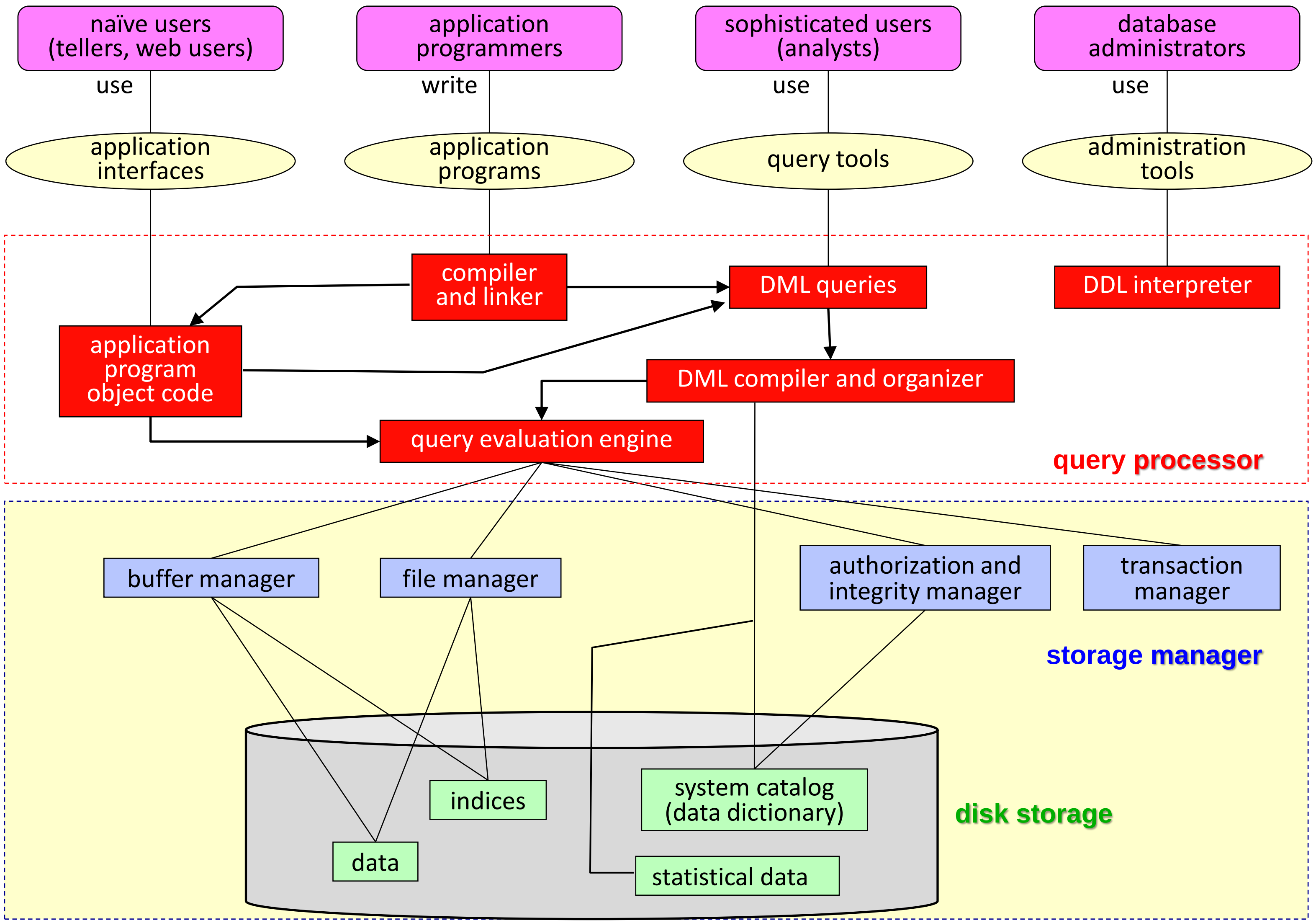
DSAA 5012
Advanced Data Management for Data Science

