

Module 4

Physical Database Design and Query

File Organization - Indexing: Single level indexing, multi-level indexing, dynamic multilevel Indexing - B+ Tree Indexing – Hashing Techniques: Static and Dynamic Hashing – Relational Algebra - Translating SQL Queries into Relational Algebra – Query Processing – Query Optimization: Algebraic Query Optimization, Heuristic query optimization Rules, Join Query Optimization using Indexing and Hashing - Tuple Relational Calculus.

Reference :

R. Elmasri & S. B. Navathe, Fundamentals of Database Systems, Addison Wesley, 7th Edition, 2016

A. Silberschatz, H. F. Korth & S. Sudarshan, Database System Concepts, McGraw Hill, 7th Edition 2019.

Indexing and Hashing

- Basic Concepts
- Ordered Indices
- B+-Tree Index Files
- B-Tree Index Files
- Static Hashing
- Dynamic Hashing
- Comparison of Ordered Indexing and Hashing
- Index Definition in SQL
- Multiple-Key Access

Introduction

- Database data must be stored physically on storage media.
- DBMS retrieves, updates, and processes stored data.
- Storage hierarchy consists of **primary, secondary, and tertiary storage**.

Primary Storage

- Operated directly by CPU (e.g., **main memory, cache memory**).
- Provides **fast access** to data.
- **Limited storage capacity** compared to enterprise database needs.
- **Expensive** relative to disks.
- **Volatile** – data lost during power failure or system crash.

Secondary Storage

- Primary medium for online storage of enterprise databases.
- Traditionally **magnetic disks** used.
- **Flash memory** (SSDs) increasingly popular.
- **Non-volatile** and permanent storage.
- Suitable for large databases.

Tertiary Storage

- **Optical disks** (CD-ROM, DVD) and **tapes** used for archiving.
- **Removable, offline** storage.
- Provides **large capacity** at **lower cost**.
- **Slower access** compared to primary/secondary.
- Data must be copied to **primary storage** before CPU can process.

Memory Hierarchies and Storage Devices

Memory Hierarchy Concept

- Data resides in multiple storage levels → **speed vs. cost trade-off**.
- **High-speed memory** = expensive, low capacity (e.g., cache).
- **Low-speed memory** = cheaper, very high capacity (e.g., tapes).

Primary Storage

- **Cache Memory (SRAM)**
 - ✓ Fastest, most expensive.
 - ✓ Used by CPU for **prefetching & pipelining**.
- **DRAM (Main Memory)**
 - ✓ Cheaper, larger than cache.
 - ✓ Volatile & slower than SRAM.
 - ✓ Holds program instructions & data.
 - ✓ Now available in **GB to hundreds of GB** range.
 - ✓ **Main-memory databases** possible → useful for **real-time applications** (e.g., telephone switching).

Memory Hierarchies and Storage Devices

Secondary & Tertiary Storage

- **Magnetic Disks (HDDs)**
 - Traditional medium for databases.
- **Flash Memory (SSD, USB, NAND/NOR flash)**
 - ✓ Non-volatile, high performance.
 - ✓ Used in **cameras, mobiles, laptops, USB drives**.
 - ✓ NAND → higher storage capacity per cost.
- **Optical Drives (CD/DVD/Blu-ray)**
 - ✓ CD: ~700 MB, DVD: 4.5–15 GB, Blu-ray: 27–54 GB.
 - ✓ Types: **CD-ROM (read-only), CD-R/DVD-R (WORM – write once, read many)**.
 - ✓ Declining usage due to cheap, high-capacity disks.
- **Magnetic Tapes**
 - ✓ Used for **backup & archiving**.
 - ✓ Very large capacity (terabytes to petabytes with jukeboxes).
 - ✓ Example: **NASA EOS satellite data archives**.

Memory Hierarchies and Storage Devices

Storage Measurement Units

KB = 1,000 bytes

MB = 1 million bytes

GB = 1 billion bytes

TB = 1,000 GB

PB = 1,000 TB (10^{15} bytes) – now relevant for big science & enterprise databases.

Memory Hierarchies and Storage Devices

Comparison of Storage Devices (2014 Commodity Prices)

| Type | Capacity | Access Time | Max Bandwidth | Cost |
|---------------------|-----------------|-------------|---------------------|----------------|
| Main Memory (RAM) | 4 GB – 1 TB | 30 ns | 35 GB/s | \$100 – \$20K |
| Flash SSD | 64 GB – 1 TB | 50 μ s | 750 MB/s | \$50 – \$600 |
| Flash USB | 4 GB – 512 GB | 100 μ s | 50 MB/s | \$2 – \$200 |
| Magnetic Disk (HDD) | 400 GB – 8 TB | 10 ms | 200 MB/s | \$70 – \$500 |
| Optical Storage | 50 – 100 GB | 180 ms | 72 MB/s | ~\$100 |
| Magnetic Tape | 2.5 – 8.5 TB | 10–80 sec | 40–250 MB/s | \$2.5K – \$30K |
| Tape Archival | 25 TB – 2.4M TB | 10–80 sec | 250 MB/s – 1.2 PB/s | \$3K – \$1M+ |

Storage Organization of Databases

Persistent Data

- Stored for long periods, accessed repeatedly.
- Contrast: **Transient data** exists only during program execution.

Why stored on magnetic disks (secondary storage):

- Databases are **too large** for main memory.
- Disk storage is **non-volatile**, unlike volatile main memory.
- **Cheaper cost per unit** compared to primary storage.

Emerging Technologies:

- **SSDs** may replace magnetic disks in some cases.
- Future DBs may use **different hierarchy levels** (main memory → SSD → tape jukebox).
- But **magnetic disks** expected to remain dominant for large DBs.

Other Storage Media:

- **Magnetic tapes** – cheap, used for **backup/archiving**.
- Require operator/auto-loader; **offline access** (slower).
- **Disks** – online, **direct access** anytime.

Storage Organization of Databases

Physical Database Design:

- DB designers/DBA must know **advantages/disadvantages** of storage methods.
- DBMS offers **multiple organization techniques**.
- Choice depends on **application requirements**.

Data Access:

- Applications usually need only a **small portion** of DB at a time.
- Process: **Disk → Main Memory → CPU → (back to Disk if modified)**.
- Data organized as **files of records** (entities, attributes, relationships).

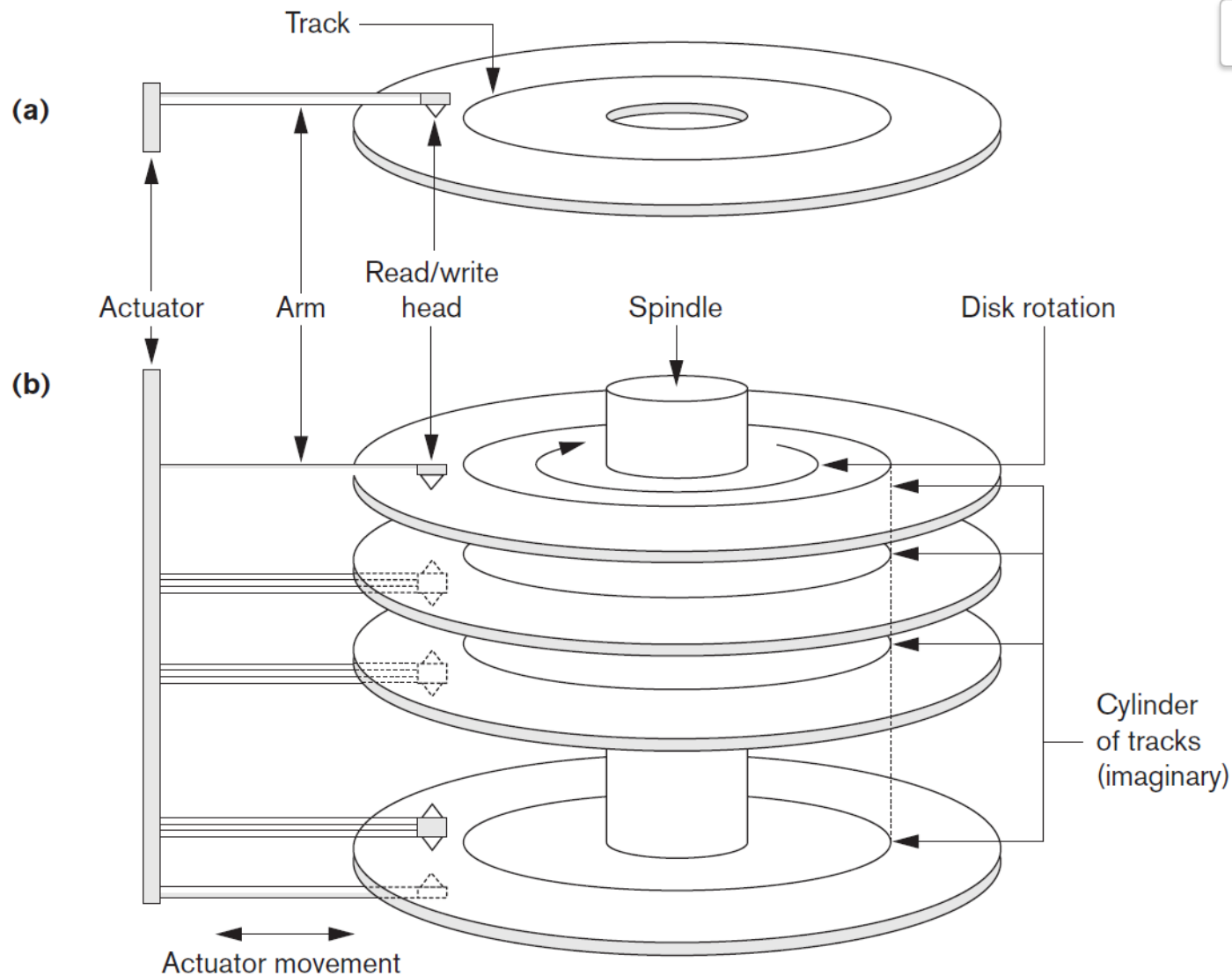
File Organization Techniques:

- **Heap (Unordered file)**: Records appended without order.
- **Sorted (Sequential file)**: Ordered by a field (**sort key**).
- **Hashed file**: Uses **hash function** on a **hash key**.
- **B-trees (and variants)**: Tree-based structure for efficient access.

