STORAGE AND INDEXING

Chapters 8-11

CHAPTER 8: OVERVIEW OF STORAGE AND INDEXING

- How does a DBMS store and access data
- How is data organized to minimize I/O cost
- How are indexes used and what are their properties

BASIC CONCEPTS

- Basic abstraction: data in DBMS is a *collection of records*, or a *file*, and each file consists of one or more *pages*
- Files and access methods define how we can access data
- File organization is a method of arranging records in a file when it is stored on disk.
  - Will define properties of operations (efficient or expensive)
- Choosing the right indexes is a powerfull tool

DATA ON EXTERNAL STORAGE

- Vast quantities of data ⇒ must store on disk
- Data is stored on disk in pages - typically 4KB or 8KB in size
- Cost of page i/O dominates typical cost of operations
  - DBMS is optimized to minimize this cost

DATA ON EXTERNAL STORAGE

- Disks are the most important external storage device
- Retrieve a page in a fexed cost
  - but if we read several pages in order they are stored ⇒ less cost
- Tapes are only used for archive data (off-site)
- Each record in a file has a unique identifier: **record id / rid**

DATA ON EXTERNAL STORAGE

- Data is read into memory for processing + back to disk for persistency by **buffer manager**
  - When files and access methods layer need some page it asks buffer manager for a page by its **page id**
- Space on disk is managed by **disk space manager**
  - Keeps track of used and unused pages in files

FILE ORGANIZATIONS AND INDEXING

- **File of records:** important abstraction in DBMS
  - Implemented by the files and access methods layer

FILE

A file can:

- be created
- be destroyed
- have records inserted into it
- have records deleted from it
- can be scanned (scan steps through all records one at a time)

FILE

A **relation** is typically stored as a file of records

Simplest file structure is an unordered file or **heap file**

INDEX

An index is a data structure that organizes data records on disk to optimize certain retrieval operations

- Must have a **search key** defined
- We can create additional indexes on a given collection of data

DATA ENTRY

- **Data entry:** the data records stored in an index file
- Data entry with key value k is denboted k^*

INDEXES AND DATA ENTRIES

3 main alternatives for data entry in an index

1. Data entry *k** is an actual data record (with seach key value k)
2. Data entry is a <*k*, *rid*> pair; *rid* is *record id* of data record with search key value *k*
3. Data entry is a <*k*, *rid-list*> pair; *rid-list* is list of record ids of data with key *k*

Option 1 is special file organization: **indexed file organization**

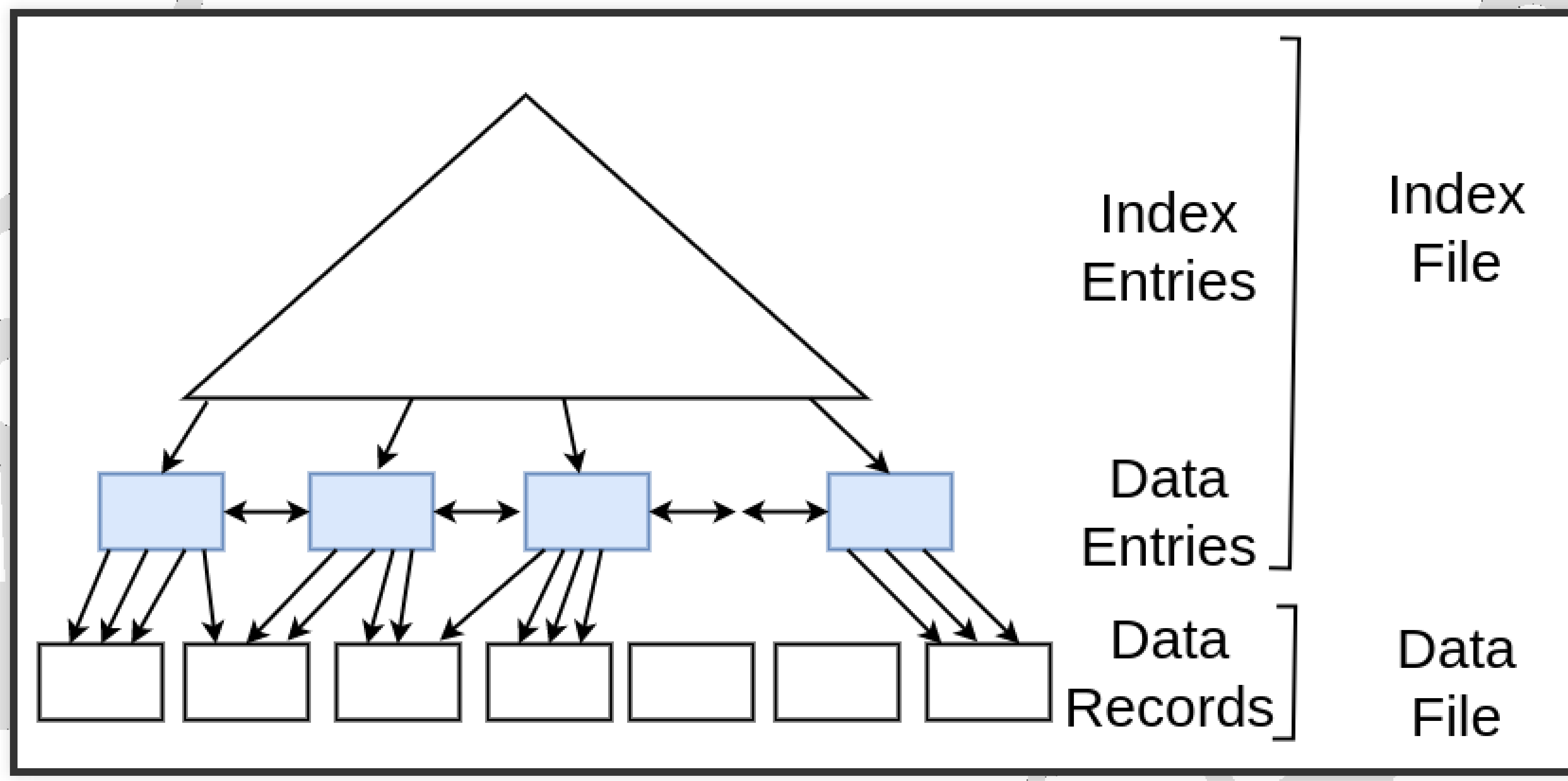
Option 2 & 3 are independant of the file of data records

At most one index of type 1 - to avoid data redundancy

CLUSTERED VS UNCLUSTERED

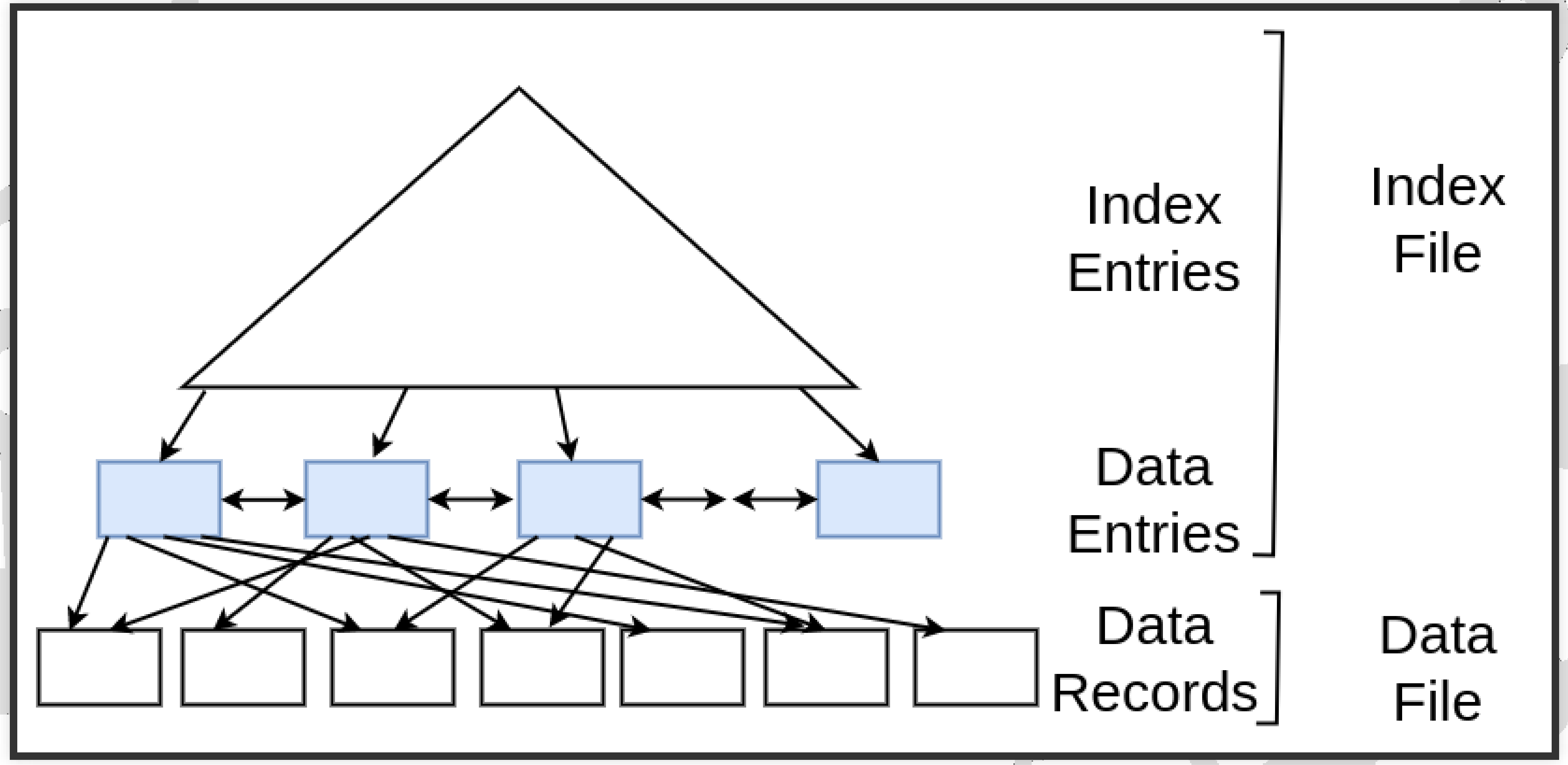
When a file is organized so ordering of data records is the same (or close to) the ordering of data entries in some index we say this index is clustered - otherwise the index is unclustered.

CLUSTERED



Index Entries
Data Entries
Data Records
Index File
Data File

UNCLUSTERED



CLUSTERED VS UNCLUSTERED

- Typically to expensive to maintain sorted indexes → Alternative 1) is called a sorted file
- Only one sorted index can exist
- Range queries on indexes can vary depending of clustering
  - Clustered: We need only retrieve a few data record pages
  - Unclustered: Each qualifying entry could be on a separate page

PRIMARY AND SECONDARY INDEXES

- **Primary index:** Contains the *primary key*
- **Secondary index:** Does NOT contain *primary key*