LECTURE 11
STORAGE AND FILE STRUCTURE

DBMS ARCHITECTURE



STORAGE AND FILE STRUCTURE: OUTLINE

Overview of Physical Storage Media

Database Buffer

Record Organization

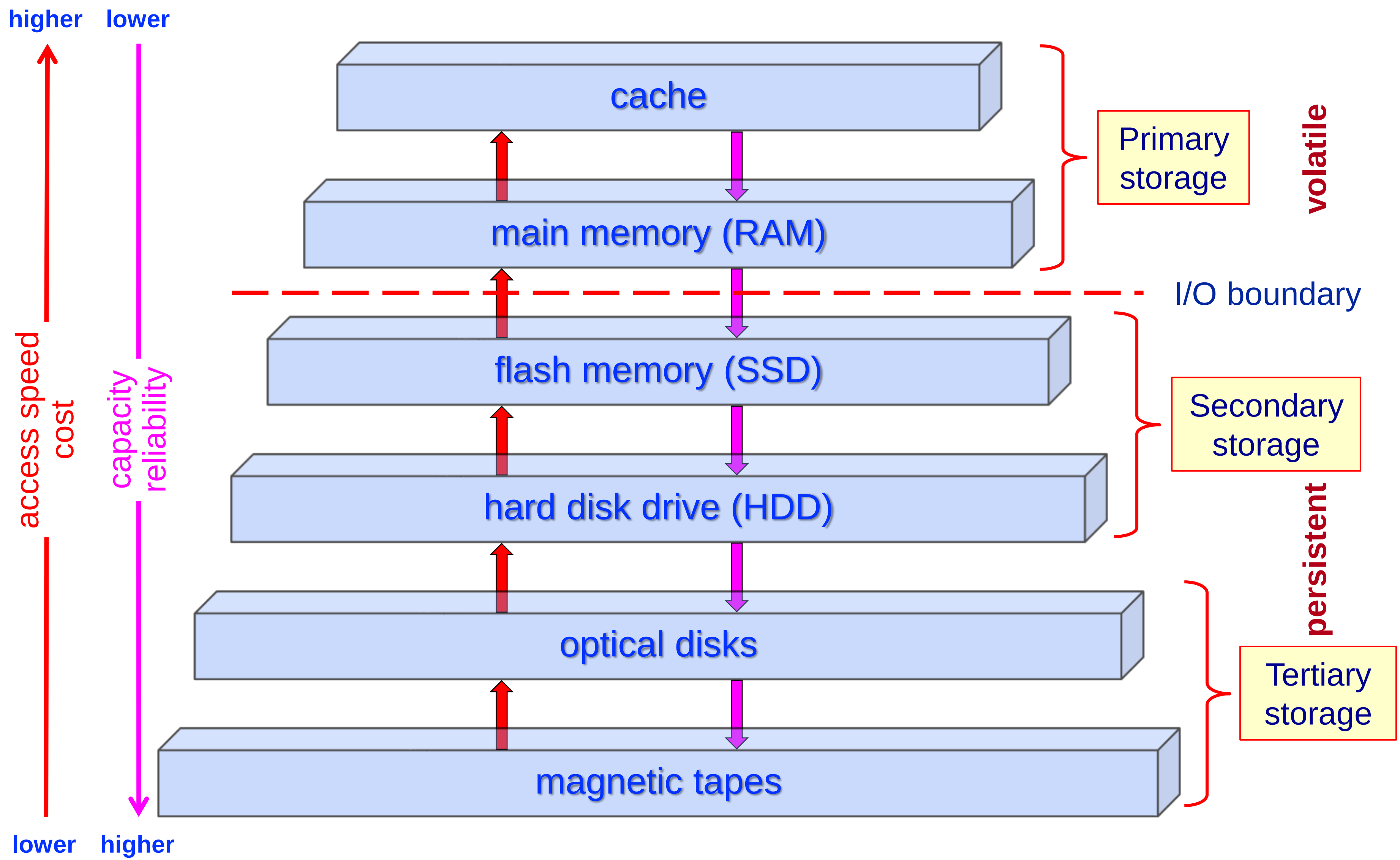
- Fixed Length Records
- Variable Length Records

