Database Management Systems 2010/11

– Chapter 2: Storage and File Structure –

J. Gamper

- Overview of Physical Storage Media
- Magnetic Disks
- Storage Access
- File Organization
- Organization of Records in Files
- Data-Dictionary Storage

These slides were developed by:
– Michael Böhlen, University of Zurich, Switzerland
– Johann Gamper, University of Bozen-Bolzano, Italy

Physical Storage Media/1

- Several types of storage media exist in computer systems and must be considered when studying DBMS
- **Classification** of Storage media
  - **Speed** with which data can be accessed
  - **Cost** per unit of data
  - **Reliability**
    - data loss on power failure or system crash
    - physical failure of the storage device
  - **Volatile** vs. **non-volatile** storage
    - Volatile storage: Loses contents when power is switched off
    - Non-Volatile storage: Contents persist even when power is switched off

Physical Storage Media/2

- **Cache**
  - Volatile
  - Fastest and most costly form of storage
  - Managed by the computer system hardware
- **Main memory**
  - Volatile
  - Fast access (10s to 100s of nanosecs; 1 nanosec $= 10^{-9}$ secs)
  - Generally too small (or too expensive) to store the entire DB
    - Capacities of up to a few Gigabytes widely used currently
    - Capacities have gone up and per-byte costs have decreased steadily and rapidly (roughly factor of 2 every 2 to 3 years)

Physical Storage Media/3

- **Flash memory**
  - Non-volatile
  - Reads are roughly as fast as main memory
  - Writes are slow (few microseconds) and more complicated
    - Data cannot be overwritten, but need first to be erased
    - Only a limited number of write/erase cycles is supported
  - Cost per unit of storage roughly similar to main memory
  - Widely used in embedded devices such as digital cameras
  - Also known as EEPROM (Electrically Erasable Programmable Read-Only Memory)

Physical Storage Media/4

- **Magnetic disk**
  - Non-volatile
  - Data is stored on spinning disk, and read/written magnetically
  - Much slower access than main memory
  - Much larger capacities than main memory; typically up to roughly 100 GB
    - Growing rapidly with technology improvements (factor 2 to 3 every 2 years)
  - Primary medium for the long-term storage of data
    - Data must be moved from disk to main memory for access, and written back for storage
  - Direct data access, i.e., data on disk can be read in any order (unlike magnetic tape)
  - Hard disks vs. floppy disks

Physical Storage Media/5

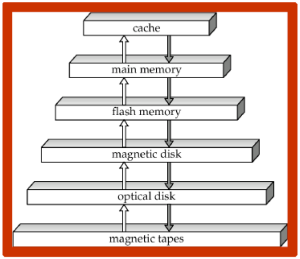
- **Optical disk**
  - Non-volatile
  - Data is read optically from a spinning disk using a laser
  - Reads and writes are slower than with magnetic disk
  - Different types
    - CD-ROM (640 MB) and DVD (4.7 to 17 GB) most popular forms
    - Write-one, read-many (WORM) optical disks used for archival storage
    - Multiple write versions also available (CD-RW, DVD-RW, and DVD-RAM)
  - Juke-box systems, with large numbers of removable disks, a few drives, and a mechanism for automatic loading/unloading of disks available for storing large volumes of data

Physical Storage Media/6

- **Tape storage**
  - Non-volatile
  - Much slower than disk due to sequential access only
  - Very high capacity (40 to 300 GB tapes available)
  - Used primarily for backup and for archival data
  - Tape can be removed from drive storage costs much cheaper than disk
  - Tape juke-boxes available for storing massive amounts of data
    - Hundreds of terabytes (1 terabyte $= 10^{12}$ bytes) to even a petabyte (1 petabyte $= 10^{15}$ bytes)

Physical Storage Media/6

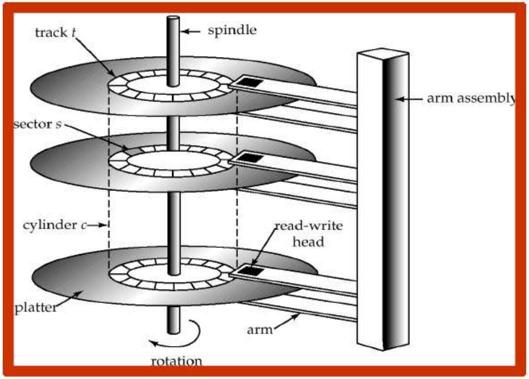
- The storage media can be organized in a **hierarchy** according to their **speed** and **cost**
- **Primary storage**: fastest media, but volatile
  - e.g., cache, main memory
- **Secondary storage**: non-volatile, moderately fast access
  - e.g., flash memory, magnetic disks
  - also called on-line storage
- **Tertiary storage**: non-volatile, slow access time
  - e.g., magnetic tape, optical storage
  - also called off-line storage



