3 : Database Indexing and Tuning

IT3306 – Data Management

Level II - Semester 3

Overview

- This lesson on Database Indexing and Tuning discusses the gradual development computer memory hierarchy and hence the evolution of the Indexing Methods.
- Here we look into types of computer memory and types of indexes in detail.

Intended Learning Outcomes

- At the end of this lesson, you will be able to;
  - Describe how the different computer memories evolved.
  - Identify the different types of computer memory and the storage organizations of databases
  - Recognize the importance of implementing indexes on databases
  - Explain the key concepts of the different types of database indexes.

List of subtopics

List of subtopics

3.2 Introduction to indexing

Introduce index files, indexing fields, index entry (record pointers and block pointers)

3.3 Types of Indexes

3.3.1 Single Level Indexes: Primary, Clustering and Secondary indexes

3.3.2 Multilevel indexes: Overview of multilevel indexes

3.4 Indexes on Multiple Keys

3.5 Other types of Indexes

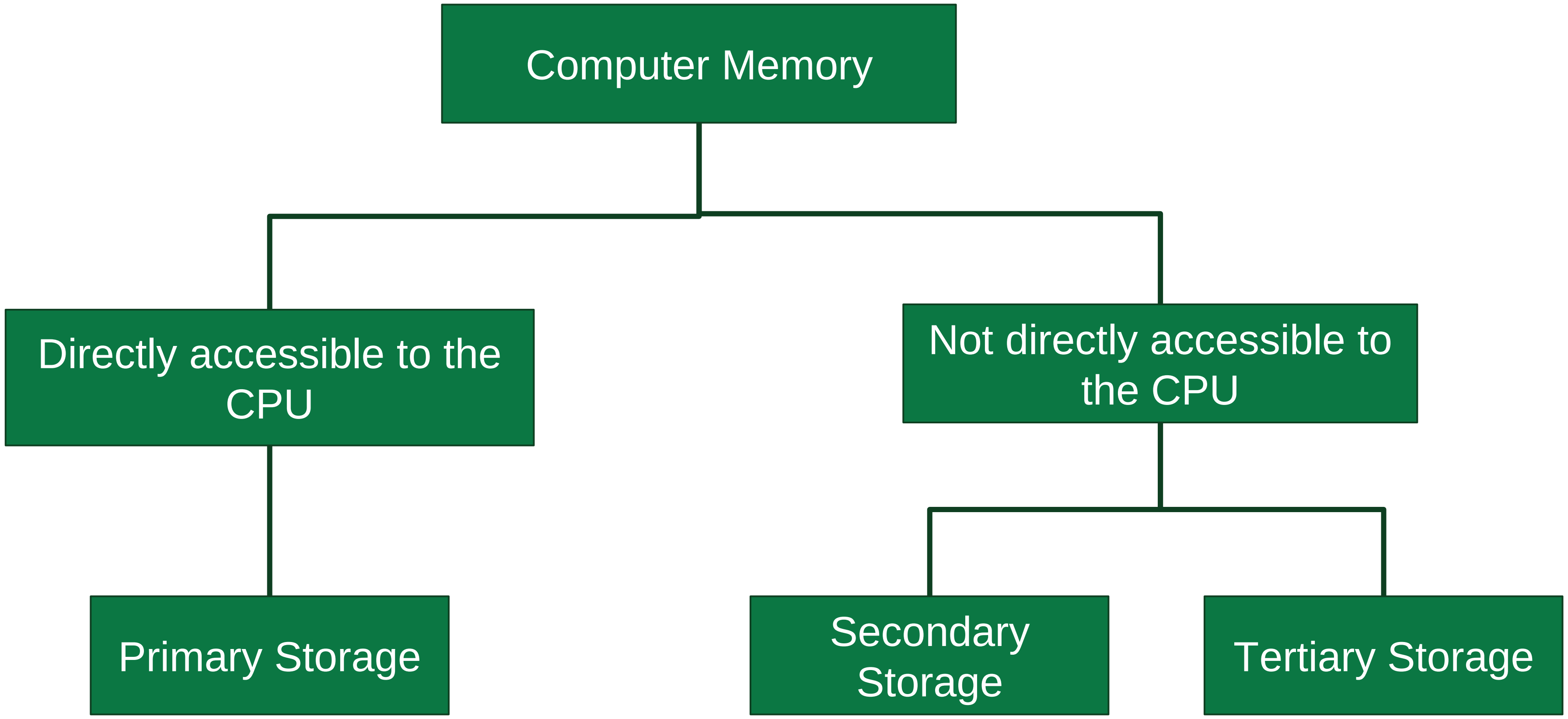
Hash indexes, bitmap indexes, function based indexes

3.6 Index Creation and Tuning

3.7 Physical Database Design in Relational Databases

3.1 Disk Storage and Basic File Structures

3.1.1. Computer Memory Hierarchy


Computer Memory
Directly accessible to the CPU
Not directly accessible to the CPU
Primary Storage
Secondary Storage
Tertiary Storage

3.1 Disk Storage and Basic File Structures

3.1.1. Computer Memory Hierarchy

- The data collected via a computational database should be stored in a physical storage medium.
- Once stored in a storage medium, the database management software can execute functions on that to retrieve, update and process the data.
- In the current computer systems, data is stored and moved across a hierarchy of storage media.
- As for the memory organization, the memory with the highest speed is the most expensive option and it also has the lowest capacity.
- When it comes to lowest speed memory, they are the options with the highest available storage capacity.

3.1 Disk Storage and Basic File Structures

3.1.1. Computer Memory Hierarchy

Now let's explain the hierarchy given in the previous slide (slide number 7).

1. Primary Storage

This operates directly in the computer's Central Processing Unit.

Eg: Main Memory, Cache Memory.

- Provides fast access to data.
- Limited storage capacity.
- Contents of primary storage will be deleted when the computer shuts down or in case of a power failure.
- Comparatively more expensive.

