

# COMP 3311

# DATABASE MANAGEMENT

# SYSTEMS

## LECTURE 11

## STORAGE AND FILE STRUCTURE

# STORAGE AND FILE STRUCTURE: OUTLINE

Overview of Physical Storage Media

Database Buffer

Record Organization

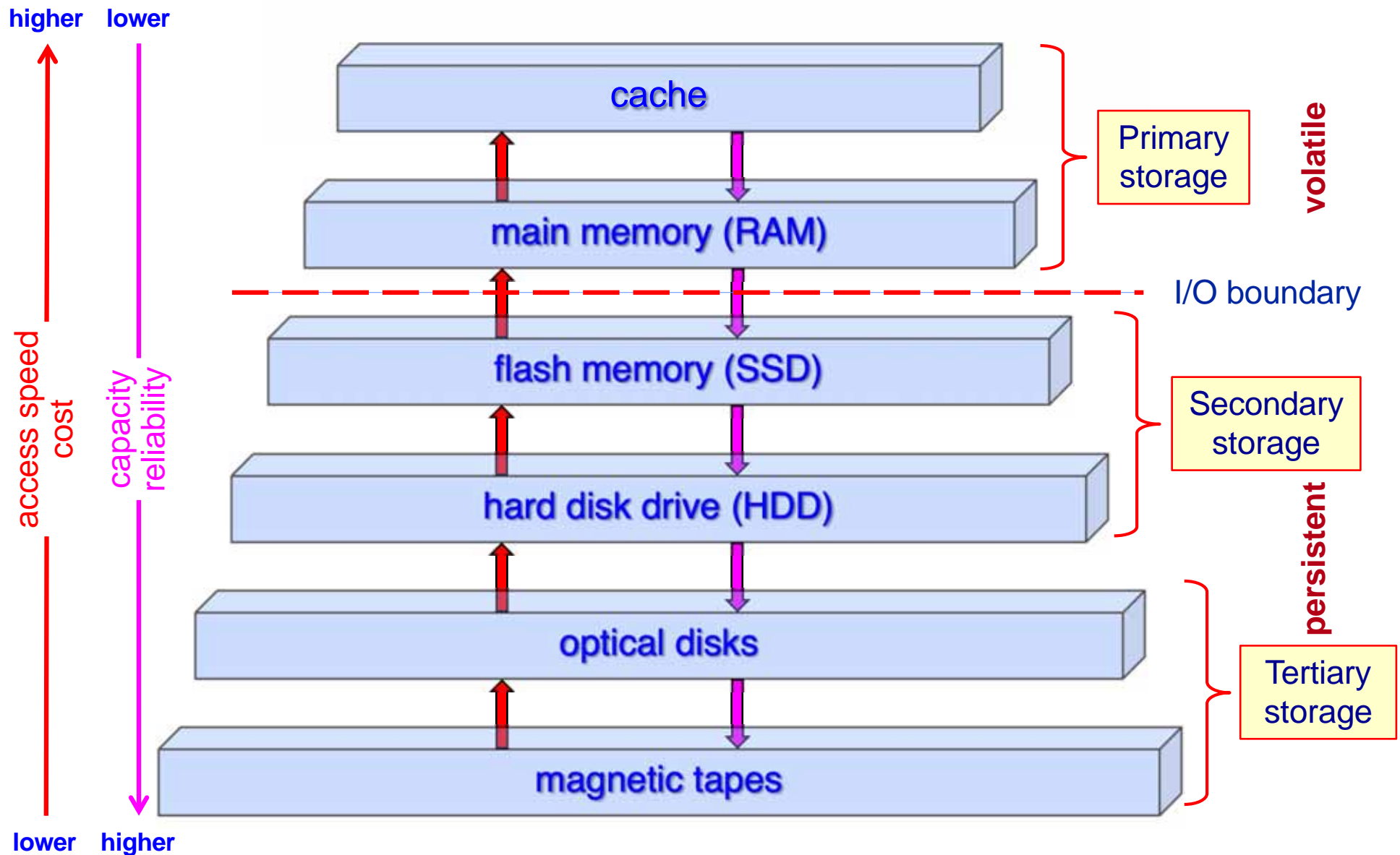
- Fixed Length Records
- Variable Length Records

File Organization

- Heap File
- Sequential File
- Hash File

Data-Dictionary Storage

# STORAGE DEVICE HIERARCHY



# STORAGE DEVICE HIERARCHY

- A typical storage hierarchy is:
  - **main memory (RAM)** for currently used data.
  - **flash memory (SSD)/hard disk disks (HDD)** for the main database (secondary storage).
  - **optical disks/tapes** for archiving older versions of the data (tertiary storage).