Secondary (Auxiliary) Organization:

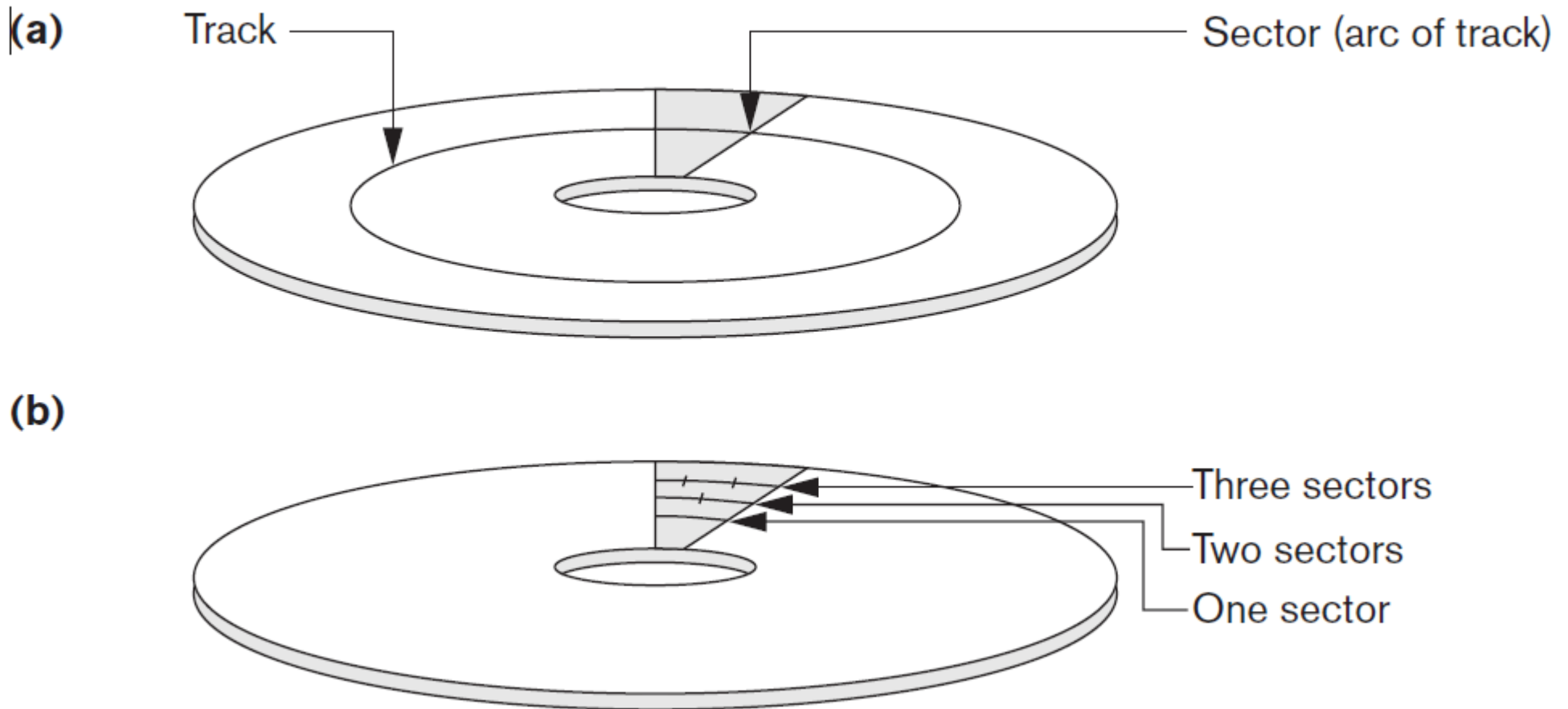
- Provides **efficient access via alternate fields**.
- Implemented mostly as **indexes**

Secondary Storage Devices



(a) A single-sided disk with read/write hardware. (b) A disk pack with read/write hardware.

Secondary Storage Devices



Different sector organizations on disk.

(a) Sectors subtending a fixed angle. (b) Sectors maintaining a uniform recording density

Techniques to Make Disk Data Access More Efficient

1. Buffering of Data

- Balances speed mismatch between CPU and slow HDD.
- Uses memory buffers to hold data temporarily.
- Double buffering improves throughput.

2. Proper Organization of Data on Disk

- Store related data in **contiguous blocks/cylinders**.
- Reduces arm movement and seek time.

3. Reading Data Ahead of Request (Prefetching)

- Reads additional consecutive blocks even if not requested.
- Works well for sequential access. Not effective for random access.

4. Proper Scheduling of I/O Requests

- Organizes block access to minimize arm movement.
- **Elevator algorithm (SCAN)** → services requests in one direction, then reverse

5. Use of Log Disks for Writes

- Dedicated disk (or area) used to store writes sequentially.
- Eliminates seek time.
- Improves write performance but adds cost.

6. Use of SSDs/Flash for Recovery

- SSD/flash used as fast **non-volatile buffer** for updates.
- Prevents update loss in case of system crash.
- Data is then written to HDD, reducing the time taken to write data to disk.

Secondary Storage Devices

Hardware Description of Disk Devices

Magnetic Disks (HDDs):

- ✓ Store large amounts of data.
- ✓ Basic unit: **bit** (0 or 1 via magnetization).
- ✓ Bits grouped into **bytes** (8 bits = 1 character, most common).
- ✓ Capacity = number of bytes it can store (modern disks → **hundreds of GBs to TBs**).

Disk Structure:

- ✓ Made of thin **circular magnetic material**, protected by cover.
- ✓ **Single-sided** (one surface used) or **double-sided** (both surfaces).
- ✓ Disks assembled into **disk packs** (multiple disks/surfaces).
- ✓ Common form factors: **3.5 inch, 2.5 inch**.
- ✓ Data stored in **tracks** (concentric circles).
- ✓ Tracks of same diameter across surfaces = **cylinder** → faster retrieval if data stored in one cylinder.

Secondary Storage Devices

Tracks, Sectors, and Blocks:

- ✓ Disk has **thousands of tracks** (up to ~152,000).
- ✓ Each track divided into **sectors** (fixed, hard-coded).
- ✓ Sector sizes: commonly **512 bytes**.
- ✓ Tracks also divided into **blocks/pages** (set during formatting).
- ✓ Typical block sizes: **512 bytes – several KB**.
- ✓ Several blocks = **cluster** (transferred as a unit).

Read/Write Mechanism:

- ✓ Performed by **disk head** attached to mechanical arm.
- ✓ Multiple surfaces → multiple heads, controlled together by **actuator motor**.
- ✓ Disk rotates continuously (speed: **5,400 – 15,000 rpm**).
- ✓ **Fixed-head disks**: one head per track → faster, but costly.
- ✓ **Movable-head disks**: common, cheaper.

Secondary Storage Devices

Disk Interfaces:

- ✓ **Older:** SCSI (Small Computer System Interface).
- ✓ **Modern:** SATA (Serial ATA) and SAS (Serial Attached SCSI).
- ✓ SATA: common in PCs (1.5 – 6 Gbps).
- ✓ SAS: used in servers, better **IOPS (Input/Output per second)**.
- ✓ Drive sizes:
- ✓ **3.5-inch:** up to 8 TB, 7,200–10,000 rpm.
- ✓ **2.5-inch:** up to 1.2 TB, up to 15,000 rpm.

Disk Access Times:

- ✓ **Seek Time:** Time to move head to correct track (3–10 ms).
- ✓ **Rotational Delay (Latency):** Time for block to rotate under head (depends on rpm; e.g., ~2 ms at 15,000 rpm).
- ✓ **Block Transfer Time:** Time to actually transfer data (smaller).
- ✓ Total = Seek Time + Rotational Delay + Transfer Time.
- ✓ Access time: ~**9 – 60 ms**.
- ✓ Contiguous block transfer much faster (saves seek/latency).

Secondary Storage Devices

Performance Bottleneck:

- ✓ Disk I/O is **slower** compared to CPU/main memory speeds.
- ✓ Hence, **file organization techniques** try to minimize block transfers.
- ✓ Goal: store **related info in contiguous blocks** to reduce delays.

Secondary Storage Devices

Performance Bottleneck:

- ✓ Disk I/O is **slower** compared to CPU/main memory speeds.
- ✓ Hence, **file organization techniques** try to minimize block transfers.
- ✓ Goal: store **related info in contiguous blocks** to reduce delays.

Placing File Records on Disk

- Data in a database is stored as **records** organized into **files**.
- A **record**: a collection of related data values (fields) representing an entity.
- A **record type**: defines the fields and their data types.
- Files contain sequences of records; placement affects performance
Each record describes an **entity** (e.g., EMPLOYEE).
- Fields have **data types**:
 - ✓ Numeric: integer, long, floating point
 - ✓ String: fixed-length or variable-length
 - ✓ Boolean: 0/1 or TRUE/FALSE
 - ✓ Date/Time: fixed-size format

Example in C:

```
struct employee {  
    char name[30];  
    char ssn[9];  
    int salary;  
    int job_code;  
    char department[20];  
};
```

File Types

Large Data Items

- ✓ **BLOB (Binary Large Object):** stores images, audio, video, or free text.
- ✓ Usually stored **separately** from the main record.
- ✓ Example: CLOB (Character Large Object) for free text.
- ✓ Alternatives:
 - Store as files in file systems
 - Store as files managed by database
 - Break into pieces and store in multiple tuples in separate relation
 - PostgreSQL TOAST
- **Fixed-length records:** all records same size.
- **Variable-length records:** record sizes differ due to:
 1. Variable-length fields (e.g., Name field)
 2. Repeating fields (multi-valued)
 3. Optional fields
 4. Mixed record types

File Types- Fixed-Length Records

- ✓ All fields have **fixed sizes** → predictable record length.
- ✓ Example: EMPLOYEE record = 71 bytes
- ✓ Easy to locate field values by byte offset.
- ✓ Limitation: wasted space for optional or repeating fields.
- ✓ Simple approach:
 - 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
 - Modification: do not allow records to cross block boundaries

File Types- Fixed-Length Records

| | | | | |
|-----------|-------|------------|------------|-------|
| record 0 | 10101 | Srinivasan | Comp. Sci. | 65000 |
| record 1 | 12121 | Wu | Finance | 90000 |
| record 2 | 15151 | Mozart | Music | 40000 |
| record 3 | 22222 | Einstein | Physics | 95000 |
| record 4 | 32343 | El Said | History | 60000 |
| record 5 | 33456 | Gold | Physics | 87000 |
| record 6 | 45565 | Katz | Comp. Sci. | 75000 |
| record 7 | 58583 | Califieri | History | 62000 |
| record 8 | 76543 | Singh | Finance | 80000 |
| record 9 | 76766 | Crick | Biology | 72000 |
| record 10 | 83821 | Brandt | Comp. Sci. | 92000 |
| record 11 | 98345 | Kim | Elec. Eng. | 80000 |

Record 3 deleted

| | | | | |
|-----------|-------|------------|------------|-------|
| record 0 | 10101 | Srinivasan | Comp. Sci. | 65000 |
| record 1 | 12121 | Wu | Finance | 90000 |
| record 2 | 15151 | Mozart | Music | 40000 |
| record 4 | 32343 | El Said | History | 60000 |
| record 5 | 33456 | Gold | Physics | 87000 |
| record 6 | 45565 | Katz | Comp. Sci. | 75000 |
| record 7 | 58583 | Califieri | History | 62000 |
| record 8 | 76543 | Singh | Finance | 80000 |
| record 9 | 76766 | Crick | Biology | 72000 |
| record 10 | 83821 | Brandt | Comp. Sci. | 92000 |
| record 11 | 98345 | Kim | Elec. Eng. | 80000 |

■ Deletion of record i : alternatives:

- move records $i + 1, \dots, n$ to $i, \dots, n - 1$
- move record n to i
- do not move records, but link all free records on a *free list*

File Types- Fixed-Length Records

- Deletion of record i : alternatives:
 - move records $i + 1, \dots, n$ to $i, \dots, n - 1$
 - **move record n to i**
 - do not move records, but link all free records on a *free list*