3.1 Disk Storage and Basic File Structures

3.1.1. Computer Memory Hierarchy

Primary Storage - Static RAM

- Static Random Access Memory (RAM) is the memory where as long as power is provided.
- Cache memory in CPU is identified as the Static RAM
- Data is kept as bits in its memory.
- The most expensive type of memory.
- Using techniques like prefetching and pipelining, the Cache memory speeds up the execution of program instructions for the CPU.

3.1 Disk Storage and Basic File Structures

3.1.1. Computer Memory Hierarchy

Primary Storage - Dynamic RAM

- Dynamic Random Access Memory (DRAM) is the CPU's space for storing application instructions and data.
- Main memory of the computer is identified as the DRAM.
- The advantage of the DRAM is its low cost.
- When it is compared with the Static RAM, the speed is lesser.

3.1 Disk Storage and Basic File Structures

3.1.1. Computer Memory Hierarchy

2. Secondary Storage

Operates external to the computer’s main memory.

Eg: Magnetic Disks, Flash Drives, CD-ROM

- The CPU cannot process data in secondary storage directly. It must first be copied into primary storage before the CPU can handle it.
- Mostly used for online storage of enterprise databases.
- With regards to enterprise databases, the magnetic disks have been used as the main storage medium.
- Recently there is a trend to use flash memory for the purpose of storing moderate amounts of permanent data.
- Solid State Drive (SSD) is a form of memory that can be used instead of a disk drive.

3.1 Disk Storage and Basic File Structures

3.1.1. Computer Memory Hierarchy

2. Secondary Storage

- Least expensive type of storage media.
- The storage capacity is measured in:
  - kilobytes(kB)
  - Megabytes(MB)
  - Gigabytes(GB)
  - Terabytes(TB)
  - Petabytes (PB)

3.1 Disk Storage and Basic File Structures

3.1.1. Computer Memory Hierarchy

3. Tertiary Storage

Operates external to the computer’s main memory.

Eg: CD - ROMs, DVDs

- The CPU cannot process data in tertiary storage directly. It must first be copied into primary storage before the CPU can handle it.
- Removable media that can be used as offline storage falls in this category.
- Large capacity to store data.
- Comparatively less cost.
- Slower access to data than primary storage media.

3.1 Disk Storage and Basic File Structures

3.1.1. Computer Memory Hierarchy

4. Flash Memory

- Popular type of memory with its non-volatility.
- Use the technique of EEPROM (Electronically Erasable and Programmable Read Only Memory)
- High performance memory.
- Fast access.
- One disadvantage is that the entire block must be erased and written simultaneously.
- Two Types:
  - NAND Flash Memory
  - NOR Flash Memory
- Common examples:
  - Devices in Cameras, MP3/MP4 Players, Cellphones, USB Flash Drives

3.1 Disk Storage and Basic File Structures

3.1.1. Computer Memory Hierarchy

5. Optical Drives

- Most popular type of Optical Drives are CDs and DVDs.
- Capacity of a CD is 700-MB and DVDs have capacities ranging from 4.5 to 15 GB.
- CD - ROM reads the data by laser technology. They cannot be overwritten.
- CD-R(compact disk recordable) and DVD-R: Allows to store data which can be read as many times as required.
- Currently this type of storage is comparatively declining due to the popularity of the magnetic disks.

3.1 Disk Storage and Basic File Structures

3.1.1. Computer Memory Hierarchy

6. Magnetic Tapes

- Used for archiving and as a backup storage of data.
- Note that Magnetic Disks (400 GB–8TB) and Magnetic Tapes (2.5TB–8.5TB) are two different storage types.

Activity

Categorize the following devices as Primary, Secondary or Tertiary Storage Media.

1. Random Access Memory
2. Hard Disk Drive
3. Flash Drive
4. Tape Libraries
5. Optical Jukebox
6. Magnetic Tape
7. Main Memory

3.1 Disk Storage and Basic File Structures

3.1.2. Storage Organization of Databases

- Usually databases have **Persistent** data. This means large volumes of data stored over long periods of time.
- These persistent data are continuously retrieved and processed in the storage period.
- The place where the databases are stored permanently in the computer memory is the secondary storage.
- Magnetic disks are widely used here since:

  - If the database is too large, it will not fit in the main memory.

  - Secondary storage is non-volatile, but the main memory is volatile.

  - The cost of storage per unit of data is lesser in secondary storage.

3.1 Disk Storage and Basic File Structures

3.1.2. Storage Organization of Databases

- Solid State Drive (SSD) is one of the latest technologies identified as an alternative for magnetic storage disks.
- However, it is expected that the primary option for the storage of large databases will continue to be the magnetic disks.
- Magnetic tapes are also used for database backup purposes due to their comparatively lower cost.
- But the data in them need to be loaded and read before processing. Opposing to this, magnetic disks can be accessed directly at anytime.

3.1 Disk Storage and Basic File Structures

3.1.2. Storage Organization of Databases

- **Physical Database Design** is a process that entails selecting the techniques that best suit the application requirements from a variety of data organizing approaches.
- When designing, implementing, and operating a database on a certain DBMS, database designers and DBAs must be aware of the benefits and drawbacks of each storage medium.

3.1 Disk Storage and Basic File Structures

3.1.2. Storage Organization of Databases

- The data on disk is grouped into **Records** or **Files.**
- These records include data about entities, attributes and relationships.
- Whenever a certain portion of the data retrieved from the DB for processing, it needs to be found on disk, copied to main memory for processing, and then rewritten to the disk if the data gets updated.
- Therefore, the data should be kept on disk in a way that allows them to be quickly accessed when they are needed.

3.1 Disk Storage and Basic File Structures

3.1.2. Storage Organization of Databases

- **Primary File Organization** defines how the data is stored physically in the disk and how they can be accessed.

File Organization	Description
Heap File	No particular order in storing data. Appends new records to the end.
Sorted File	Maintains an order for the records by sorting data on a particular field.
Hashed File	Uses the hash function of a field to identify the record’s place in the database.
B Trees	Use Tree structures for record storing.