- DBMS must explicitly deal with storage media at all levels of the hierarchy

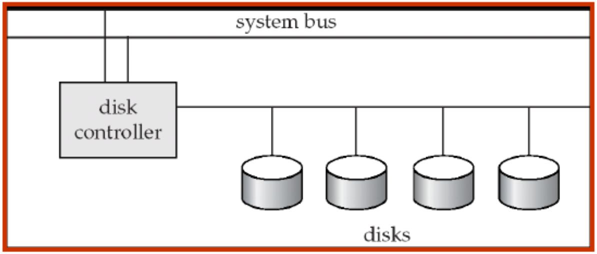
Magnetic Hard Disks/1

- Most DBs are stored on magnetic disks for the following reasons:
  - Generally, DBs are too large to fit entirely in main memory
  - Data on disks is non-volatile
  - Disk storage is cheaper than main memory
- Simplified and schematic structure of a magnetic disk



Magnetic Hard Disks/2

- **Disk controller:** Interface between the computer system and the HW of the disk drive. Performs the following tasks:
  - Translates high-level commands, such as read or write a sector, into actions of the disk HW, such as moving the disk arm or reading/writing the ector.
  - Adds a checksum to each sector
  - Ensures successful writing by reading back a sector after writing it



Magnetic Hard Disks/3

- **Performance measures** of hard disks
  - **Access time:** the time it takes from when a read or write request is issued to when the data transfer begins; is composed of:
    - **Seek time:** time it takes to reposition the arm over the correct track
    - Avg. seek time is 1/2 the worst case seek time (2-10 ms on typical disks)
    - **Rotational latency:** time it takes for the sector to be accessed to appear under the head
    - Avg. seek time is 1/2 the worst case seek time (e.g., 4-11 ms for 5400-15000 r.p.m.)
  - **Data-transfer rate:** rate at which data can be retrieved from or stored to disk (e.g., 25-100 MB/s)
    - Multiple disks may share a single controller
  - **Mean time to failure (MTTF)**: average time the disk is expected to run continuously without any failure
    - Typically several years

Blocks and Storage Access/1

- **Block:** a logical unit consisting of a fixed number of contiguous sectors from a single track
- A block is a unit of storage allocation and data transfer
  - Data between disk and main memory is transferred in blocks
  - A database file is partitioned into fixed-length blocks
  - Typical size ranges from 4 to 16 kilobytes
    - Smaller blocks: more transfers from disk
    - Larger blocks: more space wasted due to partially filled blocks

Blocks and Storage Access/2

- One of the **major goals** of a DBMS is to make the transfer of data between disk and main memory as efficient as possible.
- Two ways:
  1. **Optimize/Minimize** the disk-block access time
     - Disk-arm-scheduling
     - Appropriate file organization
     - Write buffers
     - Log disks
  2. Keep as many blocks as possible in memory ($\rightarrow$ **buffer manager**),thus minimize the number of block transfers

Optimization of Disk-Block Access/1

- **Disk-arm-scheduling:** Order pending accesses to tracks so that disk arm movement is minimized.
- **Elevator** algorithm
  1. Disk controller orders the requests by track in one direction (from outer to inner or vice versa)
  2. Move disk arm in the direction of the ordering and process the next request until no more requests in that direction exist
  3. Reverse the direction and go to step 1

Optimization of Disk-Block Access/2

- **File organization:** Optimize block access time by organizing the blocks to correspond to how data will be accessed, e.g., store related information on the same or nearby cylinders.
- Files may get **fragmented** over time
  - e.g., if data is inserted to or deleted from the file
  - Or free blocks on disk are scattered, and newly created file has its blocks scattered over the disk
  - Sequential access to a fragmented file results in increased disk arm movement
- Some systems have utilities to **defragment** the file system, in order to speed up file access

Optimization of Disk-Block Access/3

- **Non-volatile write buffers:** Speed up disk writes by writing blocks to a non-volatile, battery backed up RAM or flash memory immediately; the controller then writes to disk whenever the disk has no other requests or request has been pending for some time.
  - Even if power fails, the data is safe.
  - Writes can be reordered to minimize disk arm movement.
  - Database operations that require data to be safely stored before continuing can continue immediately.