File Organization

- Heap File
- Sequential File
- Hash File

Data-Dictionary Storage

STORAGE DEVICE HIERARCHY

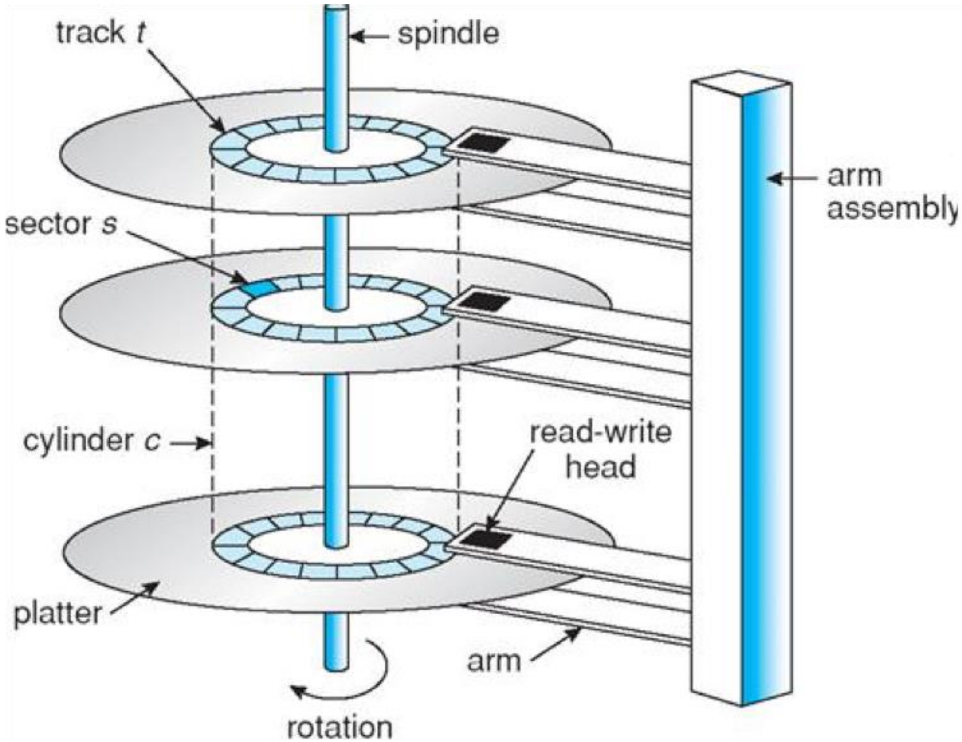


HARD DISK DRIVE (HDD)

- 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.

DISK PAGE ACCESS

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

seek time	Time to move the arms to position the disk head on a track. Seek time varies from about 4 to 15 msec.
rotational delay (latency)	Time to wait for the page (sector) to rotate under the head. Rotational delay varies from 2 to 7 msec.
transfer time	Time to 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 AND FILE STRUCTURE: OUTLINE

✓ Overview of Physical Storage Media

➡ **Database Buffer**

File Organization
- Fixed Length Records
- Variable Length Records

Record Organization in Files
- Heap File
- Sequential File
- Hash File

Data-Dictionary Storage

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.