Sometimes a clustered index is called a primary index

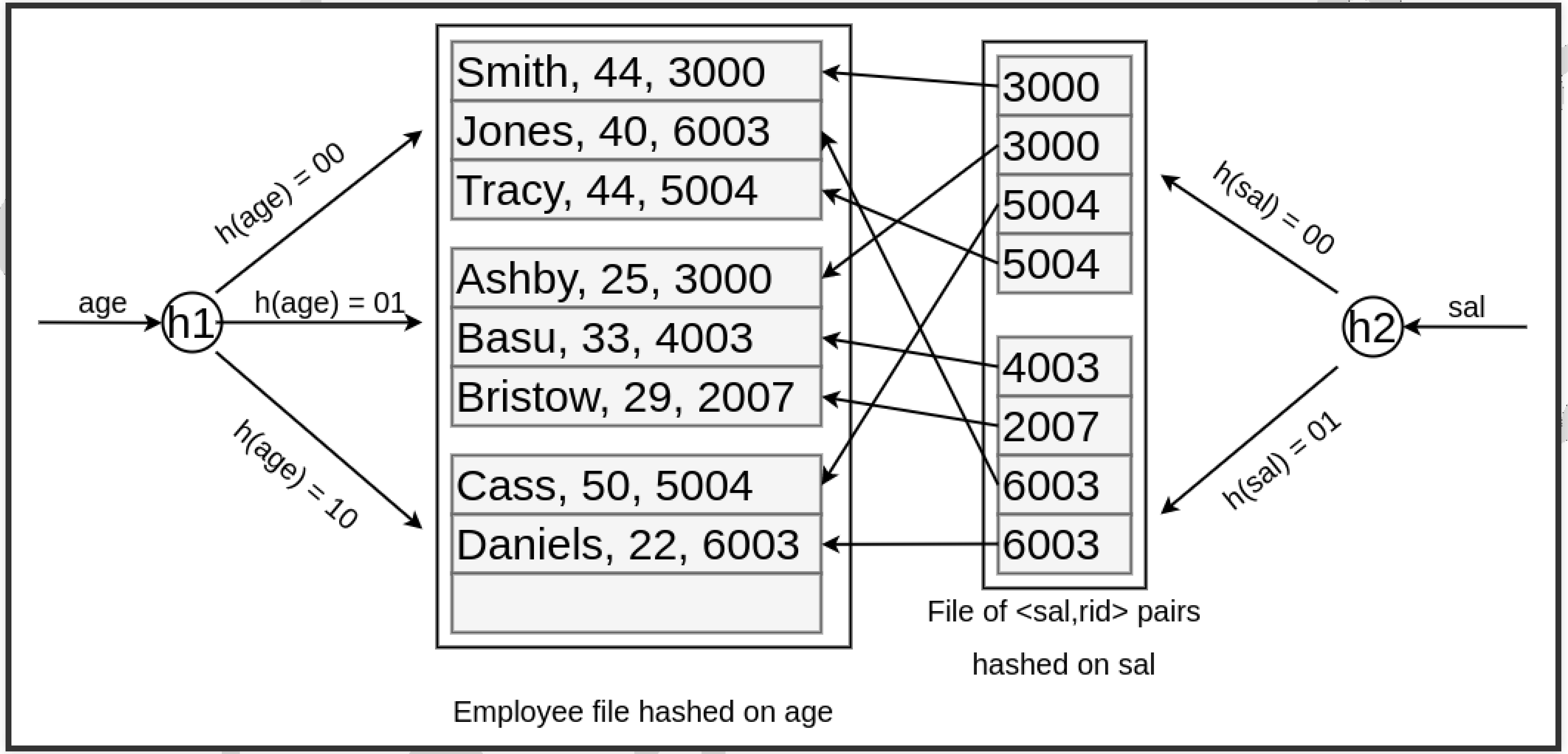
INDEX DATA STRUCTURES

Hash- and Tree-based indexing can be used with all 3 alternatives

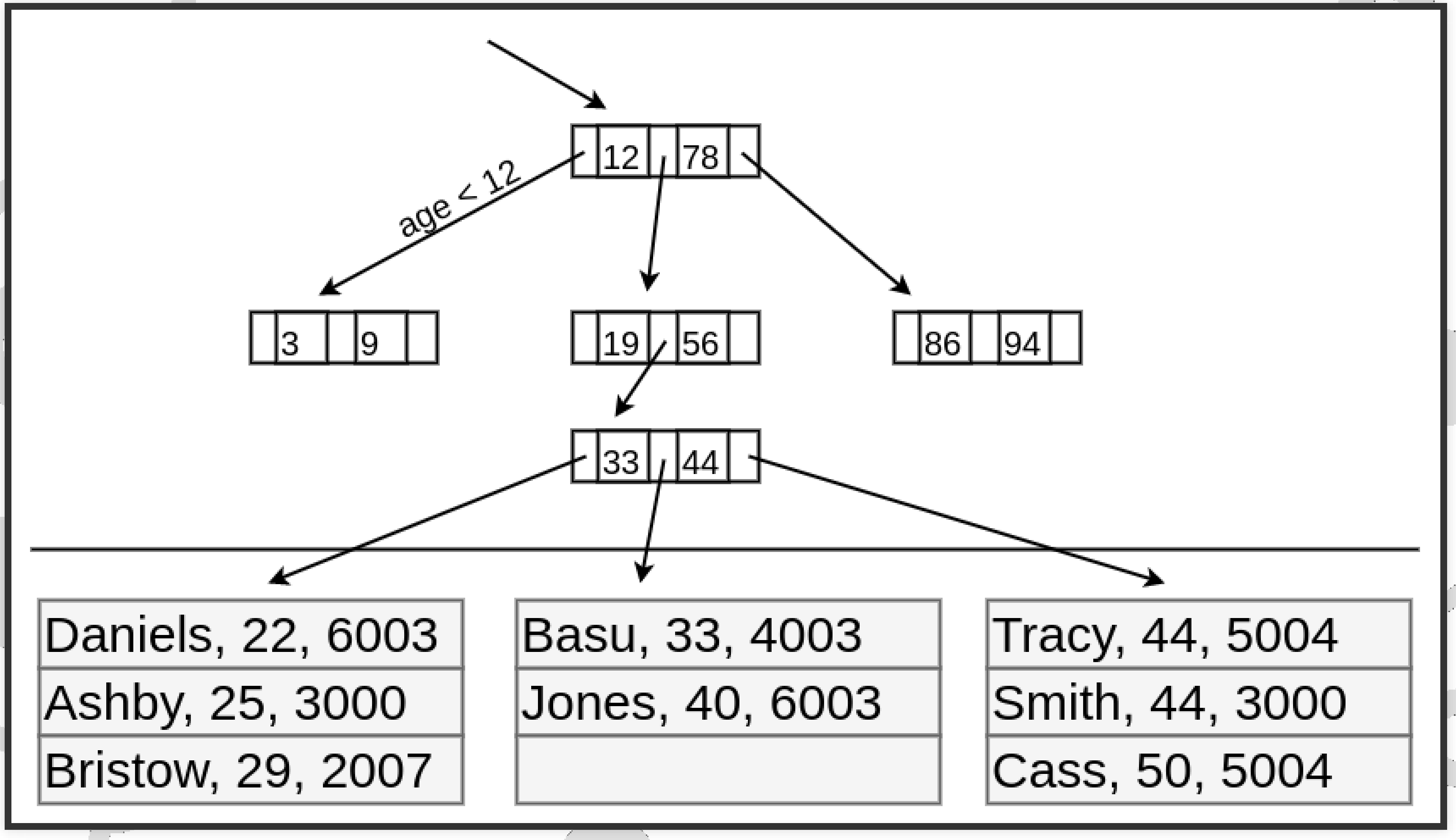
HASH-BASED INDEXING

- Quickly find records that have a given search key value
- Records in file are grouped in **buckets**
- **Bucket** consists of **primary page** and possibly additional pages linked in a chain
- Use **hash function** on search key to find bucket

HASH-BASED INDEXING



TREE-BASED INDEXING



I/OS FOR INDEX SEARCHES

- Hash based: ~1 I/O Per data page
- Tree based: Height of tree (Rarely more than 3 in practice)

COMPARISONS OF FILE ORGANIZATIONS

SCENARIO

- Relation of employee records.
- Indexes are organized according to composite search key *<age,sal>*
- All selection criteria are on those fields

FILE ORGANIZATIONS CONSIDERED

- File of randomly ordered employee records (heap file)
- Sorted file on *<age,sal>*
- Clustered B+ tree file with search key *<age,sal>*
- Heap file with an unclustered B+ tree index with search key *<age,sal>*
- Heap file with an unclustered hash index on *<age,sal>*

OPERATIONS

- Scan
- Search with equality selection
- Search with range selection
- Insert a record
- Delete a record

COST MODEL

- **B:** Number of pages when records are packed with no wasted space
- **R:** Number of records per page
- **D:** Average time to read/write a page
- **C:** Average time to process a record
- **H:** Time to apply hash function
- **F:** Fann-out of tree (>100)

COST MODEL

Typical values:

- D = 15 millis
- C, H = 100 nannoseconds

So I/O dominates totally ⇒ We focus on I/O aspect ⇒ Simplifycation

HEAP FILES

- Scan: B(D+RC)
  - Retrieve each of the B pages, taking D time per page, process R records taking time C per record
- Eq. search: B(D+RC)
  - If we know only 1 fits: 0.5B(D+RC) on average otherwise
- Range search: B(D+RC)
- Insert: 2D + C
  - Fetch last page, insert, write
- Delete: Cost of searching + B + D
  - If deleted by *rid* read page directly, i.e. D. If many fits, more expensive

SORTED FILES

- Scan: B(D+RC) - all records examined
- Eq. search: Assume match sort order, we can locate page in $\log_2$(B) steps
  - Find record in page C $\log_2$®
  - Total: D $\log_2$(B) + C $\log_2$®
- Range search: Similar to equality search - all matching will be the following items
- Insert: On average we insert in the middle of file. Must move all later records.
  - Searching + 2(0.5B(D+RC)) = Searching + B(D+RC)
- Delete: Same as search, we must move all records following

CLUSTERED FILES

Clustered files have typically 67% occupancy on data pages ⇒ Number of physical pages is 1.5B

- Scan: 1.5B(D+RC)
- Eq. search: Find page $\log_F(1.5B)$
  - Total: $D \log_F(1.5B) + C \log_2$®
- Range search: As equality search + size of output
- Insert: Search + 1 write: $D \log_F(1.5B) + C \log_2$® + D
- Delete: $D \log_F(1.5B) + C \log_2$®

HEAP FILE WITH UNCLUSTERED TREE INDEX

Number of leaf pages depends on size of data entry! Assume 1/10 the size of data record.

Number of leaf pages therefore: 0.1(1.5B) = 0.15B

Number of data entries on a page is: 10(0.67R) = 6.7R

HEAP FILE WITH UNCLUSTERED TREE INDEX

- Scan: 1.5B(D+RC)
- Eq. search: Find page $\log_F(1.5B)$
  - Total: D $\log_F(1.5B)$ + C $\log_2$(R)
- Range search: D(1+ $\log_F(0.15B)$ + # matching pages
- Insert: D(3 + $\log_F(0.15B)$)
- Delete: Search + 2D

HEAP FILE WITH UNCLUSTERED HASH INDEX

- Scan: BD(R+0.125)
- Eq. search: 2D
- Range search: BD
- Insert: 4D
- Delete: Search + 2D

COMPARISONS OF I/O COSTS

File Type	Scan	EQ. Search	Range Search
Heap	BD	0.5BD	BD
Sorted	BD	$D \log_2 B$	$D \log_2 B$ + # matching pages
Clustered	1.5BD	$D \log_F(1.5B)$	$D \log_F(1.5B)$ + # matching pages
Unclustered Tree Index	BD(R + 0.15)	$D(1 + \log_F(0.15B))$	$D(1 + \log_F(0.15B)$ + # matching pages)
Unclustered Hash Index	BD(R+0.125)	2D	BD

COMPARISONS OF I/O COSTS

File Type	Insert	Delete
Heap	2D	Search + D
Sorted	Search + BD	Search + BD
Clustered	Search + D	Search + D
Unclustered Tree Index	$D(3 + \log_F(0.15B))$	Search + 2D
Unclustered Hash Index	4D	Search + 2D

INDEXES AND PERFORMANCE TUNING

> Indexes can have a tremendous impact on system performance!

Choise of index made in the context of

- Expected workload (typical mix of)
  - Queries
  - Update operations

IMPACT OF THE WORKLOAD

Different file organizations and indexes support different operations.

- Indexes support efficient retrieval of queries with selection criteria
  - Hash based: Only equality
  - Tree based: Equality and range (if clustered)

ADVANTAGES OF TREE INDEX OVER SORTED FILE

- Inserts and deletes more efficient
- Finding leaf page when searching by search key more efficient

Disadvantage:

- Sorted file can be allocated in physical order on disk

CLUSTERED INDEX ORGANIZATION

- At most 1 clustred index (without duplication)
- More expensive to maintain than unclustered
- No reason to make hash index clustered
  - Does not support range queries

INDEX-ONLY EVALUATION

- Clustered index is expensive to maintain
- If we can evaluate just by the values in the index key ⇒ index-only evaluation
- Equally efficient for unclustered index

Example:

Index on age and calculate average age ⇒ Enough to scan the index pages

INDEX EXAMPLES

Consider:

```
SELECT E.dno
FROM Employees E
WHERE E.age > 40
```

With B+ tree index on age.

Does this index help?

INDEX EXAMPLES

```
SELECT E.dno, COUNT(*)
FROM Employees E
WHERE E.age > 10
GROUP BY E.dno
```

- Is the index on age helpfull?
- What about an index on dno?

INDEX EXAMPLES

```
SELECT E.dno
FROM Employees E
WHERE E.hobby = 'Stamps'
```

- What about an index on hobby?

INDEX EXAMPLES

```sql
SELECT E.dno, COUNT(*)
FROM Employees E
GROUP BY E.dno
```

- What index could help?

