Database 2
Lecture I

Alessandro Artale

Faculty of Computer Science – Free University of Bolzano

Room: 221

artale@inf.unibz.it

http://www.inf.unibz.it/∼artale/

2003/2004 – First Semester

Course Overview

- Techniques of record storage on disk;
- Indexing techniques for efficient records retrieval;
- Query processing and optimization;
- Transaction management
  - Concurrency control
  - Error recovery techniques
- Transaction management in Distributed Databases

Readings

Database Systems: The Complete Book, Hector Garcia-Molina, Jeff Ullman, and Jennifer Widom, Prentice-Hall, 2002.

Fundamentals of Database Systems, R. Elmasri and S. B. Navathe, Addison Wesley, 4^{th} Edition, 2003.

Database System Concepts, A.Silberschatz, H.F.Korth, S.Sudarshan, 4^{th} edition, Mc Graw Hill, 2002.

Summary of Lecture I

- **I**ntroduction to Databases.
- Storing a Database.
  - Storing a toy database;
  - An overview of secondary storage devices;
  - Physical layout of data;
  - Efficient techniques for storing DB Relations.

What is a Database?

- A **Database** is a collection of data with a correlated meaning which are *stored* and *manipulated*.
- A database represents some aspect of the real world called the *mini-world* or *Universe of Discourse* (UoD).

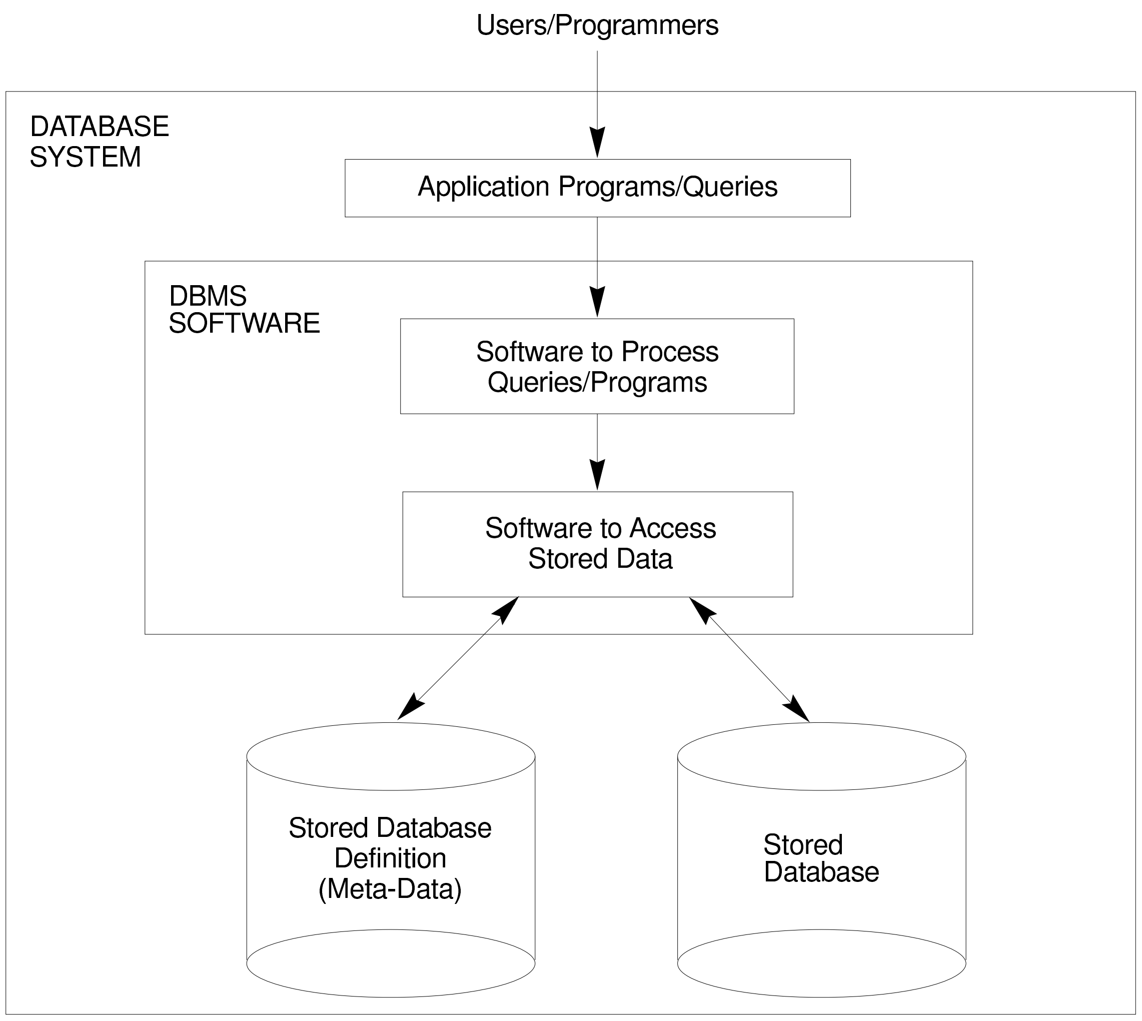
What is a Database Management System?

