File Organization

Database System Concepts, 5th Ed.

File Organization

- **File:** A file is logically a **sequence of records**, where
  - a record is a sequence of fields;
  - the file header contains information about the file.
- Usually, a relational table is mapped to a file and a tuple to a record.
- A DBMS has the choice to
  - Use the file system of the operating system (reuse code)
  - Manage disk space on its own (OS independent, better optimization, e.g., Oracle)
- Two approaches to represent files (or records) on disk blocks:
  - **Fixed length** records
  - **Variable length** records

Fixed-Length Records

- Suppose we have a table that has the following organization:

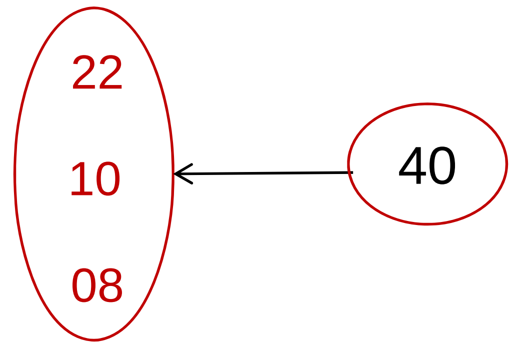
type deposit = record

branch-name : char(22);

account-number : char(10);

balance : real;

End



- Assumptions: If each character occupies 1 byte and a real occupies 8 bytes, then this record occupies 40 bytes. If the first record occupies the first 40 bytes and the second record occupies the second 40 bytes.

Fixed-Length Records

Problems with this approach are:

- **Difficult to delete a record**, because there is no way to identify deleted record. **How it can be used for another record ?**
- If fix the size then it may possible some records will cross block boundaries and it would require two block access to read or write such a record.

Fixed-Length Records

- Store record i starting from byte $n * (i - 1)$, where n is the size of each record.
- Record access is simple but records may cross blocks
- Deletion of record i is more complicated. Several alternatives exist:
  - Move records $i + 1, \ldots, n$ to $i, \ldots, n - 1$
  - Move record n to i
  - Do not move records, but link all free records on a *free list*

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 2	A-215	Mianus	700
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

Fixed-Length Records

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 2	A-215	Mianus	700
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

Record 2 Deleted and All Records Moved

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

Fixed-Length Records

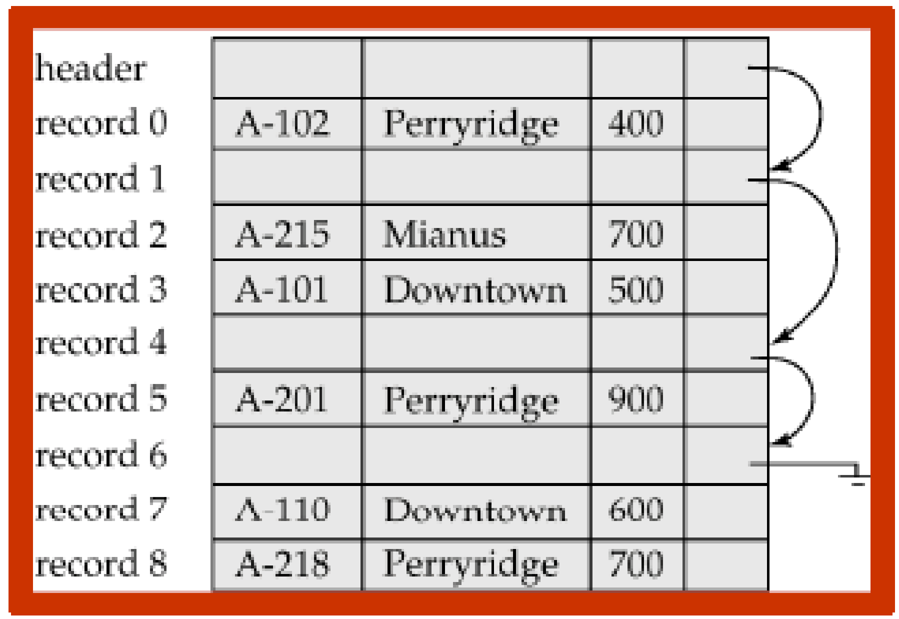
record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 2	A-215	Mianus	700
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

Record 2 deleted and Final Record Moved

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 8	A-218	Perryridge	700
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600

Fixed-Length Records

- **Free list**
  - Store the address of the first deleted record in the file header.
  - Use this first record to store the address of the second deleted record, and so on
  - Note the additional field to store pointers!
- More space efficient representation is possible
  - Hint: No pointers are stored in records that contain data.



