Database-Connection Libraries

Host/SQL Interfaces Via Libraries

- The third approach to connecting databases to conventional languages is to use library calls
  1. C + CLI
  2. Java + JDBC
  3. Python + psycopg2

Three-Tier Architecture

- A common environment for using a database has three tiers of processors:
  1. *Web servers* – talk to the user.
  2. *Application servers* – execute the business logic
  3. *Database servers* – get what the app servers need from the database

Example: Amazon

- Database holds the information about products, customers, etc.
- Business logic includes things like "what do I do after someone clicks 'checkout'?"
  - Answer: Show the "how will you pay for this?" screen

Environments, Connections, Queries

- The database is, in many DB-access languages, an *environment*
- Database servers maintain some number of *connections*, so app servers can ask queries or perform modifications
- The app server issues *statements:* queries and modifications, usually

JDBC

- *Java Database Connectivity* (JDBC) is a library similar for accessing a DBMS with Java as the host language
- >200 drivers available: PostgreSQL, MySQL, Oracle, ODBC, ...
- `http://jdbc.postgresql.org/`

Making a Connection

The JDBC classes

```
import java.sql.*;
...
Class.forName("org.postgresql.Driver");
Connection myCon =
  DriverManager.getConnection(…);
...
```

Loaded by forName

URL of the database your name, and password go here

The driver for postgresql; others exist

URL for PostgreSQL database

- getConnection(jdbc:postgresql://<host>[:<port>]/<database>?user=<user>&password=<password>);
- Alternatively use getConnection variant:
- getConnection("jdbc:postgresql://<host>[:<port>]/<database>", <user>, <password>);
- DriverManager.getConnection("jdbc:postgresql://10.110.4.32:5434/postgres", "petersk", "geheim");

Statements

- JDBC provides two classes:
  1. *Statement* = an object that can accept a string that is a SQL statement and can execute such a string
  2. *PreparedStatement* = an object that has an associated SQL statement ready to execute

Creating Statements

- The Connection class has methods to create Statements and PreparedStatements

```
Statement stat1 = myCon.createStatement();
PreparedStatement stat2 =
    myCon.createStatement(
        ”SELECT beer, price FROM Sells ” +
        ”WHERE bar = ’C.Ch.’ ”
    );
```

createStatement with no argument returns a Statement; with one argument it returns a PreparedStatement

Executing SQL Statements

- JDBC distinguishes queries from modifications, which it calls "updates"
- Statement and PreparedStatement each have methods executeQuery and executeUpdate
  - For Statements: one argument – the query or modification to be executed
  - For PreparedStatements: no argument

Example: Update

- stat1 is a Statement
- We can use it to insert a tuple as:

```
stat1.executeUpdate(
  ”INSERT INTO Sells ” +
  ”VALUES(’C.Ch.’,’Eventyr’,30)”
);
```

Example: Query

- stat2 is a PreparedStatement holding the query ”SELECT beer, price FROM Sells WHERE bar = ’C.Ch.’ ”
- executeQuery returns an object of class ResultSet – we’ll examine it later
- The query:

ResultSet menu = stat2.executeQuery();

Accessing the ResultSet

- An object of type ResultSet is something like a cursor
- Method next() advances the "cursor" to the next tuple
  - The first time next() is applied, it gets the first tuple
  - If there are no more tuples, next() returns the value false

Accessing Components of Tuples

- When a ResultSet is referring to a tuple, we can get the components of that tuple by applying certain methods to the ResultSet
- Method get*X* (*i*), where *X* is some type, and *i* is the component number, returns the value of that component
  - The value must have type *X*

Example: Accessing Components

- Menu = ResultSet for query "SELECT beer, price FROM Sells WHERE bar = 'C.Ch.' "
- Access beer and price from each tuple by:

```
while (menu.next()) {
  theBeer = menu.getString(1);
  thePrice = menu.getFloat(2);
     /*something with theBeer and
       thePrice*/
}
```

Important Details

- Reusing a Statement object results in the ResultSet being closed
  - Always create new Statement objects using createStatement() or explicitly close ResultSets using the close method
- For transactions, for the Connection con use con.setAutoCommit(false) and explicitly con.commit() or con.rollback()
  - If AutoCommit is false and there is no commit, closing the connection = rollback

Python and Databases

- many different modules for accessing databases
- commercial: mxodbc, ...
- open source: pygresql, psycopg2, ...
- we use psycopg2
  - install using `easy_install psycopg2`
  - import with `import psycopg2`

Connection String

- Database connection described by a connection string
- Example: `con_str = """`

```
host='10.110.4.32'
port=5434
dbname='postgres'
user='petersk'
password='geheim'
"""
```

Making a Connection

- With the DB library imported and the connection string `con_str` available:

```
con = psycopg2.connect(con_str);
```

Function connect in the DB API

Class is connection because it is returned by psycopg2.connect(…)

Cursors in Python

- Queries are executed for a cursor
- A cursor is obtained from connection
- Example:

```
cursor = con.cursor()
```