A **Database Management System (DBMS)** is a general-purpose software which facilitates the process of **creating**, **storing** and **manipulating** databases in an *efficient* way (both in space and time).

DBMS main features:

1. Data must **persist** over long period of time;
2. Supports efficient **access** to very large amounts of data (gigabytes or more);
3. Support for **concurrent access** to data: Many different processes (*transactions*) can access the Database at once;
4. **Recovery** from system failures;
5. **Security**: Users must be authenticated.

A Database Management System


Users/Programmers
DATABASE SYSTEM
Application Programs/Queries
DBMS SOFTWARE
Software to Process Queries/Programs
Software to Access Stored Data
Stored Database Definition (Meta-Data)
Stored Database

Summary

- Introduction to Databases.
- **S**toring a Database.
  - **S**toring a toy database;
  - An overview of secondary storage devices;
  - Physical layout of data;
  - Efficient techniques for storing DB Relations.

Storing a Database

Databases are generally stored on Disk rather than in main memory.

- **Size.** Databases for enterprise contain giga or tera-byte of data.
- **Volatility.** Data must be preserved and main memory is volatile.

Storing a Toy Database

STUDENTS

NAME	ID	DEPT
Smith	123	CS
Johnson	522	ES
. . .	. . .	. . .

DEPTS

NAME	OFFICE
CS	F10
ES	L12
. . .	. . .

- DB schema stored in a special file /usr/db/schema:

```
Students#name#STR#id#INT#dept#STR
Depts#name#STR#office#STR
...
```

- Relations stored in files, e.g., relation *Students* in /usr/db/Students:

```
Smith#123#CS
Johnson#522#ES
...
```

Query Execution: Selection

SELECT * **FROM** R **WHERE** <Condition>

1. Read the schema file to determine attributes of R and their types;
2. Read the R file and for each line:
   (a) Check the condition;
   (b) Display the line if the condition is true.

Query Execution: Join

```
SELECT office
FROM   Students, Depts
WHERE  Students.name = 'Smith' AND
       Students.dept = Depts.name.
```

1. Read the schema file to determine attributes of `Students` and `Depts` and their types;
2. Read in turn each line/tuple from files Students and Depts, and:
   (a) Check whether the name of the student is 'Smith', and if true
   (b) Check whether the current pair of tuples represent the same department, and if true
   (c) Display the office.