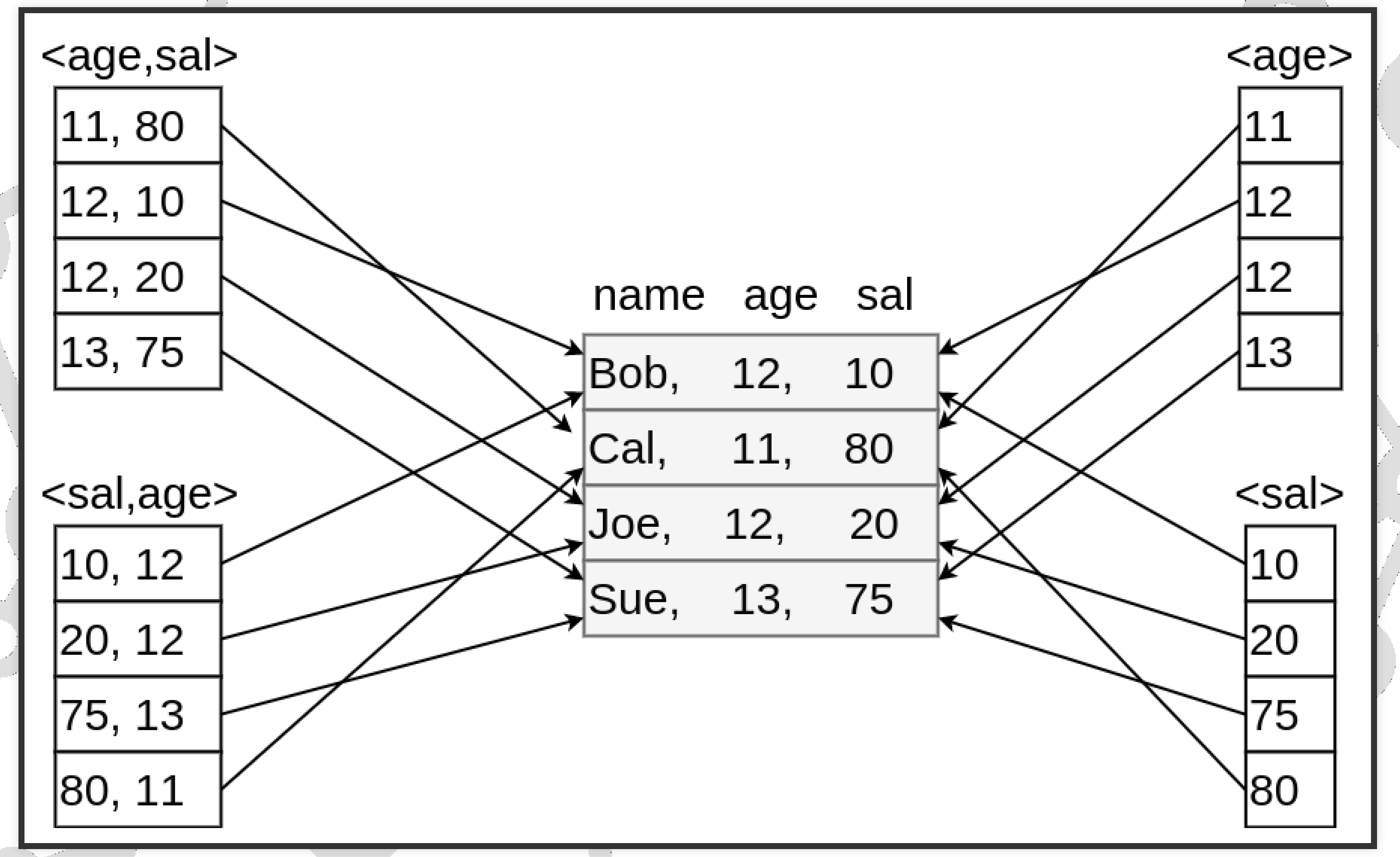
COMPOSITE SEARCH KEY

A search key for an index with multiple fields are called **composite search keys** or **concatenated keys**

If the search key is composite, an **equality query** is one where *each* field in the search key is bound to a constant

Range queries is where not all search keys are bound to constants or operator is not =

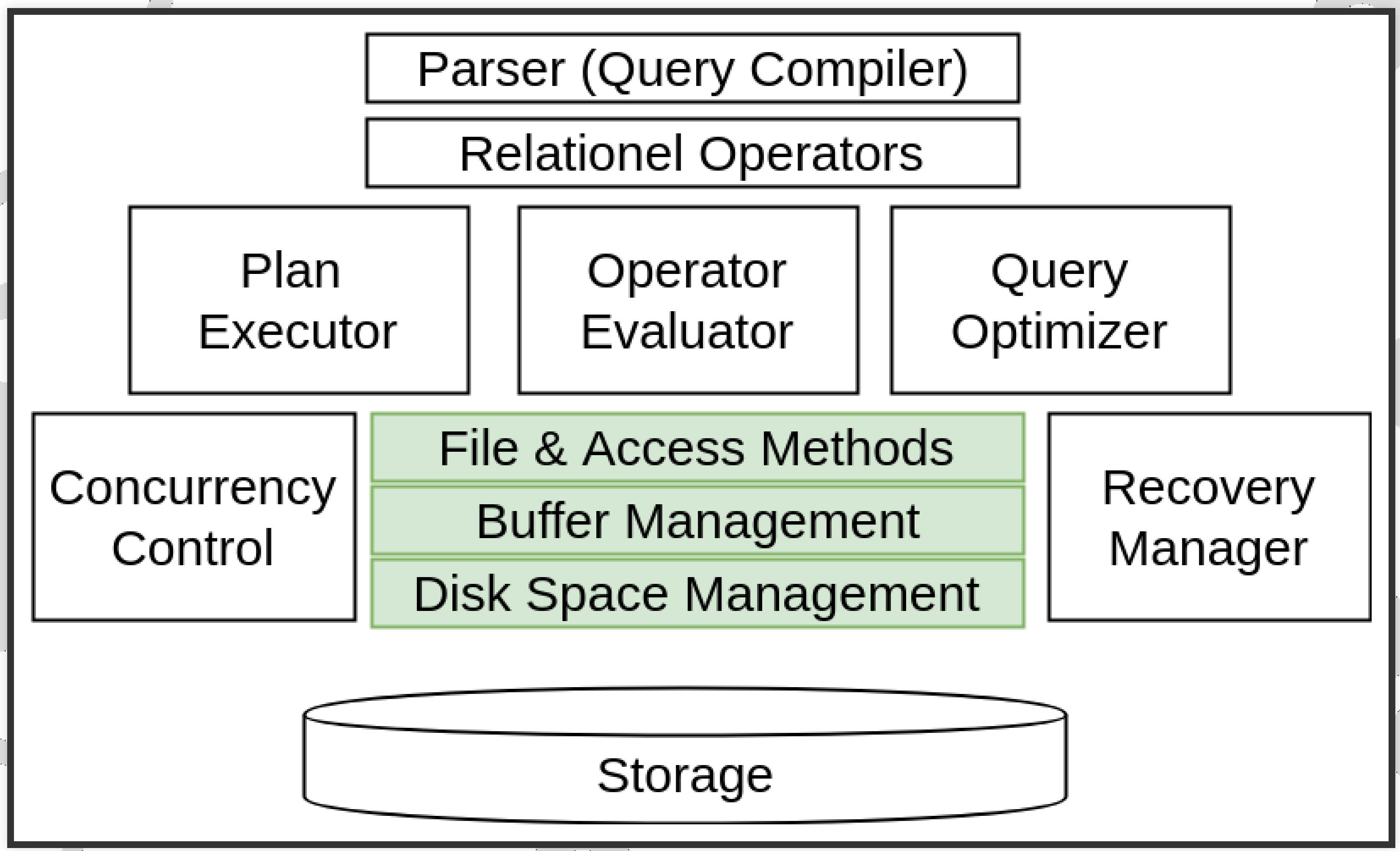
COMPOSITE KEY INDEXES



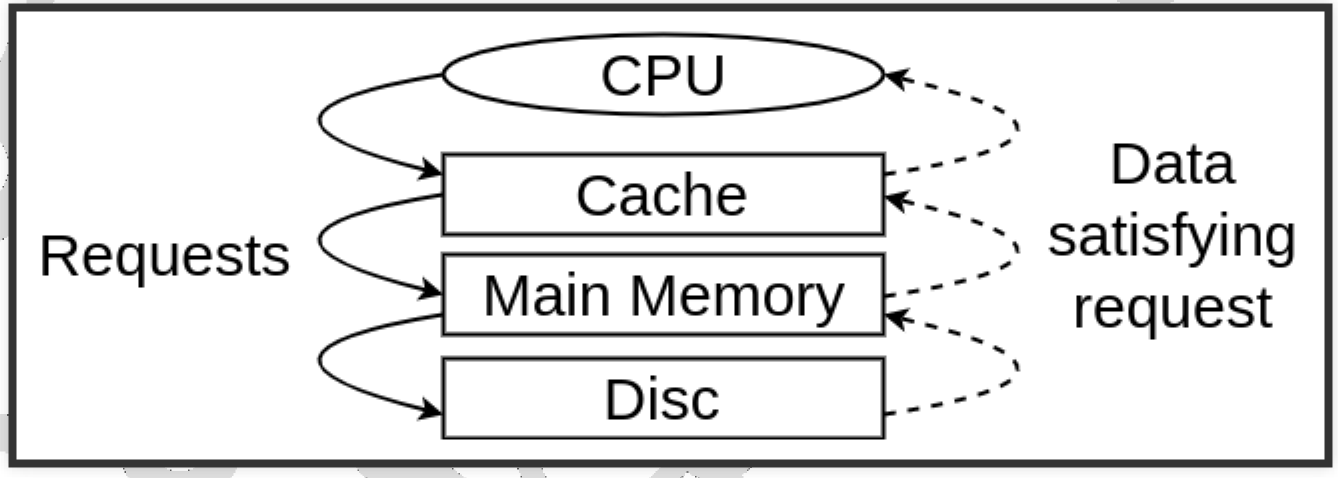
COMPOSITE KEY INDEXES

- Must be updated if any of the fields in the key is updated
- Can support a broader range of queries
  - Higher chance of index-only queries
- Larger due to more data in search key

CHAPTER 9: STORING DATA: DISKS AND FILES

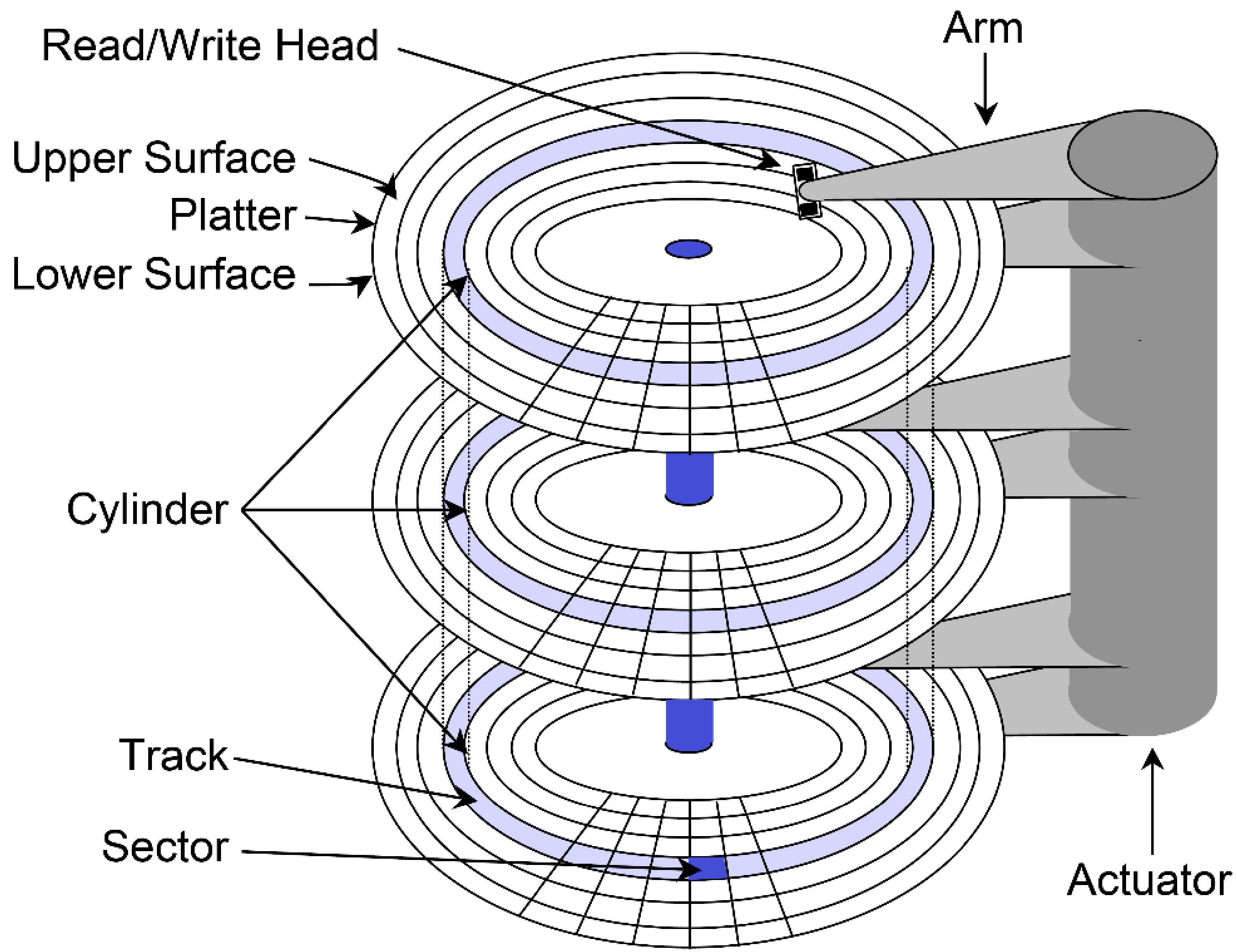


THE MEMORY HIERACHY



MAGNETIC DISKS

Read/write/transfer in blocks (pages)