## Why not store everything in main memory?

 Capacity **versus** Cost

 **Main memory is volatile!**  
(We want the database to be saved **permanently!**)

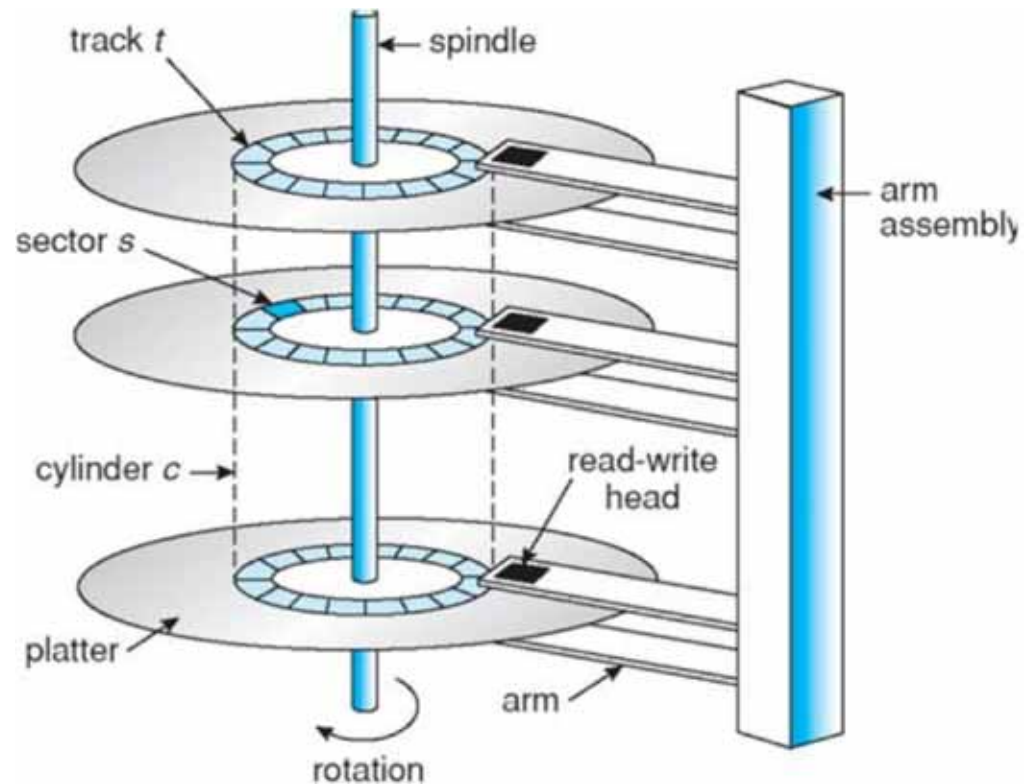


# HARD DISK DRIVE (HDD)

- The platters spin.
- The arm assembly moves in or out to position a head on a desired track.
- Tracks under all the heads make a (imaginary) **cylinder**.
- Only one head reads/writes at any one time since it is hard for all heads to line up on a track exactly.
- The **page/block size** is often the unit of retrieval and is a multiple of the **sector size** (which is fixed).

A **sector** is the smallest unit that can be physically read/written.

Most common secondary storage device.



Disk **READ/WRITE** operations are much slower than in-memory operations.

# HARD DISK DRIVE (HDD)

- Data is stored and retrieved in units called *pages* or *blocks* (typically a page/block is 512 to 16K bytes).
- DBMS operations on disks:
  - **READ:** transfer data (page) from disk to main memory (RAM).
  - **WRITE:** transfer data (page) from main memory (RAM) to disk.
  - Both are much slower than in-memory operations, so must be planned carefully!
- The time to retrieve a page from disk varies depending upon its location on disk.

 **The placement of pages on a disk has a major impact on DBMS performance!**



# DISK PAGE ACCESS

- Time to access (read/write) a page on a disk consists of:

<b>seek time</b>	Time to move the arms to position the disk head on a track. Seek time varies from about 4 to 15 msec.
<b>rotational delay (latency)</b>	Time to wait for the page (sector) to rotate under the head. Rotational delay varies from 2 to 7 msec.
<b>transfer time</b>	Time to actually move data to/from the disk surface. The transfer rate is about 1 msec. per 4KB page.

 **Seek time and rotational delay dominate.**

 **Sequential I/O is much faster than random I/O.**

 **The key to better I/O performance:  
reduce seek time/rotational delay!**

# STORAGE ACCESS

- To reduce seek time/rotational delay, database systems try to **minimize page transfers** (read/write) between disk and memory.
- The number of disk accesses can be reduced by **keeping** as many **pages** as possible **in main memory**.