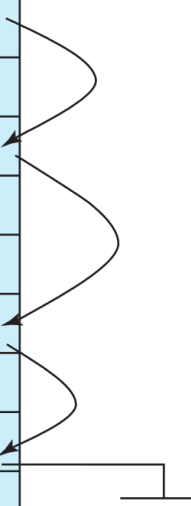
Record 3 deleted and replaced by record 11

| | | | | |
|-----------|-------|------------|------------|-------|
| record 0 | 10101 | Srinivasan | Comp. Sci. | 65000 |
| record 1 | 12121 | Wu | Finance | 90000 |
| record 2 | 15151 | Mozart | Music | 40000 |
| record 11 | 98345 | Kim | Elec. Eng. | 80000 |
| record 4 | 32343 | El Said | History | 60000 |
| record 5 | 33456 | Gold | Physics | 87000 |
| record 6 | 45565 | Katz | Comp. Sci. | 75000 |
| record 7 | 58583 | Califieri | History | 62000 |
| record 8 | 76543 | Singh | Finance | 80000 |
| record 9 | 76766 | Crick | Biology | 72000 |
| record 10 | 83821 | Brandt | Comp. Sci. | 92000 |

File Types- Fixed-Length Records

- Deletion of record i : alternatives:
 - move records $i + 1, \dots, n$ to $i, \dots, n - 1$
 - move record n to i
 - **do not move records, but link all free records on a *free list***

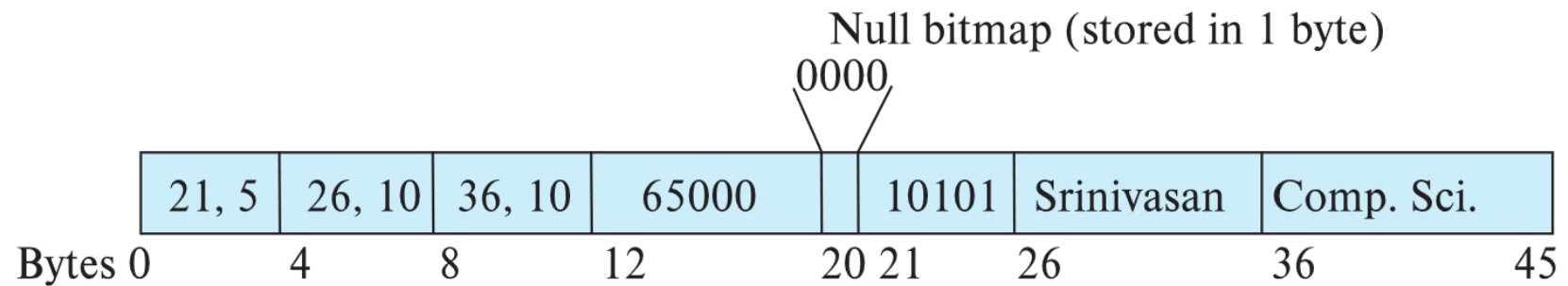
| | | | | |
|-----------|-------|------------|------------|-------|
| header | | | | |
| record 0 | 10101 | Srinivasan | Comp. Sci. | 65000 |
| record 1 | | | | |
| record 2 | 15151 | Mozart | Music | 40000 |
| record 3 | 22222 | Einstein | Physics | 95000 |
| record 4 | | | | |
| record 5 | 33456 | Gold | Physics | 87000 |
| record 6 | | | | |
| record 7 | 58583 | Califieri | History | 62000 |
| record 8 | 76543 | Singh | Finance | 80000 |
| record 9 | 76766 | Crick | Biology | 72000 |
| record 10 | 83821 | Brandt | Comp. Sci. | 92000 |
| record 11 | 98345 | Kim | Elec. Eng. | 80000 |



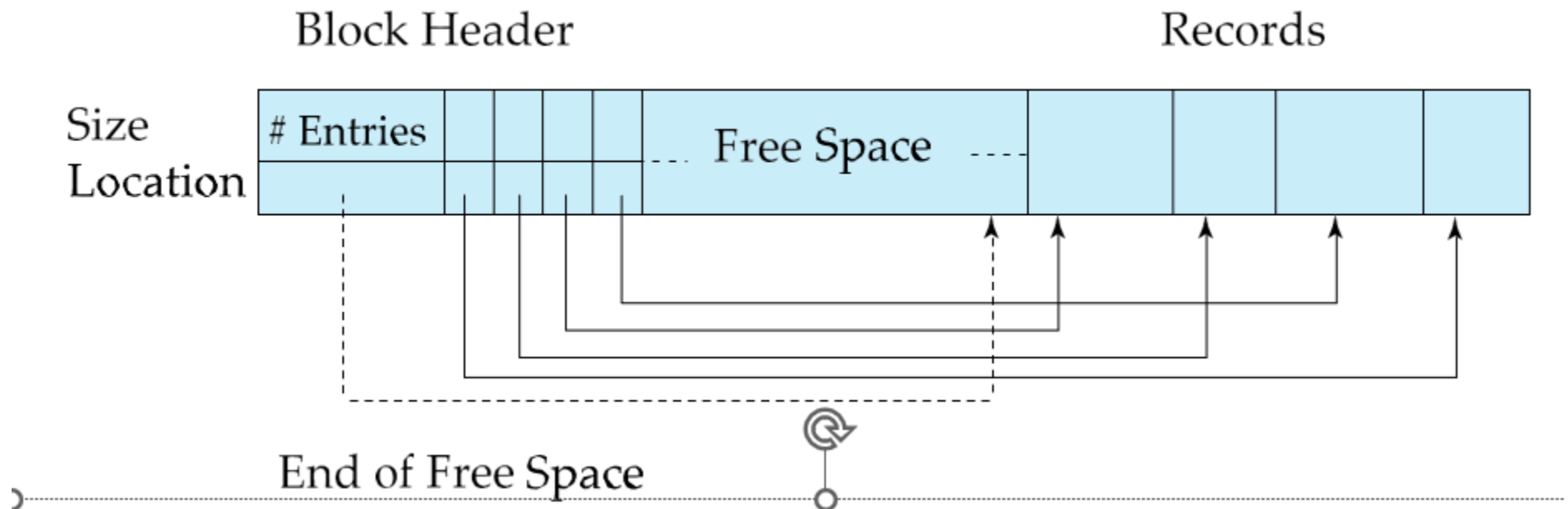
File Types- Variable-Length Records

- ✓ Field lengths can vary; require **special handling**:
 - **Separator characters** (e.g., ?, %, \$) between fields
 - **Field-name/value pairs** for optional fields
 - **Field-type/value pairs** for efficiency
 - Repeating fields: separators between values, terminator at end
- ✓ 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 such as strings (**varchar**)
 - Record types that allow repeating fields (used in some older data models).
- ✓ Attributes are stored in order
- ✓ Variable length attributes represented by fixed size (offset, length), with actual data stored after all fixed length attributes
- ✓ Null values represented by null-value bitmap

File Types- Variable-Length Records



Null values represented by null-value bitmap



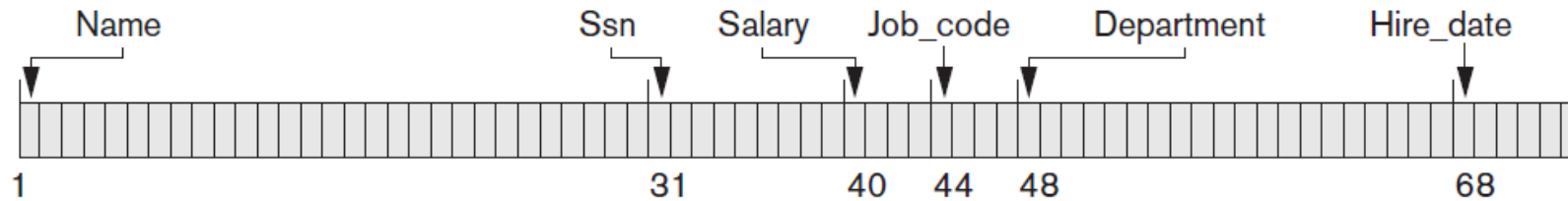
Slotted Page Structure

File Types- Variable-Length Records

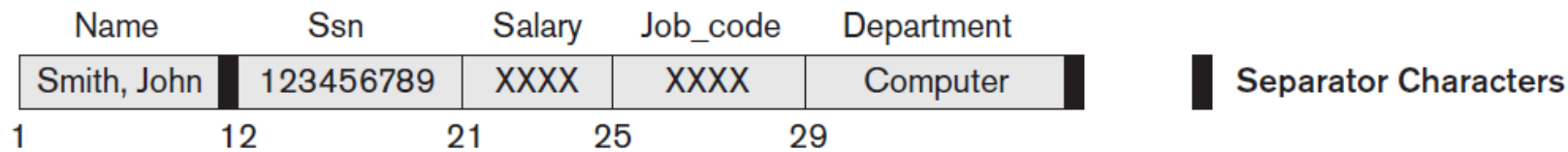
- **Slotted page** header contains:
 - number of record entries
 - end of free space in the block
 - location and size of each record
- Records can be moved around within a page to keep them contiguous with no empty space between them; entry in the header must be updated.
- Pointers should not point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

File Types- Three record storage formats.

(a)



(b)



(c)



| Separator Characters | |
|----------------------|---------------------------------------|
| = | Separates field name from field value |
| █ | Separates fields |
| X | Terminates record |

Three record storage formats. (a) A fixed-length record with six fields and size of 71 bytes. (b) A record with two variable-length fields and three fixed-length fields. (c) A variable-field record with three types of separator characters.

Record Blocking and Spanned vs Unspanned Records

Disk Storage and File Organization

- ✓ Disk blocks are the **unit of data transfer** between disk and memory.
- ✓ Each block can store multiple records if the **block size (B) > record size (R)**.
- ✓ **Blocking factor (bfr)**: number of records per block

$$bfr = \lfloor B/R \rfloor$$

- ✓ Unused space per block:

$$B - (bfr \times R) \text{ bytes}$$

Record Blocking and Spanned vs Unspanned Records

Spanned Records :

- ✓ Part of a record can be stored on one block and the rest on another.
- ✓ Requires a **pointer** to the next block containing the remainder.
- ✓ **Advantages:**
 - Utilizes unused space efficiently
 - Useful for **records larger than a block**
- ✓ **Applicable to:** Fixed-length or variable-length records when record size $>$ block size

Record Blocking and Spanned vs Unspanned Records

UnSpanned Records :

- ✓ Records cannot cross block boundaries..
- ✓ Requires a **pointer** to the next block containing the remainder.
- ✓ **Advantages:**
 - Each record starts at a **known position** within a block
 - Simplifies **record processing**
- ✓ **Applicable to:** Fixed-length records where block size > record size

Record Blocking and Spanned vs Unspanned Records

Choosing Between Spanned and Unspanned

| Organization Type | Pros | Cons | Use Case |
|-------------------|-----------------------|---|----------------------------------|
| Spanned | Efficient space usage | More complex processing | Large or variable-length records |
| Unspanned | Simple processing | Wasted space in partially filled blocks | Fixed-length small records |

File Block Allocation Techniques

Efficiently store file blocks on disk for fast access and expansion.