Activity

State whether the following statement are true or false.

1. The place of permanently storing databases is the primary storage.
2. A Heap File has a specific ordering criterion where the new records are added at the end.
3. Upon retrieval of data from a file, it needs to be found on disk and copied to main memory for processing.
4. The database administrators need to be aware of the physical structuring of the database to identify whether they can be sold to a client.
5. Solid State Drives are identified alternatives for magnetic disks.

3.1 Disk Storage and Basic File Structures

3.1.3. Secondary Storage Media

- The device that holds the magnetic disks is the Hard Disk Drive (HDD).
- Basic unit of data on a HDD is the **Bit**. Bits together make **Bytes**. One character is stored using a single byte.
- **Capacity** of a disk is the number of bytes the disk can store.
- Disks are composed of magnetic material in the shape of a thin round disk, with a plastic or acrylic cover to protect it.
- **Single Sided Disk** stores information on one of its surfaces.
- **Double Sided Disk** stores information on both sides of its surfaces.
- A few disks assembled together makes a **Disk Pack** which has higher storage capacity.

3.1 Disk Storage and Basic File Structures

3.1.3. Secondary Storage Media

- On a disk surface, information is stored in concentric circles of small width, each with its own diameter.
- Each of these circles is called a **Track**.
- A **Cylinder** is a group of tracks on different surfaces of a disk pack that have the same diameter.
- Retrieval of data stored on the same Cylinder is faster compared to data stored in different Cylinders.
- A track is broken into smaller **Blocks** or **Sectors** since it typically includes a vast amount of data.

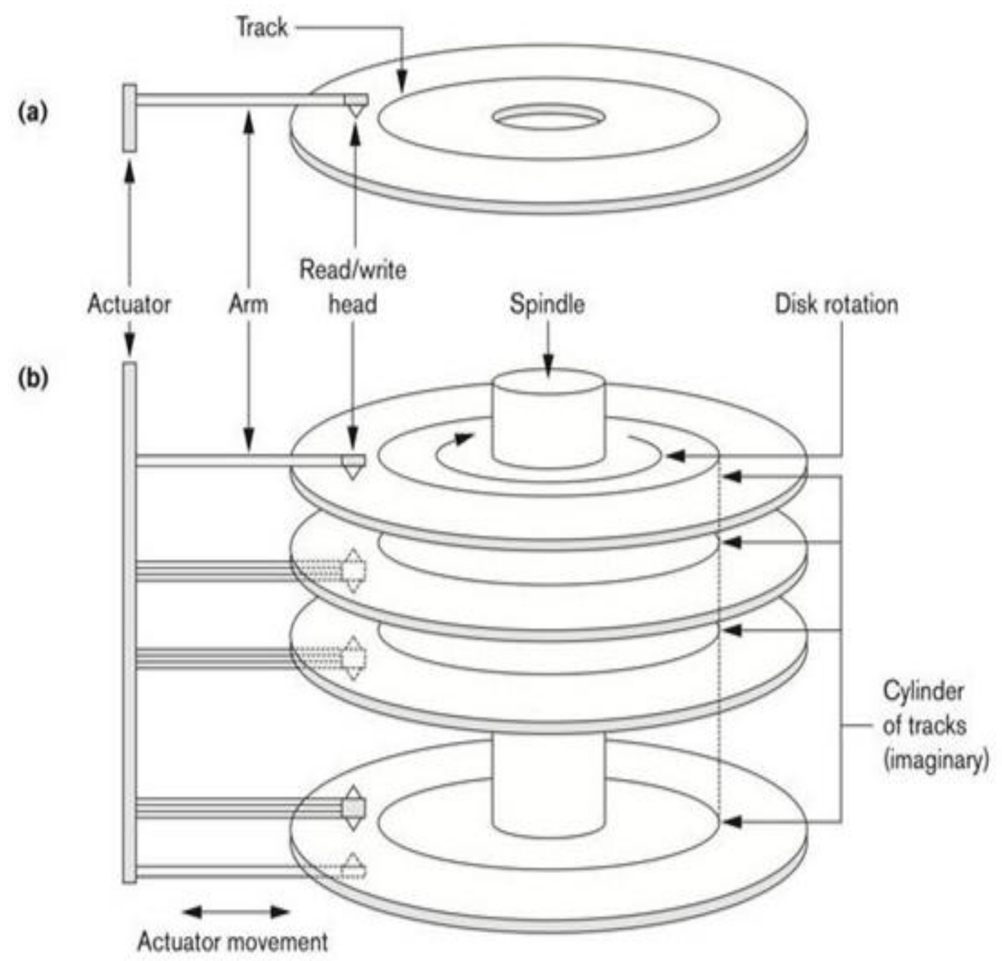
3.1 Disk Storage and Basic File Structures

3.1.3. Secondary Storage Media

Hardware components on disk:

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

b) A disk pack with read/write.