**Buffer** – the portion of main memory available to store **copies of disk pages**.

 **NOTE: main memory space for storing disk pages is limited!**

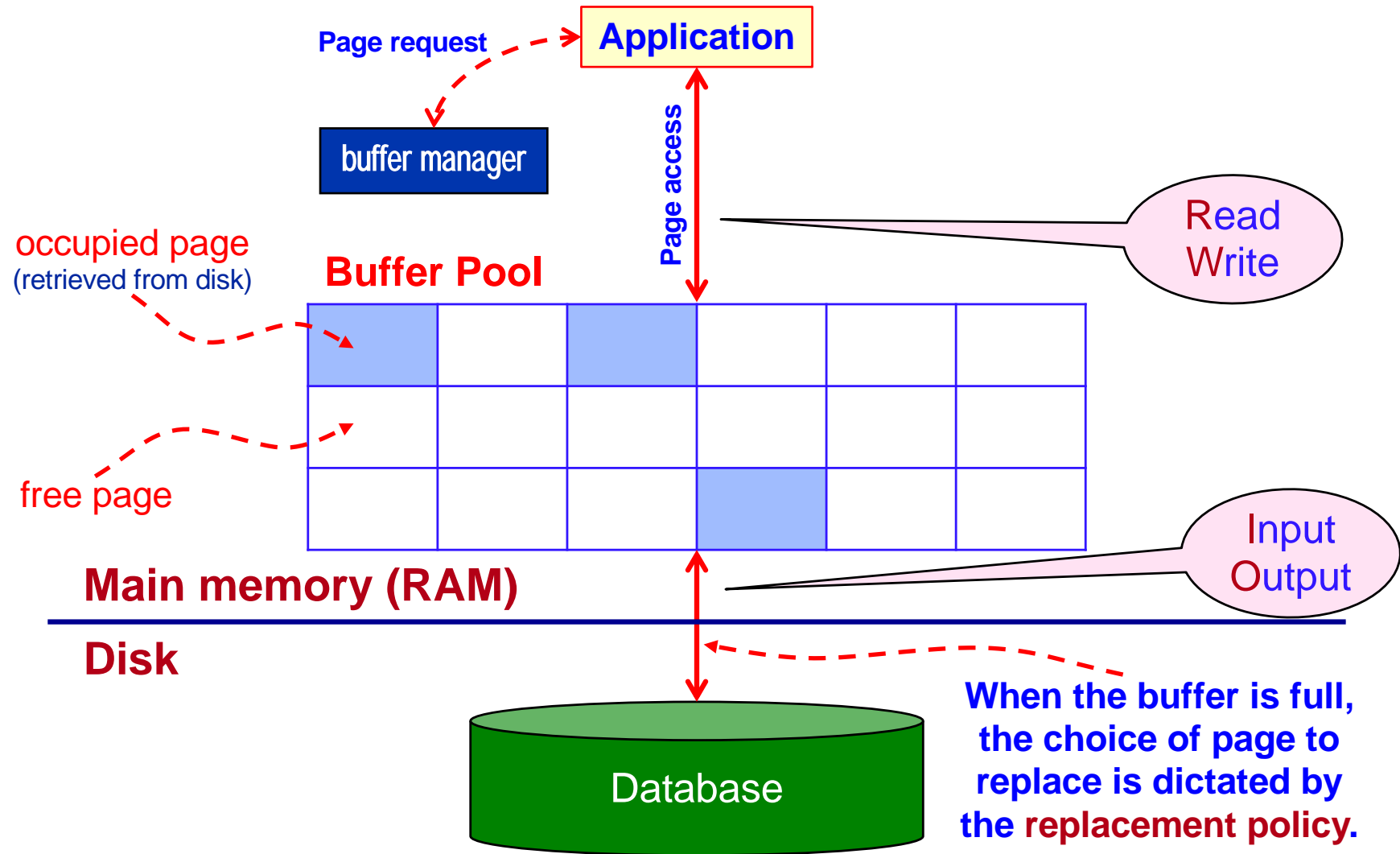
**Buffer manager** – the subsystem responsible for managing the in-memory buffer space.

 **The goal of the buffer manager is to minimize disk accesses.**

**Data must be in RAM for an application to operate on it!**



# BUFFER MANAGEMENT



# BUFFER MANAGER

- Applications make requests to the buffer manager when a page is needed from disk.
- If the page is already in the buffer, the requesting application is given the address of the page in main memory and accesses it directly.
- If the page is not in the buffer:
  - The buffer manager allocates space in the buffer for the page, replacing (throwing out) some other page, if necessary, to make space for the new page.
  - The page 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 page from the disk into the buffer, and passes the address of the page in main memory to the requesting application



# BUFFER REPLACEMENT POLICIES

- Most operating systems replace (evict) the page *least recently used* (LRU strategy).
- LRU can be a *bad strategy* for certain access patterns involving *repeated scans of data*.