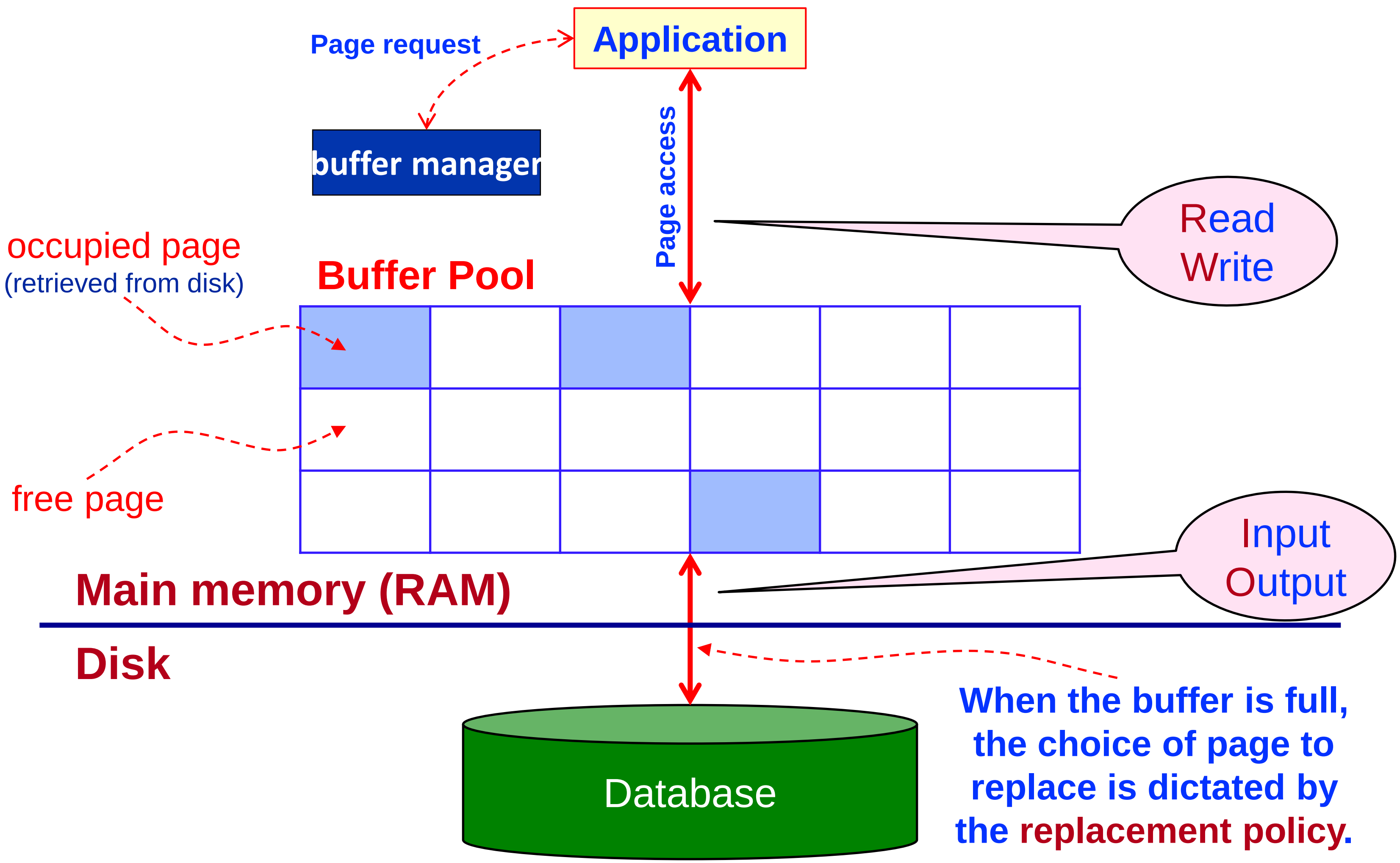
☞ **NOTE: main memory space for storing disk pages is limited!**

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



When the buffer is full, the choice of page to replace is dictated by the **replacement policy**.

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.

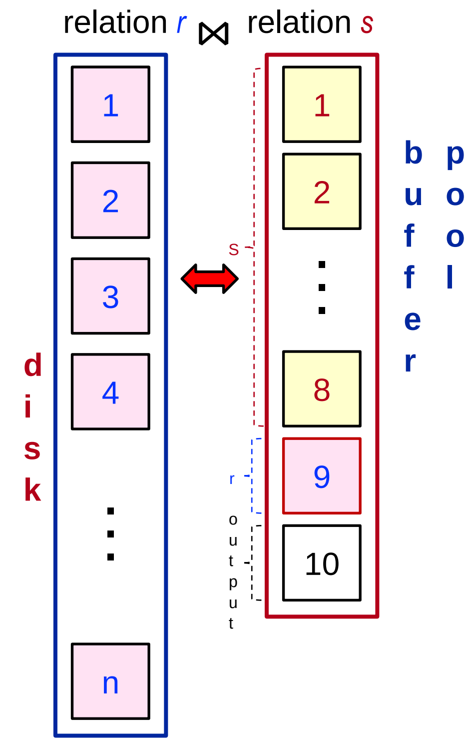
  Example: compute the join of two relations *r* and *s* using a nested loop.

  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*
        ⋮



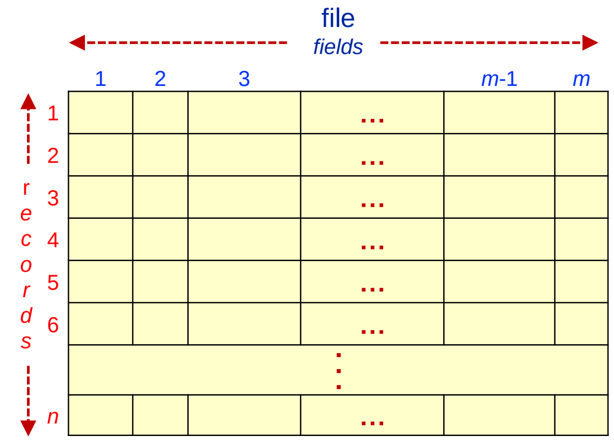
STORAGE AND FILE STRUCTURE: OUTLINE

- ✓ 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

RECORD ORGANIZATION

Record organization is 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 several 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 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 <u>not good</u> when records are pointed to by:

1. other records (e.g., foreign keys).
2. 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.