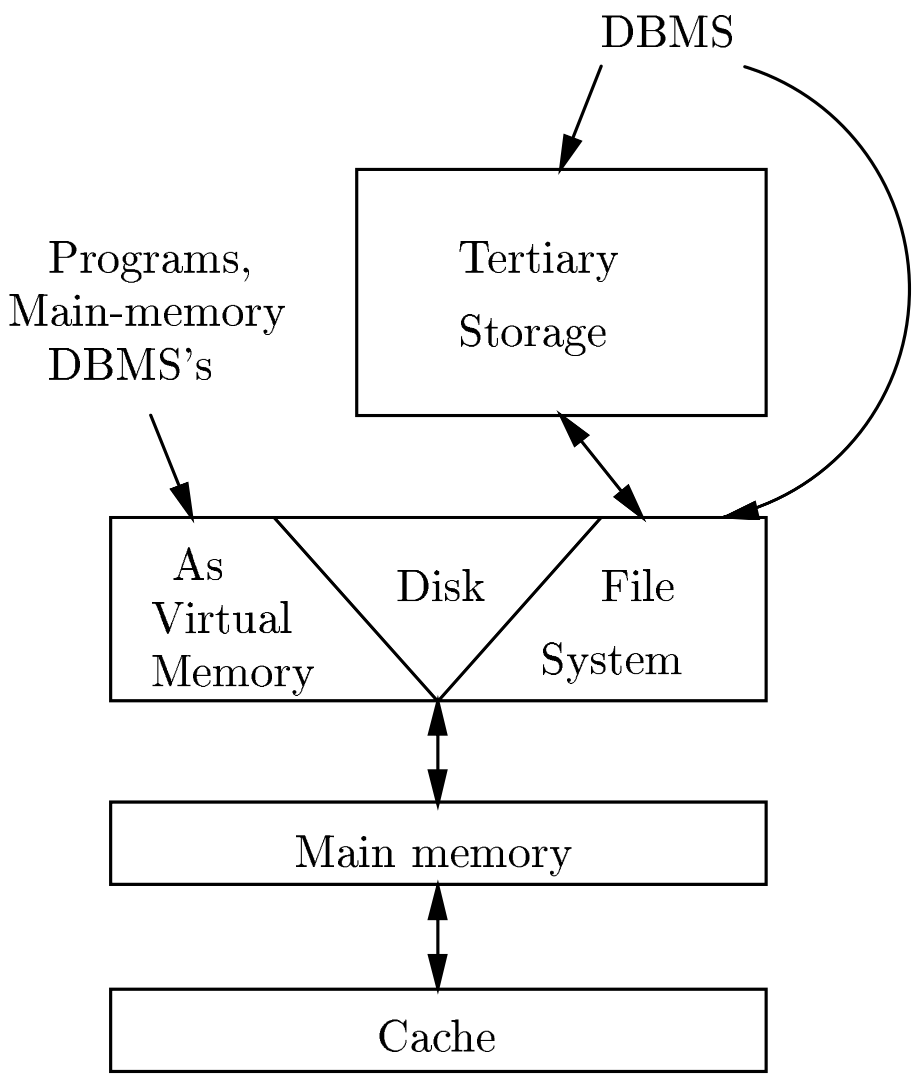
What is Wrong

1. **No flexibility for DB modifications:** For each change the entire file has to be rewritten.
2. **Search Expensive:**
   - Selection: Always read entire relation;
   - Join: It is not necessary to look at all pairs of tuples – more efficient join operations.
3. **No buffered data:** Everything comes off the disk all the time.
4. **No concurrency control:** Several users can modify a file at the same time with unpredictable results.
5. **No reliability:** We can lose data in a crash.
6. **No security:** File system protection too coarse.

Summary

- Introduction to Databases.
- Storing a Database.
  - Storing a toy database;
  - An overview of secondary storage devices;
  - Physical layout of data;
  - Efficient techniques for storing DB Relations.