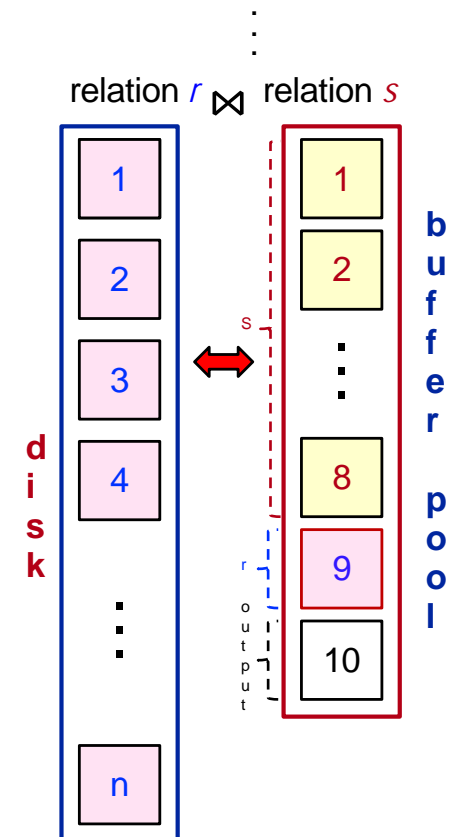
**Example:** compute the join of two relations  $r$  and  $s$  using a nested loop.

**Best in this case:** replace the page *most recently used* (MRU strategy).

- A DBMS usually has its *own buffer manager* that uses *statistical information* regarding the probability that a request will reference a particular relation or page.

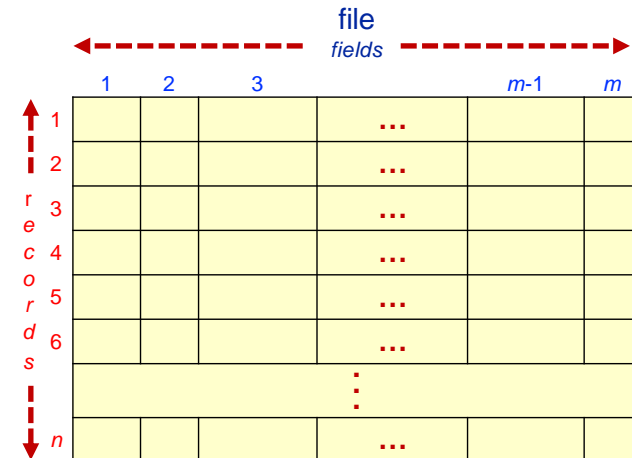
Suppose memory can hold 10 pages and relation  $s$  needs 8 pages.

For each page in  $r$   
For each page in  $s$



# RECORD ORGANIZATION

- The **organization of data items**, which usually represent attributes, **into physically stored records**.
- A database is stored as
  - a collection of *files* where
  - each file is a sequence of *records* and
  - each record is a sequence of *fields* that occupy a **number of bytes**.
- Most common organization:
  - assume the **record size is fixed** (i.e., each record occupies a **fixed number of bytes**).
  - each file has **records of one particular type** only.
  - **different files** are used for **different relations**.



 **This organization is easiest to implement.**

# FIXED-LENGTH RECORDS: RELATIVE LOCATION

- In each page, store record  $i$  starting from byte  $n * (i - 1)$ , where  $n$  is the size of each record in bytes.

- Record access is simple, but records may cross (span) pages.

✎ Normally file systems do not allow records to cross page boundaries (unspanned).

✎ Consequently, there may be some unused space at the end of a page.

Catalog: account#, branchName, balance

record 1	A-102	Perryridge	400
record 2	A-305	Round Hill	350
record 3	A-215	Mianus	700
record 4	A-101	Downtown	500
record 5	A-222	Redwood	700
record 6	A-201	Perryridge	900
record 7	A-217	Brighton	750
record 8	A-110	Downtown	600
record 9	A-218	Perryridge	700

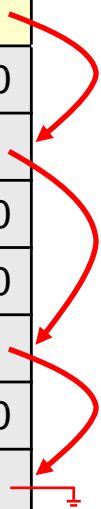
- To reclaim space when record  $i$  is deleted, shift records up.
- However, moving records inside a page is not good when records are pointed to by:
  - other records (e.g., foreign keys).
  - index entries.

# FIXED-LENGTH RECORDS: FREE LISTS

- Do not move records in a page; instead, store the address of the first deleted record in the file header (the first record).
- Use the first deleted record to store the address of the second deleted record and so on.
- Can think of these stored addresses as pointers since they “point” to the location of a record.
- This is a more space efficient representation since we can reuse space for normal attributes of free records to store the pointers. (No pointers are stored in records being used.)