- Queries or modifications are executed using the `execute(…)` method
- Cursors can then be used in a `for`-loop

Example: Executing a Query

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

```
beer = 'Od.Cl.’
cursor = con.cursor()
cursor.execute(
   "SELECT bar FROM Sells" +
   “WHERE beer = '%s’;" % beer);
```

Remember this variable is replaced by the value of `beer`

Example: Tuple Cursors

```
bar = 'C.Ch.'
cur = con.cursor()
cur.execute("SELECT beer, price" +
" FROM Sells" +
" WHERE bar = " + bar + ";")
for row in cur:
  print row[0] + “ for “ + row[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

```
cur.execute("SELECT acct FROM " +
"Accounts WHERE name = '%s' " +
“AND passwd = '%s';" % (n,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, psycopg2

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

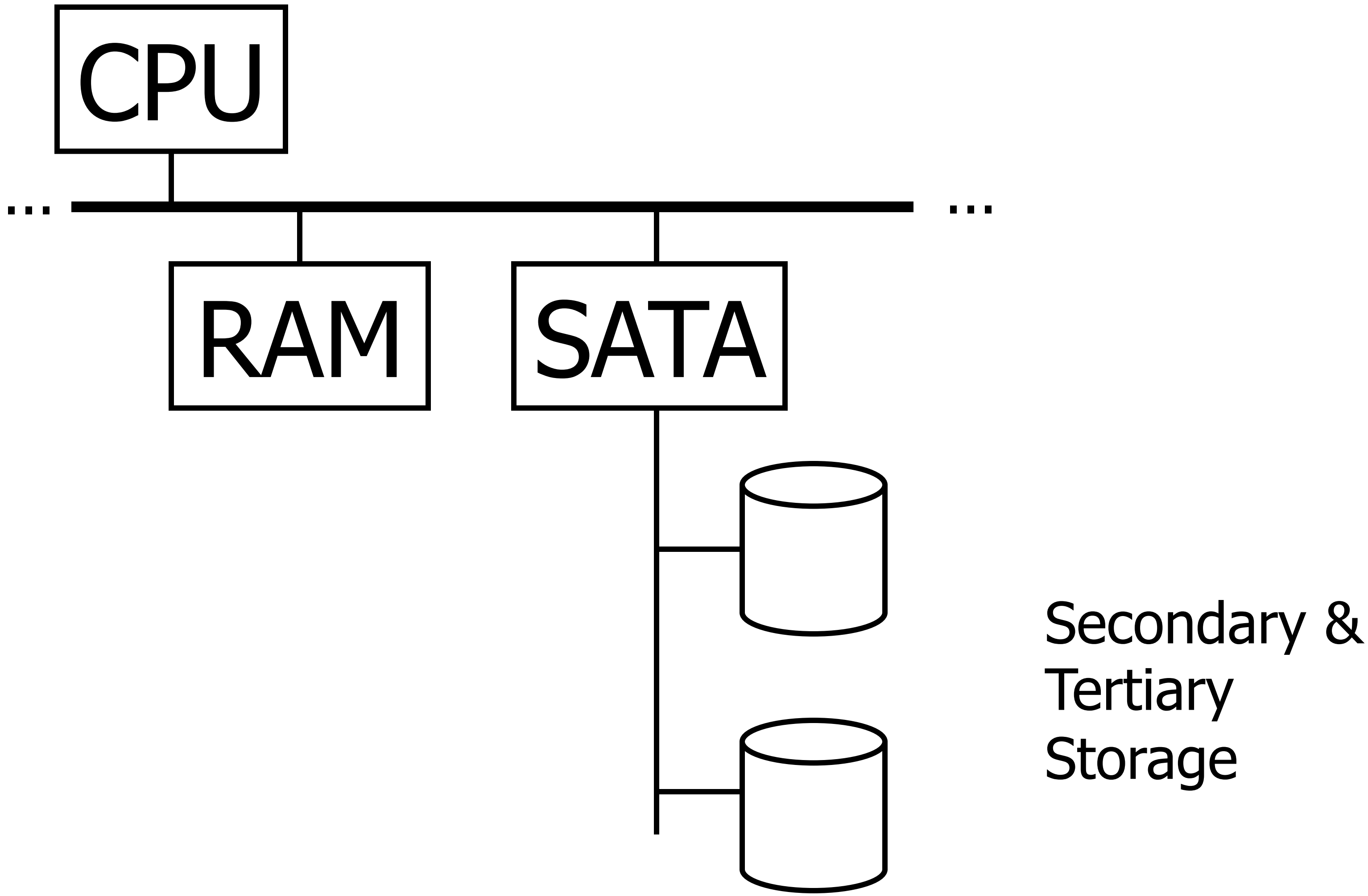
- No reliability
  - Can lose data
  - Can leave operations half done
- No security
  - File system insecure
  - File system security is too coarse
- No application program interface (API)
  - How to access the data from a real program?