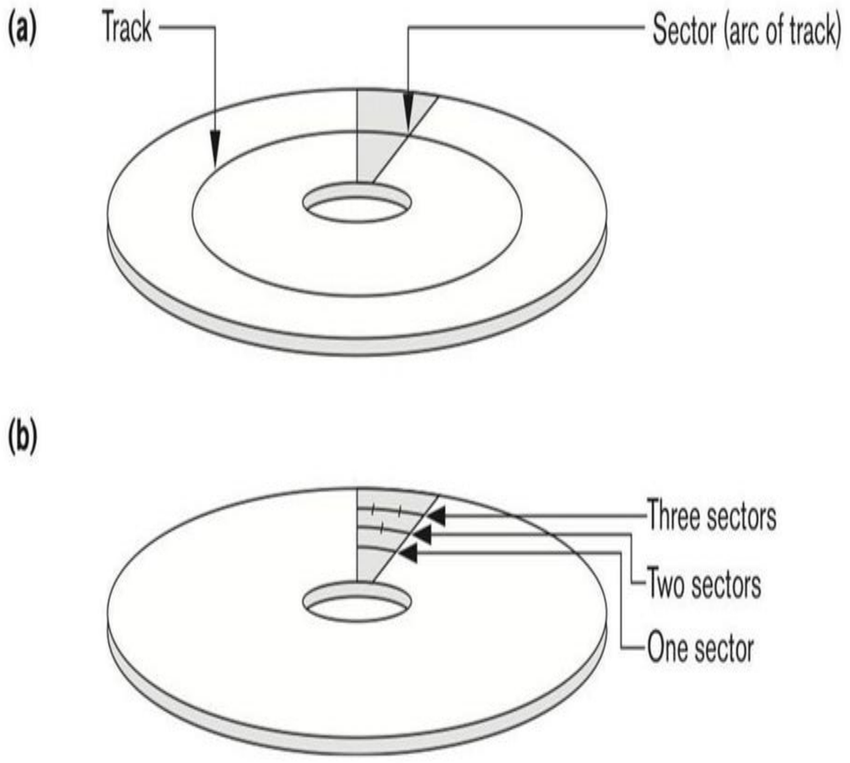
3.1 Disk Storage and Basic File Structures

3.1.3. Secondary Storage Media

Different sector organizations on disk:

(a) Sectors subtending a fixed angle

(b) Sectors maintaining a uniform recording density



3.1 Disk Storage and Basic File Structures

3.1.3. Secondary Storage Media

- During disk formatting, the operating system divides a track into equal-sized **Disk Blocks** (or pages). The size of each block is fixed and cannot be adjusted dynamically.
- **Interblock Gaps**, which are fixed in size and contain specifically coded control information recorded during disk formatting.
- **Hardware Address of a Block** is the combination of a cylinder number, track number (surface number inside the cylinder on which the track is placed), and block number (within the track).
- **Buffer** is one disk block stored in a reserved region in primary storage.
- **Read Command** - Disk block is copied into the buffer
- **Write Command** - Contents of the buffer are copied into the disk block.

3.1 Disk Storage and Basic File Structures

3.1.3. Secondary Storage Media

- A collection of several shared blocks is called a **Cluster**
- The hardware mechanism that reads or writes a block of data is the **Read / Write Head**
- An electronic component is coupled to a mechanical arm in a read/write head.
- **Fixed Head Disks** - The read/write heads on disk units are fixed, with as many heads as there are tracks.
- **Movable Head Disks** - Disk units with an actuator connected to a second electrical motor that moves the read/write heads together and accurately positions them over the cylinder of tracks defined in a block address.

Activity

Match the description with the relevant technical term out of the following.

[Capacity, Track, Buffer, Hardware Address of a Block, Cluster]

1. The concentric circles on a disk where information is stored.
2. Combination of a cylinder number, track number, and block number
3. Number of Bytes that a disk can store
4. Collection of shared blocks
5. A disk block stored in a reserved location in primary storage.

3.1 Disk Storage and Basic File Structures

3.1.4. Solid State Device Storage

- Solid State Device (SSD) Storage is sometimes known as Flash Storage.
- They have the ability to store data on secondary storage without requiring constant power.
- A controller and a group of interconnected flash memory cards are the essential components of an SSD.
- SSDs can be plugged into slots already available for mounting Hard Disk Drives (HDDs) on laptops and servers by using form factors compatible with HDDs.
- SSDs are identified to be more durable, run silently, faster in terms of access time, and delivers better transfer rates than HDD because there are no moving parts.

3.1 Disk Storage and Basic File Structures

3.1.4. Solid State Device Storage

- As opposed to HDD, where Blocks and Cylinders should be pre-assigned for storing data, any address on an SSD can be directly addressed, since there are no restrictions on where data can be stored.
- With this direct access, data is less likely to be fragmented, and the need for restructuring is not available.
- Dynamic Random Access Memory (DRAM)-based SSDs are also available in addition to flash memory.
- DRAM based SSDs are more expensive than flash memory, but they provide faster access. However, they need an internal power supplier to perform.

Activity

State four key features of a Solid State Drive (SSD).