Catalog: account#, branchName, balance

header			
record 1	A-102	Perryridge	400
record 2			
record 3	A-215	Mianus	700
record 4	A-101	Downtown	500
record 5			
record 6	A-201	Perryridge	900
record 7			
record 8	A-110	Downtown	600
record 9	A-218	Perryridge	700



# VARIABLE-LENGTH RECORDS: BYTE-STRING REPRESENTATION

- Variable-length records arise in database systems due to:
  - Storage of multiple types of records in a file.
  - Records that allow variable lengths for one or more fields (e.g., varchar data type).
  - Records that allow repeating fields (not allowed in relational DBMSs).

file

1 ⊥	2 ⊥	3 ⊥	4 ⊥	5 ⊥	...
-----	-----	-----	-----	-----	-----

- Simple (but bad) solution: **byte-string representation**
  - Store each record one after the other as a string of bytes.
  - Attach a special **end-of-record** ( $\perp$ ) symbol to the end of each record.
  - Problems with record
    - **deletion** (results in fragmentation of free space).
    - **growth** (may require movement of records if ordered).

# VARIABLE-LENGTH RECORDS: EMBEDDED IDENTIFICATION

- Precede fields with metadata (e.g., attribute name).
- Similar to approaches used for semi-structured data (e.g., XML, JSON).
- Requires extra storage space, but efficient if record size is variable or data is missing (e.g., not applicable).
- Similar problems as byte-string representation.

record 1	account#	A-102	branchName	Perryridge	balance	400
record 2	account#	A-305	branchName	Round Hill	balance	350
record 3	account#	A-215	branchName	Mianus	balance	700
record 4	account#	A-101	branchName	Downtown	balance	500
record 5	account#	A-222	branchName	Redwood	balance	700
record 6	account#	A-217	branchName	Brighton	balance	750



## VARIABLE-LENGTH RECORDS: 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.
- Can result in much empty, unused space in a file if record lengths vary widely.

Catalog: branchName, account#, balance, account#, balance, ...

record 1	Perryridge	A-102	400	A-210	900	A-218	700
record 2	Round Hill	A-305	350	⊥	⊥	⊥	⊥
record 3	Mianus	A-215	700	⊥	⊥	⊥	⊥
record 4	Downtown	A-101	500	A-110	600	⊥	⊥
record 5	Redwood	A-222	700	⊥	⊥	⊥	⊥
record 6	Brighton	A-217	750	⊥	⊥	⊥	⊥

# VARIABLE-LENGTH RECORDS: POINTER METHOD

- Useful for certain types of records with repeating attributes.
- Requires two kinds of pages in a file:

## Anchor page

Contains the first records of a chain.

## Overflow page

Contains records other than those that are the first records of a chain.

## Pointers

Link related records together.

Catalog: branchName, account#, balance

Anchor page	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	
Catalog: account#, balance				
Overflow page	A-210	900		
	A-218	700		
	A-110	600		

# VARIABLE-LENGTH RECORDS: SLOTTED-PAGE STRUCTURE



- The **page header** contains:
  - the number of record entries.
  - the end location of the free space in the page.
  - the location and size of each record.
- Records can be moved within a page to keep them contiguous with no empty space between them.
  - 👉 **The page header must be updated when a record is moved.**
- References to records do not point directly to the records — instead they point to the entry for the record in the page header.

# FILE ORGANIZATION

**Search key:** one or more attributes by which records are retrieved.

- The records in a physical file are usually organized to facilitate efficient retrieval on a search key.
- If the search key is a:  
 primary or candidate key  $\Rightarrow$  at most one record is retrieved  
 non-key  $\Rightarrow$  multiple records can be retrieved.
- Most common file organizations are
  - heap
  - sequential
  - hash (random)

## File Blocking Factor

The **blocking factor of a file**  $r$ ,  $bf_r$ , is the number of records that fit in a page and is equal to (for unspanned records)