Common techniques:

1. Contiguous Allocation

- ✓ Blocks stored in consecutive disk locations
- ✓ Pros: Fast sequential read (double buffering)
- ✓ Cons: Difficult to expand file

2. Linked Allocation

- ✓ Each block contains a pointer to the next block
- ✓ Pros: Easy to expand file
- ✓ Cons: Slow sequential read

File Block Allocation Techniques

3. Combined/Clustered Allocation

- Groups of consecutive blocks (clusters) are linked
- Clusters also called file segments or extents

4. Indexed Allocation

- One or more index blocks store pointers to file blocks
- Often combined with other techniques

File Headers

- Metadata about the file needed for access.
- **Contents of a file header:**
 - Disk addresses of file blocks
 - Record format description
 - **Fixed-length unspanned records:** field lengths & field order
 - **Variable-length records:** field-type codes, separator characters, record type codes
- Programs use header information to locate and read records efficiently.

Searching Records Using File Headers

- When searching for a record:
 - Copy one or more blocks into **main memory buffers**
 - Use **file header info** to identify record location
- Without block address info: **linear search** through all blocks is needed
 - Time-consuming for large files
- **Goal of file organization:** Avoid linear search and minimize full file scans

Operations on Files

Two main categories:

1. **Retrieval operations:** Locate records without changing data.
2. **Update operations:** Change data by inserting, deleting, or modifying records.
 - Selection condition (filtering) used to locate records:
 - Example: (Ssn = '123456789')_{or} (Salary >= 30000 AND Department = 'Research')
 - Complex conditions are **decomposed** into simple conditions for locating records.

Record-at-a-Time Operations

Operations applied to a single record rather than multiple records simultaneously.

- **Open:** Prepare file for access, allocate buffers, read file header.
- **Reset:** Move file pointer to beginning.
- **Find / Locate:** Search for first record satisfying condition, load block into buffer.
- **Read / Get:** Copy current record into program variable, may advance pointer.
- **FindNext:** Locate next record satisfying condition.
- **Delete:** Remove current record from file.
- **Modify:** Change fields of current record.
- **Insert:** Add new record into file block and update disk.
- **Close:** Release buffers and complete file access.

Scan Operation and Set-at-a-Time Operations

Scan Operation

- Combines **Find**, **FindNext**, and **Read** into a single operation.
- Returns first or next record satisfying a condition.
- Simplifies sequential access for retrieval operations.

Set-at-a-Time Operations

Operate on multiple records at once:

- **FindAll**: Retrieve all records satisfying a condition.
- **Find n**: Retrieve first n records satisfying a condition.
- **FindOrdered**: Retrieve records in a specified order.
- **Reorganize**: Reorder records (e.g., by sorting) for efficiency.

File Organization vs Access Method and Considerations for File Organization

- ✓ **File organization:** How data is physically stored in records, blocks, and structures.
- ✓ **Access method:** Set of operations applied to a file (e.g., Find, Scan).
- ✓ Multiple access methods can operate on a single file organization.
- ✓ Indexed access requires an index; not all methods are universally applicable

- Expected usage patterns influence organization:
- **Static files:** mostly read-only, few updates
- **Dynamic files:** frequent insert, delete, modify operations
- Optimizing for common search conditions improves efficiency

Organization of Records in Files

- **Heap** – record can be placed anywhere in the file where there is space
- **Sequential** – store records in sequential order, based on the value of the search key of each record
- In a **multitable 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
- **B⁺-tree file organization**
 - Ordered storage even with inserts/deletes
- **Hashing** – a hash function computed on search key; the result specifies in which block of the file the record should be placed

Heap File Organization

- Heap file = simplest and most basic file organization.
- Records are stored in the order they are inserted (**no sorting**).
- **New records are appended to the end of the file.**
- Commonly used for:
 - Collecting and storing records for future use.
 - Supporting secondary indexes for faster access.
 - Easy insertion, but searching and deletion are less efficient.

Insertion

Very efficient operation:

- Copy the last disk block into buffer.
- Add new record into buffer.
- Write block back to disk.
- File header keeps address of the last block.
- Minimal overhead compared to other organizations.

Heap File Organization

Searching

- No ordering of records → requires **linear search**.
- Must scan block by block until condition is met.
- Average cost:
 - If one record satisfies condition → $\sim b/2$ blocks must be checked (where b = number of blocks).
 - If no records (or multiple records) match → all b blocks must be scanned.
- Searching is expensive for large files.

Deletion

Two main methods:

1. Physical Deletion

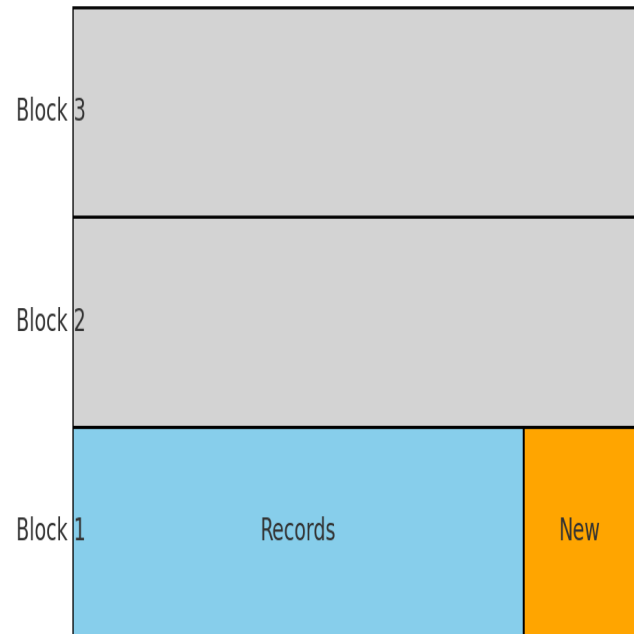
- Find the block, load into buffer, delete record, write back.
- Leaves unused space in the block.
- Deleting many records wastes storage space.

2. Logical Deletion (Deletion Marker)

- Add an extra bit/byte to mark record as deleted.
- Valid records are those without deletion marker.
- Both methods require **periodic reorganization** to reclaim unused space.

Heap File Organization

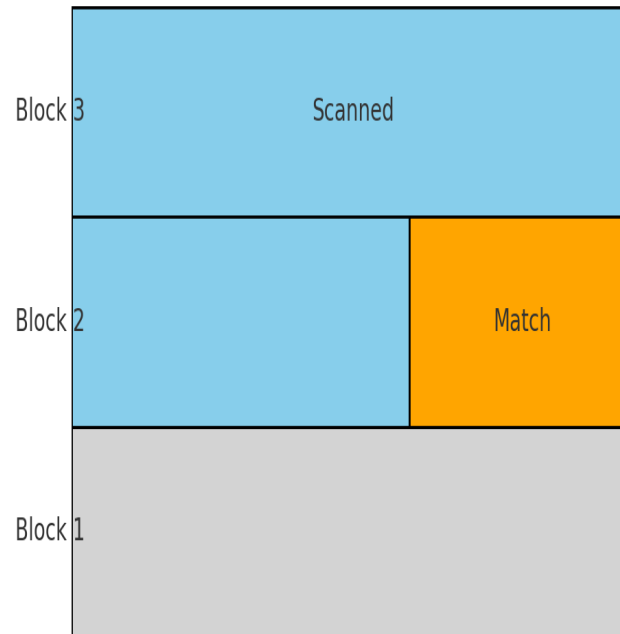
Heap File: Insertion



Insertion

- New records are always added at the **end of the file (last block)**.
- Example: Orange block = newly inserted record.

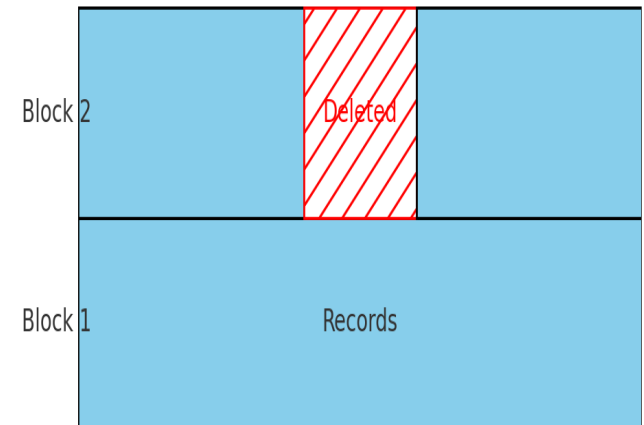
Heap File: Searching



Searching

- Searching requires a **linear scan**.
- Blocks are checked one by one until the match is found.
- Example: Blue = scanned, Orange = matching record found.

Heap File: Deletion



Deletion

- To delete, the record is located and removed.
- This leaves **unused space** (shown with red hatched area).
- Over time, too many deletions waste space → requires **reorganization**.

Heap File Organization

Reorganization

- Required when many deletions have occurred.
- Process:
 - Sequentially access all blocks.
 - Pack valid records together, remove deleted ones.
 - Fill blocks to full capacity again.
- Alternative approach:
 - Reuse deleted record space during new insertions.
 - Requires bookkeeping to track free space.

Heap - Record Organization Options

Spanned vs. Unspanned Organization:

- Spanned: Records can span across multiple blocks.
- Unspanned: Each record fits entirely within a single block.

Fixed-Length vs. Variable-Length Records:

Fixed: Easier to manage and access.

Variable: More flexible, but modification may require deletion + reinsertion.

Sorting records:

To read in sorted order, file must be copied and sorted.

Sorting is expensive for large files → **external sorting** is used.

Heap File Organization

Relative (Direct) Files

- Special type of unordered file with **fixed-length records + unspanned blocks**.
- Allows **direct access to records by position**:
 - If file has r records numbered $0 \dots r-1$, and block has bfr (blocking factor) records:
 - Record i is located in block $\lfloor i / bfr \rfloor$
 - Position = $(i \bmod bfr)$ in that block.
- Provides **direct positional access** (efficient).
- Still does not help with conditional searches.
- Useful for building access paths like indexes.

Heap File Organization

Need for Free-Space Tracking

- Since records can go into **any available free space**, the system must be able to **quickly find blocks with enough empty space**.
- If we scan every block to find free space, it becomes **very expensive**.
- Solution: use a **Free-Space Map**.

Free-Space Map

- A **free-space map** is an **array** with one entry per block in the file
- Each entry is only a few **bits or a byte**, recording the **fraction of the block that is free**.