DanDB 3000 Problems

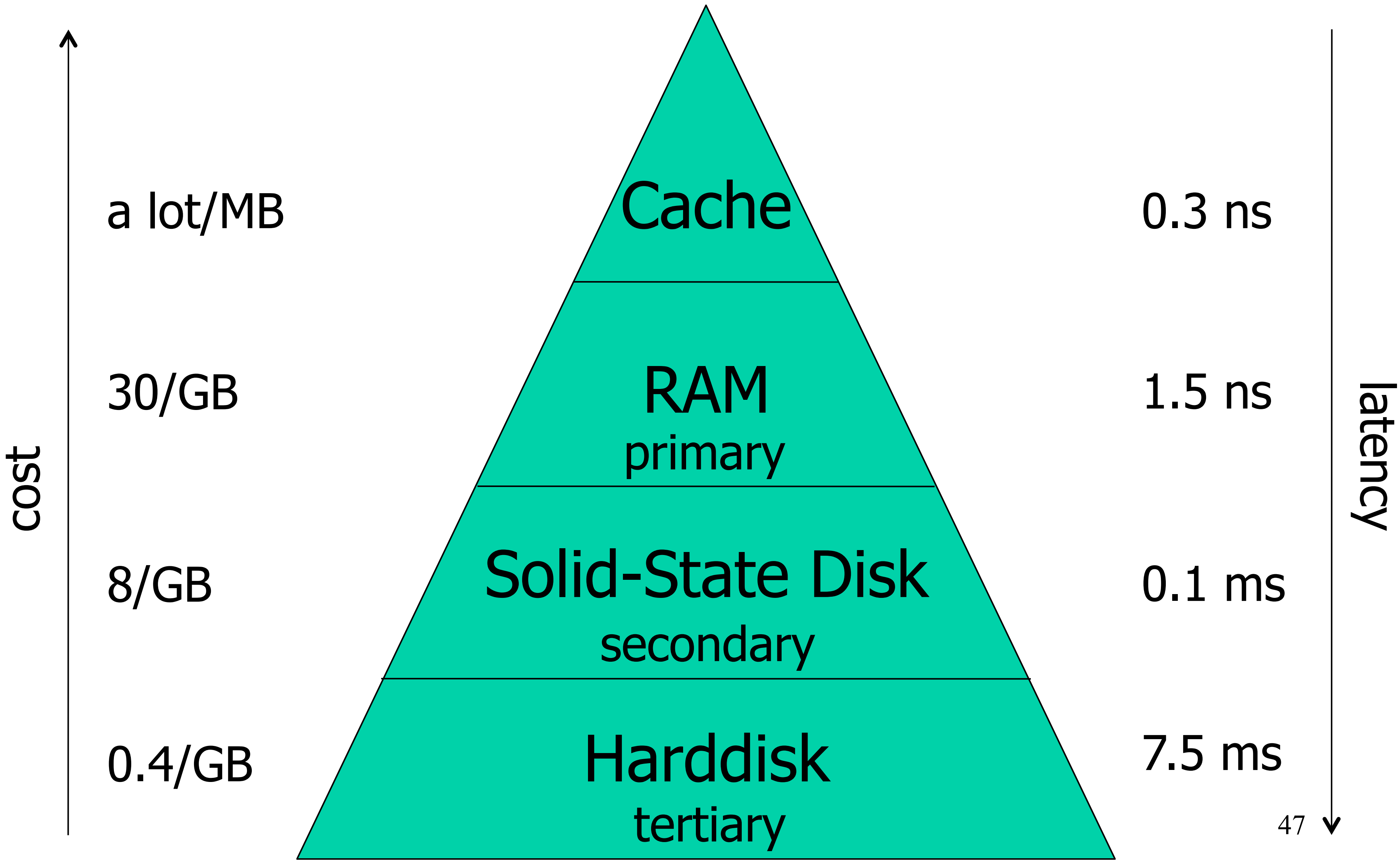
- Cannot interact with other DBMSs
  - Very limited support of SQL
- No constraint handling etc.
- No administration utilities, no web frontend, no graphical user interface, ...
- Lousy salesmen!