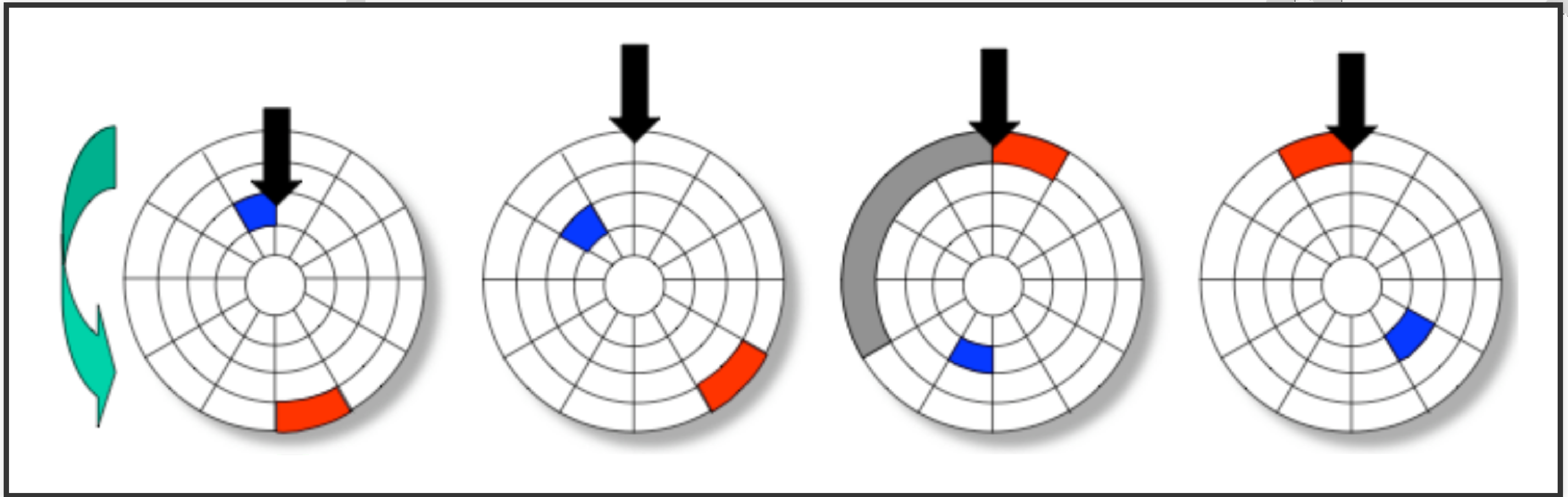


DISK CONTROLLER

Interfaces a disk drive to computer

- Implements read and write commands
- Checksum is computed - to detect bad reads/writes
  - Tries to read again if error
  - Fails if errors ultiple times

DISK PERFORMANCE



access time = seek time + rotational delay + transfer time

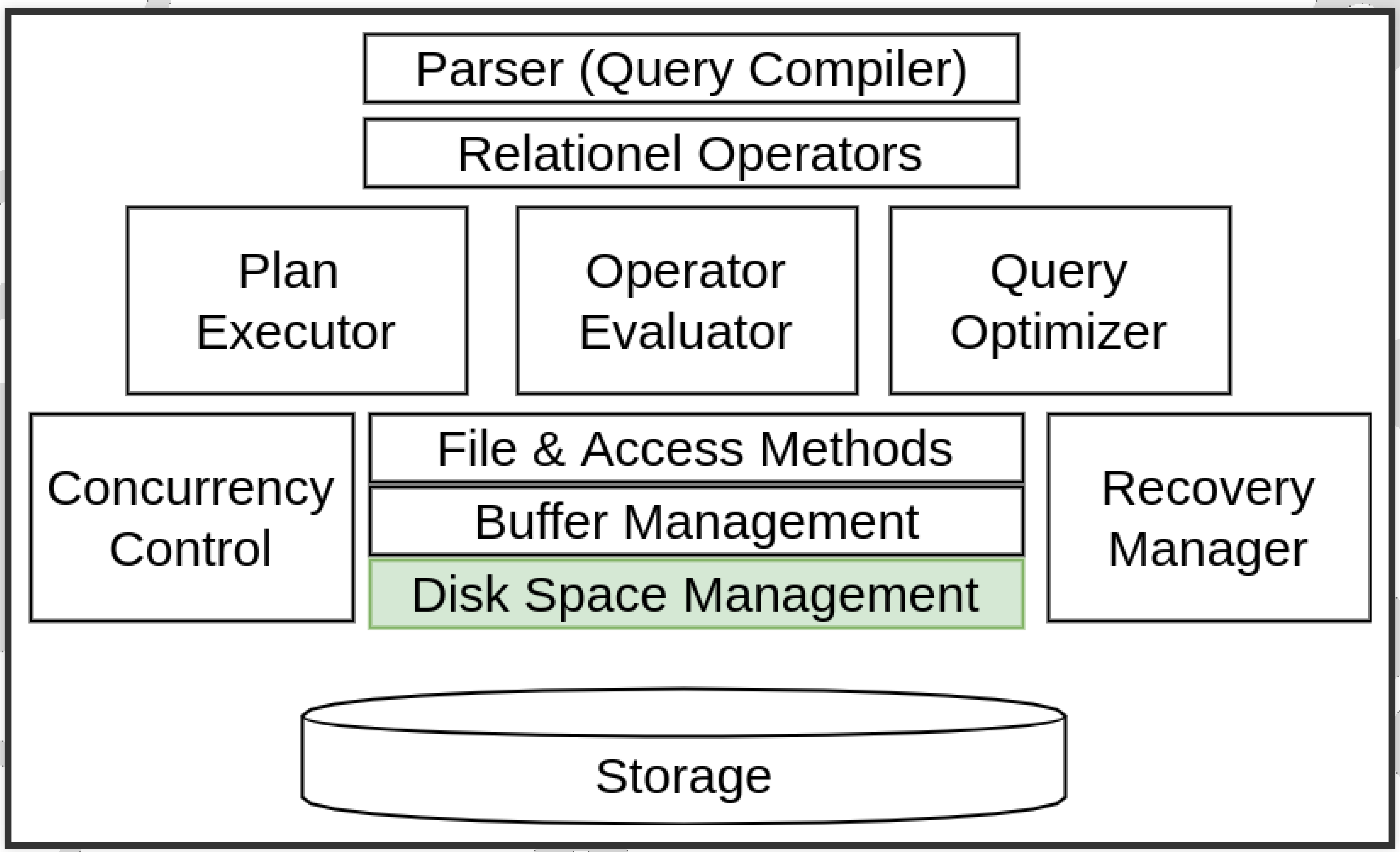
If single item on a block is needed, the entire block is transferred

THE FUTURE

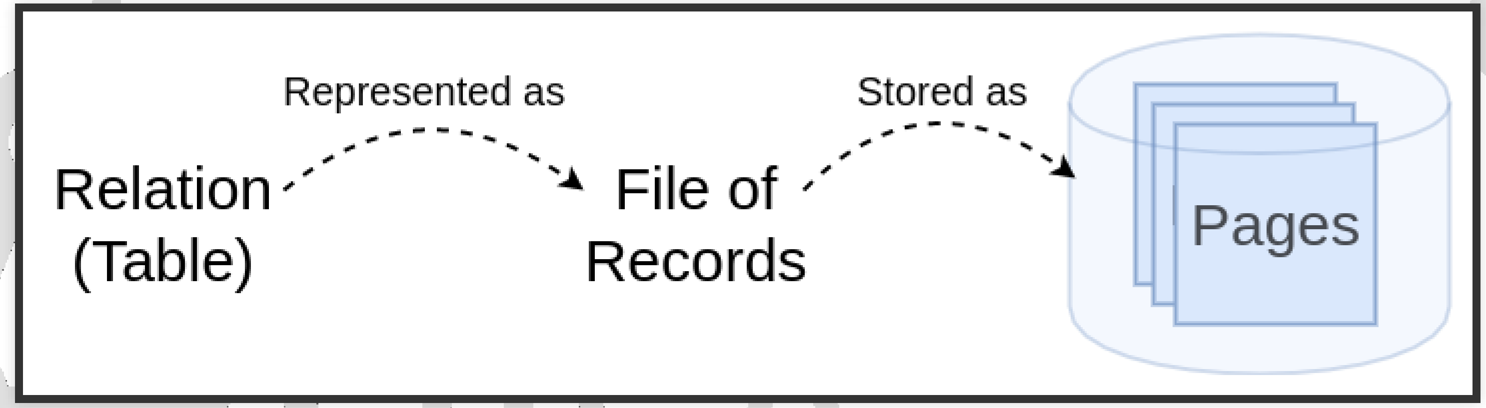
Non-Volatile Memory (NVM)

- Low access latency
- Byte addressable
- Persistent storage
- No more difference between random and sequential access

DISK SPACE MANAGEMENT



DISK SPACE MANAGER



- Preferable to have contigous blocks if sequential access is often
- Disk space manager must provide this
- And still hide details of underlying hardware and OS

KEEPING TRACK OF FREE BLOCKS

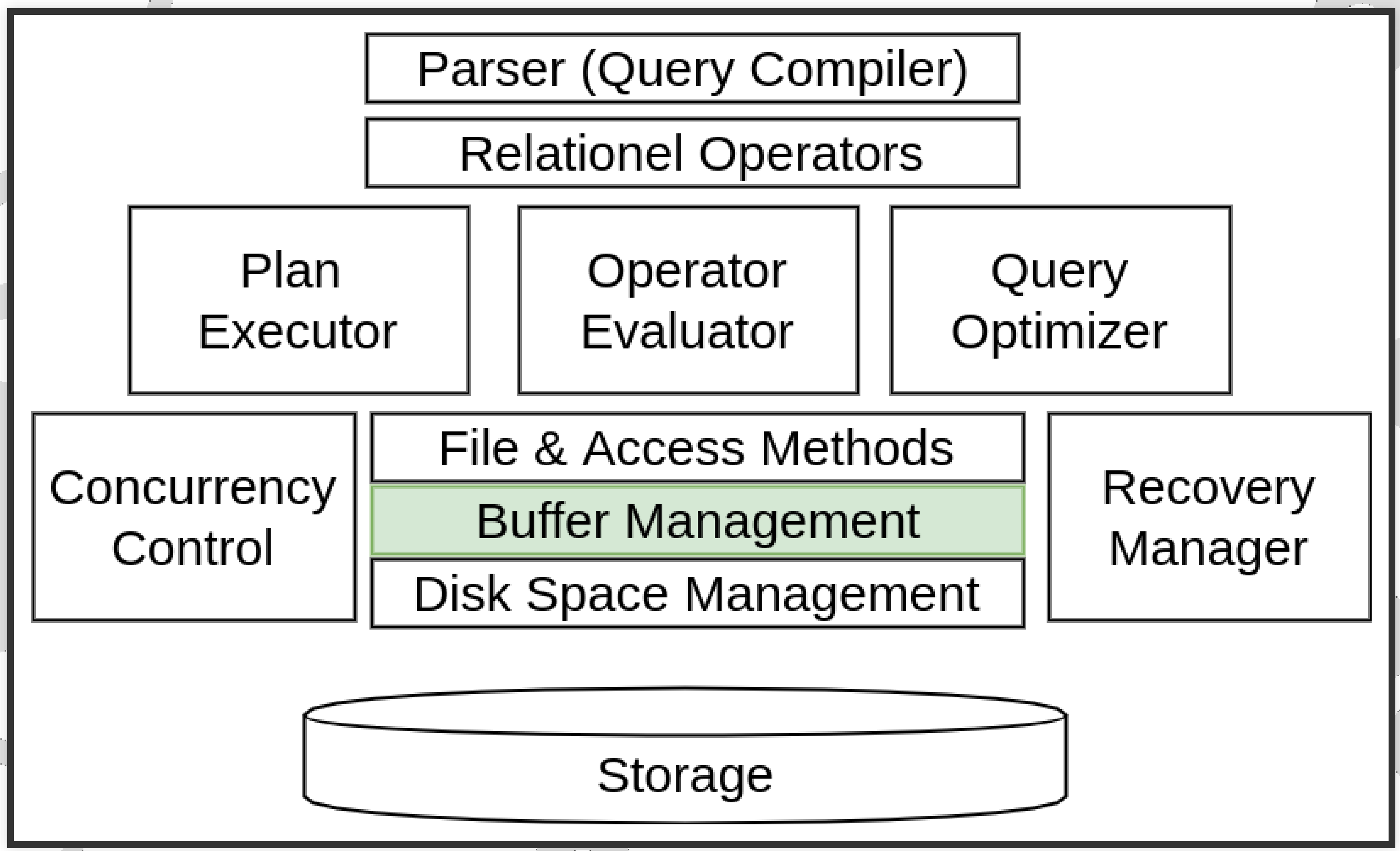
- DB grows and shrinks over time
- Files migh be contigous from start but holes appear
- Need to keep track of free blocks

1. Maintain linked list of free blocks
   - Just need pointer to head of list
2. Maintain bitmap, 1 buit for each block
   - Allows for fast identification of free contiguous blocks

OS FILE SYSTEMS TO MANAGE DISK SPACE

- OS also manages space on disk
  - Typically: *file as a sequence of bytes*
- DBMS could be build using OS Files
  - Disk Space manager responsible for managing space in these OS files
- Many DB systems do not rely on this and handle their own filesystems

BUFFER MANAGER



BUFFER MANAGER

Assume db file contains 1.000.000 pages, and only 1.000 pages of main memory.