$$\lfloor \# \text{ bytes per page} / \# \text{ bytes per record} \rfloor$$

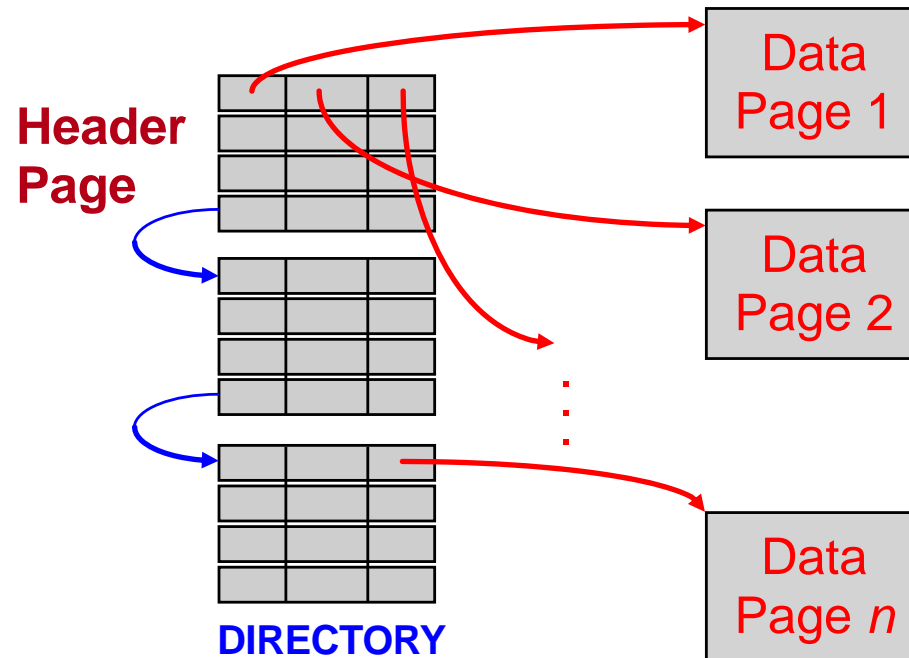
Consequently, the **number of pages** needed to store a file is equal to

$$\lceil \# \text{ records} / bf_r \rceil$$

# HEAP FILE ORGANIZATION

- A record can be placed anywhere in the file where there is space (usually at the end).
  - There is no relationship between search key and a record's location.
  - As a file grows and shrinks, disk pages are allocated/de-allocated.
- To support record level operations, need to keep track of the:
  - pages in a file.
  - free space in pages.
  - records in a page.
- Some ways to manage this information:
  - linked list: a header page points to full and free data pages, which link to each other.
  - page directory: header pages, which link to each other, point to data pages.

# HEAP FILE USING A PAGE DIRECTORY



- The entry for a page can include the number of free bytes on the page.
- The directory is a collection of pages; linked list implementation is just one alternative.

# SEQUENTIAL FILE ORGANIZATION

Catalog: account#, branchName, 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	

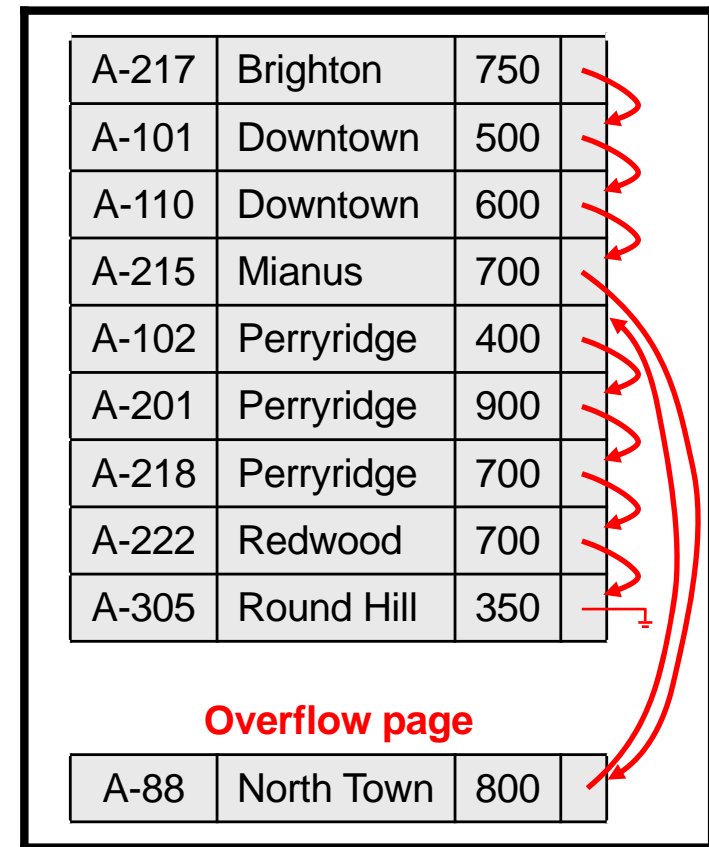
Ordered by  
branchName.

- The records are **stored in sequential order**, based on the **value of a search key** (usually, but not always, the primary key).
- Retrieval of records based on the search key is efficient.

# SEQUENTIAL FILE ORGANIZATION (cont'd)

- **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 page (can be a heap file).
  - In either case, the pointer chain must be updated.
- **Deletion** – use pointer chains.
- **Search** – binary search on search key; else linear scan.
- The file needs to be reorganized periodically to maintain the benefits of the sequential order.

Catalog: account#, branchName, balance



Ordered by branchName.