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

- 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.)

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* (⊥) symbol to the end of each record.
  - Problems with
    - record deletion (results in fragmentation of free space).
    - file 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

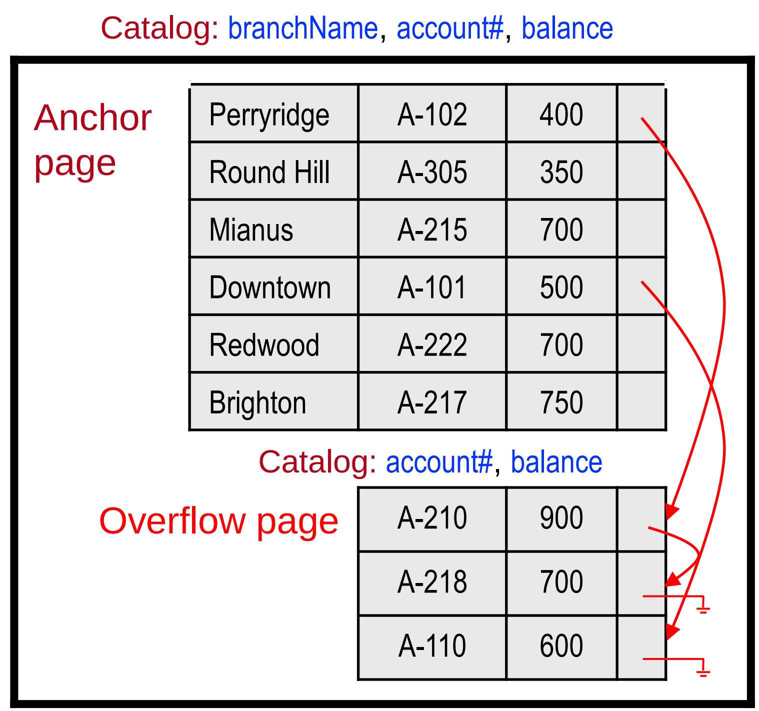
  Contains the first records of a chain.

  Overflow page

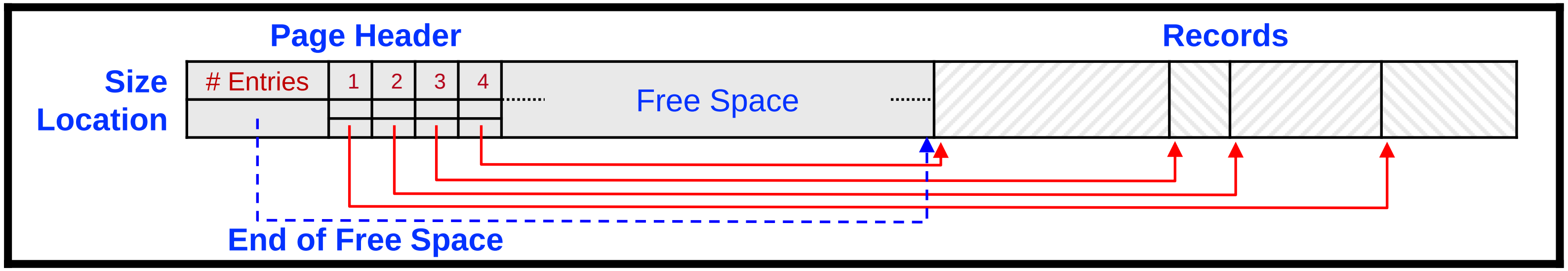
  Contains records other than those that are the first records of a chain.

  Pointers

  Link related records together.



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.