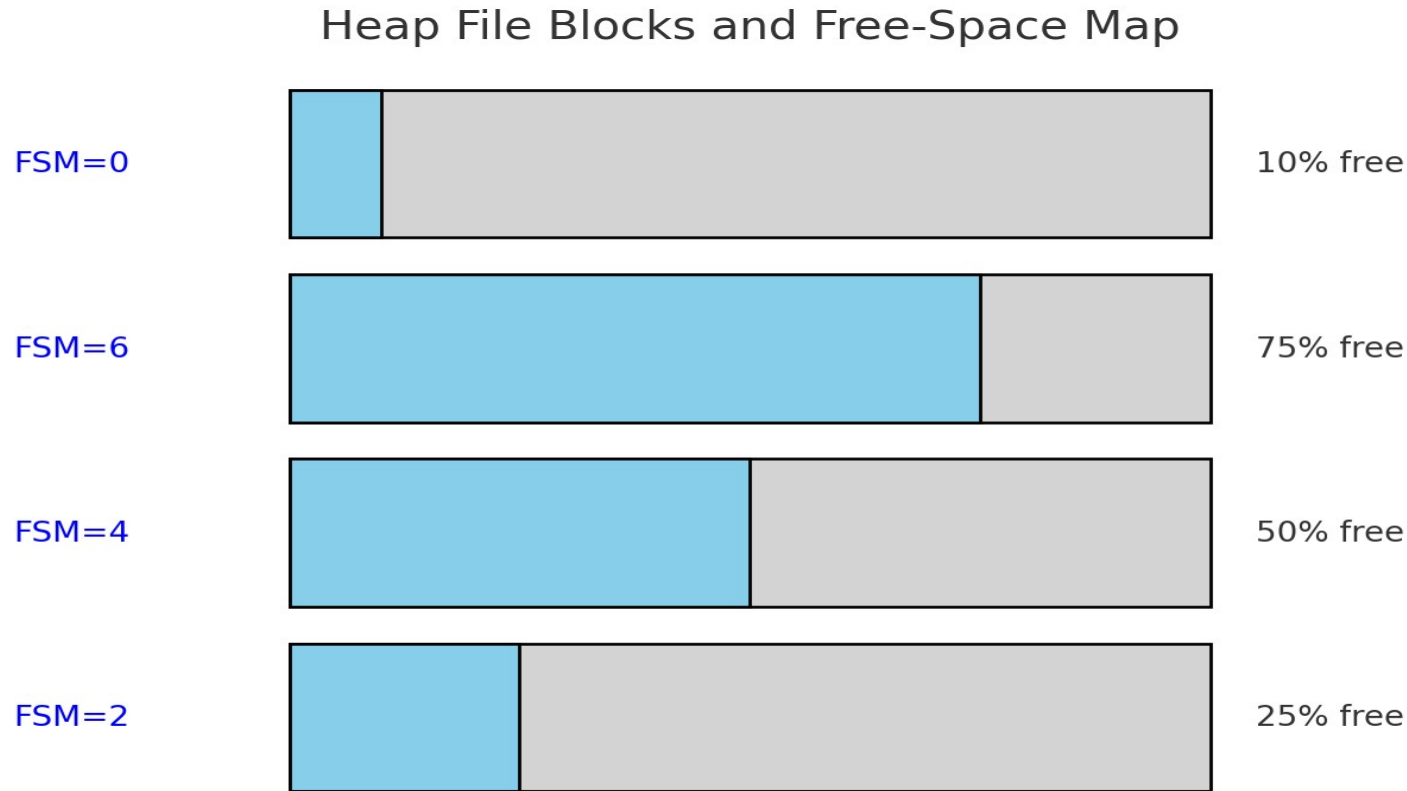
Example:

Using 3 bits per block.

Value $\div 8$ gives the fraction of free space.

If value = 4, it means half ($4/8 = 0.5$) of the block is free.

Heap File Organization- Free space map



- Each horizontal bar = one disk block.
- **Blue portion** = free space in the block.
- **Grey portion** = occupied records.
- **FSM value** (on the left) shows what is stored in the free-space map (scaled using few bits).
- **% free** (on the right) indicates the actual free space available.

Heap File Organization- Free space map

Suppose each disk block has a fixed size, say **8 KB (8192 bytes)**
Free-Space Map (FSM) does **not store exact bytes free**.

Example:

if FSM uses **3 bits per block**, it can represent values from 0 to 7 .

$$\text{Fraction Free} = \text{FSM Value} / 8$$

If FSM = 2

$$\text{Fraction Free} = 2 \div 8 = 0.25$$

So **25% of block is free**.

$$\text{If block size} = 8 \text{ KB} \rightarrow \text{Free space} = 0.25 \times 8192 = \mathbf{2048 \text{ bytes}}.$$

Similarly,

- FSM = 0 \rightarrow 0% free \rightarrow Block is full.
- FSM = 4 \rightarrow 50% free \rightarrow 4 KB free in an 8 KB block.
- FSM = 6 \rightarrow 75% free \rightarrow 6 KB free in an 8 KB block.
- FSM = 7 \rightarrow 100% free \rightarrow Entire block is empty.

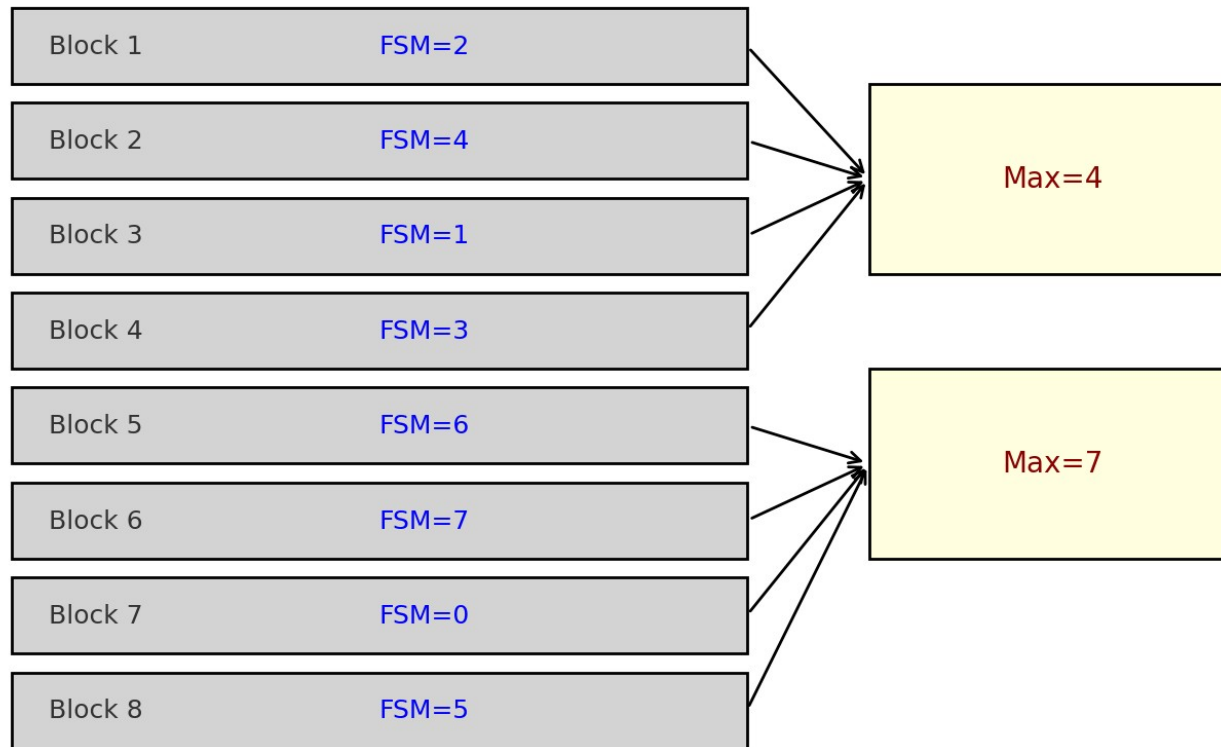
Heap File Organization

Second-Level Free-Space Map

- For **very large files**, scanning the entire first-level map is still expensive.
- A **second-level map** is created, where each entry stores the **maximum free space value** of a group of first-level entries.
- Example:
 - Each second-level entry summarizes 4 first-level entries.
 - This allows the system to **quickly locate a region** where enough free space exists, and then check only a few blocks in detail.

Heap File Organization

Second-Level Free-Space Map



- **Left side:** First-level FSM values (one entry per block, 0–7 scale).
- **Right side:** Second-level FSM entries. Each summarizes **4 blocks** by storing the **maximum free space value** among them.
- Arrows show how groups of 4 blocks are linked to one second-level entry. This hierarchy lets the system quickly skip over groups of blocks and find a region with enough space for insertion, instead of scanning every block's entry.

Heap File Organization

- **Block size** = 8 KB (8192 bytes)
- **FSM uses 3 bits per block** → values 0–7 (bucketed free space)
 - We'll store **$\text{floor}(8 \times \text{free_fraction})$** in FSM, so an FSM value of k means **at least $k/8$ of the block is free** (may be more).

- **Record to insert = 3 KB**

Required fraction = $3/8 = 0.375$ → **required FSM bucket = 3**
(since $\text{floor}(8 \times 0.375) = 3$).

Maps (same numbers as the diagram)

First-level FSM (8 blocks):

[2, 4, 1, 3, 6, 7, 0, 5]

(so Block1=2, Block2=4, ..., Block8=5)

Second-level FSM (groups of 4 blocks):

- Group A (Blocks 1–4) → $\text{max} = 4$
- Group B (Blocks 5–8) → $\text{max} = 7$

Heap File Organization

Insertion using the maps (step-by-step)

1. Use second-level to choose a group needing at least bucket 3.

1. Group A has $\text{max}=4$ (≥ 3) \rightarrow **eligible**

2. Group B has $\text{max}=7$ (≥ 3) \rightarrow also eligible

Pick the first eligible group to minimize seek (say **Group A**, Blocks 1-4).

2. Scan first-level entries within Group A for the first block with value ≥ 3 :

1. Block1 = 2

2. Block2 = 4 (means $\geq 4/8 = 50\%$ free $\rightarrow \geq 4$ KB free)

Choose **Block2**.

3. Validate actual free bytes in Block2 (quick header check).

1. Suppose Block2 actually has **4.2 KB free** (FSM rounded down to bucket 4).

2. 4.2 KB \geq 3 KB, so the record **fits**.

4. Insert record into Block2.

1. New free space in Block2 $\approx 4.2 - 3.0 = 1.2$

Heap File Organization

Update maps

1. First-level: Block2 value goes **4** → **1**.
2. Second-level Group A becomes $\max(2, 1, 1, 3) = 3$ (was 4 before).

(We write the updated entries eventually; it's OK if the persisted map lags a bit.)

What if a value was stale?

Say we had picked **Block8** (FSM=5) but it actually had only **2.5 KB free** (because the FSM entry wasn't updated yet).

- On validation, we detect **2.5 KB < 3 KB**, so we **fail the placement, downgrade Block8's FSM** to the correct bucket (2), and **retry**:
 - Group B's max might still be ≥ 3 (because of other blocks), so we try the next eligible block in that group; otherwise we fall back to Group A.