Data Storage

Computer System



The Memory Hierarchy

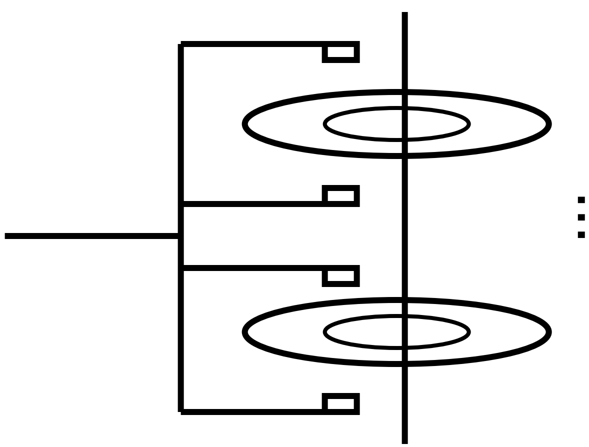


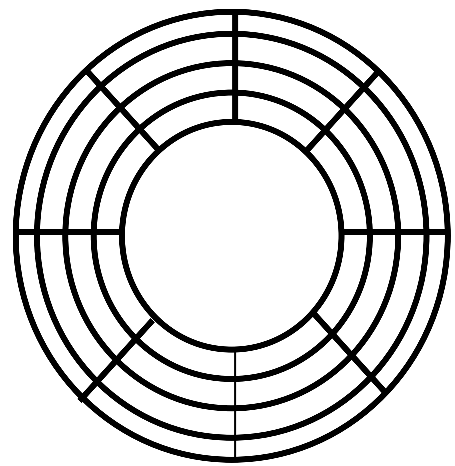
DBMS and Storage

- 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 (or even on tape robot)
- Indexes and current tables *cached* in primary storage