The Memory Hierarchy



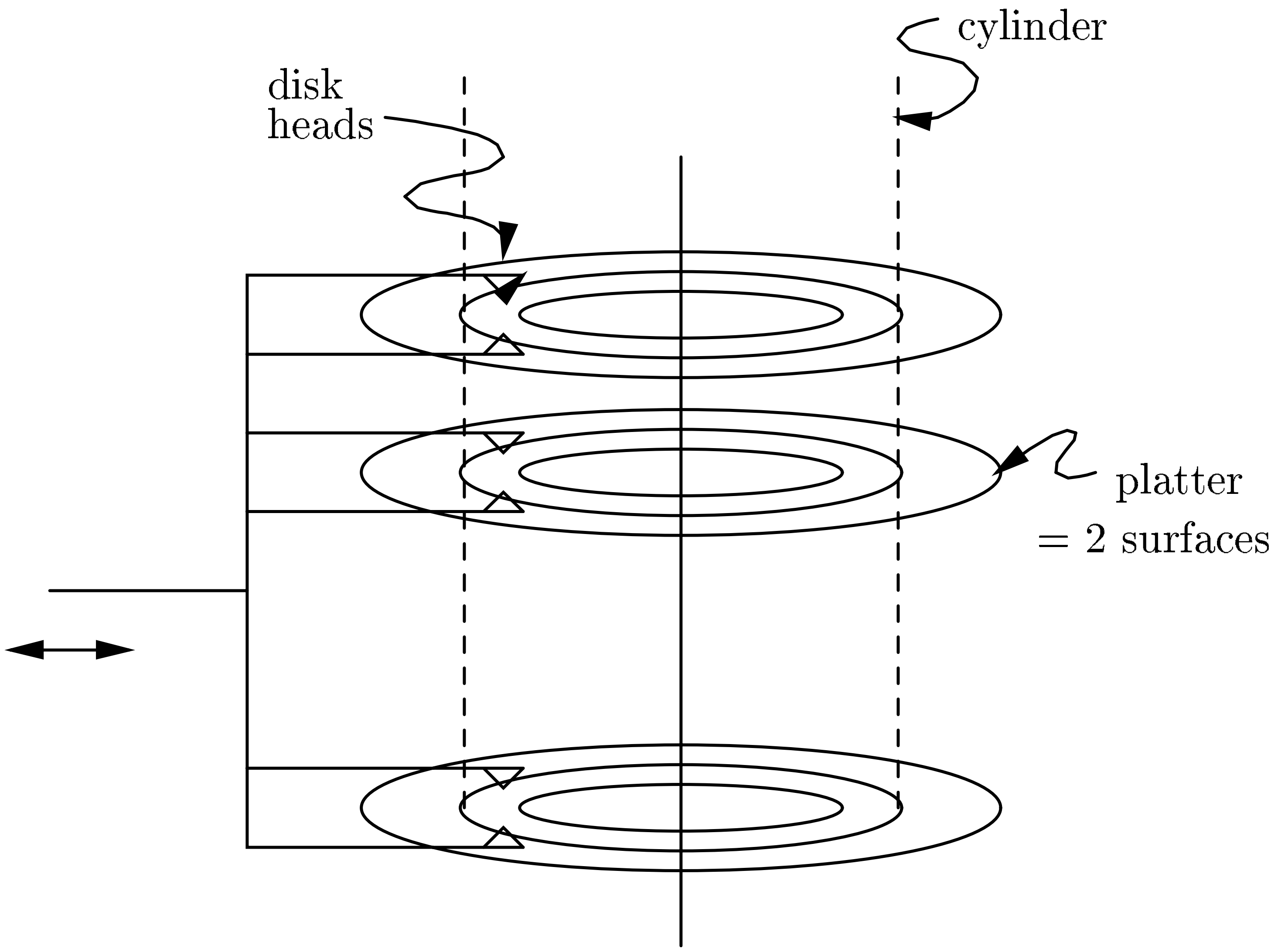
Database System Implementation, H. Garcia-Molina, J. Ullman, and J. Widom, Prentice-Hall, 2000.

The Memory Hierarchy (cont.)

- **C**ache: fastest. Access time: nanoseconds (10^{-9}), size: perhaps 1Mb (10^{6}).
- **M**ain memory. Access time: under a microsecond (10^{-6}), size: perhaps 128Mb.
- **S**econdary storage: Typically magnetic disks. Access times in milliseconds (unit of read/write = *block* or *page*, typically 4Kb). Capacities in gigabytes (10^{9}).
- **T**ertiary storage. Access times in seconds, capacities in terabytes (10^{12}).

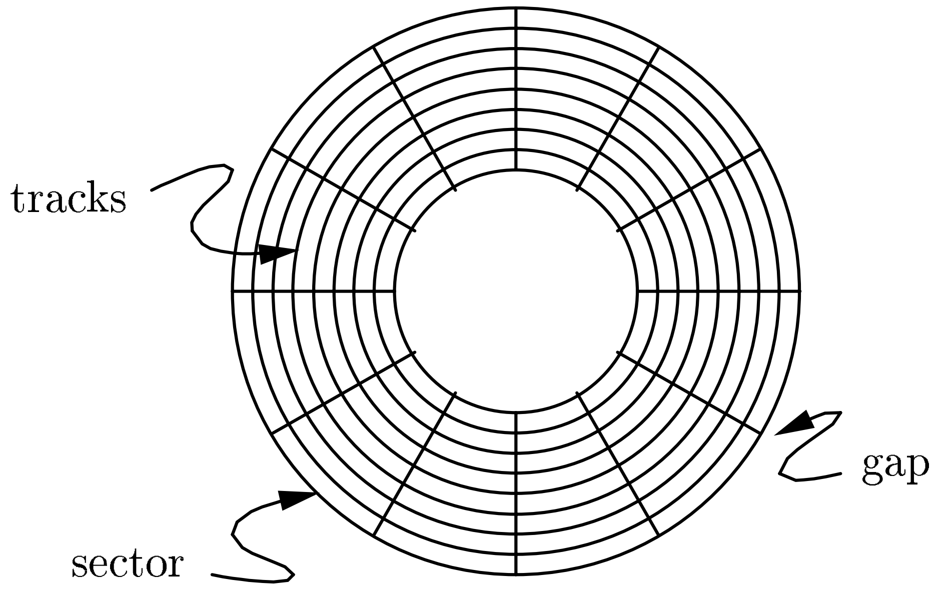
  Most common: racks of tapes, CD-ROMs.