Heap File Organization

- Required bucket for a record of size S in a block of size B :

$$k_{\text{req}} = \left\lceil 8 \times \frac{S}{B} \right\rceil$$

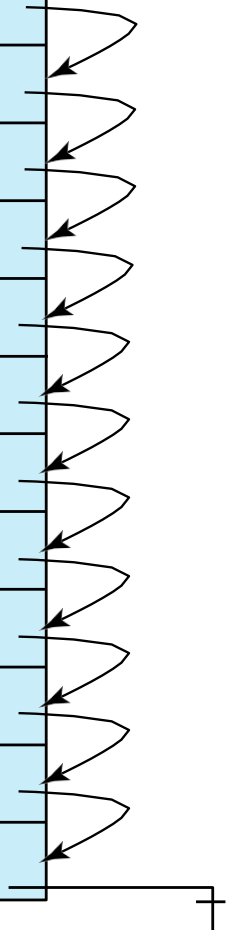
- FSM bucket for a block with F bytes free:

$$k = \left\lceil 8 \times \frac{F}{B} \right\rceil$$

Sequential File Organization

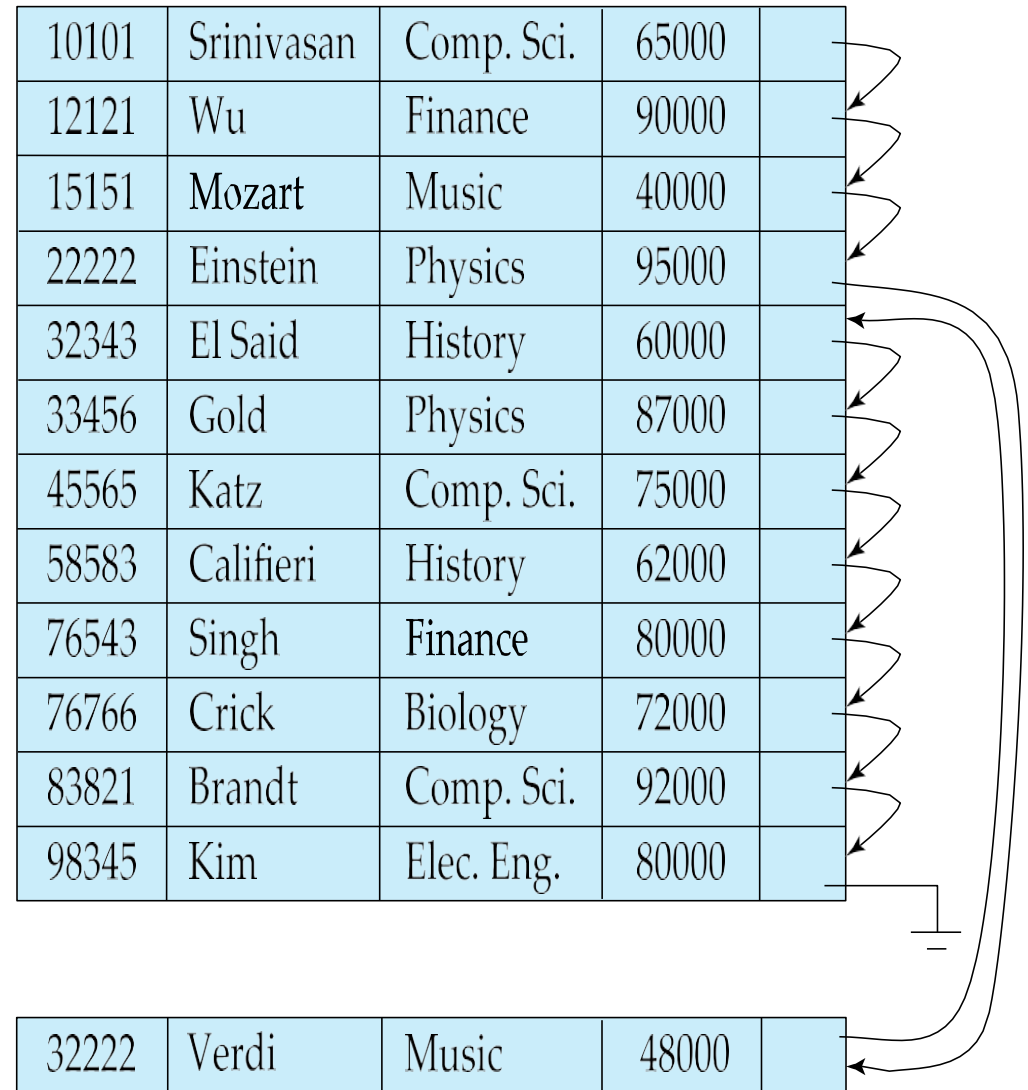
- Suitable for applications that require sequential processing of the entire file
- The records in the file are ordered by a **search-key**

| | | | | |
|-------|------------|------------|-------|--|
| 10101 | Srinivasan | Comp. Sci. | 65000 | |
| 12121 | Wu | Finance | 90000 | |
| 15151 | Mozart | Music | 40000 | |
| 22222 | Einstein | Physics | 95000 | |
| 32343 | El Said | History | 60000 | |
| 33456 | Gold | Physics | 87000 | |
| 45565 | Katz | Comp. Sci. | 75000 | |
| 58583 | Califieri | History | 62000 | |
| 76543 | Singh | Finance | 80000 | |
| 76766 | Crick | Biology | 72000 | |
| 83821 | Brandt | Comp. Sci. | 92000 | |
| 98345 | Kim | Elec. Eng. | 80000 | |



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



Multitable Clustering File Organization

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

department

| <i>dept_name</i> | <i>building</i> | <i>budget</i> |
|------------------|-----------------|---------------|
| Comp. Sci. | Taylor | 100000 |
| Physics | Watson | 70000 |

instructor

| <i>ID</i> | <i>name</i> | <i>dept_name</i> | <i>salary</i> |
|-----------|-------------|------------------|---------------|
| 10101 | Srinivasan | Comp. Sci. | 65000 |
| 33456 | Gold | Physics | 87000 |
| 45565 | Katz | Comp. Sci. | 75000 |
| 83821 | Brandt | Comp. Sci. | 92000 |

multitable clustering
of *department* and
instructor

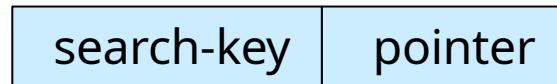
| | | | |
|------------|------------|------------|-------|
| Comp. Sci. | Taylor | 100000 | |
| 10101 | Srinivasan | Comp. Sci. | 65000 |
| 45565 | Katz | Comp. Sci. | 75000 |
| 83821 | Brandt | Comp. Sci. | 92000 |
| Physics | Watson | 70000 | |
| 33456 | Gold | Physics | 87000 |

Multitable Clustering File Organization (cont.)

- good for queries involving *department* \bowtie *instructor*, and for queries involving one single department and its instructors
- bad for queries involving only *department*
- results in variable size records
- Can add pointer chains to link records of a particular relation

Indexing- Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
 - E.g., author catalog in library
- **Search Key** - attribute to set of attributes used to look up records in a file.



- An **index file** consists of records (called **index entries**) of the form
 - Index files are typically much smaller than the original file
 - Two basic kinds of indices:
 - **Ordered indices:** search keys are stored in sorted order
 - Hash indices: search keys are distributed uniformly across “buckets” using a “hash function”.

Index Evaluation Metrics

- Access types supported efficiently.
 - records with a specified value in the attribute
 - or records with an attribute value falling in a specified range of values.
- Access time
- Insertion time
- Deletion time
- Space overhead

Ordered Indices

Indexing techniques evaluated on basis of:

- In an **ordered index**, index entries are stored sorted on the search key value. E.g., author catalog in library.
- **Primary index**: in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
 - Also called **clustering index**- defined on non-key ordering field (multiple records can share value).
 - The search key of a primary index is usually but not necessarily the primary key.
- **Secondary index**: an index whose search key specifies an order **different from the sequential order of the file**. Also called **non-clustering index**.
- **Index-sequential file**: ordered sequential file with a primary index.

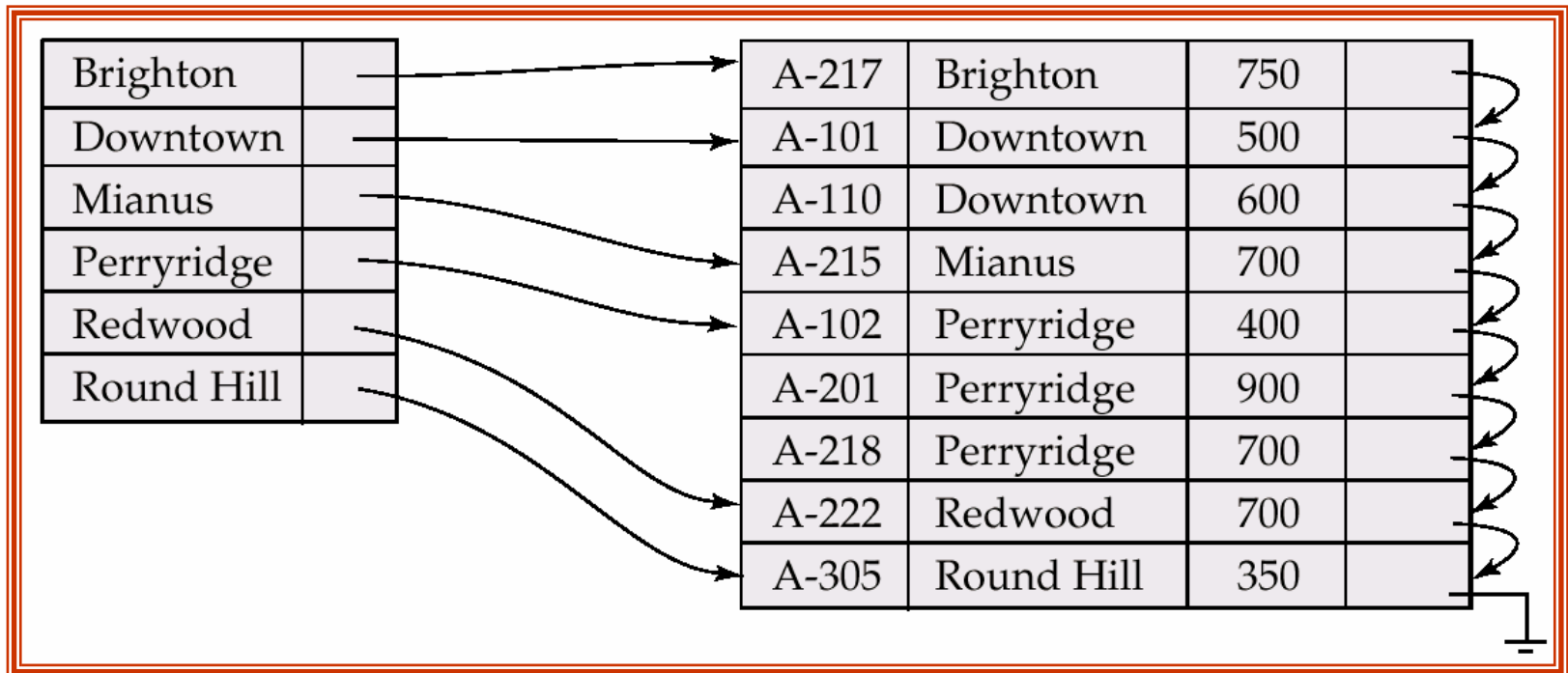
Ordered Indices

Indexing techniques evaluated on basis of:

- In an **ordered index**, index entries are stored sorted on the search key value. E.g., author catalog in library.
- **Primary index**: in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
 - Also called **clustering index- defined on non-key ordering field (multiple records can share value)**.
 - The search key of a primary index is usually but not necessarily the primary key.
- **Secondary index**: an index whose search key specifies an order different from the sequential order of the file. Also called **non-clustering index**.
- **Index-sequential file**: ordered sequential file with a primary index.