Harddisk

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



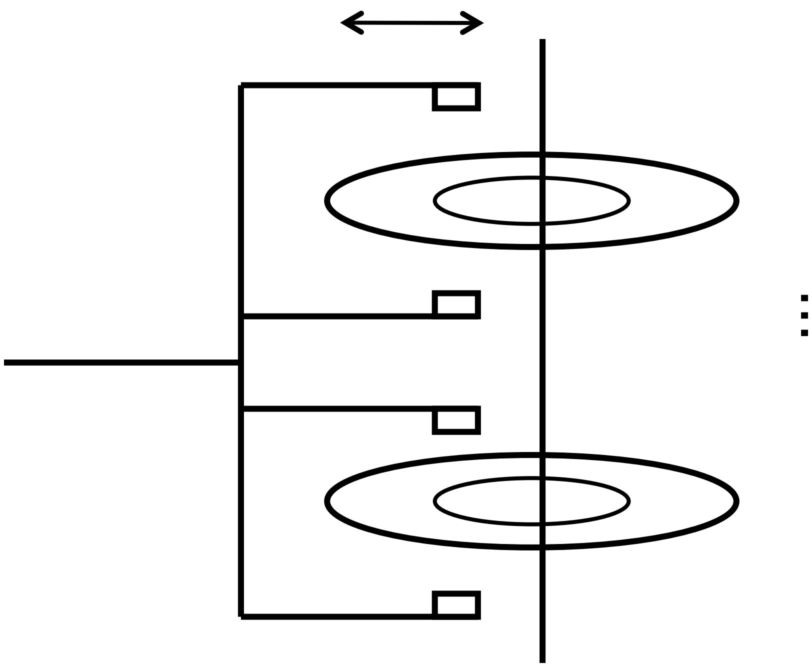


Harddisk Access

- 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

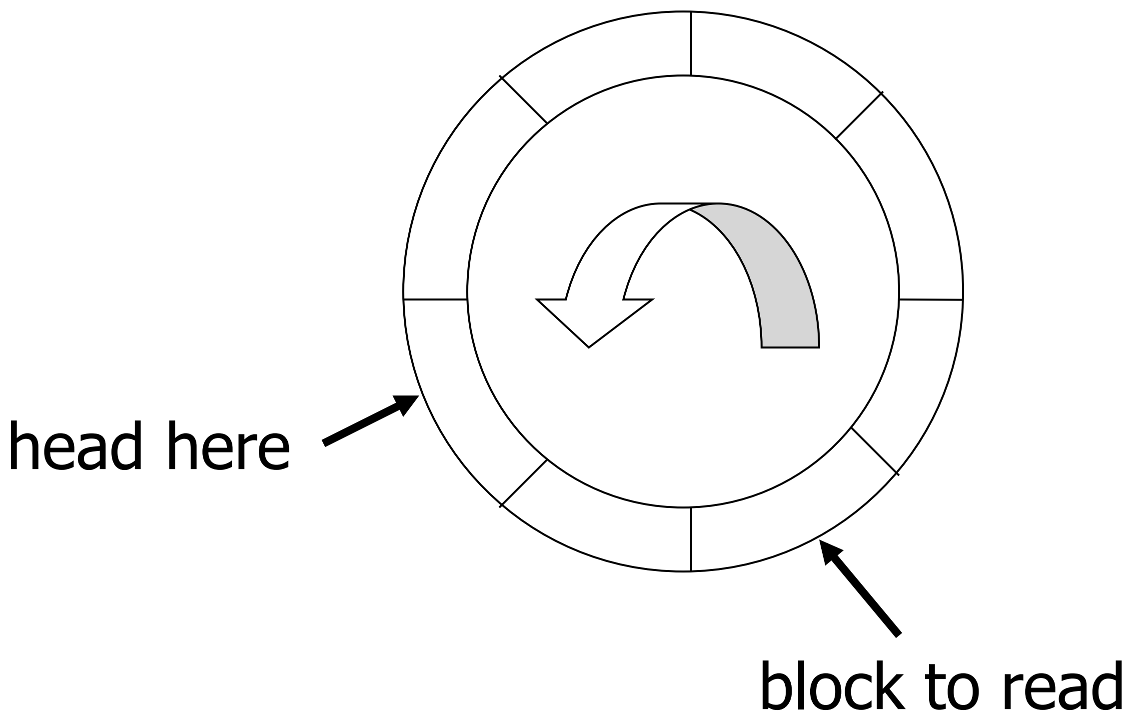
Seek Time

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



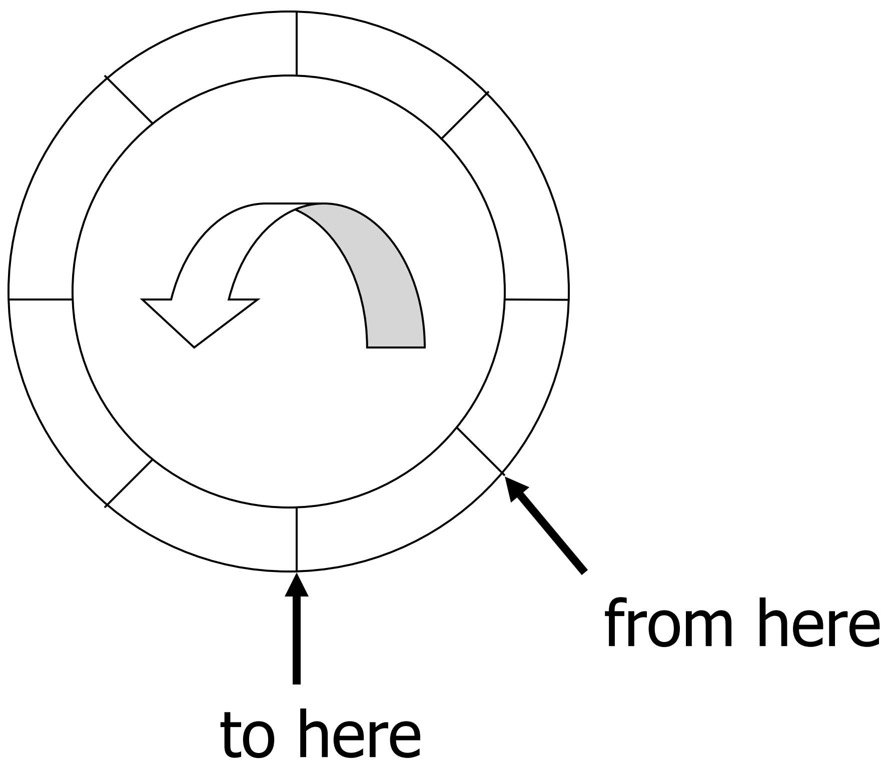
Rotational Delay

- average rotational delay = ½ rotation



Transfer Time

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



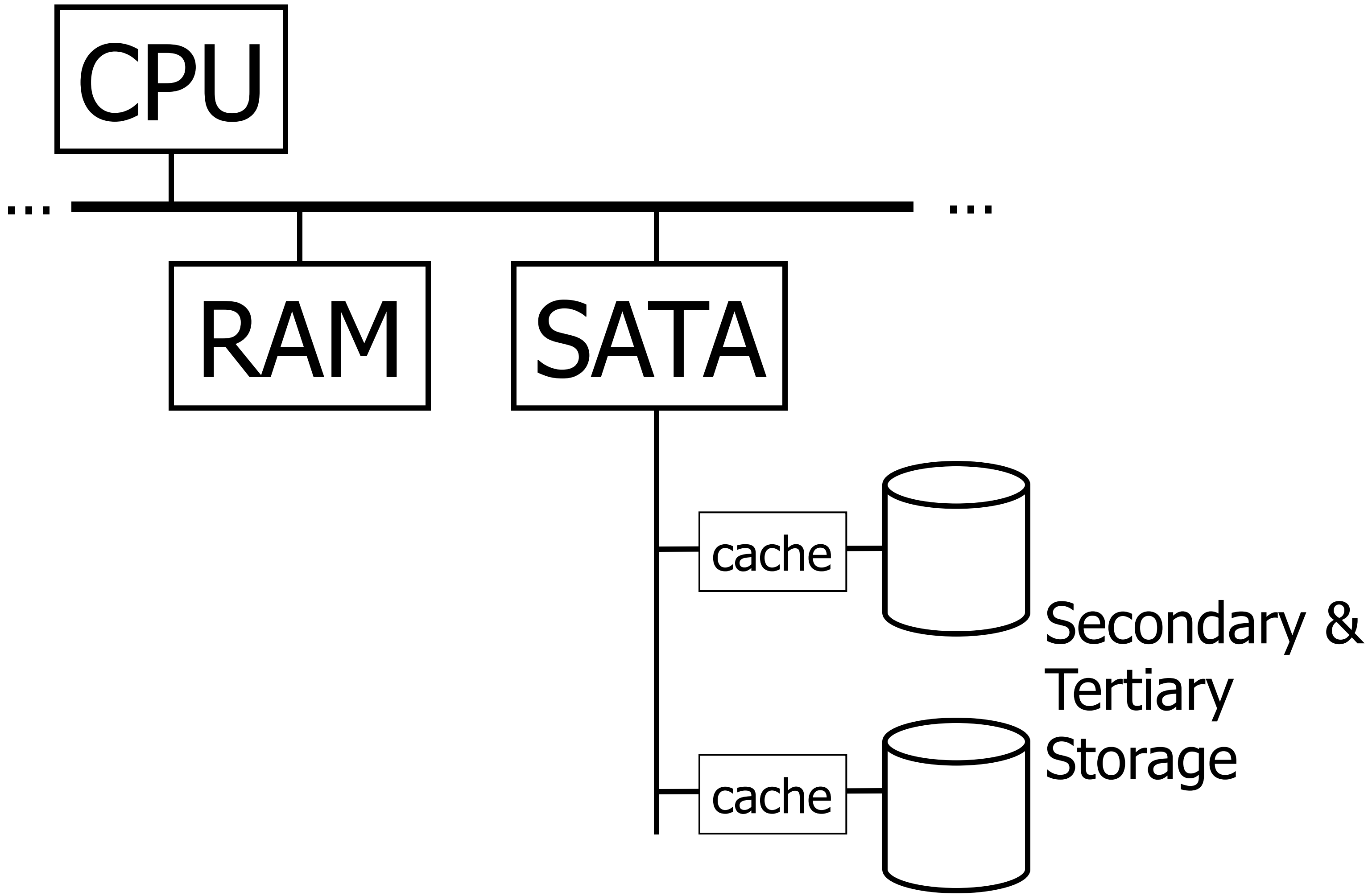
Access Time