Consider query that require a scan of entire file.

DBMS must bring pages into main memory as they are needed, and in the process decide what in main memory to replace.

Policy used to decide which page to replace: **replacement policy**

BUFFER MANAGER

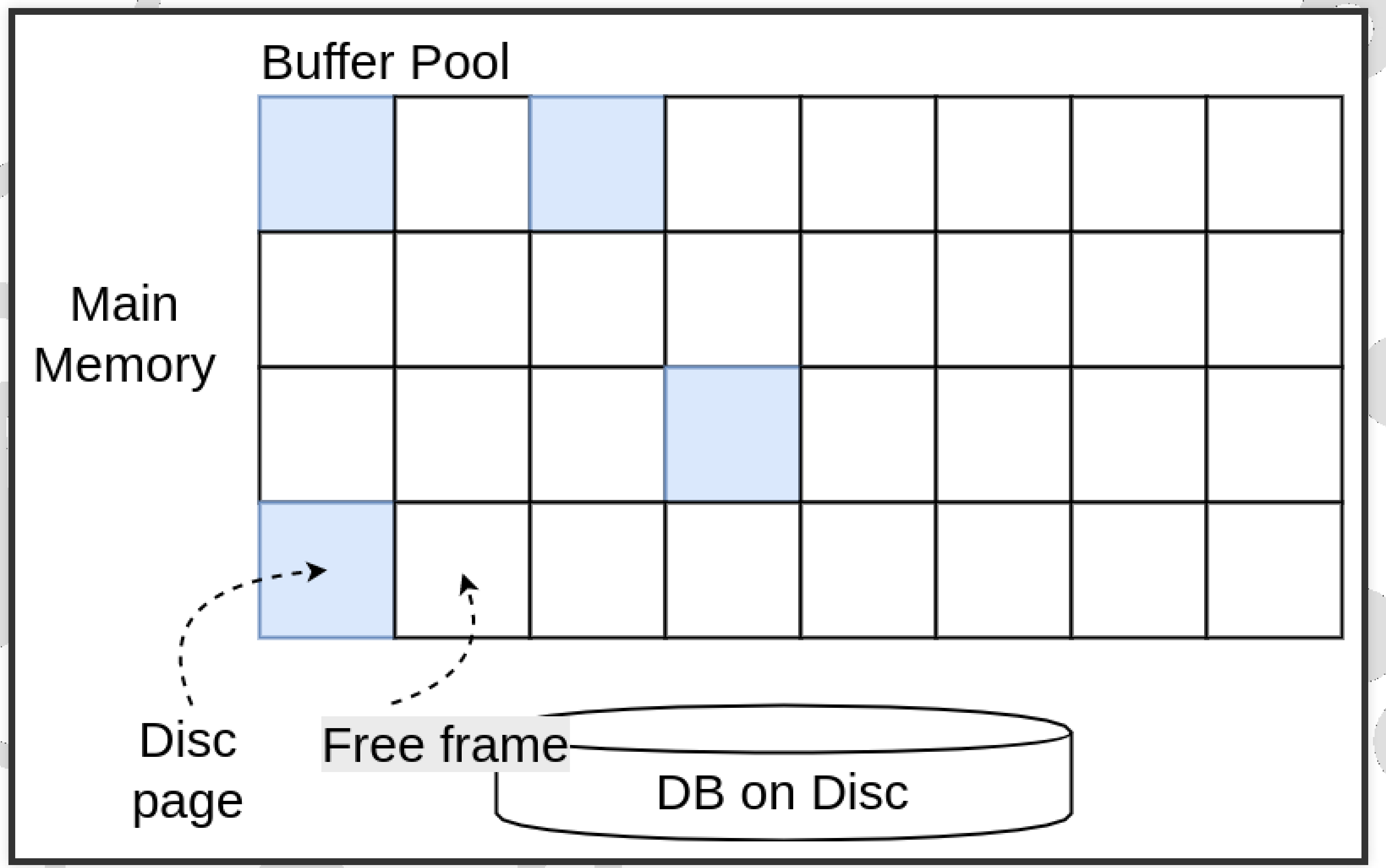
- **Buffer manager** is software layer responsible for bringing pages from disk to main memory as needed.

Manages available free memory by partitioning it into a collection of pages: the **buffer pool**

The main memory pages in the buffer pool is called **frames**

- **Frame** is a slot in main memory that can hold a page

BUFFER POOL



BUFFER MANAGER

Higher layers of DBMS does not have to worry if a page is available or not, this is handled by buffer manager.

The higher layers must however:

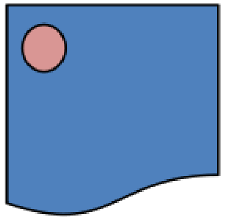
- Release a page when not needed anymore
- Inform if the page has been modified
  - Buffer manager makes sure the page is propagated back to disk

PAGE MAINTENANCE IN A BUFFER POOL

Buffer manager maintains some bookkeeping information

- *pin_count* - Number of times the page has been requested but not released
  - Think of it as number of current users of the page
  - Initially 0 for all pages
- *dirty* - boolean indicating if the page has been modified after being brought in from disk

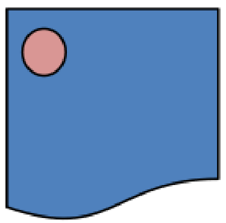
PAGE MAINTENANCE IN A BUFFER POOL



(pin_count = 1)

Keep the page in the buffer when it is being used:

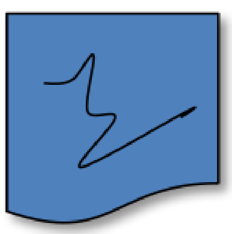
- Pin a frame when its page is requested (pin_count++)



(pin_count = 0)

Considered for replacement while not being used

- Unpin a frame when its page is released (pin_count--)



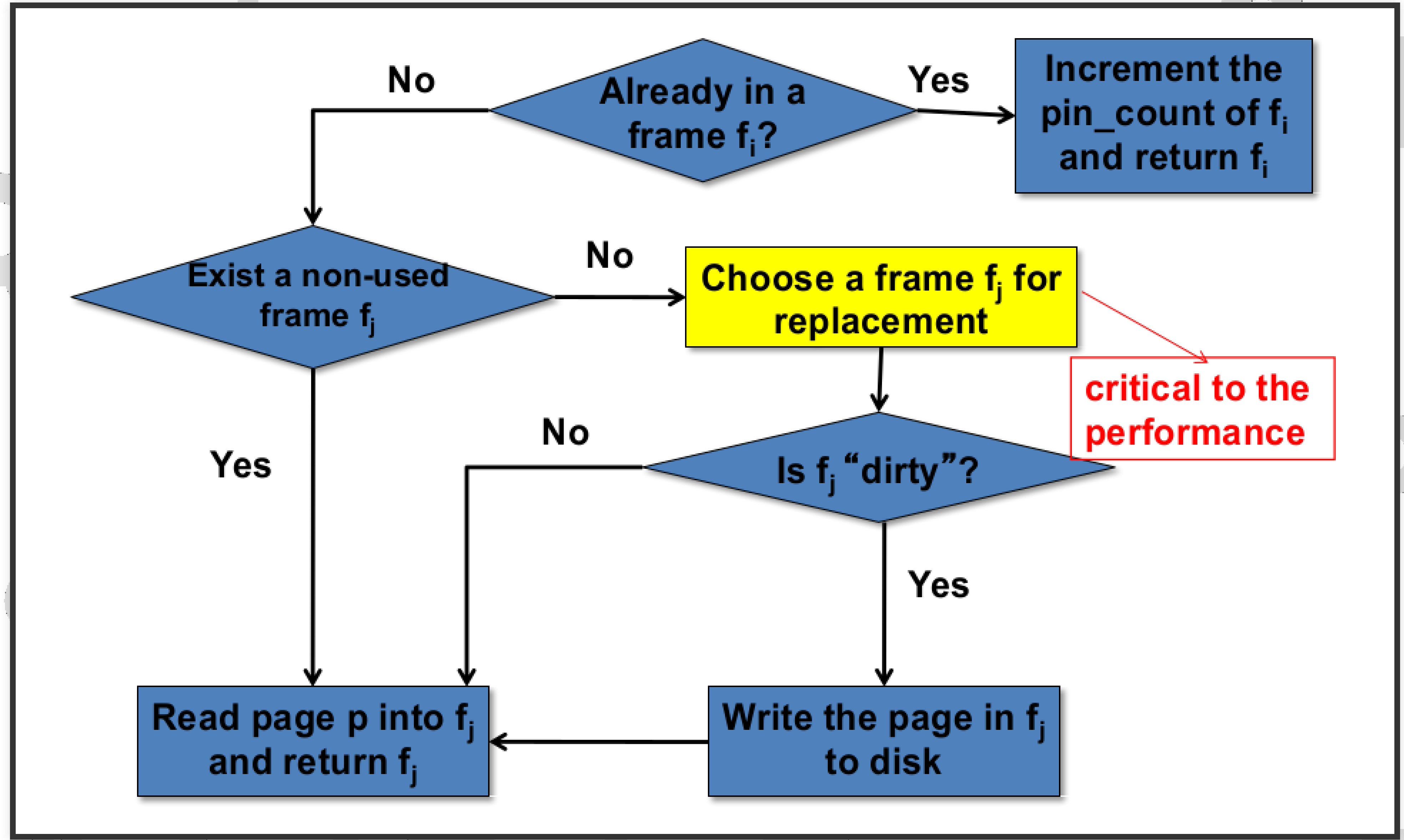
Update in the buffer should be put in the disk

- A page is dirty if it has been modified but not updated on the disk yet

PINNING

- Incrementing the *pin_count* is called **pinning**
- Decrementing the *pin_count* is called **unpinning**

HOW TO PROCESS A PAGE REQUEST?



BUFFER REPLACEMENT POLICIES

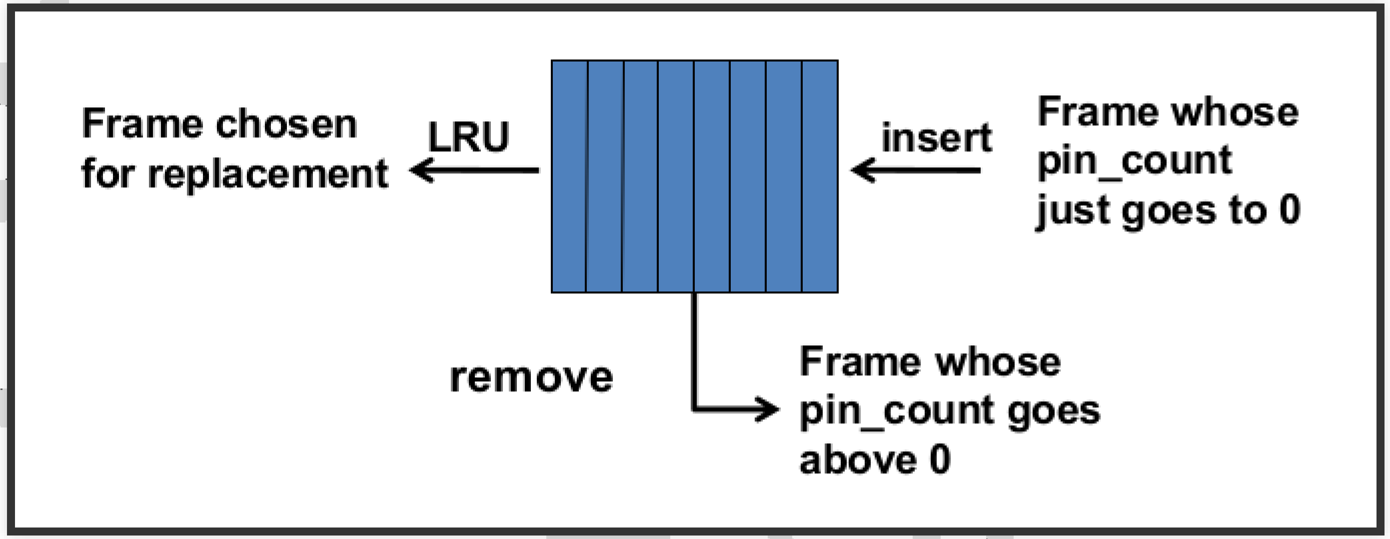
Critical for performance

- General Rule: Keep those pages that might be accessed soon in the future
- A frame is only considered for replacement if `pin_count == 0`