Dense Index Files

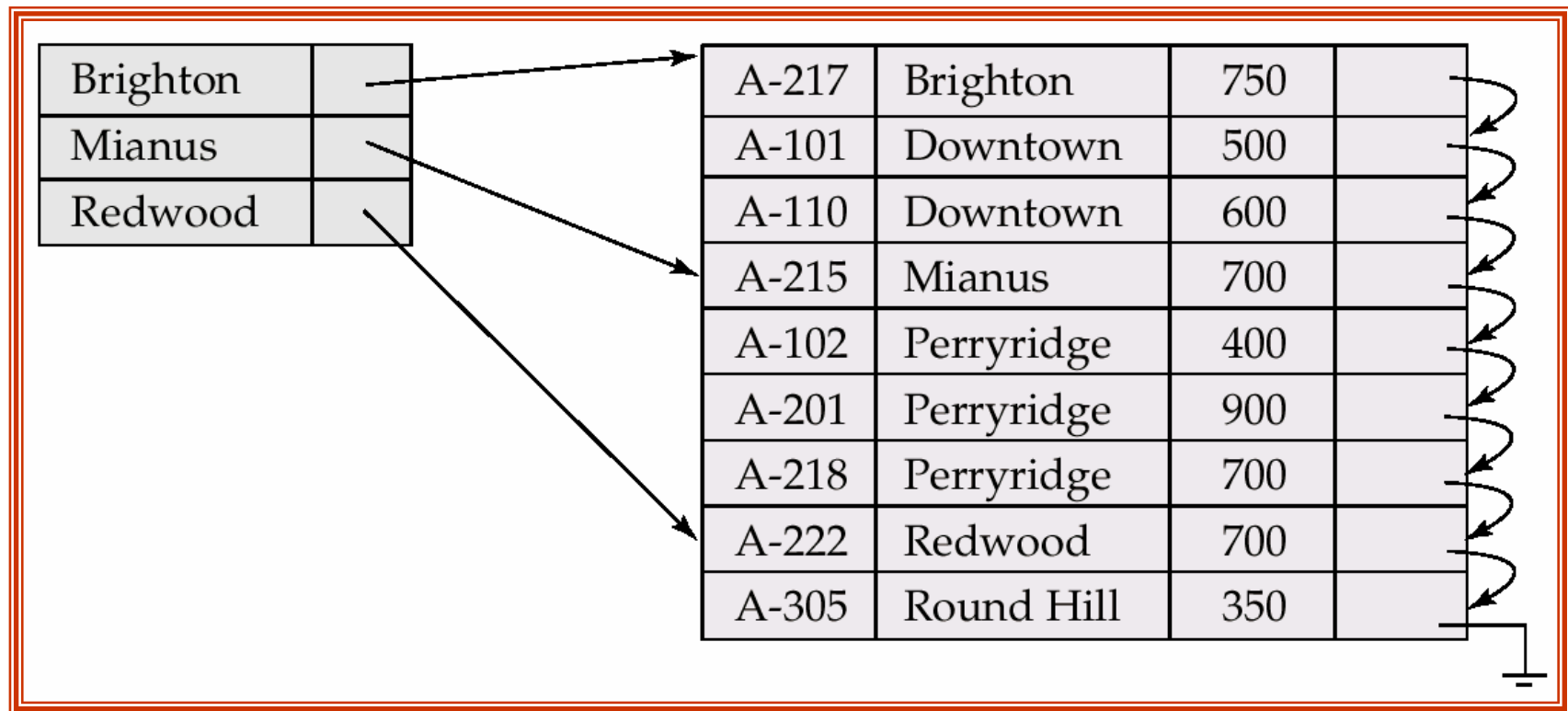
- **Dense index** — Index record appears for every search-key value in the file.



Sparse Index Files

- **Sparse Index:** contains index records for only some search-key values.
 - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value K we:
 - Find index record with largest search-key value $< K$
 - Search file sequentially starting at the record to which the index record points
- Less space and less maintenance overhead for insertions and deletions.
- Generally slower than dense index for locating records.
- Good tradeoff: sparse index with an index entry for every block in file, corresponding to least search-key value in the block.

Example of Sparse Index Files



Dense vs Sparse Index

- **Dense Index:** One entry per record.
- **Sparse Index:** One entry per block (Primary Index).
 - Primary Index = **Sparse**.
 - Advantage:
 - Index file much smaller than data file.
 - Faster binary search.

Searching with Primary Index

- Binary search performed on index file (with b_i blocks).
- Total accesses = $\log_2(b_i) + 1$
 - $\log_2(b_i) \rightarrow$ binary search on index file.
 - $+1 \rightarrow$ access data block.
- Example:
 - Data file = 1,000 blocks.
 - Index file = 100 blocks.
 - Binary search index = $\log_2(100) \approx 7$.
 - Access data block = 1.
 - Total \approx **8 accesses vs 10 on data file.**

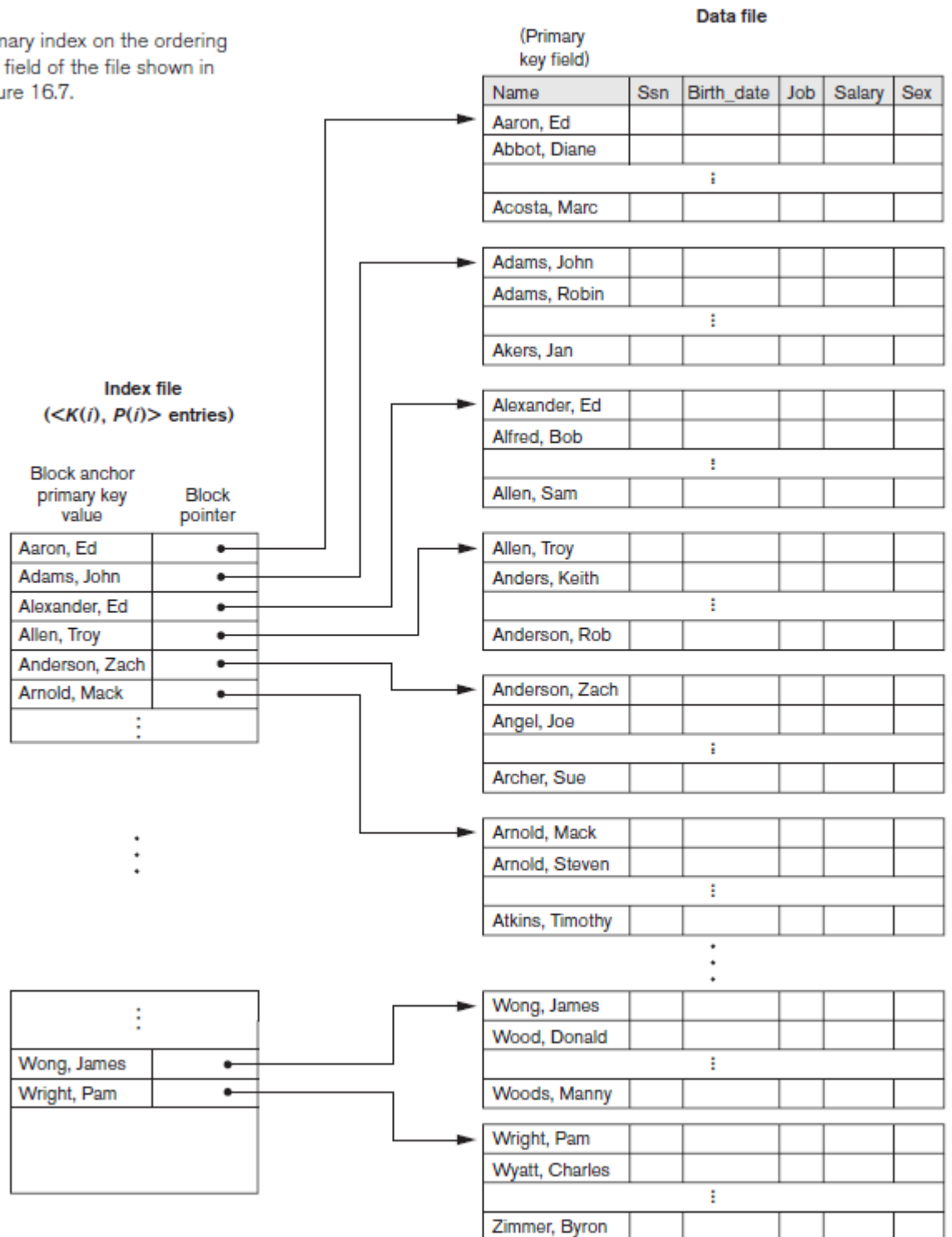
Clustering Index

- Defined on a **non-key ordering field**.
- File = **clustered file**.
- Example: Records ordered by *Department*, but many employees may share same department.
- Index entry contains:
- Value of clustering field.
- Pointer to block containing first occurrence.

Primary Index

- Built on **ordering key field** of an ordered file.
- One entry per block.
- Each entry $\langle K(i), P(i) \rangle$ contains:
 - $K(i)$** : first key (anchor record) in block i .
 - $P(i)$** : pointer to block i .
- Example:
 - $\langle (\text{Aaron, Ed}), P(1) \rangle \rightarrow$ Block 1
 - $\langle (\text{Adams, John}), P(2) \rangle \rightarrow$ Block 2

Primary index on the ordering key field of the file shown in Figure 16.7.