- 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

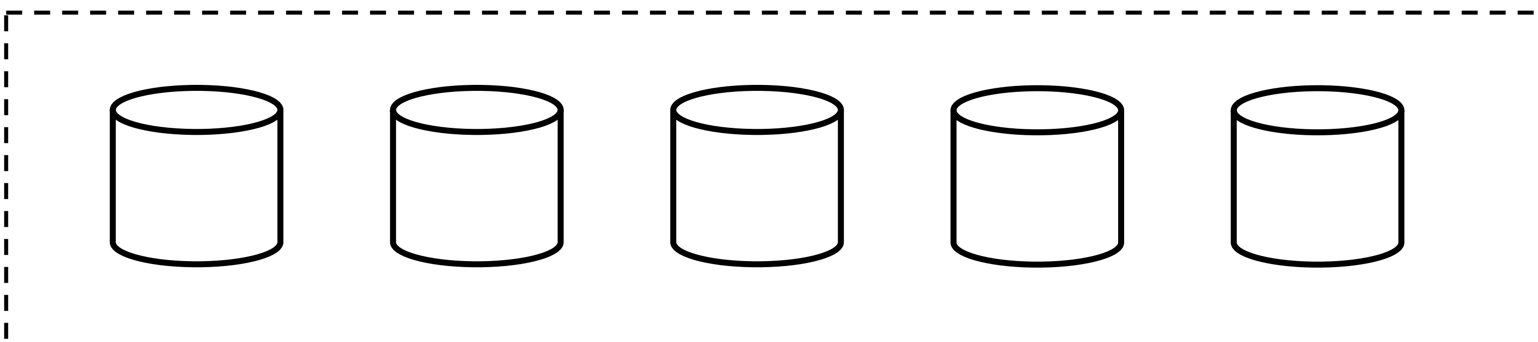
- Even with caches, harddisk remains bottleneck for DBMS performance
- Harddisks can fail:
  - Intermittent failure
  - Media decay
  - Write failure
  - Disk crash
- Handle intermittent failures by rereading the block in question

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

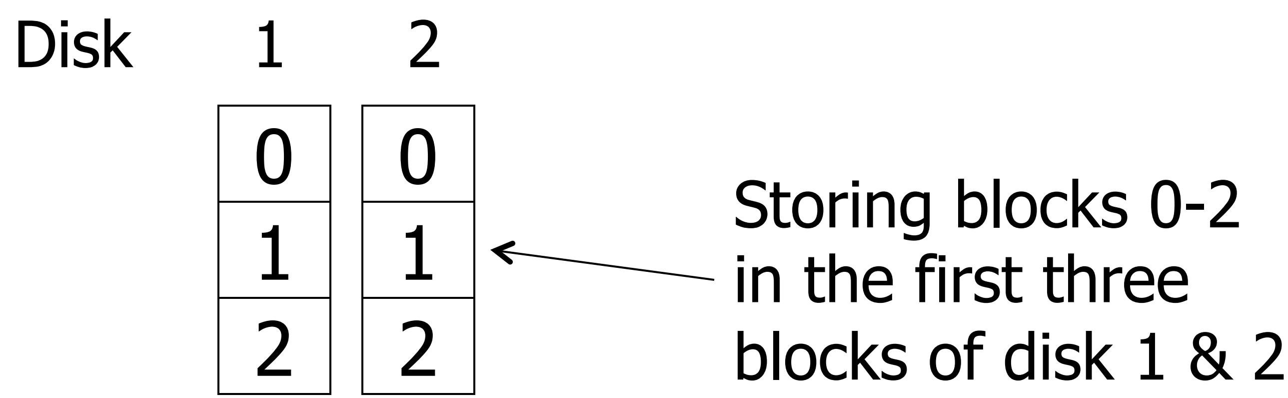
- Alternate blocks between two or more disks (“Striping“)
- Increases performance both for writing and reading
- No increase in reliability