1.______________________

2.______________________

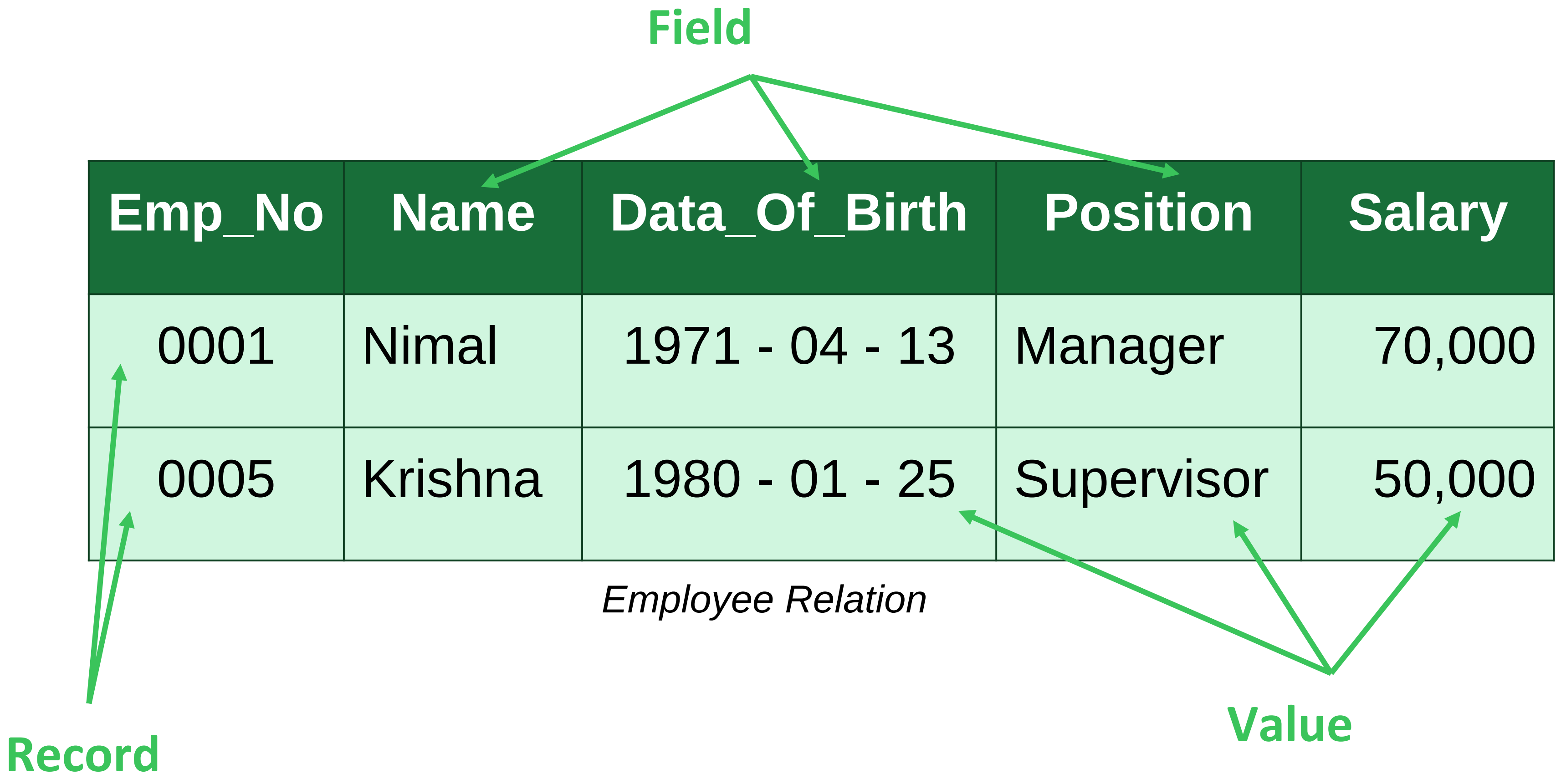
3.______________________

4.______________________

3.1 Disk Storage and Basic File Structures

3.1.5. Placing File Records on Disk

Explanation can be found next slide.



Emp_No	Name	Data_Of_Birth	Position	Salary
0001	Nimal	1971 - 04 - 13	Manager	70,000
0005	Krishna	1980 - 01 - 25	Supervisor	50,000

Employee Relation

3.1 Disk Storage and Basic File Structures

3.1.5. Placing File Records on Disk

- As shows in the previous slide, columns of the table are called *fields*; rows are called *records*; each cell data item is called *value*.
- The **Data Type** is one of the standard data types that are used in programming.

  - Numeric (Integer, Long Integer, Floating Point)
  - Characters / Strings (Fixed length, varying length)
  - Boolean (True or False and 0 or 1)
  - Date, Time

- For a particular computer system, the number of bytes necessary for each data type is fixed.

3.1 Disk Storage and Basic File Structures

3.1.5. Placing File Records on Disk

```
Create table Employee
(
	Emp_No Int,
	Name Char (50),
	Date_Of_Birth Date,
	Position Char (50),
	Salary Int
);
```

An example for the Creation of Employee relation using MySQL with *data types*.

Activity

Select the Data Type that best matches the description out of the following.

[Integer, Floating Point, Date and Time, Boolean, Character]

1. NIC Number of Sri Lankans
2. The access time of users for the Ministry of Health website within a week
3. The number of students in a class
4. Cash balance of a bank account
5. Response to the question by a set of students whether they have received the vaccination for Rubella.

3.1 Disk Storage and Basic File Structures

3.1.5. Placing File Records on Disk

- **File** is a sequence of records. Usually all records in a file belong to the same record type.
- If the size of each record in the file is the same (in bytes) the file is known to be made up of **Fixed Length Records**.
- **Variable Length Records** means that different records of the file are of different sizes.
- Reasons to have variable length records in a file:
  - One or more fields are of different sizes.
  - For individual records, one or more of the fields may have multiple values (**Repeating Group/ Field**)
  - One or more fields are optional (**Optional Fields**)
  - File includes different record types (**Mixed File**)

3.1 Disk Storage and Basic File Structures

3.1.5. Placing File Records on Disk

- **In a Fixed Length Record**;
  - The system can identify the starting byte location of each field relative to the starting position of the record since each record has equal fields and field lengths. This makes it easier for programs that access such files to locate field values.
  - However, variable length records can also be stored as fixed length records.
  - By assigning “Null” for optional fields where data values are not available.
  - By assigning the maximum possible number of records for each repeating group.
  - In each if these cases, the space is wasted.

3.1 Disk Storage and Basic File Structures

3.1.5. Placing File Records on Disk

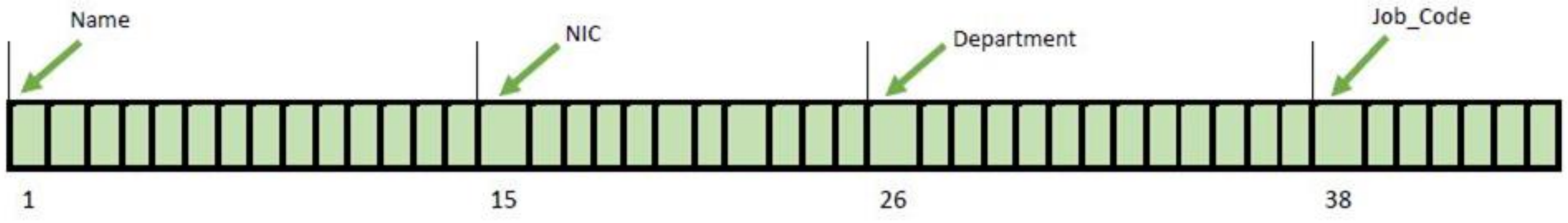
- **In a Variable Length Record**;
  - Each field in each record contains a value, but the precise length of some field values is not correctly known.
  - To determine and terminate variable lengths special characters can be used.
  - They represent the number of bytes for a particular record in each field.
  - Separators that can be used are: **?**, **$**, **%**

3.1 Disk Storage and Basic File Structures

3.1.5. Placing File Records on Disk

Record Storage Format 1

Eg: A fixed-length record with four fields and size of 44 bytes.