Hard Disk Physical Structure



Database System Implementation, H. Garcia-Molina, J. Ullman, and J. Widom, Prentice-Hall, 2000.

Hard Disk Logical Structure



Database System Implementation, H. Garcia-Molina, J. Ullman, and J. Widom, Prentice-Hall, 2000.

Surfaces covered with concentric **T**racks.

- Tracks at a common radius = **C**ylinder. Important because all data of a cylinder can be read quickly, without moving the heads.
- Floppy disk: 40 cylinders; typical magnetic disk: 10,000 cylinders (optical disk: several times that).

Hard Disk Logical Structure (cont.)

- Tracks divided into **S**ectors by unmagnetized gaps.
- Sector is indivisible unit for read/write and error detection.
  Usually one sector = 512 Bytes.
- A bad sector is “cancelled” by the disk controller, so it is never used.
- Sectors are grouped into **B**locks: logical units for read/write.
  Typical one block $= 4KB = 8 \times 512$ Bytes, i.e., 8 sectors.

Accessing Disk Blocks

The following events occur when an application requests a particular tuple T_1:

1. The DBMS determines which of the blocks contains T_1;
2. The block is read into a cache buffer (if not already there);
3. T_1 ready to be read from the application.

In particular, modifying a block requires:

1. Read the block into a cache buffer;
2. Modify the block there;
3. Write the block back to disk.

Efficient Use of Disk

Data in DBMS applications do *not* fit in main memory. Algorithms should limit the number of disk accesses.

- **D**isk I/O. Read or write of a block is very expensive compared with what is likely to be done with the block once it arrives in main memory.
  - Perhaps 1,000,000 machine instructions in the same time a disk I/O is performed.
- **I**/O Model of Computation. The number of block reads and writes is a good approximation of the running time of the algorithm and should be minimized.

I/O Strategies

Here are some techniques that make secondary-memory algorithms more efficient.

Group related blocks by cylinder.

- If we are going to read or write blocks in a known order, place them by cylinder, so once we reach that cylinder we can read block after block.

Prefetching.

- Load in cache buffers the blocks that are needed in advance. Similarly, keep output blocks in cache buffers until it is convenient to write them to disk. LRU (Least Recently Used) technique: The block that has not been referenced for the longest time is written back to the disk.

Summary

- Introduction to Databases.
- Storing a Database.
  - Storing a toy database;
  - An overview of secondary storage devices;
  - **P**hysical layout of data;
  - Efficient techniques for storing DB Relations.

Physical Layout of Data

Tuples and relations are stored in secondary storages following the block model.

MOVIE

TITLE	YEAR	LENGTH	FILMTYPE	STUDIONAME
Star Wars	1977	124	color	Fox
Mighty Ducks	1991	104	color	Disney
Wayne's World	1992	95	color	Paramount

- Attribute. Sequence of bytes representing the value of an attribute in a tuple, called **Field**.
- Tuple. Sequence of bytes divided into fields, called **Record**.
- Relation. A collection of records that form the extent of a relation is stored in a **File** as a collection of blocks.

Physical Layout of Data (Cont.)

We need to discuss the following issues:

1. How do we represent datatypes as fields?
2. How do we represent tuples as records?
3. How do we represent collections of records in blocks of memory?
4. How do we cope with record sizes that maybe different for different tuples?
5. How do we represent relations as collections of blocks?
6. What happens if the size of a record changes because of an update?

Attributes: How we Represent Datatypes as Fields (1)

The representation of an attribute depends by the associated SQL datatype. In general, datatypes can be represented by a *fixed length field* made by a sequence of bytes.

- *Integers*, *Reals*: 2-8 byte fields.
- *Fixed-length Character Strings*, `CHAR(n)`: array of n bytes (possibly filled with *pad* characters.) Ex. of a `CHAR(10)` and `cat` as a string:

  `cat`$\perp\perp\perp\perp\perp\perp\perp$

- *Variable-length Character Strings*, `VARCHAR(n)`: array of $n+1$ bytes, regardless of how effectively long the string is. Two methods:
  - *Lenght plus Content*. Ex. of a `CHAR(10)` and `cat` as a string:

    3`cat`

  - *Null Terminated String*. Ex. of a `CHAR(10)` and `cat` as a string:

    `cat`$\perp$

    and the remaining positions are irrelevant.