# HASH FILE ORGANIZATION

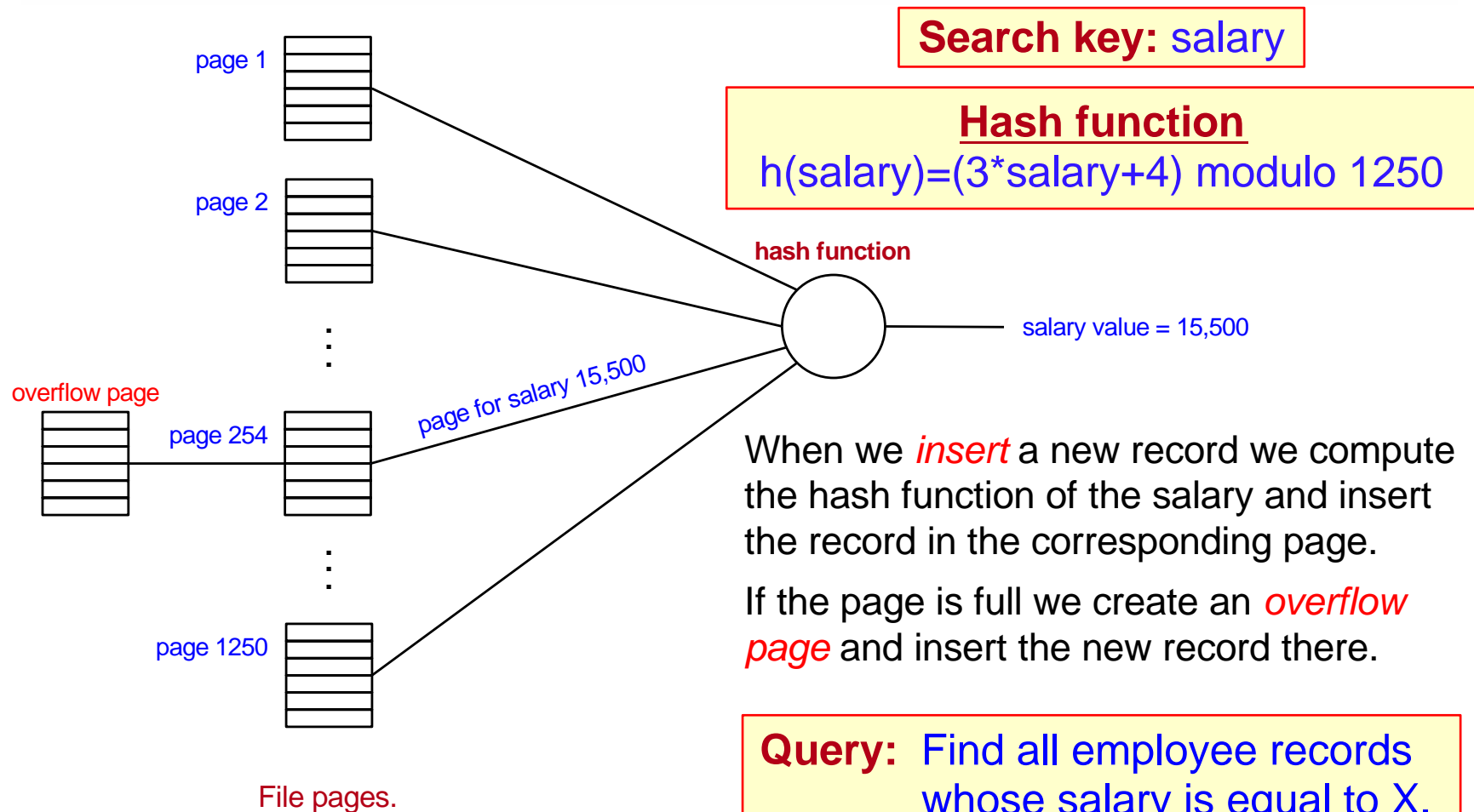
- A **hash function** defines a **key-to-address transformation** such that a record's **physical address** can be **calculated from** its **search key value**.
- The result of a hash function **specifies** in which **page of the file** a **record should be stored**. (Pages are also referred to as **buckets**.)
- **Insertion** – apply the hash function to the search key value and store the record in the page calculated by the hash function.
  - There is a **direct relationship** between the search key value and a record's physical location.
- **Search** – apply the same hash function to the search key value and retrieve the record from the page calculated by the hash function.
  - A record can often be **retrieved with only one page access**.