Optimization of Disk-Block Access/4

- **Log disk:** A disk devoted to write a sequential log of block updates.
  - Used exactly like non-volatile RAM
  - Writing to log disk is very fast since no seeks are required
  - No need for special hardware (NV-RAM), thus less expensive

Buffer Manager

- **Buffer:** Portion of main memory available to store copies of disk blocks (when they are transferred from disk).
- **Buffer Manager:** Subsystem of DBMS that is responsible for buffering disk blocks in main memory with the overall goal to **minimize** the number of disk accesses (similar to a virtual-memory manager of an OS).
- Programs call the buffer manager when they need a block from disk.
- Buffer manager **algorithm**
  1. If the block is already in the buffer:
     - The requesting program is given the address of the block in main memory.
  2. If the block is not in the buffer:
     - The buffer manager allocates space in the buffer for the new block (replacing/throwing out some other block, if required).
     - The block that is thrown out is written back to disk only if it was modified since the most recent time that it was written to/fetched from the disk.
     - Once space is allocated in the buffer, the buffer manager reads the block from the disk to the buffer, and passes the address of the block in memory to the requesting program.
- Different **buffer replacement** strategies/policies exist.

Buffer-Replacement Policies/1

- **LRU strategy:** Replace the block least recently used.
  - Idea: Use past pattern of block references to predict future references.
  - Applied successfully by most operating systems.
- **MRU strategy:** Replace the block most recently used.

- LRU can be a bad strategy in DBMS for certain access patterns involving repeated scans of data.
- Queries in DBs have well-defined access patterns (such as sequential scans), and a database system can use the information in a user's query to predict future references.

Buffer-Replacement Policies/2

- **Example:** compute a join $r \bowtie s$ with a nested loop evaluation strategy

```
for each tuple tr of r do
  for each tuple ts of s do
    if the tuples tr and ts match then ...
```

- Different access pattern for r and s
  - An r-block is no longer needed after the last tuple in the block is processed (even if it has been used recently), thus should be removed immediately.
  - An s-block is needed again after all other s-blocks are processed, thus MRU is the best strategy.
  - A mixed strategy with hints on replacement strategy provided by the query optimizer is preferable.

Buffer-Replacement Policies/3

- **Pinned block:** Memory block that is not allowed to be written back to disk (as long as it is pinned).
  - e.g., the *r*-block before processing the last tuple *tr*
- **Toss-immediate strategy:** Frees the space occupied by a block as soon as the final tuple of that block has been processed.
  - e.g., the *r*-block after processing the last tuple *tr*
- MRU + pinned block is the best choice for the nested-loop join

Buffer-Replacement Policies/4

- **B**uffer-replacement policies in DBMS can use various information
  - Queries have **well-defined access patterns** (e.g., sequential scan)
  - **Information in a query** to predict future references
  - **Statistical information** regarding the probability that a request will reference a particular relation.
    - e.g., the data dictionary is frequently accessed;
    - hence, keeping data-dictionary blocks in main memory buffer is a good heuristic

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 of code);
  - manage disk space on its own (OS independent, better optimization, e.g., Oracle)
- Two approaches to represent files on disk blocks:
  - **fixed length** records
  - **variable length** records

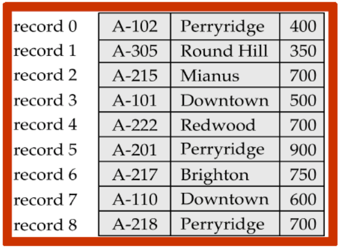
Fixed-Length Records/1

- **Example:** Consider a bank application with an account relation that stores the following account records:

```
type account =  record
                  account-number:  char(10);
                  branch-name:  char(22);
                  balance:  real;
                end
```

Fixed-Length Records/2

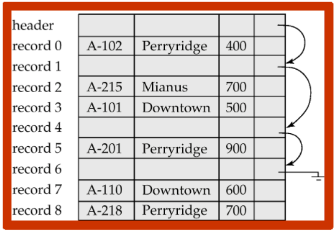
- **Fixed-length records** store record i starting from byte $n * (i-1)$, where n is the size of each record.
- Record access is simple but records may cross blocks.
- Deletion of record i is more complicated. Several alternatives exist:
  - move records $i+1, \ldots, n$ to $i, \ldots, n-1$;
  - move 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/3