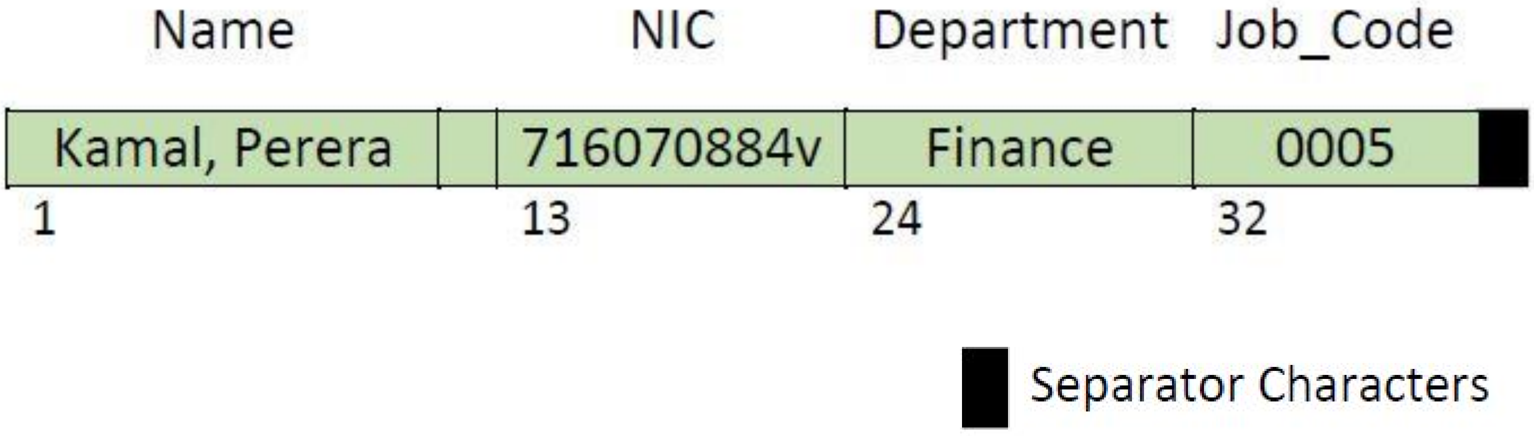
3.1 Disk Storage and Basic File Structures

3.1.5. Placing File Records on Disk

<u>Record Storage Format 2</u>

Eg: A record with two variable-length fields (Name and Department) and two fixed-length fields (NIC and Job_Code).

Separator Character is used to mark the record separation.

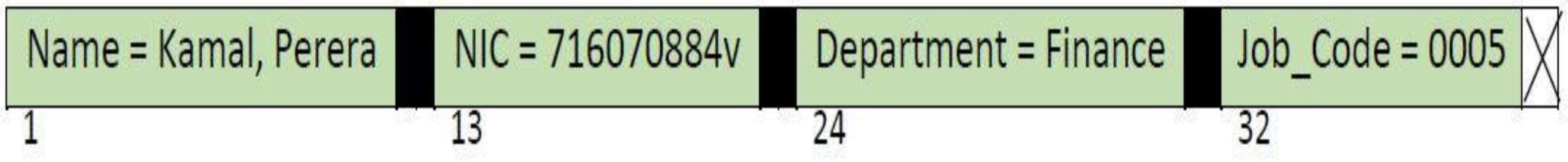


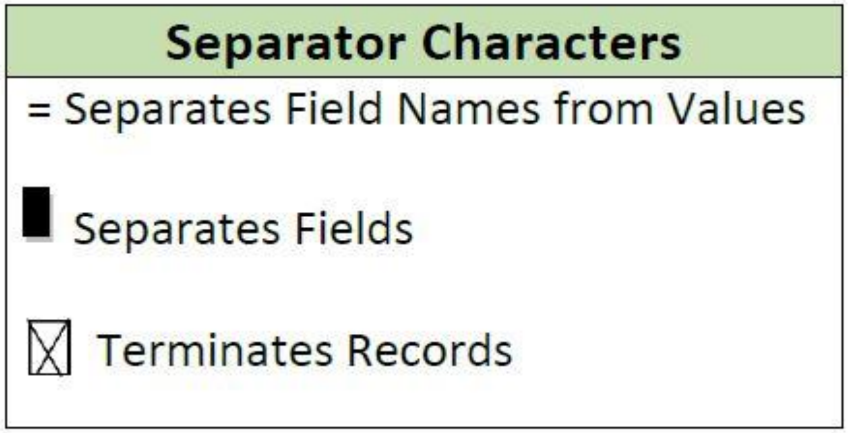
3.1 Disk Storage and Basic File Structures

3.1.5. Placing File Records on Disk

<u>Record Storage Format 3</u>

Eg: A variable-field record with three types of separator characters





3.1 Disk Storage and Basic File Structures

3.1.5. Placing File Records on Disk

- **In a record with Optional fields**;
  - A series of <Field-Name, Field-Value> pairs can be added in each record instead of the field values if the overall number of fields for the record type is high but the number of fields that actually occur in a typical record is low.
  - It will be more practical to store a **Field Type** code, to each field and include in each record a series of <Field-Type, Field-Value>.
- **In a record with a Repeating Field**;
  - One separator character can be used to separate the field's repeated values and another separator character can be used to mark the field's end.

Activity

Fill in the blanks in the following statements.