Variable-Length Records

- Variable-length records arise in database systems in several ways:
  - Storage of multiple record types in a file.
  - Record types that allow variable lengths for one or more fields.
  - Record types that allow repeating fields (used in some older data models).
- Different methods to represent variable-length records
  - Byte string representation
  - Slotted page structure
  - Fixed-length representation

Variable-Length Records

- **Example**: Bank application with an *account* relation, where one variable-length record is used for each branch name and all the account information for that branch.

```
Type account-list =
        record
          branche-name: char(22);
          account-info: array[1..n] of
                          record
                            account-number: char(10);
                            balance: real;
                          end
        end
```

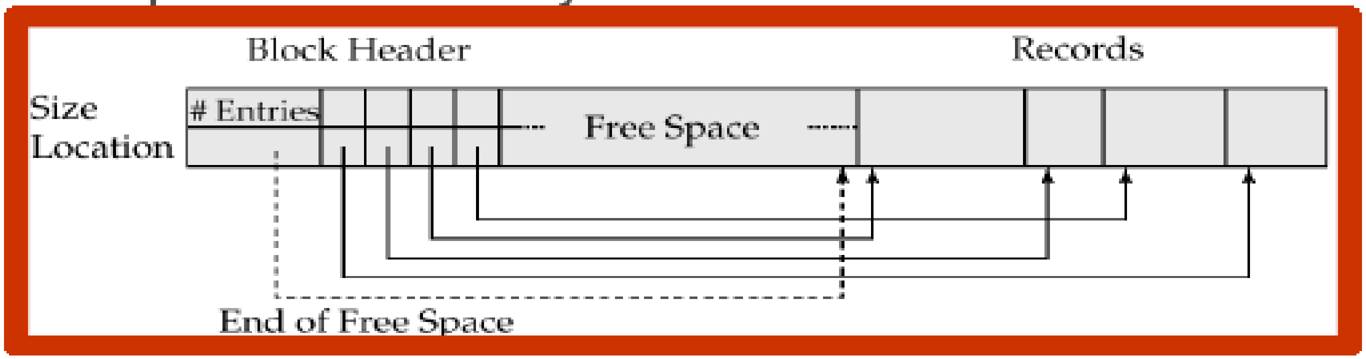
Variable-Length Records

- **Byte string representation**
  - Attach an *end-of-record (^) control character to the end of* each record
  - Difficulty with deletion and growth (how to reuse deleted space?)
  - No space, in general, for a record to grow

0	Perryridge	A-102	400	A-201	900	A-218	700	⊥
1	Round Hill	A-305	350	⊥				
2	Mianus	A-215	700	⊥				
3	Downtown	A-101	500	A-110	600	⊥		
4	Redwood	A-222	700	⊥				
5	Brighton	A-217	750	⊥				

Variable-Length Records

- Slotted page structure
  - Slotted page header contains:
    - number of record entries
    - end of free space in the block
    - location and size of each record
  - Records can be moved around in a page to keep them contiguous with no empty space between them; entry in the header must be updated.
  - Pointers should not point directly to record — instead they should point to the entry for the record in header.



Variable-Length Records

- Use one or more fixed length records:
  - **reserved space**
  - **pointers**
- **Reserved space –** can use fixed-length records of a known maximum length; unused space in shorter records filled with a null or end-of-record symbol.
  - Disadvantage: useful when most of records are of length near to maximum otherwise wastage of space

0	Perryridge	A-102	400	A-201	900	A-218	700
1	Round Hill	A-305	350	⊥	⊥	⊥	⊥
2	Mianus	A-215	700	⊥	⊥	⊥	⊥
3	Downtown	A-101	500	A-110	600	⊥	⊥
4	Redwood	A-222	700	⊥	⊥	⊥	⊥
5	Brighton	A-217	750	⊥	⊥	⊥	⊥