Disk	1	2
	0	1
	2	3
	4	5

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

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

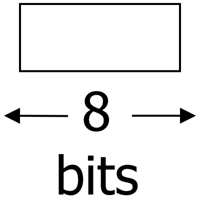
Disk	1	2	3
	0	1	P
	P	2	3
	5	P	4

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 

- Integer: 4 bytes
  Example: 42 is
  $\boxed{00000000}$ $\boxed{00000000}$ $\boxed{00000000}$ $\boxed{00101010}$
- Real: n bits for mantissa, m for exponent
- Characters: ASCII, UTF8, ...
- Boolean: $\boxed{00000000}$ and $\boxed{11111111}$

Storage of Values

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

L	a	r	s		
4	L	a	r	s	

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

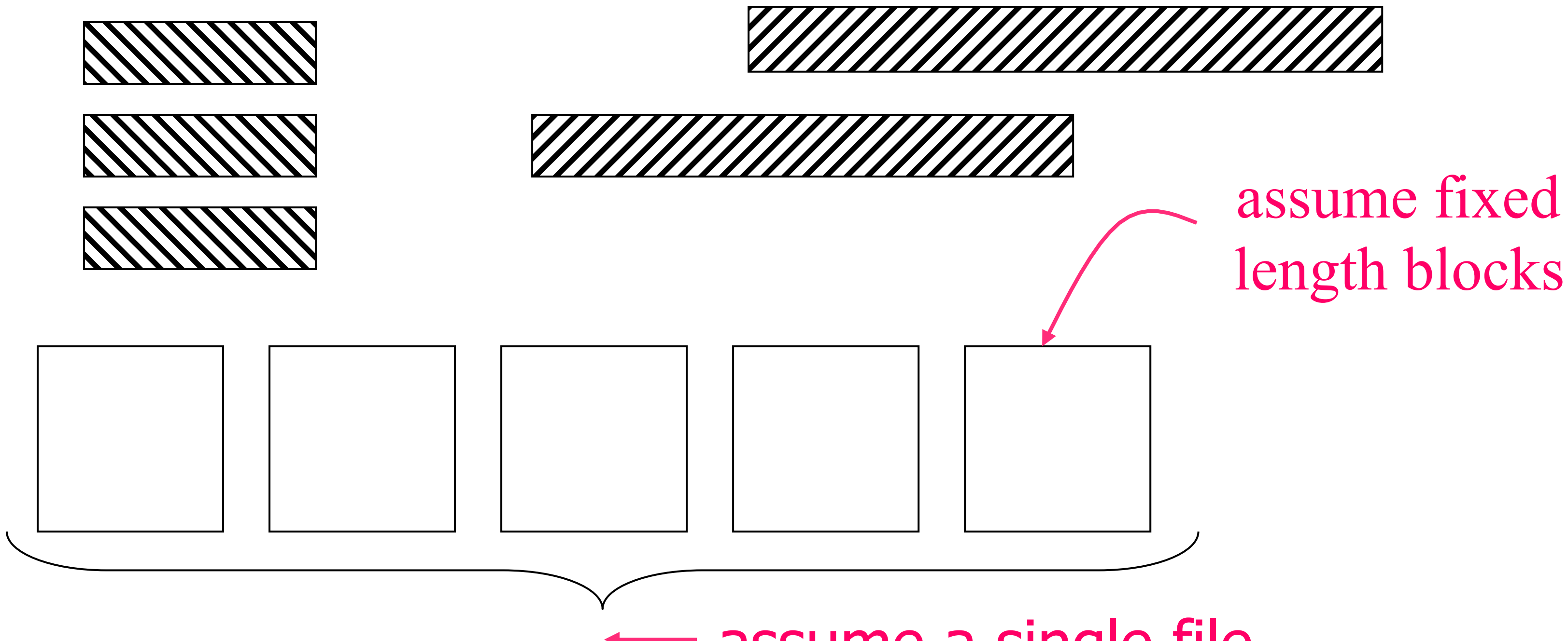
- For fixed-length records, schema contains the following information:
  - Number of fields
  - Type of each field
  - Order in record
- For variable-length records, every record contains this information in its header

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?