- **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.



header				
record 0	A-102	Perryridge	400	
record 1				
record 2	A-215	Mianus	700	
record 3	A-101	Downtown	500	
record 4				
record 5	A-201	Perryridge	900	
record 6				
record 7	A-110	Downtown	600	
record 8	A-218	Perryridge	700	

Variable-Length Records/1

- **Variable-length records** arise in DBMS in several ways:
  - Storage of multiple record types in a file.
  - Records types that allow variable lengths for one or more fields.
  - Record types that allow repeating fields (used in some older models).

- Different methods to represent variable-length records
  1. Byte string representation
  2. Slotted page structure
  3. Fixed-length representation with reserved space
  4. Fixed-length representation with pointers (list representation)

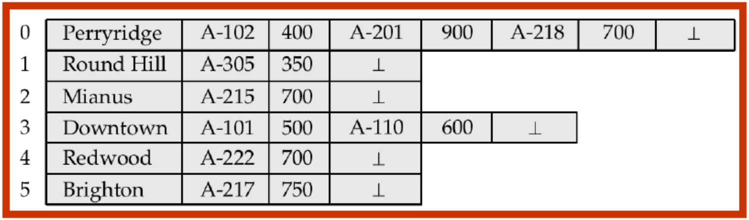
Variable-Length Records/2

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

Variable-Length Records/3

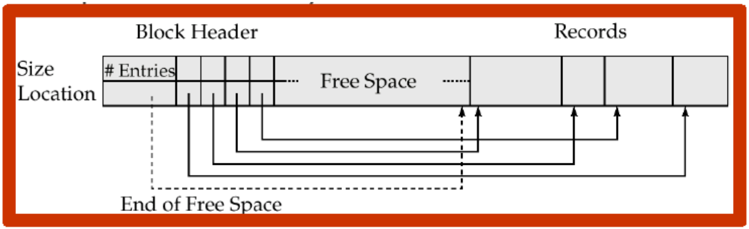
- **Byte string representation**
  - Attach an end-of-record (⊥) control character to the end of each record
  - Difficulty with deletion and growth:
    - it is not easy to reuse space occupied formerly by a deleted record;
    - no space, in general, for a record to grow.



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/4

- **Slotted page structure**
  - Variation of byte-string representation for organizing records within a block.
  - A page header contains information about the record organisation:
    - 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.

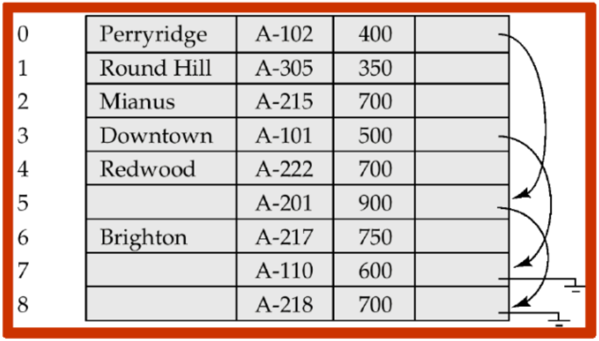
Variable-Length Records/5

- **Fixed-length representation with reserved space**
  - Use fixed-length records of a known maximum length
  - Unused space in shorter records is filled with a null or end-of-record symbol.

0	Perryridge	A-102	400	A-201	900	A-218	700
1	Round Hill	A-305	350	$\perp$	$\perp$	$\perp$	$\perp$
2	Mianus	A-215	700	$\perp$	$\perp$	$\perp$	$\perp$
3	Downtown	A-101	500	A-100	600	$\perp$	$\perp$
4	Redwood	A-222	700	$\perp$	$\perp$	$\perp$	$\perp$
5	brighton	A-217	750	$\perp$	$\perp$	$\perp$	$\perp$

Variable-Length Records/6

- **Fixed-length representation with pointer**
  - A variable-length record is represented by a list of fixed-length records that are chained together via pointers.
  - Can be used even if the maximum record length is not known.