1. A file where the sizes of records in it are different in size is called a ________________.

2. A __________________ includes different types of records inside it.

3. In a file, the records belong to _________ record type.

4. A ___________ length record can be made by assigning “Null” for optional fields where data values are not available

5. To determine and terminate variable lengths special characters named as __________ can be used.

3.1 Disk Storage and Basic File Structures

3.1.5. Placing File Records on Disk

- **Block** is a unit of data transfer between disk and memory.
- When the block size exceeds the record size, each block will contain several records, however, certain files may have exceptionally large records that cannot fit in a single block.
- **Blocking Factor (*bfr*)** is the number of records per block in bytes.
  - If Block Size> Record Size,

    bfr can be calculated using the below equation.

$$\boxed{bfr = B / R}$$

Block Size = *B* bytes

Record Size = *R* bytes

3.1 Disk Storage and Basic File Structures

3.1.5. Placing File Records on Disk

- In calculating the *bfr* a floor function rounds down the number to the nearest integer.
- But, when the bfr is calculated, there may be some additional space remaining in each block.
- The unused space can be calculated with the equation given below.

$$\boxed{\textit{Unused Space in bytes} = \underset{\text{Block size}}{B} - \underbrace{bfr * R}_{\text{Space dedicated for blocks}}}$$

3.1 Disk Storage and Basic File Structures

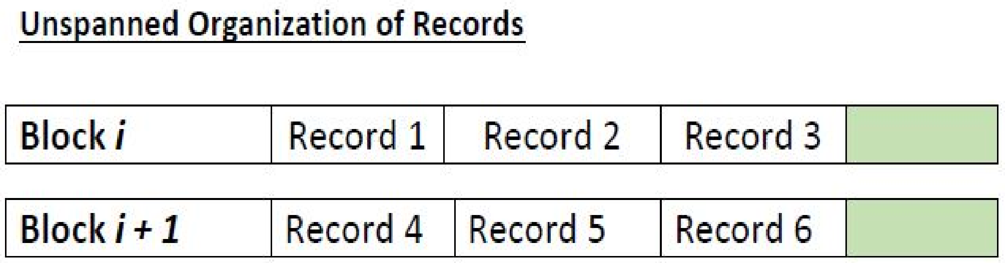
3.1.5. Placing File Records on Disk

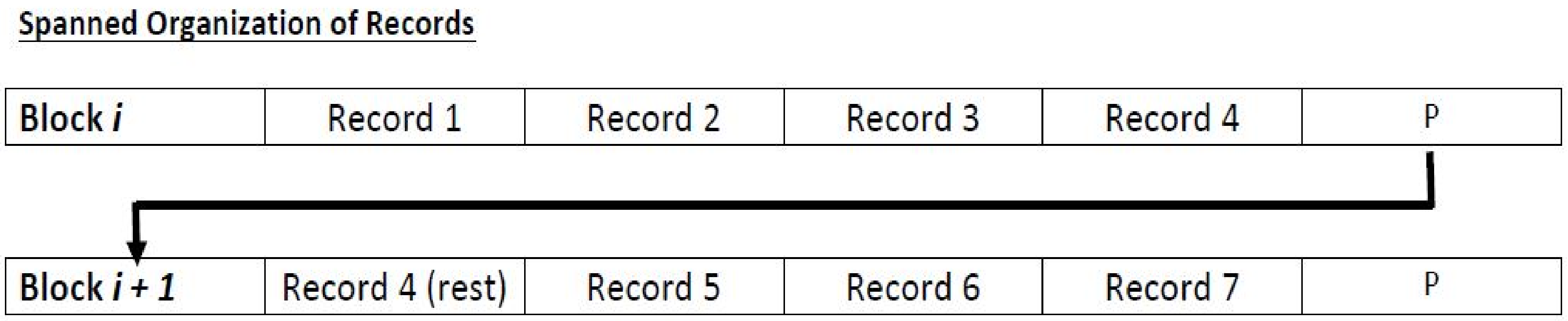
- Upon using the unused space, to minimize waste of space, a part of a record can be stored in one block and the other part can be stored in another block.
- If the next block on disk is not the one holding the remainder of the record, a **Pointer** at the end of the first block refers to it.
- **Spanned Organization of Records** - One record spanning to more than one block.
  - Used when a record is larger than the block size.
- **Unspanned Organization of Records** - Not allowing records to span into more than one block.
  - Used with fixed length records.

3.1 Disk Storage and Basic File Structures

3.1.5. Placing File Records on Disk

Let’s look at the representation of Spanned and Unspanned Organization of Records.





3.1 Disk Storage and Basic File Structures

3.1.5. Placing File Records on Disk

- A spanned or unspanned organization can be utilized in variable-length records.
- If it is a spanned organization, each block may store a different number of records.
- Here the *bfr* would be the average number of records per block.
- Hence the number of blocks b needed for a file of r records is,

$$\boxed{b = r / bfr}$$

3.1 Disk Storage and Basic File Structures

3.1.5. Placing File Records on Disk

Example of Calculation

There is a disk with **block size B=256 bytes**. A file has **r=50,000** STUDENT records of fixed-length. Each record has the following fields:

NAME (55 bytes),	STDID (4 bytes),
DEGREE(2 bytes),	PHONE(10 bytes),
SEX (1 byte).	

(i) Calculate the record size in Bytes.

Record Size R = (55 + 4 + 2 + 10 + 1) = 72 bytes

3.1 Disk Storage and Basic File Structures

3.1.5. Placing File Records on Disk

Example of Calculation Continued…

(ii) Calculate the blocking factor (bfr)

Blocking factor bfr = floor (B/R)

= floor(256/72)

= 3 records per block

Floor Function = Rounds the value down to the previous integer.

3.1 Disk Storage and Basic File Structures

3.1.5. Placing File Records on Disk

Example of Calculation Continued...

(iii) Calculate the number of file blocks (b) required to store the STUDENT records, assuming an unspanned organization.

Number of blocks needed for file = ceiling(r/bfr)

= ceiling(50000/3)

= 16667

Ceiling Function = Rounds the value up to the next integer.

