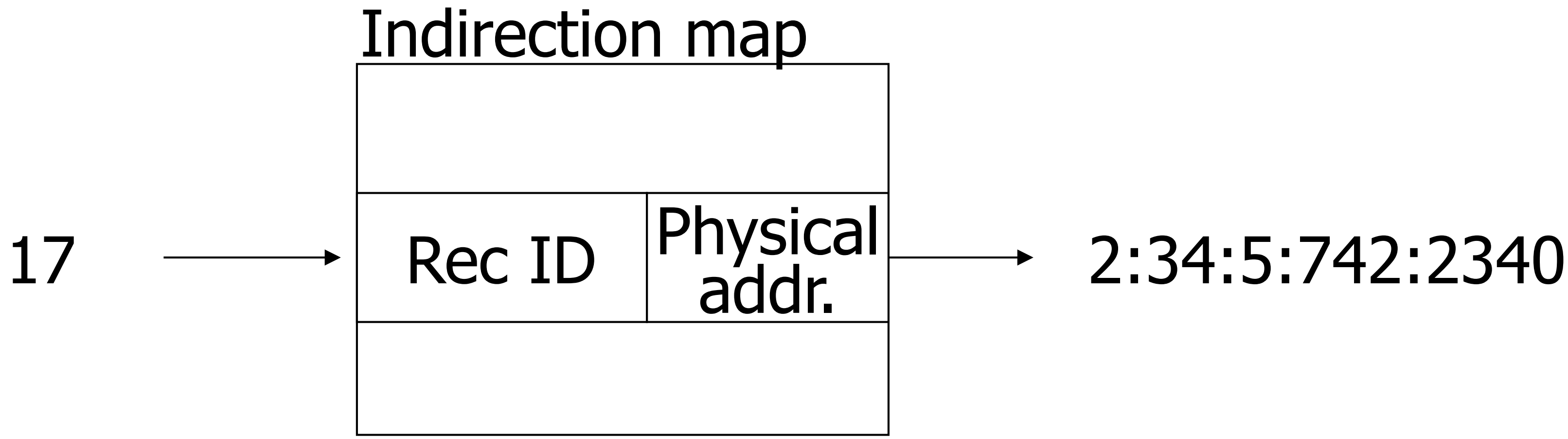


  b. Records are linked



4. Indirection

- How does one refer to records?
  a. Physical address (disk id, cylinder, head, sector, offset in block)
  b. Logical record ids and a mapping table



- Tradeoff between flexibility and cost

Modification of Records

How to handle the following operations on the record level?

1. Insertion
2. Deletion
3. Update

1. Insertion

- Easy case: records not in sequence
  - Insert new record at end of file
  - If records are fixed-length, insert new record in deleted slot
- Difficult case: records are sorted
  - Find position and slide following records
  - If records are sequenced by linking, insert overflow blocks

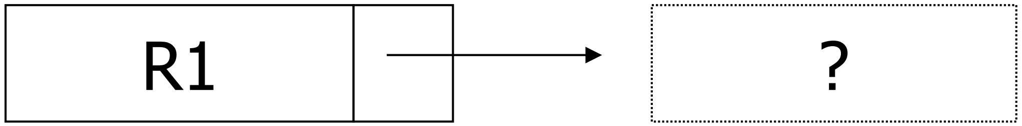
2. Deletion

a. Immediately reclaim space by shifting other records or removing overflows
b. Mark deleted and list as free for re-use

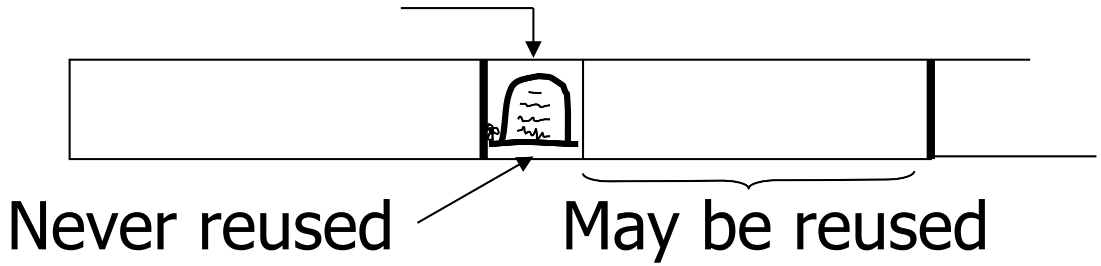
- Tradeoffs:
  - How expensive is immediate reclaim?
  - How much space is wasted?

Problem with Deletion

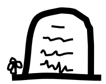
- Dangling pointers:



- When using physical addresses:



- When using logical addresses:

ID	LOC
7788	

Never reuse
ID 7788 nor
space in the map