BUFFER REPLACEMENT POLICIES

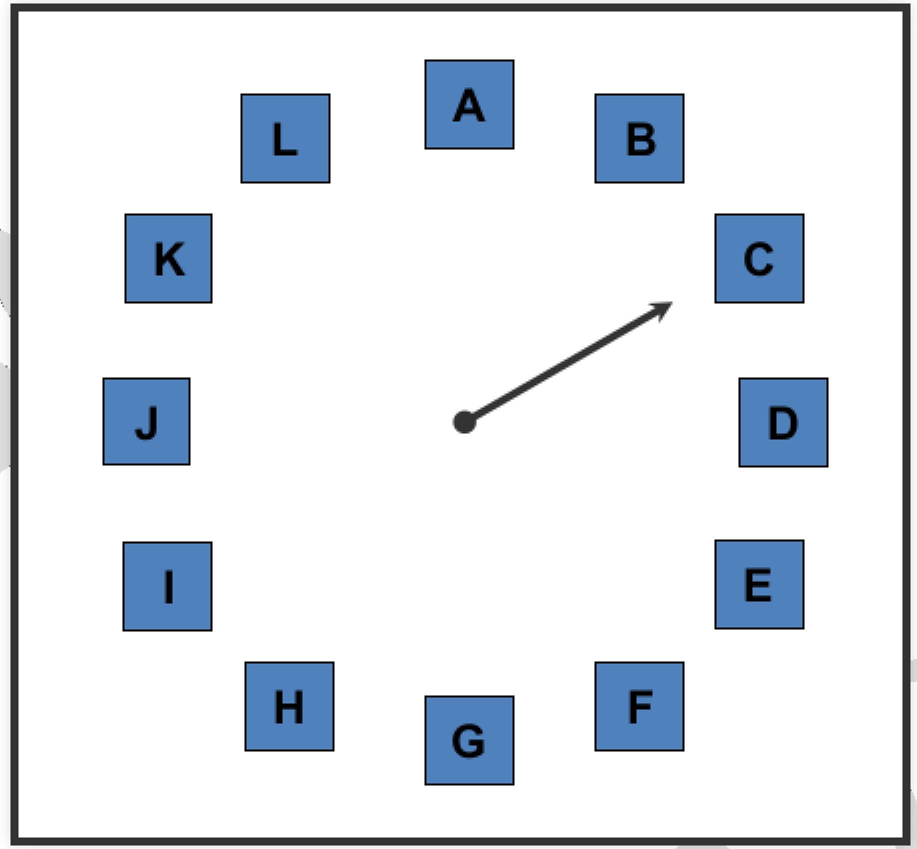
- LRU: Least Recently Used
- CLOCK: Approximating LRU (also called second chance)
- FIFO: First In First Out
- MRU: Most Recently Used
- Random

LEAST RECENTLY USED



Q: What is the assumption of LRU?

CLOCK



CLOCK

- Every frame is associated with a Reference Bit *R*.
- *R* is set to 1 when a frame's pin_count goes down to 0.

On replacement request:

```
1. Advance the pointer.
2. If R == 0 and pin_count==0
    choose the frame.
   Else if R == 1
      set R to 0
   go to step 1.
```

BUFFER MANAGEMENT IN DBMS VS OS

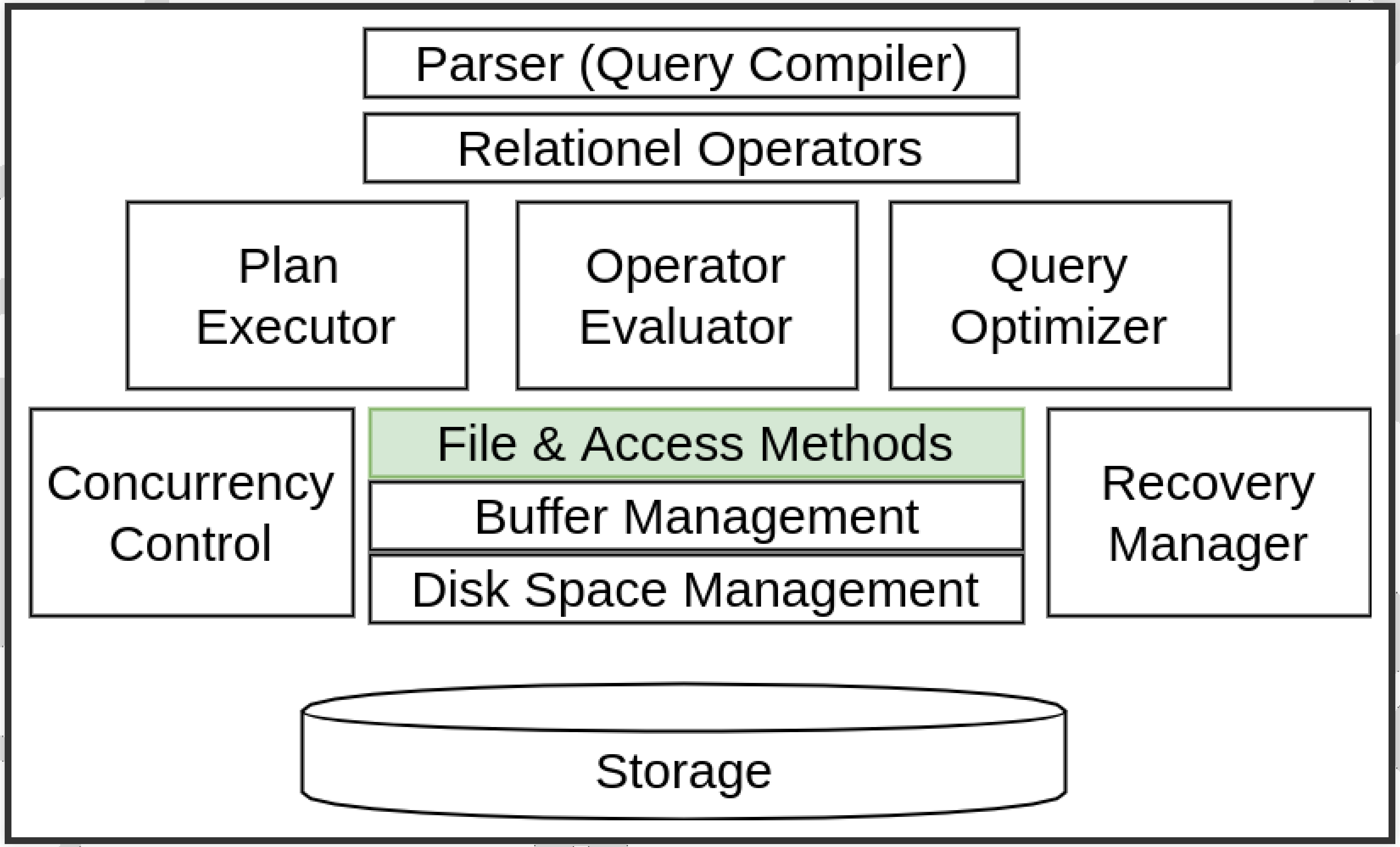
- DBMS can often predict the order in which pages are referenced
  - **page reference patterns**
- **Prefetching of pages:** Anticipate the next several pages and bring them into memory before they are requested
- Require the ability to force a page to disk (WAL)

FILES OF RECORDS

The way pages are used to store records and organized into logical collection of files

First: How a collection of pages can be organized as a file

FILE AND ACCESS METHODS



IMPLEMENTING HEAP FILES

- Only guarantee: You can retrieve all records if you repeat requests for next record.
- Each record in the file has a unique rid
- Every page in the file is the same size

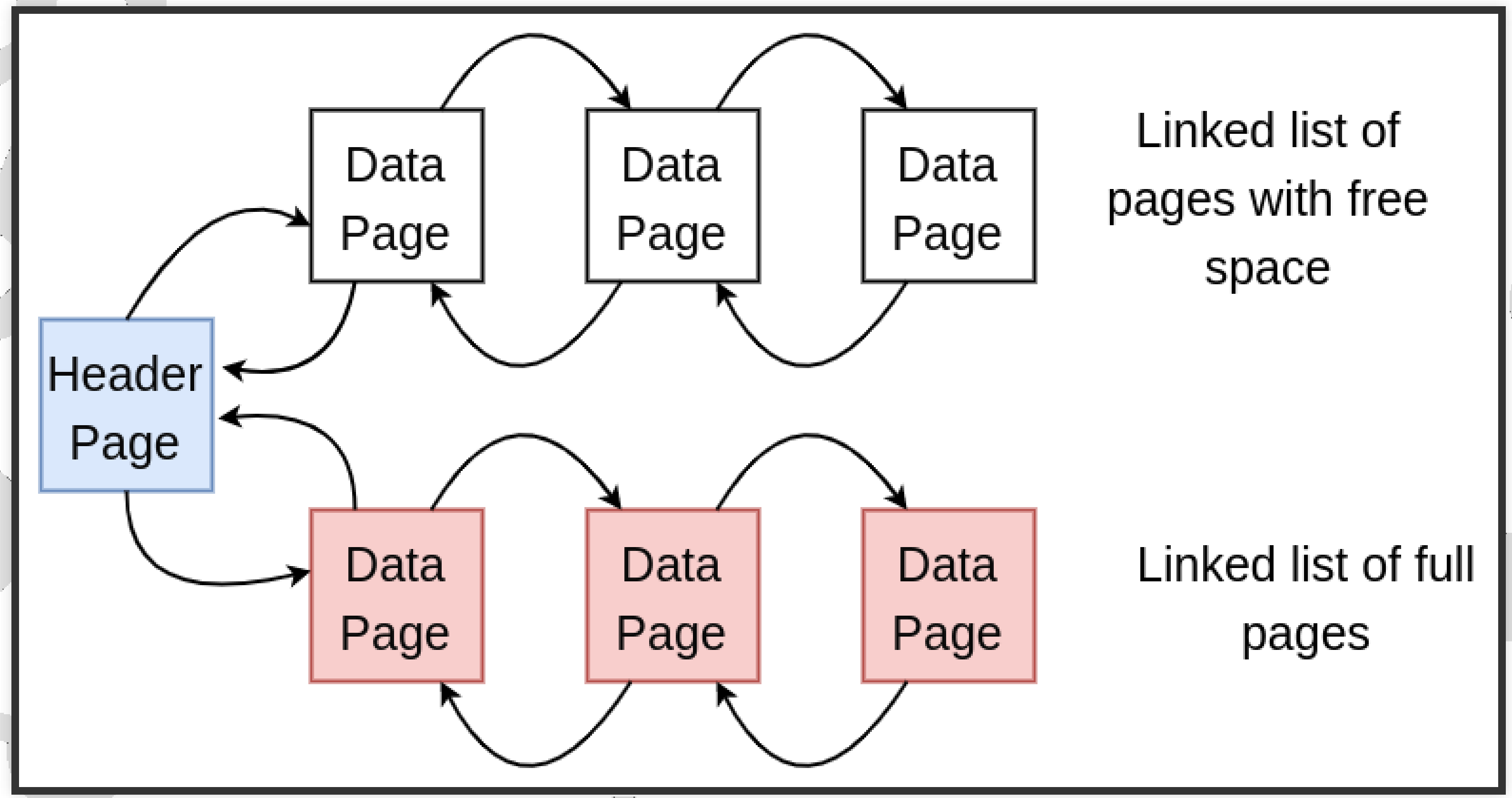
IMPLEMENTING HEAP FILES

- Supported operations:
  - *create* and *destroy* file
  - *insert* a record
  - *delete* a record with a given rid
    - Note we must be able to find the page from the rid
  - *get* a record with a given rid
  - *scan* all records in a file