Attributes (Cont.)

- *Date*: array of fixed length (SQL2 uses arrays of 10 bytes).
- *Sequence of bits*, `BIT(n)`: each 8 bits are packed in 1 byte. Unused bits are put to 0.

Fixed-Length Records: How we Represent Tuples as Records (2)

Tuples are represented by records. When all fields of the record have a *fixed length* we simply concatenate the fields.

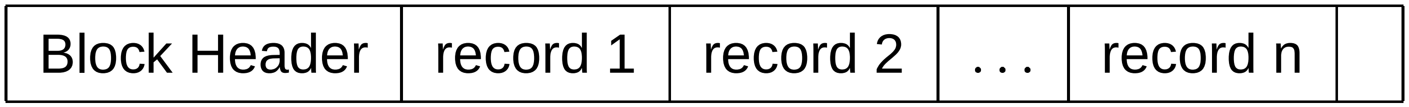
```
CREATE TABLE  EMPLOYEE
        ( INITIAL CHAR,
          LAST-NAME VARCHAR(15),
          ADDRESS VARCHAR(255),
          SSN CHAR(5),
          BDATE DATE,
          EMP-ID INT,
          SALARY DECIMAL(10,2) );
```

Each record of type `Employee` takes $1 + 16 + 256 + 5 + 10 + 4 + 8 = 300$ bytes.

To locate a single record's field the associated schema file is consulted.

Storing Collections of Fixed-Length Records: How we Represent Collections of Records in Blocks of Memory (3)

- Fixed-Length Records representing tuples of a relation are packed in blocks following the block header.
- The Bloch Header contains the following information:
  - Info about wich relation the records of this block belong to;
  - Pointers to other blocks storing tuples of the same relation;
  - Offset for each record in the block;
  - Timestamp indicating the block's last update;
  - etc...



Variable-Length Records

1. Fields that vary in length.

   We can choose to allocate exact space for storing a VARCHAR string.

2. Very large fields.

   Fields can contain pictures, movies or sounds: These fields are very large and they do not fit within blocks.

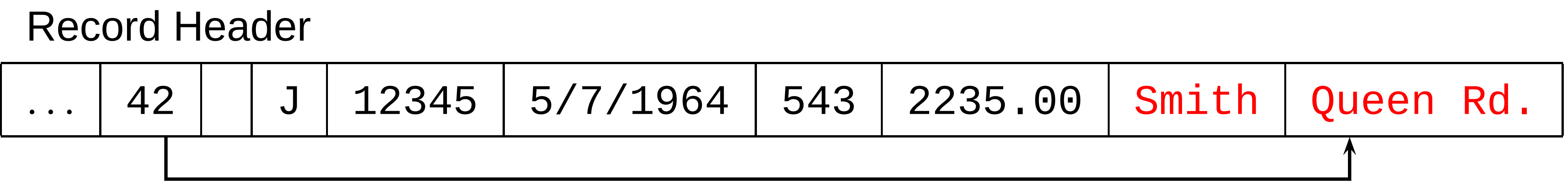
Records with Variable-Length Fields

Put the Fixed-Length fields before the Variable-Length fields. The record header contains additional information:

1. The length of the whole record;
2. Pointers (i.e., an offset table) to the beginning of each Variable-Length field (excluding the first of them).

Example.

`EMPLOYEE(INITIAL,LAST-NAME,ADDRESS,SSN,BDATE,EMP-ID,SALARY`



Space allocated for this record with variable-length fields:

$$1 + 5 + 9 + 5 + 10 + 4 + 8 = 42 \text{ bytes (instead of 300 bytes).}$$

Records with Large Fields

Problem: The record does not fit in one block.

Solution: *Spanned Records*, records are split between two or more blocks. *Record Fragment*: portion of the spanned record allocated in one block.