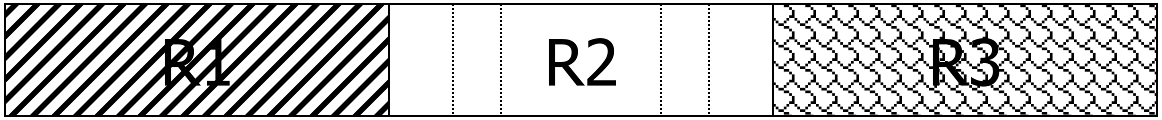


Files

- 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

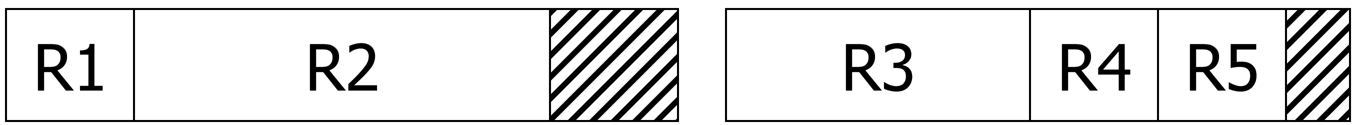
c. give record lengths (or offsets)

  i. within each record

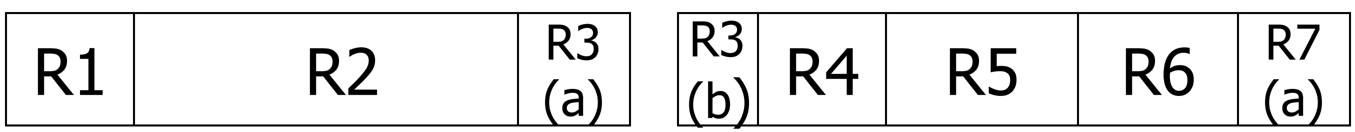
  ii. in block header

2. Spanned vs Unspanned

- Unspanned: records must be in one block



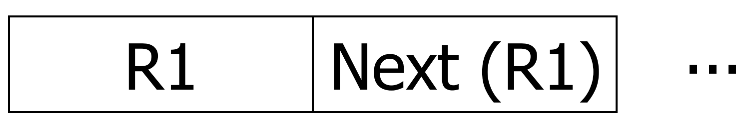
- Spanned: one record in two or more blocks



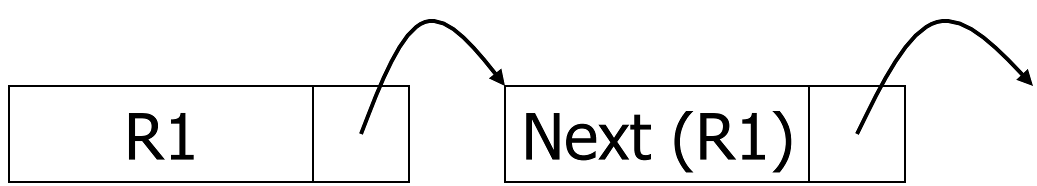
- Unspanned much simpler, but wastes space
- Spanned essential if record size > block size

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

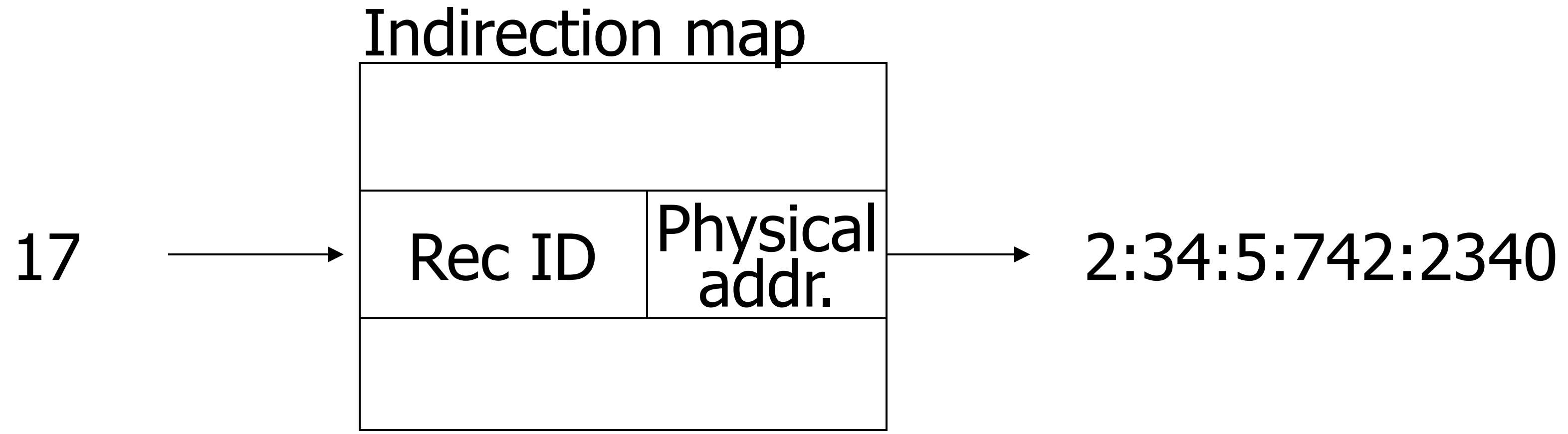


  b. Records are linked



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