Must keep track of pages with free space to implement insertion efficiently

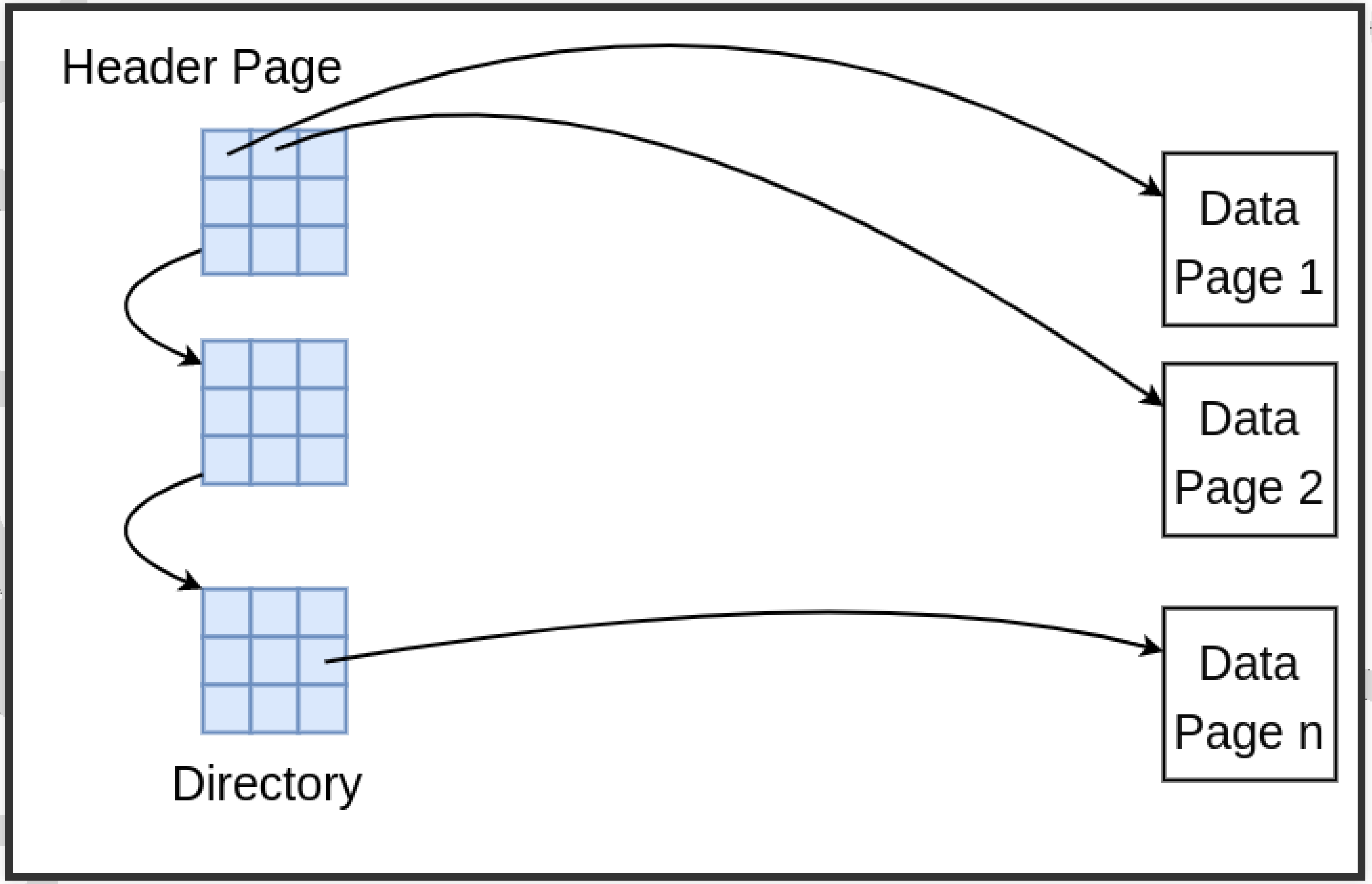
LINKED LIST OF PAGES

Linked-list format maintains two linked list of data pages used by a file



Where to find the header page? → System Catalog

DIRECTORY OF PAGES



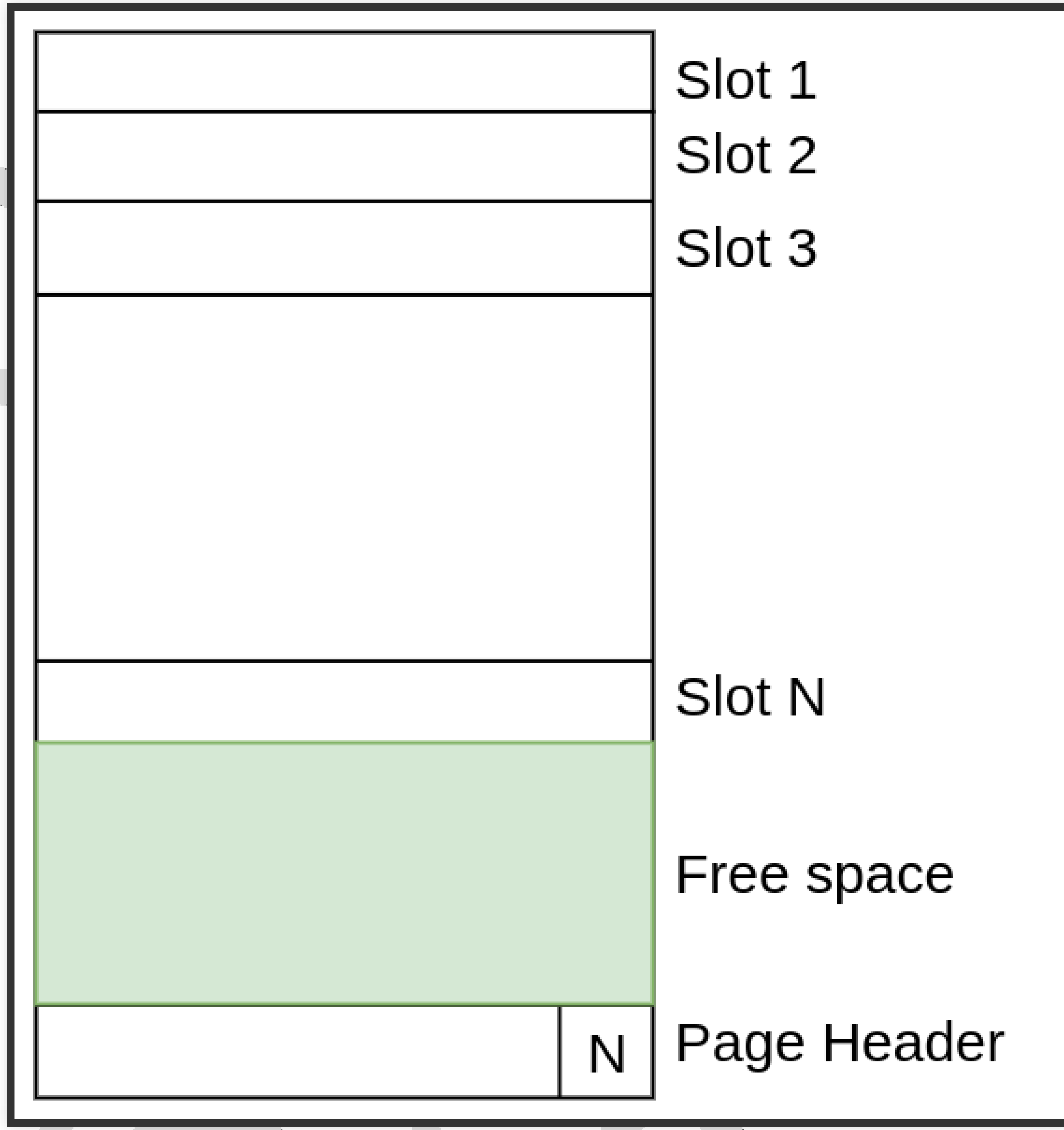
PAGE FORMATS

- Page abstraction fine when dealing with I/O
- Higher level of DBMS see data as a collection of records
- Here: consider how collection of records can be arranged on a page
- Think of page as a collection of **slots**, each containing a record
- record is identified by using the pair `<page id, slot number>`

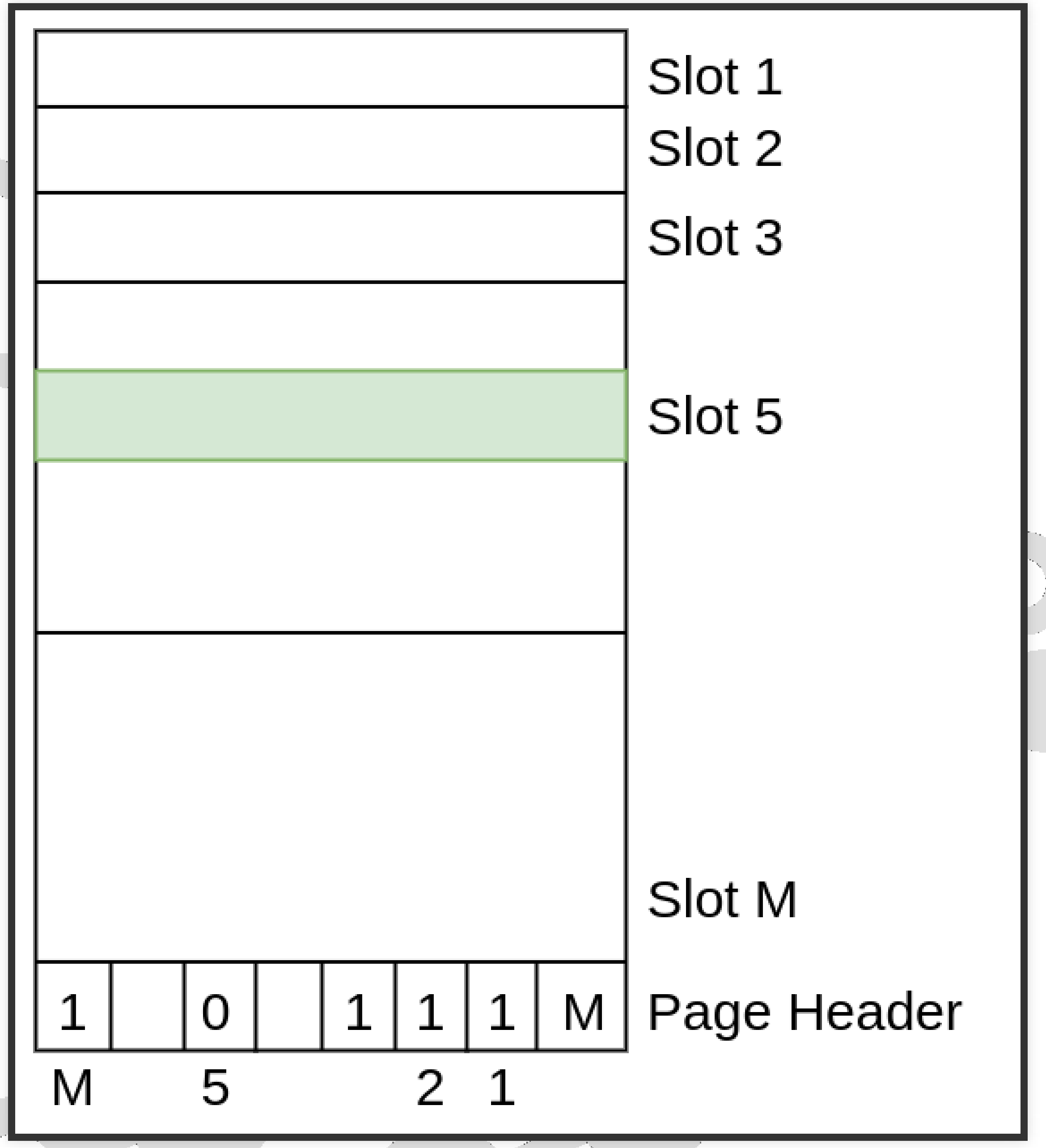
FIXED-LENGTH RECORDS

If all records on the page ar guaranteed to be the same length, record slots are uniform ⇒ Easy to handle