The record (fragment) header contains additional information:

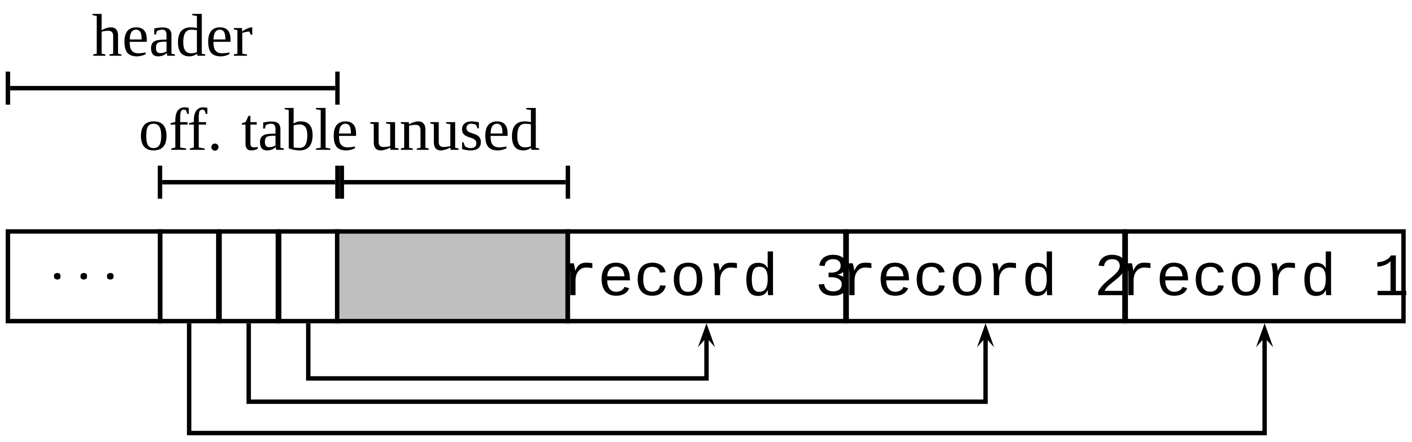
1. A bit indicating if the record is a fragment;
2. If it is a fragment, then two pointers for the next and previous fragment for the same record (the first fragment has a null previous pointer, while the last fragment has a null successive pointer).

Spanned Records are also used for storing records slightly larger then half a block for saving space.

Relations with Variable-Length Records: How we Cope with Records with Different Sizes (4)

The header of the block contains an *Offset Table* holding the offset of the records within the block.

The records are stored starting from the end of the block, while the offset table grows from the front end.



Since we don't know in advance how many records the block will hold this solution allows to dynamically allocate space for the offset table.

Summary

- Introduction to Databases.
- **S**toring a Database.
  - Storing a toy database;
  - An overview of secondary storage devices;
  - Physical layout of data;
  - **E**fficient techniques for storing DB Relations.

Storing DB Relations: How we Represent Relations as Collections of Blocks (5)

- Each Relation is stored in a separate *file of records*.
- We use the term **Data File** to refer to the file of records which stores a given Relation.
- Data files can have the following structures:
  1. **Heap File Organization.** Records are stored anywhere in the file where there is space.
  2. **Sequential File Organization.** Records are stored in sequential order, according to the value of some field(s).

NOTE: In the following we consider the Heap File Organization. The Sequential File Organization will be discussed in the next Lecture on Indexing Techniques.