# EXAMPLE HASH FILE ORGANIZATION

- Assume we want to store an **Employee** table with **100,000** records and we can put **100** records per page (i.e., the blocking factor,  $bf_{Employee}$ , is **100**).
- Consequently, we need  $\lceil \# \text{ records} / bf_{Employee} \rceil = \lceil 100,000 / 100 \rceil = 1,000$  pages to store the records.
- To allow for future insertions, we can allocate **1,250 pages** (buckets) for the table (i.e., so that pages are only 80% full).
- We want to organize the file so that we can efficiently answer equality selections on salary (e.g., “**Find all employee records whose salary is equal to \$15,500**”)  $\Rightarrow$  **salary is the search key**.
- To support this query, the hash function might have the form

$$h(\text{salary}) = (a * \text{salary} + b) \text{ modulo } 1250.$$

# EXAMPLE HASH FILE ORGANIZATION (CONT'D)



👉 **Good for equality search  
on hash value only!**

When we **insert** a new record we compute the hash function of the salary and insert the record in the corresponding page.  
If the page is full we create an **overflow page** and insert the new record there.

**Query:** Find all employee records whose salary is equal to X.

When we **search** for records we compute the hash function of the salary and find the records in the corresponding page.

## EXAMPLE HASH FILE ORGANIZATION (CONTD)

- This organization can efficiently answer queries of the form: “Find all employees whose salary is equal to X”.
- In order to answer this query we compute the hash value of X and then search only in the corresponding page.
- A page may contain records of employees with different salaries that produce the same hash value (e.g., employees with salary 20,500 also hash to page 254). This is not a problem because, since we read the page, we can check all its records and select only the ones that satisfy the query condition.
- If there are **no overflow pages**, **answering the query requires just one page I/O**.

 **If there are overflow pages, we have to retrieve all of them.**

- Hashing is not good for range search (e.g., “Find all employees with salaries between 15,000 and 16,000”) because the records of these employees may be distributed in many different pages.



# SIMPLISTIC ANALYSIS OF FILE ORGANIZATIONS

- When analyzing and comparing different file organizations, for simplicity we will:
  - ignore CPU costs (e.g., searching a page in memory).
  - approximate I/O costs by ignoring I/Os saved due to page prefetching.
  - do average-case analysis using several simplistic assumptions.
    - Single record insert and delete.
    - For heap files
      - Equality selection is on the key; therefore, there is exactly one match.
      - A record insert is always at the end of a file.
    - For sequential files
      - File compaction happens after a record deletion.
      - Selection is on sort field(s).
    - For hash files
      - No overflow pages.
      - 80% page occupancy.

# PAGE I/O COST OF OPERATIONS

file	1	2	3	4	5	6	...	B
------	---	---	---	---	---	---	-----	---

Operation	Heap File	Sequential File	Hash File
Scan all records	B	B	$1.25^1 B$
Equality search <sup>2</sup>	0.5 B	$\log_2 B^3$	1
Range search	B	$\log_2 B +$ # of pages with matches	$1.25^1 B$
Insert	$2^4$	Equality search + $B^5$	2
Delete	Equality search + 1	Equality search + $B^5$	2

B is the number of pages in a file.

<sup>1</sup> Assumes 80% occupancy of pages to allow for future additions. Thus,  $1.25B$  pages are needed to store all records.

<sup>2</sup> Assumes the search is on the key value.

<sup>3</sup> Assumes binary search is used.

<sup>4</sup> Assumes the record is inserted at the end of the file – read last page and write it back.

<sup>5</sup> Assumes insert/delete is in the middle of the file and need to read and write all pages in second half of the file.

 **Several assumptions underlie these (rough) estimates!**

# DATA-DICTIONARY STORAGE

The **data dictionary** (also called the **system catalog**) stores **metadata** (i.e., data about the data in the database).

- Information about **relations**
  - names of relations
  - names and types of attributes
  - names and definitions of views
  - integrity constraints
- Information about **physical file organization**
  - How relation is stored (sequential / hash / ...)
  - Physical location of a relation
    - Operating system file name or
    - Disk addresses of pages containing records of the relation
- Information about **indices**
- **Statistical and descriptive data**
  - number of tuples in each relation and the storage method
- Information about **user accounts**, including passwords



## DATA-DICTIONARY STORAGE (CONT'D)

- The data dictionary structure can use either
  - Specialized data structures designed for efficient access.
  - A set of relations with existing system features used to ensure efficient access.

Usually the preferred option.

- A possible data dictionary representation of some metadata.

RelationMetadata(relationName, numberOfAttributes, storageOrganization, location)

AttributeMetadata(attributeName, relationName, domainType, position, length)

UserMetadata(userName, encryptedPassword, group)

IndexMetadata(indexName, relationName, indexType, indexAttributes)

ViewMetadata(viewName, definition)





# STORAGE AND FILE STRUCTURE: SUMMARY

- Available **data storage** options have **different cost/performance**.

👉 **HDDs most commonly used DBMS storage device.**

- DBMS **performance** is highly **dependent** on the **assignment** of relation tuples **to disk pages**.

👉 **Often the bottleneck in DBMS performance.**

👉 **Buffer management is very important.**

- DBMSs use file organizations provided by operating systems to store data.

👉 **Heap, sequential or hash.**