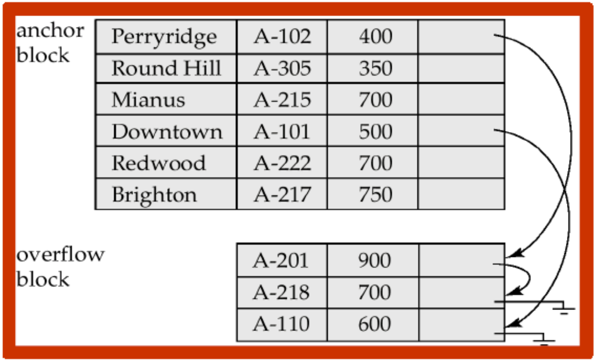


0	Perryridge	A-102	400	
1	Round Hill	A-305	350	
2	Mianus	A-215	700	
3	Downtown	A-101	500	
4	Redwood	A-222	700	
5		A-201	900	
6	Brighton	A-217	750	
7		A-110	600	
8		A-218	700	

- The main disadvantage of the pointer structure is that space is wasted in all records except the first in a chain.

Variable-Length Records/7

- **Alternative solution** (for fixed-length representation with pointer), which uses two different kinds of blocks in a file
  - **Anchor block:** Contains the first record of each chains.
  - **Overflow block:** Contains records other than those that are the first records of chains.



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

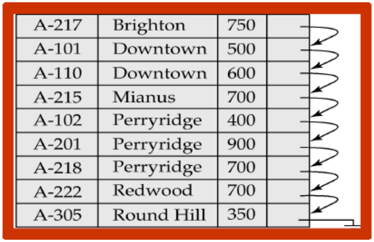
A-201	900	
A-218	700	
A-110	600	

Organization of Records in Files

- Different ways to logically organize records in a file:
  - **Heap file organisation:** A record can be placed anywhere in the file where there is space; there is no ordering in the file.
  - **Sequential file organization:** Store records in sequential order, based on the value of the search key of each record.
  - **Hashing file organization:** A hash function is computed on some attribute of each record; the result specifies in which block of the file the record is placed.
  - **Clustering file organization:** Records of several different relations can be stored in the same file
    - (Generally, each relation is stored in a separate file)

Sequential File Organization/1

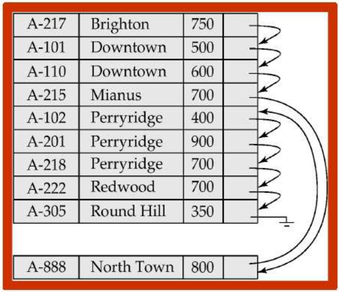
- **Sequential file (organization):** The records in the file are ordered by a search-key (one or more attributes)
  - Records are chained together by pointers
  - Suitable for applications that require sequential processing of the entire file
  - To be efficient, records should also be stored **physically** in search-key order (or close to it)
  - **Example:** Relation 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/

- It is difficult to maintain the physical order as records are inserted and deleted, since it is costly to move many records as the result of a single operation.
- Instead, the following strategy is applied:
  - Deletion: Use pointer chains to collect free records (**free list**)
  - Insertion:
    - Locate the position where the record is to be inserted
    - If there is free space insert there
    - If there is no free space, insert the record in an overflow block
  - Need to reorganize the file from time to time to restore (physical) sequential order



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
A-888	North Town	800

Clustering File Organization

- **Clustering file (organization):** Stores several relations in one file (instead of each relation in a separate file)
- Motivation: store related records on the same block to minimize I/O, e.g., for joins
- **Example:** Clustering file of the two relations *customer*(*name*, *street*, *city*) and *depositor*(*name*, *account-num*)

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

  - Good for queries involving a join of *depositor* and *customer*, and for queries involving one single customer and his account
  - Bad for queries involving only *customer*
  - Results in variable size records

Data Dictionary Storage/1

- **Data dictionary (system catalog):** stores metadata
  - 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
    - Operating system file name or disk addresses of blocks containing records of the relation
  - Information about indexes

Data Dictionary Storage/2

- **Catalog structure:** Can use either
  - specialized data structures designed for efficient access;
  - or a set of relations, with existing system features used to ensure efficient access (usually the preferred method)
- Schema for a possible catalog representation
  - relation-metadata(relation-name, number-of-attributes, storage-organization, location)
  - attribute-metadata(attribute-name, relation-name, domain-type, position, length)
  - user-metadata(user-name, encrypted-password,group)
  - index-metadata(index-name,relation-name, index-type,index-attributes)
  - view-metadata(view-name, definition)