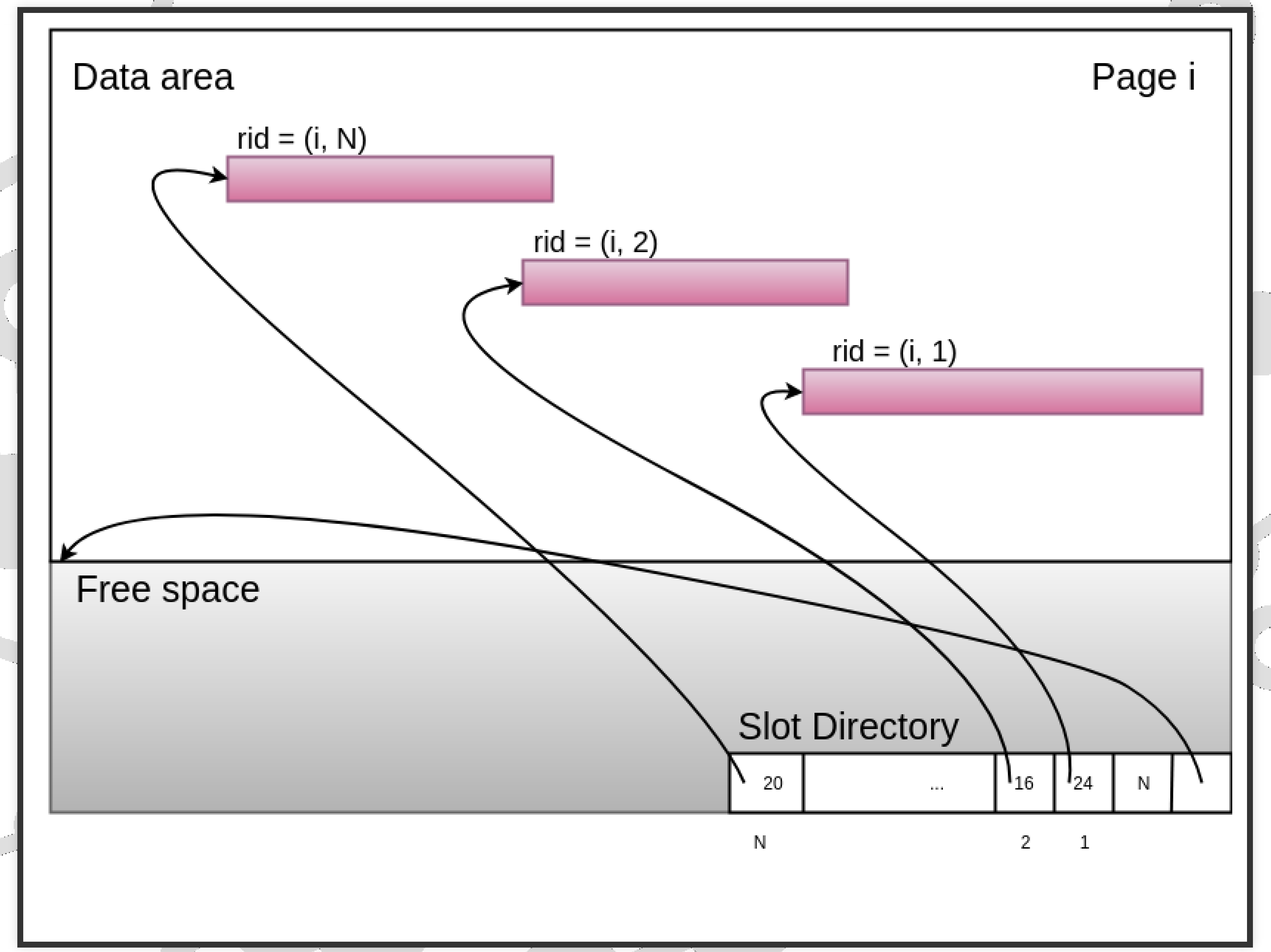
PAGE ORGANISATION - PACKED



PAGE ORGANISATION - UNPACKED



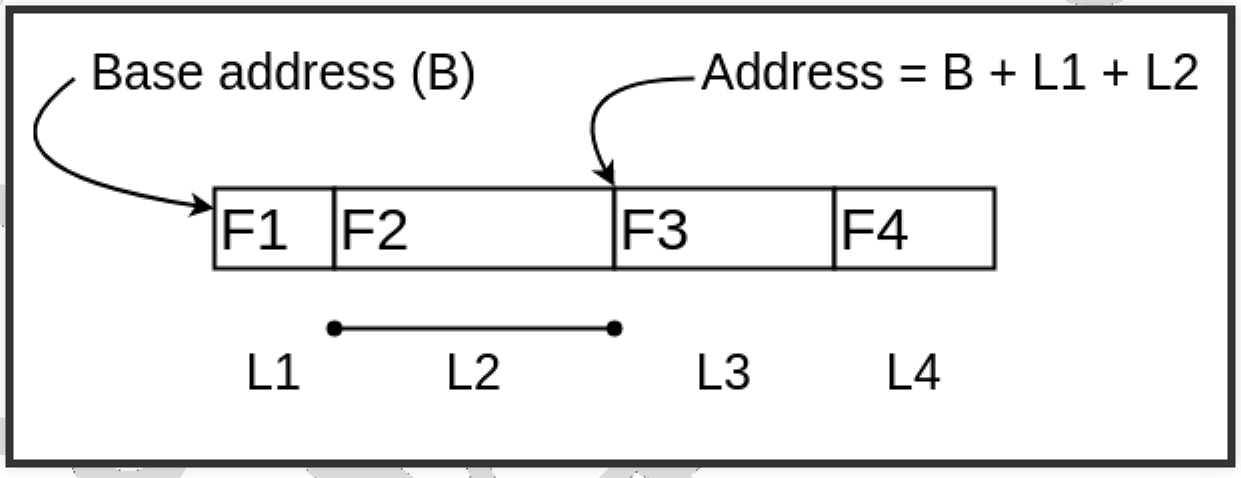
VARIABLE-LENGTH RECORDS



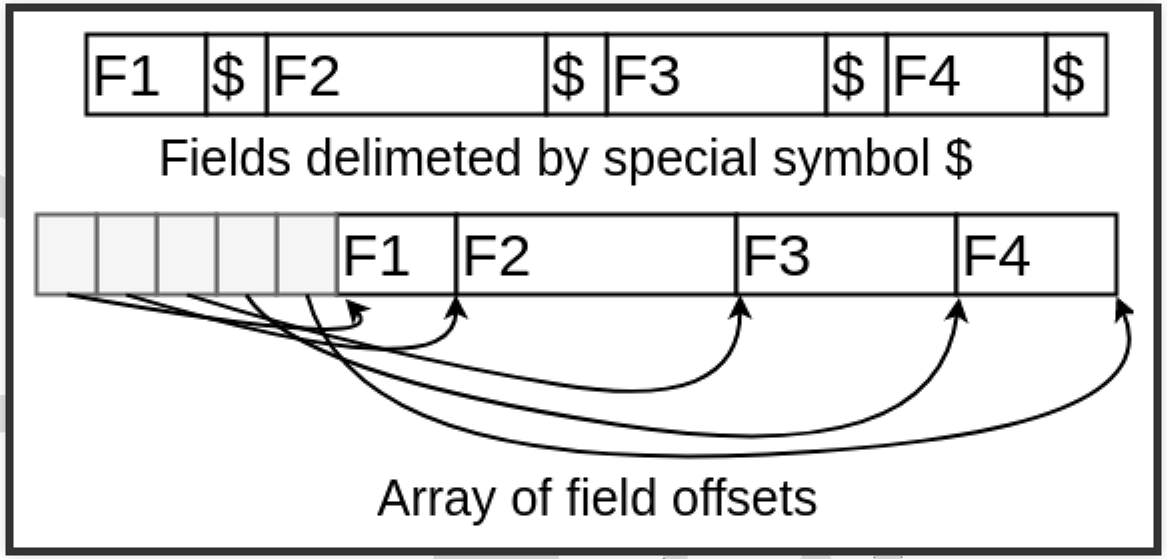
RECORD FORMATS

- How to organize fields within a record
  - Must consider if size is fixed or variable length
- Information common to all records are stored in system catalog
  - Number of fields
  - Field types

FIXED-LENGTH RECORDS



VARIABLE-LENGTH RECORDS



VARIABLE-LENGTH RECORDS

- When record modified, it might grow and have to be moved
  - Maybe to a new page
- Might have to leave a forwarding adress in the page - as page number is included in rid
- Might be it does not fit on a single page
  - Broken down into smaller records - with pointers to next part

CHAPTER 10: TREE-STRUCTURED INDEXING

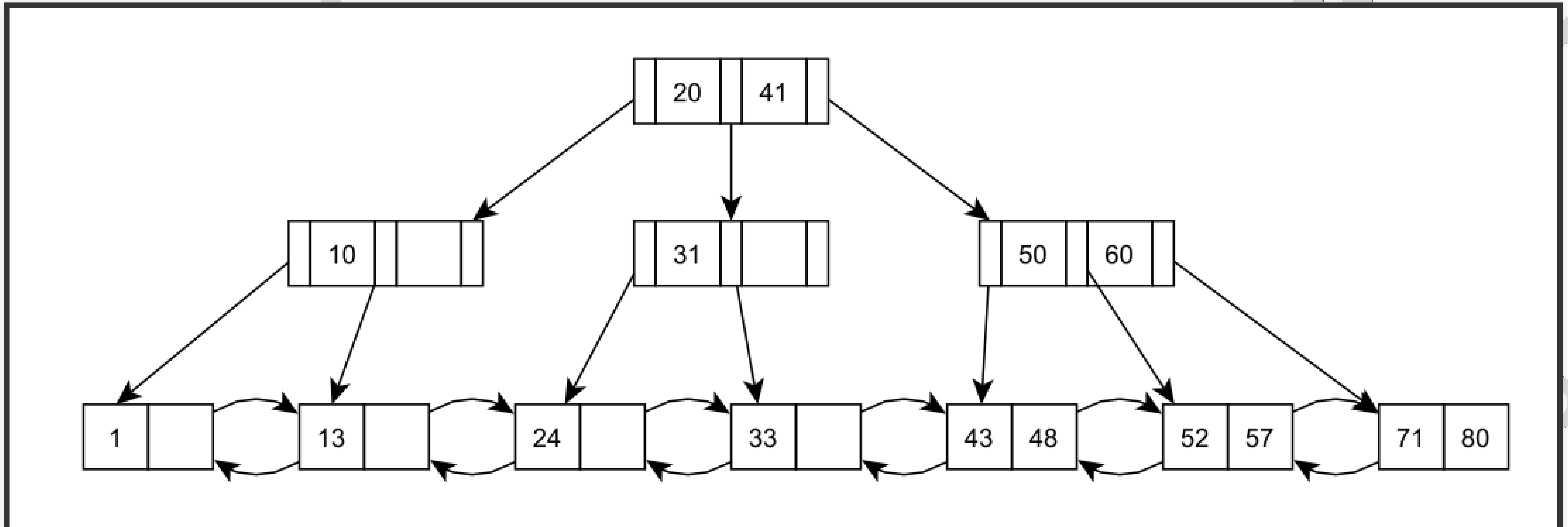
Key Issues

- Search
- Insert
- Delete
- Key-compression
- Bulk-loading
- Performance of operations

TREE-STRUCTURED INDEXING

Should be known

B+ TREE



B+ TREE PROPERTIES

- Insert and Delete keep tree balanced
- Root is either leaf or has between roundUp(M/2) and M children
  - Min occupancy of 50% if not root
- All leaves are at the same depth
- Because of high fan-out, height rarely more than 3-4

CHAPTER 11: HASH-BASED INDEXING

Key Issues

- Static hashing
- Extendible hashing
- Linear Hashing

STATIC HASHING

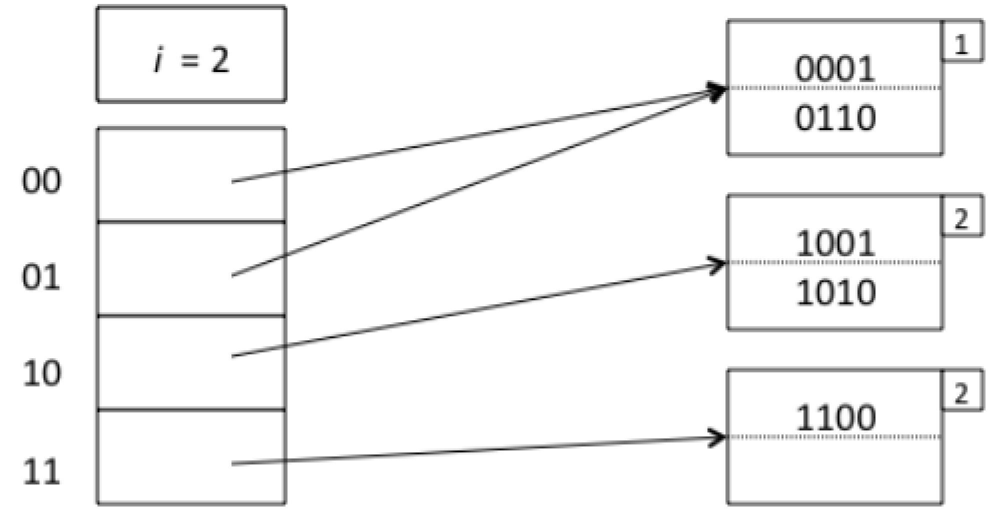
- Buckets
  - Primary page with potential overflow buckets
- hash function
  - *h(value) = (a * value + b)*
- As files tend to grow and bucket number fixed, rebuilding entire index might be necessary
  - Expensive

EXTENDIBLE HASHING

- Directory of pointers to buckets
- Double number of buckets just by doubling the directory
  - Split only bucket that overflows
- Index knows global depth
- Each bucket knows local depth

EXTENDIBLE HASHING

Extensible Hash Table with $k = 4, f = 2$

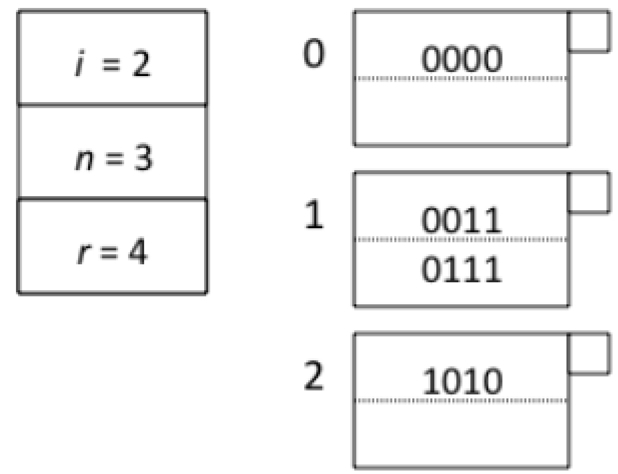


LINEAR HASHING

- Also gracefully grows
- Might not split bucket that overflows when it does
- Family of hash functions each with range twice the size of previous
- **Rounds** of splitting
  - In a round, all buckets from start of round are split
- Next pointer indicates the next bucket to split
- p_{max} indicates limit for splitting

LINEAR HASHING

Linear Hash Table with $k = 4, f = 2, p_{max} = 0.8$



QUESTIONS?