Variable-Length Records

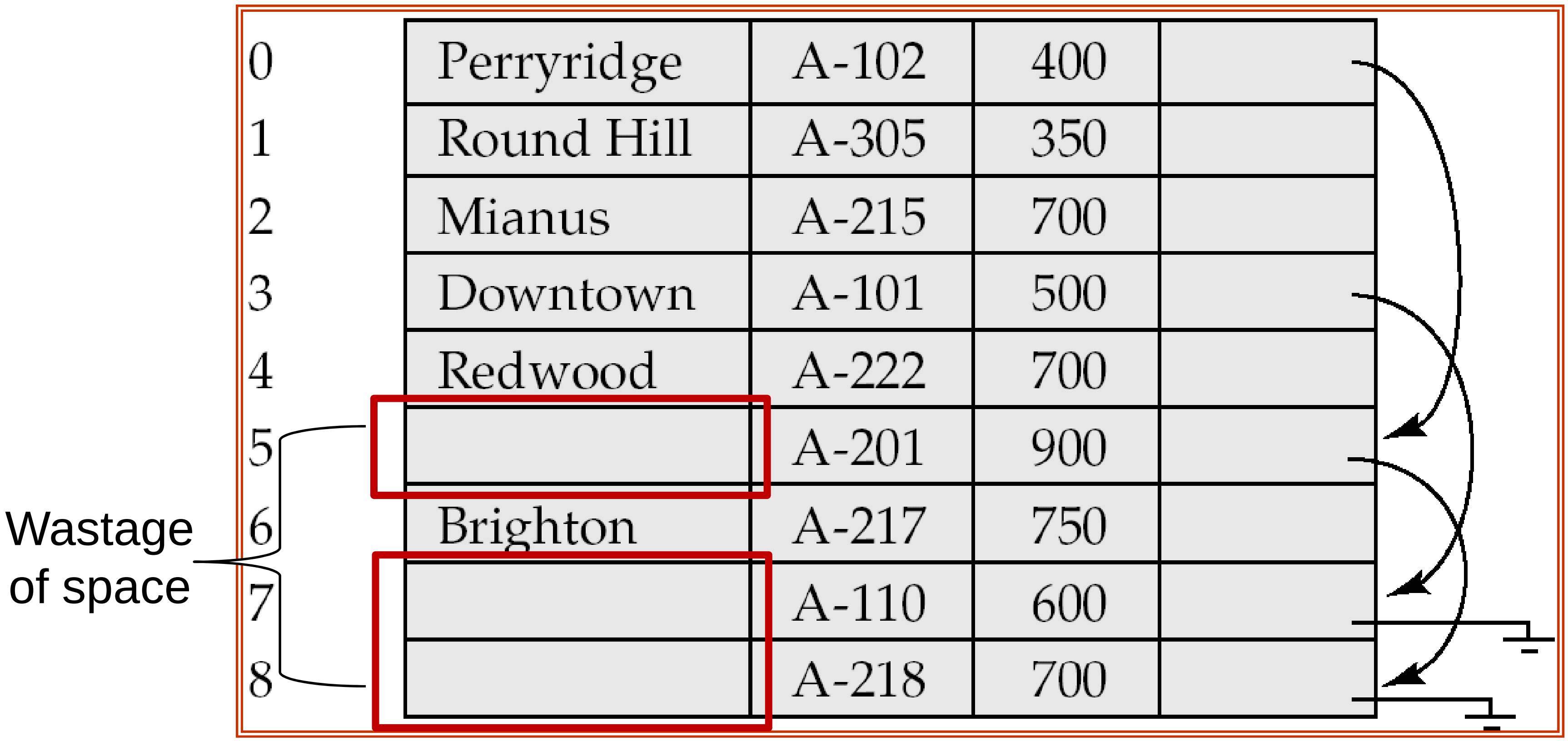
- **Pointer method**
  - A variable-length record is represented by a list of fixed-length records, chained together via pointers.
  - Can be used even if the maximum record length is not known

0	Perryridge	A-102	400	
1	Round Hill	A-305	350	
2	Mianus	A-215	700	
3	Downtown	A-101	500	
4	Redwood	A-222	700	
5		A-201	900	
6	Brighton	A-217	750	
7		A-110	600	
8		A-218	700	

Variable-Length Records

- **Pointer method**

Disadvantage to pointer structure; space is wasted in all records except the first in a chain.