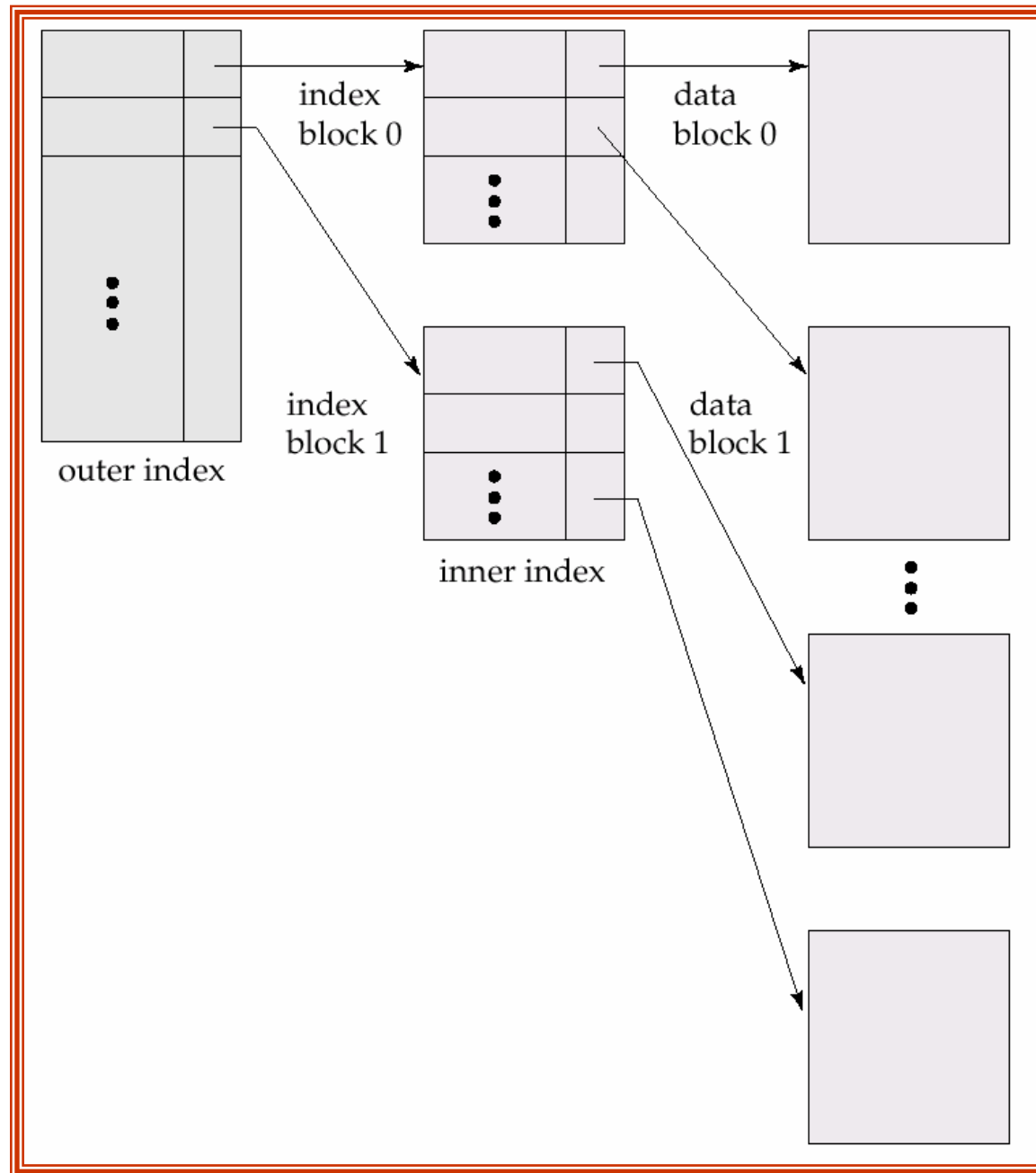
Secondary Index

- Built on a **nonordering field** (not sorted in file).
- Separate access path.
- Can be **dense** (usually 1 entry per record).
- More expensive in terms of storage.

Multilevel Index

- If primary index does not fit in memory, access becomes expensive.
- To reduce number of disk accesses to index records, treat primary index kept on disk as a sequential file and construct a sparse index on it.
 - outer index – a sparse index of primary index
 - inner index – the primary index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion from the file.

Multilevel Index (Cont.)



Index Update: Deletion

- If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also.
- Single-level index deletion:
 - Dense indices – deletion of search-key is similar to file record deletion.
 - Sparse indices – if an entry for the search key exists in the index, it is deleted by replacing the entry in the index with the next search- key value in the file (in search-key order). If the next search-key value already has an index entry, the entry is deleted instead of being replaced.

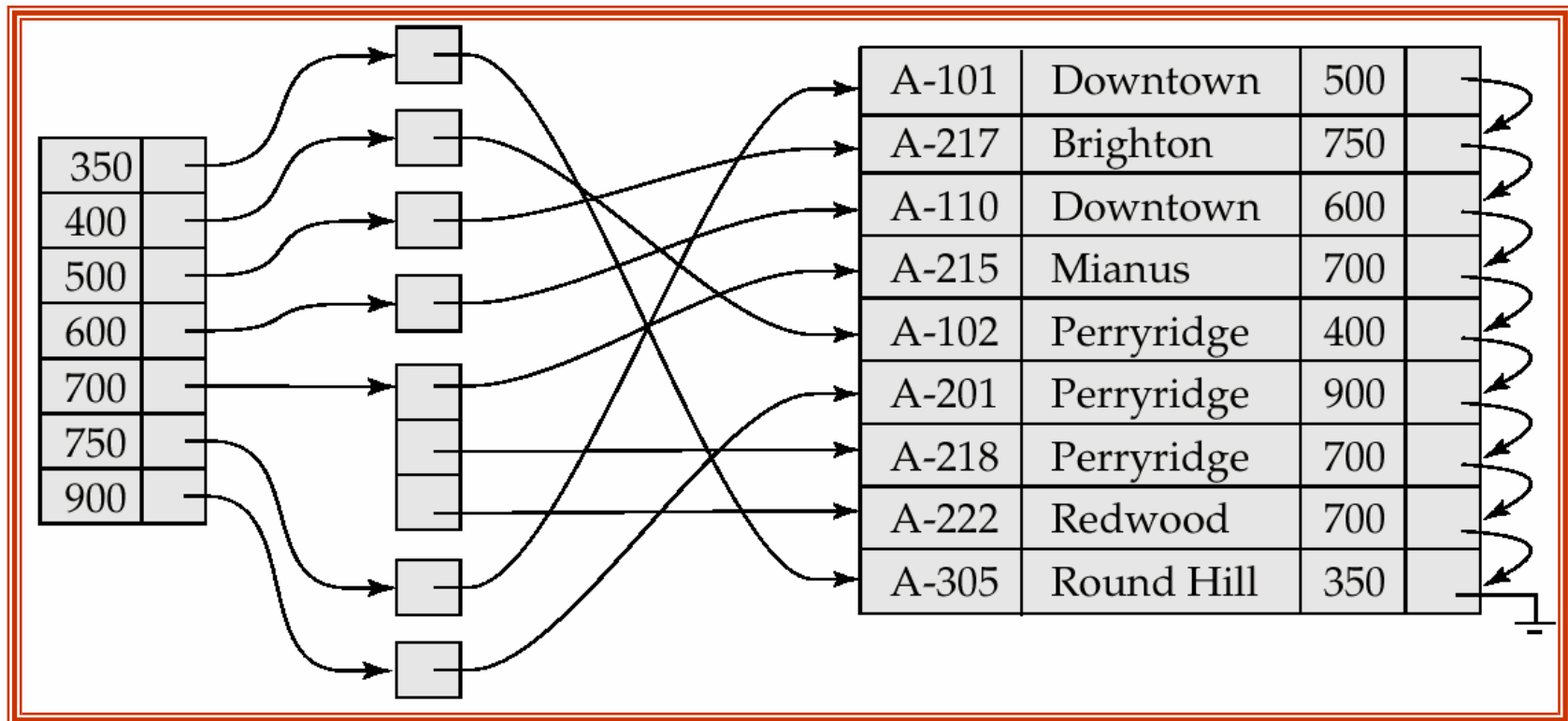
Index Update: Insertion

- Single-level index insertion:
 - Perform a lookup using the search-key value appearing in the record to be inserted.
 - Dense indices – if the search-key value does not appear in the index, insert it.
 - Sparse indices – if index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created. In this case, the first search-key value appearing in the new block is inserted into the index.
- Multilevel insertion (as well as deletion) algorithms are simple extensions of the single-level algorithms

Secondary Indices

- Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index satisfy some condition.
 - Example 1: In the *account* database stored sequentially by account number, we may want to find all accounts in a particular branch
 - Example 2: as above, but where we want to find all accounts with a specified balance or range of balances
- We can have a secondary index with an index record for each search-key value; index record points to a bucket that contains pointers to all the actual records with that particular search-key value.

Secondary Index on *balance* field of *account*



Primary and Secondary Indices

- Secondary indices have to be dense.
- Indices offer substantial benefits when searching for records.
- When a file is modified, every index on the file must be updated, Updating indices imposes overhead on database modification.
- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive
 - each record access may fetch a new block from disk

Binary Search for Disk Files

- Records in a file are stored across **b blocks** (numbered 1, 2, ..., b).
- Records are sorted by their **ordering key field**.
- Goal: efficiently search for a record with key **K**.
- Disk block addresses are stored in the **file header**

Uses **divide and conquer** principle:

- Check middle block.
- If K is smaller, search left half.
- If K is larger, search right half.
- Repeat until found or search space is empty.

Linear Search (Traditional Method)

- Sequentially checks blocks one by one.
- **Average Case:** $\sim b/2$ block accesses if record is found.
- **Worst Case:** b block accesses if record is not found.
- Inefficient for large files.

Binary Search Approach

- Works on **blocks**, not individual records.
- Uses **divide and conquer** principle:
- Check middle block.
- If K is smaller, search left half.
- If K is larger, search right half.
- Repeat until found or search space is empty

Algorithm (Binary Search on Blocks)

1. Set $low = 1$, $high = b$.
2. While $low \leq high$:
 1. $mid = (low + high)/2$.
 2. Read $block[mid]$.
 3. If K is in $block[mid] \rightarrow$ **record found**.
 4. Else if $K < \text{first key of } block[mid] \rightarrow$ set $high = mid - 1$.
 5. Else \rightarrow set $low = mid + 1$.
3. If no match \rightarrow **record not found**.

Binary Search Approach

Suppose a file has **b = 8 blocks**, each containing sorted records by **Roll No.**

| Block No | First Record Key | Last Record Key |
|----------|------------------|-----------------|
| 1 | 1 | 100 |
| 2 | 101 | 200 |
| 3 | 201 | 300 |
| 4 | 301 | 400 |
| 5 | 401 | 500 |
| 6 | 501 | 600 |
| 7 | 601 | 700 |
| 8 | 701 | 800 |

We want to **search for Roll No = 630**.

Step 1: Initialization

- Low = 1, High = 8

Step 2: Midpoint Block

- Mid = $(1+8)/2 = 4.5 \approx 4$
- Block 4 range = 301–400
- Since **630 > 400**, search right half
→ Low = 5, High = 8

Step 3: Next Midpoint

- Mid = $(5+8)/2 = 6.5 \approx 6$
- Block 6 range = 501–600
- Since **630 > 600**, search right half
→ Low = 7, High = 8

Step 4: Next Midpoint

- Mid = $(7+8)/2 = 7.5 \approx 7$
- Block 7 range = 601–700
- Since **630 lies in 601–700**, target block = 7

Step 5: Search Inside Block

- Binary search now continues **inside block 7**.
- Records in Block 7 are scanned (either sequentially or with binary search, if block itself is indexed).
- Record 630 is **found in Block 7**.

Analysis of Accesses

- Blocks checked: **Block 4** → **Block 6** → **Block 7**
- Total = **$\log_2(8) = 3$ block accesses**
- Much better than Linear Search:
 - Average = $8/2 = 4$ accesses
 - Worst case = 8 accesses

Performance Comparison

- **Binary Search:**
 - Accesses $\approx \log_2(\mathbf{b})$ blocks.
 - Independent of whether record is found or not.
- **Linear Search:**
 - Average: $\mathbf{b}/2$ blocks.
 - Worst: \mathbf{b} blocks.
- Binary search is much faster for large files.

Example

- Suppose file has $\mathbf{b} = 1024$ blocks.
- **Linear search:** ~ 512 accesses (avg).
- **Binary search:** $\log_2(1024) = 10$ accesses only.

Table 16.3 Average Access Times for a File of b Blocks under Basic File Organizations

| Type of Organization | Access/Search Method | Average Blocks to Access a Specific Record |
|----------------------|---------------------------------|---|
| Heap (unordered) | Sequential scan (linear search) | $b/2$ |
| Ordered | Sequential scan | $b/2$ |
| Ordered | Binary search | $\log_2 b$ |