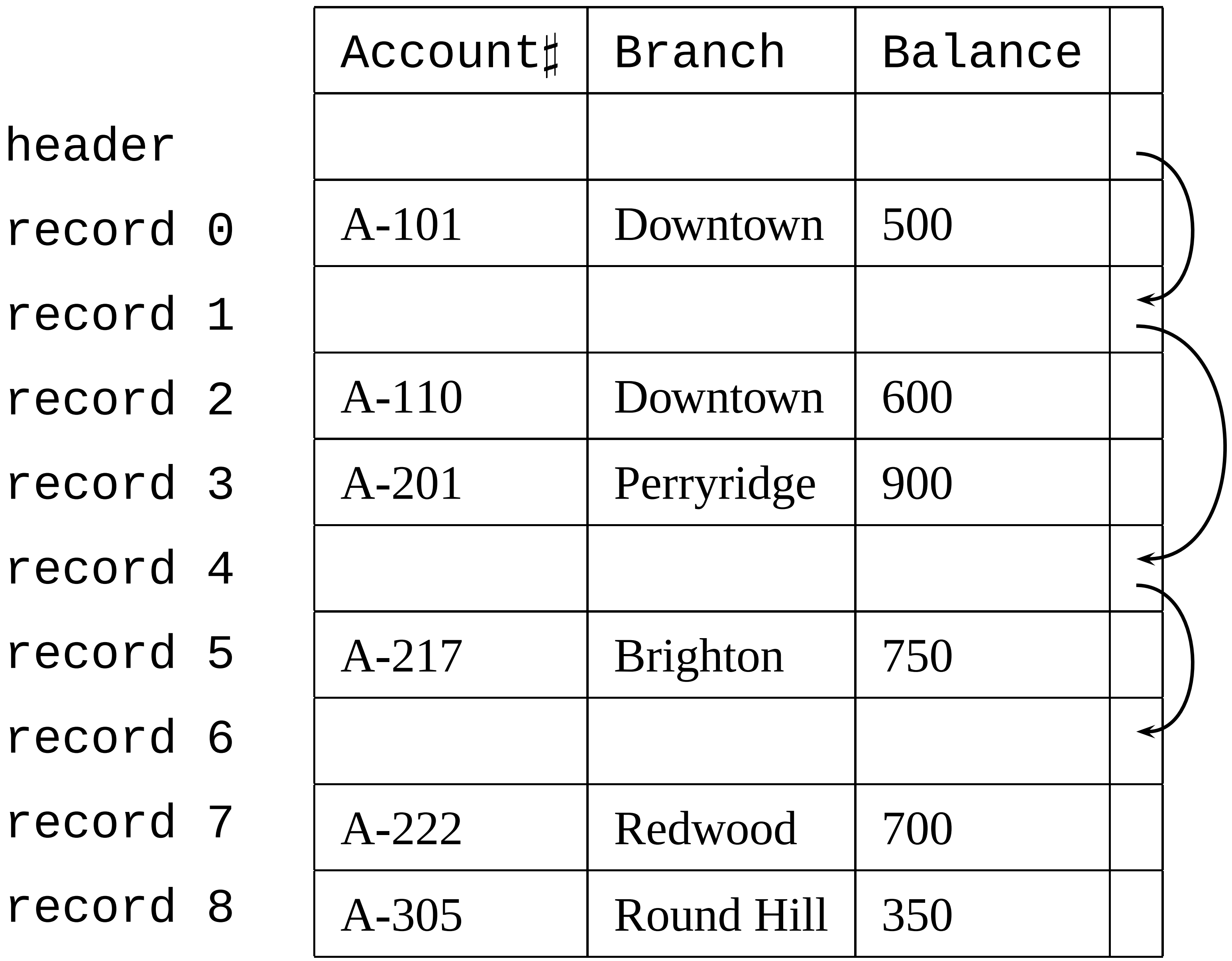
Update, Deletion and Insertion in Heap Files with Fixed-Length Records

Updating an existing record is harmless for Fixed-Length Records.

We need to optimize both *Time* and *Space* for Deletion/Insertion into a heap file.

- **Technique 1.** When a record is deleted the last record is moved into the space occupied by the deleted record.
- **Technique 2.** Usually, insertion is more frequent than deletion:
  1. Wait for a new insertion to occupy the deleted space.
  2. Maintain a list of pointers to the free space. Each block contains a *Block header* with a pointer to the first free space, while this first record contains the pointer to the next free space, and so on.
  3. A new record is inserted in the free space pointed to by the header. The header pointer is changed to point to the next free space.
  4. If the header pointer is null the record is simply appended.

Deletion and Insertion in Heap Files. An Example



	Account♯	Branch	Balance	
header				
record 0	A-101	Downtown	500	
record 1				
record 2	A-110	Downtown	600	
record 3	A-201	Perryridge	900	
record 4				
record 5	A-217	Brighton	750	
record 6				
record 7	A-222	Redwood	700	
record 8	A-305	Round Hill	350	

Deletion and Insertion in Heap Files with Variable-Length Records

- **Deletion:** Free the space for both the record and its pointer, compact both the remaining records andt the offset table so that there is always a unique unused region in the center.
- **Insertion:** Find a block with enough space, or get a new block, and put the record there introducing a new pointer in the offset table.

Updates in Heap Files with Variable-Length Records

We have problems similar to both insertion and deletion.

1. If the updated record is longer than the old version we require to either sliding records around the block or moving the record into another block. In both cases the offset table must be updated.

2. If the updated record shrinks we have the opportunity to free some space as with the deletion case.

Summary of Lecture I

- Introduction to Databases.
- Storing a Database.
  - Storing a toy database;
  - An overview of secondary storage devices;
  - Physical layout of data;
  - Efficient techniques for storing DB Relations.