Variable-Length Records

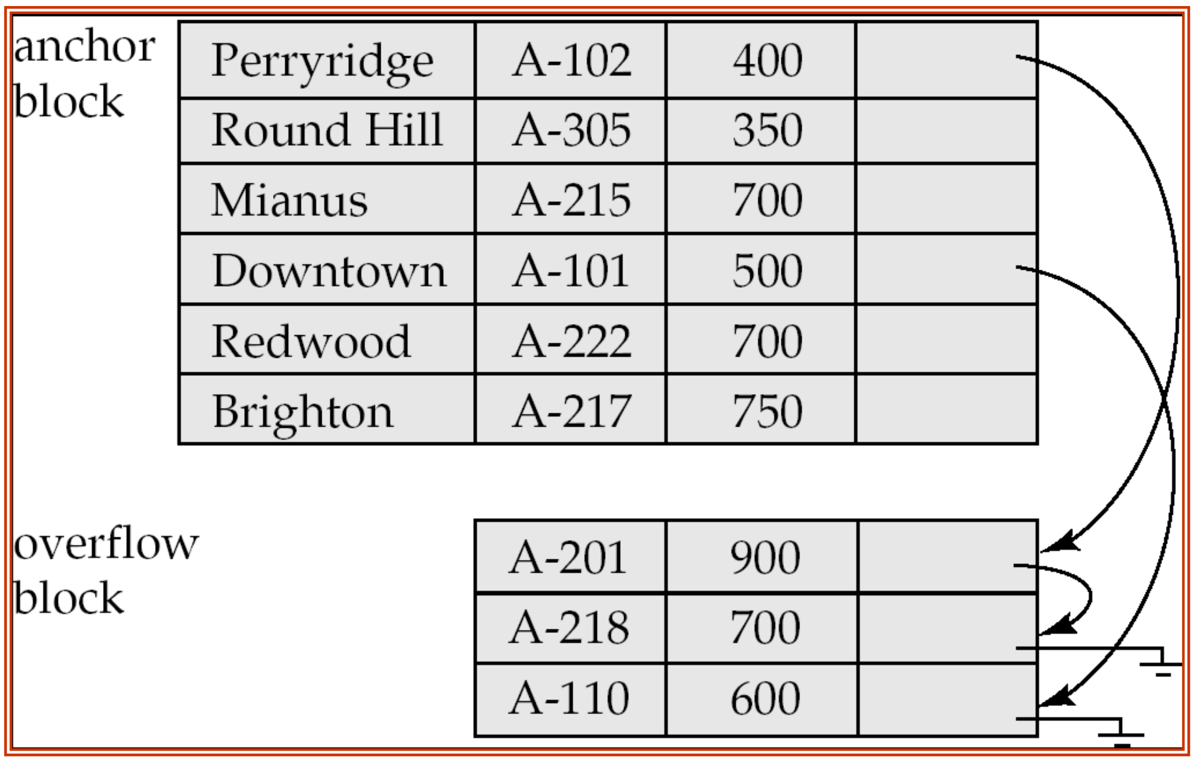
- **Pointer method**

Disadvantage to pointer structure; space is wasted in all records except the first in a chain.

❑ Solution is to allow two kinds of block in file:

Anchor block – contains the first records of chain

Overflow block – contains records other than those that are the first records of chairs.



Organization of Records in Files

- **Heap** – a record can be placed anywhere in the file where there is space; there is no ordering in the file.
- **Sequential** – store records in sequential order, based on the value of the search key of each record
- **Hashing** – a hash function computed on some attribute of each record; the result specifies in which block of the file the record should be placed. Records of each relation may be stored in a separate file.
- In a **clustering file organization** records of several different relations can be stored in the same file
  - Motivation: store related records on the same block to minimize I/O