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

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 $\Longrightarrow$ at most one record is retrieved

  non-key $\Longrightarrow$ 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)

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

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

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

STORAGE AND FILE STRUCTURE

EXERCISE 1

$bf = \lfloor \text{\# bytes per page / \# bytes per record} \rfloor$
$\text{\# pages} = \lceil \text{\# records} / bf_r \rceil$

EXERCISE 1

A Student file has 20,000 records of fixed-length. Assume the page size is 512 bytes and each record has the following fields:

name: 30 bytes, studentId: 8 bytes, address: 40 bytes, phone: 8 bytes, birthdate: 8 bytes, gender: 1 byte, majorDeptCode: 4 bytes, minorDeptCode: 4 bytes, classCode: 4 bytes, degreeProgram: 3 bytes.
An additional byte is used as a deletion marker.

a) What is the record size in bytes?

record size: 30 + 8 + 40 + 8 + 8 + 1 + 4 + 4 + 4 + 3 + 1 = 111 bytes

b) What is the blocking factor bf_{Student}?

bf_{Student}: $\lfloor$512 bytes per page / 111 bytes per Student record$\rfloor$ = 4 records/page

c) How many pages are needed to store the file?

pages needed: $\lceil$20,000 records / 4 records per page$\rceil$ = 5000

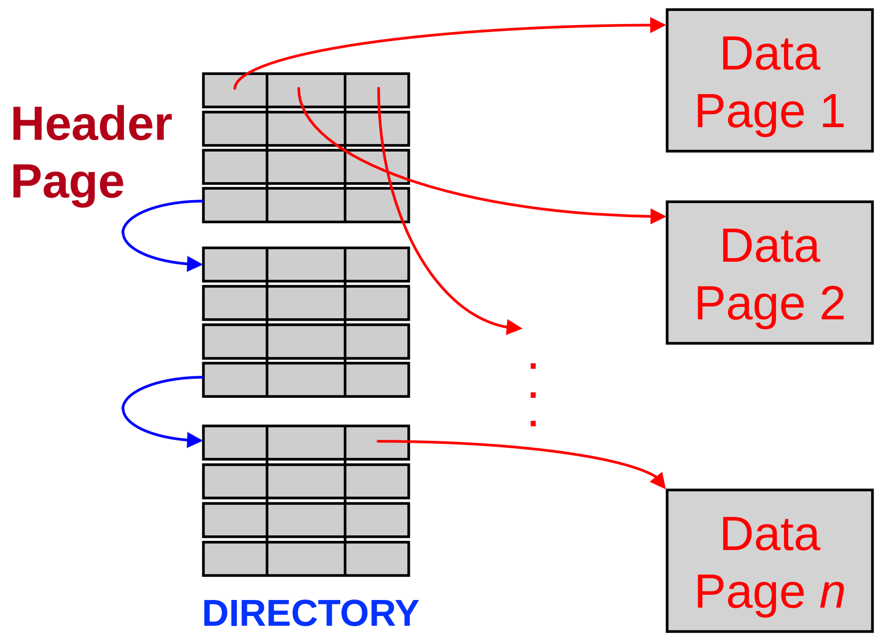
HEAP FILE ORGANIZATION

- A record can be placed anywhere in the file where there is space (usually at the end).
  - There is no relationship between a search key and a record's location.
  - As a file grows and shrinks, disk pages are allocated/de-allocated.

- To support record level operations, we 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 sequential file scan.
- The file needs to be reorganized periodically to maintain the benefits of the sequential order.

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	

Overflow page

A-88	North Town	800	

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 by accessing only one page.

EXAMPLE HASH FILE ORGANIZATION

- Assume we want to store an Employee relation 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 relation (i.e., so that pages are only 80% full).