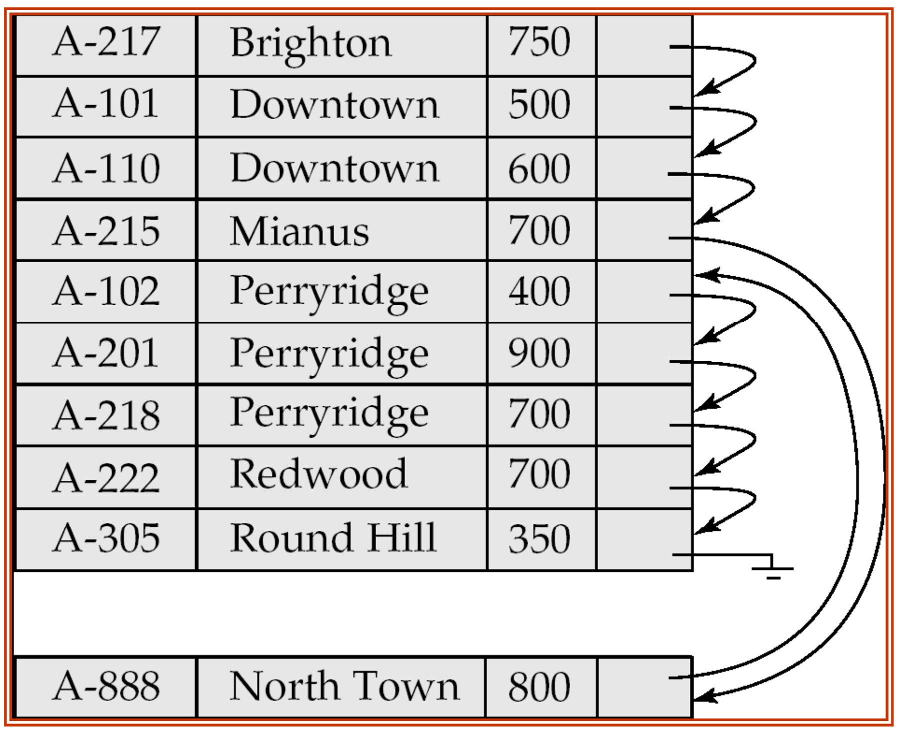
Sequential File Organization

- Suitable for applications that require sequential processing of the entire file
- The records in the file are ordered by a search-key
- **Example: account (account-number, branch-name, balance)**

A-217	Brighton	750	
A-101	Downtown	500	
A-110	Downtown	600	
A-215	Mianus	700	
A-102	Perryridge	400	
A-201	Perryridge	900	
A-218	Perryridge	700	
A-222	Redwood	700	
A-305	Round Hill	350	

Sequential File Organization (Cont.)

- Deletion – use pointer chains
- Insertion –locate the position where the record is to be inserted
  - if there is free space insert there
  - if no free space, insert the record in an overflow block
  - In either case, pointer chain must be updated
- Need to reorganize the file from time to time to restore sequential order



Clustering File Organization

Store several relations in one file using a **multitable clustering** file organization.

(instead of each relation in a separate file)

customer_name	*account_number*
Hayes	A-102
Hayes	A-220
Hayes	A-503
Turner	A-305

customer_name	*customer_street*	*customer_city*
Hayes	Main	Brooklyn
Turner	Putnam	Stamford

Multitable Clustering File Organization (cont.)

Multitable clustering organization of *customer* and *depositor:*

Hayes	Main	Brooklyn
Hayes	A-102	
Hayes	A-220	
Hayes	A-503	
Turner	Putnam	Stamford
Turner	A-305	

- good for queries involving ***depositor*** $\bowtie$ ***customer***, and for queries involving one single customer and his accounts
- bad for queries involving only customer
- results in variable size records
- Can add pointer chains to link records of a particular relation

Data Dictionary Storage

Data dictionary (also called system catalog) stores metadata; that is, data about data, such as

- Information about relations
  - names of relations
  - names and types of attributes of each relation
  - names and definitions of views
  - integrity constraints
- User and accounting information, including passwords
- Statistical and descriptive data
  - number of tuples in each relation
- Physical file organization information
  - How relation is stored (sequential/hash/…)
  - Physical location of relation
- Information about indices

Data Dictionary Storage (Cont.)

- Catalog structure
  - Relational representation on disk
  - specialized data structures designed for efficient access, in memory
- A possible catalog representation:

Relation_metadata = (*<u>relation_name</u>, number_of_attributes, storage_organization, location*)

Attribute_metadata = (*<u>attribute_name, relation_name</u>, domain_type, position, length*)

User_metadata = (*<u>user_name</u>, encrypted_password, group*)

Index_metadata = (*<u>index_name, relation_name</u>, index_type, index_attributes*)

View_metadata = (*<u>view_name</u>, definition*)