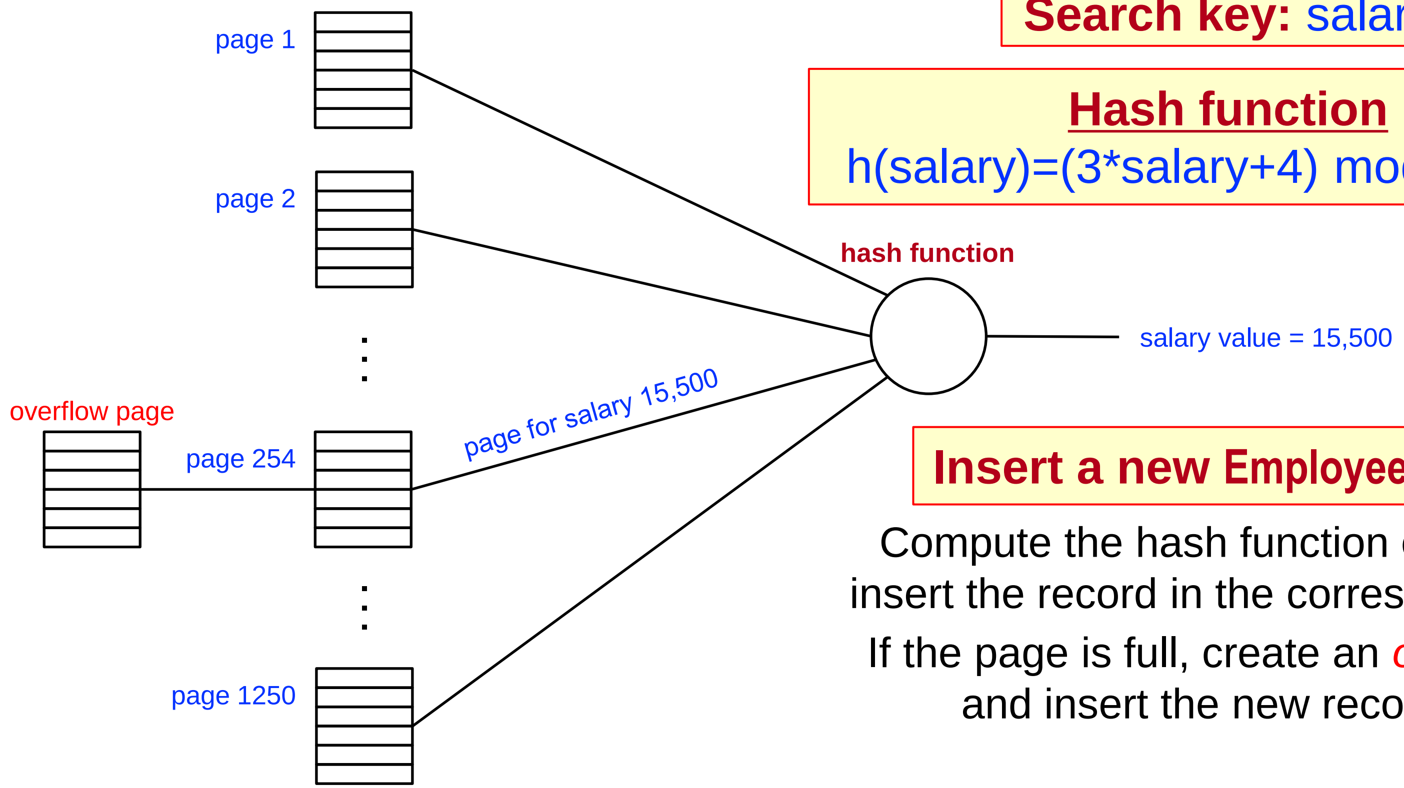
- We want to organize the file so that we can efficiently answer equality selections on salary $\Longrightarrow$ salary is the search key .

Example query: Find all employee records whose salary is equal to $15,500.

- To support this query, the hash function might have the form

h(salary)=(a*salary+b) modulo 1250.

EXAMPLE HASH FILE ORGANIZATION (CONT"D)



File pages.

Search key: salary

Hash function

h(salary)=(3*salary+4) modulo 1250

Insert a new Employee record.

Compute the hash function of salary and insert the record in the corresponding page.

If the page is full, create an *overflow page* and insert the new record there.

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

Compute the hash function of the salary and find the records in the corresponding page.

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

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[2]	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.

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

2 Assumes the search is on the key value.

3 Assumes binary search is used.

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

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

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

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.**

COMP 3311: SYLLABUS

- ✓ Introduction
- ✓ Entity-Relationship (E-R) Model and Database Design
- ✓ Relational Algebra
- ✓ Structured Query Language (SQL)
- ✓ Relational Database Design
- ✓ Storage and File Structure
- ➡ **Indexing**
- Query Processing
- Query Optimization
- Transactions
- Concurrency Control
- Recovery System
- NoSQL Databases

STORAGE AND FILE STRUCTURE

EXERCISES 2, 3, 4

Upload your completed exercise worksheet to Canvas by March 10th **11 p.m.**