3.1 Disk Storage and Basic File Structures

3.1.5. Placing File Records on Disk

- A **File Header**, also known as a File Descriptor, includes information about a file that is required by the system applications which access the file records.
- For fixed-length unspanned records, the header contains information to determine the disk addresses of the blocks, as well as record format descriptions, which may include field lengths and the order of fields within a record, and field type codes, separator characters, and record type codes for variable-length records.
- One or more blocks are transferred into main memory buffers to search for a record on disk.
- The search algorithms must do a **Linear Search** over the file blocks if the address of the block containing the requested record is unknown.

Activity

State the answer for the following calculations.

Consider a disk with **block size B=512 bytes**. A file has **r=30,000** EMPLOYEE records of fixed-length. Each record has the following fields: NAME (30 bytes), NIC (9bytes), DEPARTMENTCODE (9 bytes), ADDRESS (40 bytes), PHONE (9 bytes),BIRTHDATE (8 bytes), SEX (1 byte), JOBCODE (4 bytes), SALARY (4 bytes, real number). An additional byte is used as a deletion marker.

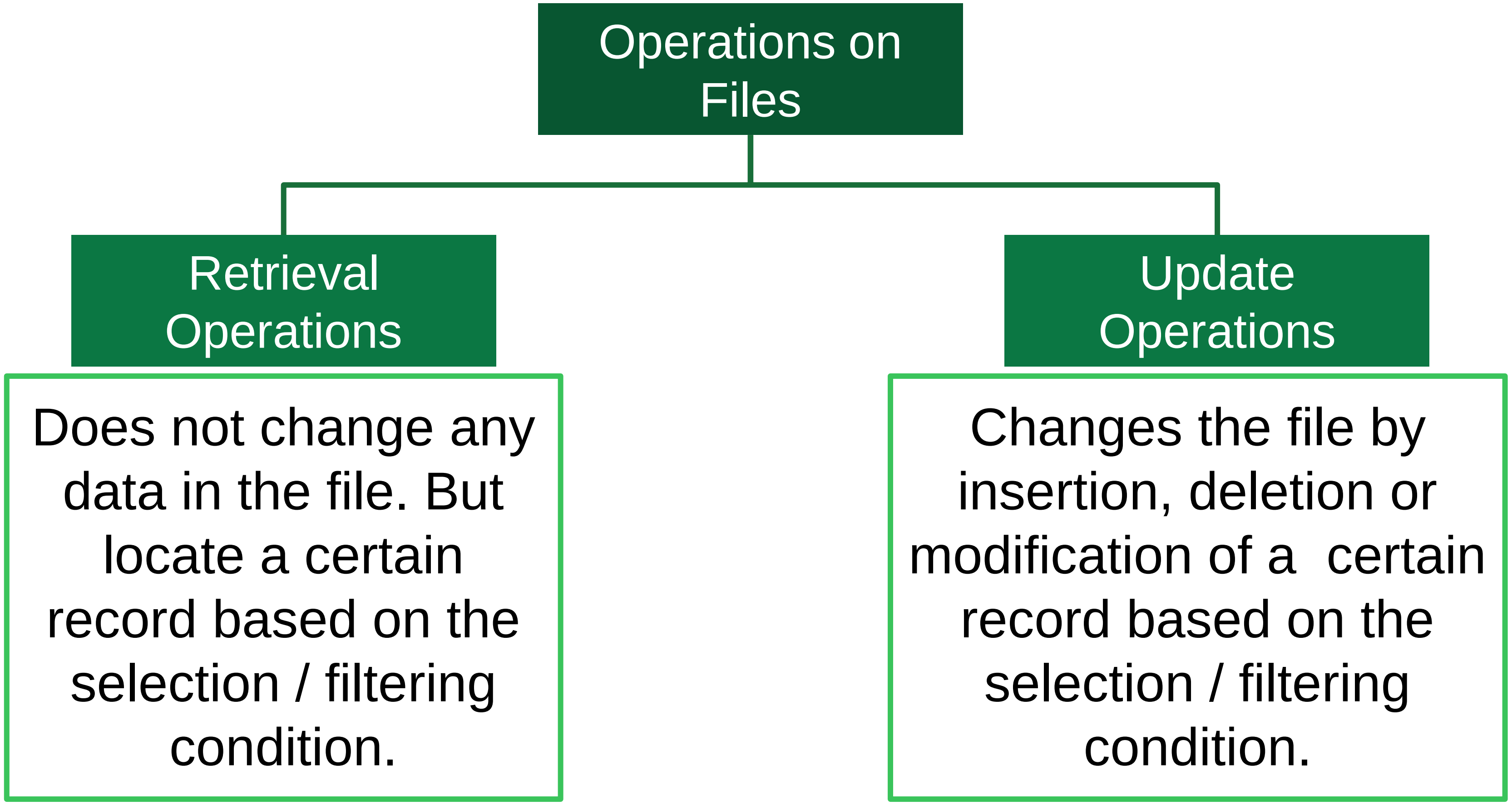
(i) Calculate the record size in Bytes.

(ii) Calculate the blocking factor (bfr)

(iii) Calculate the number of file blocks (b) required to store the EMPLOYEE records, assuming an unspanned organization.

3.1 Disk Storage and Basic File Structures

3.1.6 File Operations



Operations on Files

Retrieval Operations

Does not change any data in the file. But locate a certain record based on the selection / filtering condition.

Update Operations

Changes the file by insertion, deletion or modification of a certain record based on the selection / filtering condition.

3.1 Disk Storage and Basic File Structures

3.1.6. File Operations

Emp_No	Name	Data_Of_Birth	Position	Salary
0001	Nimal	1971 - 04 - 13	Manager	70,000
0005	Krishna	1980 - 01 - 25	Supervisor	50,000

- Simple Selection Condition

  Search for the record where Emp_No = “0005”
- Complex Selection Condition

  Search for the record where Salary>60,000

3.1 Disk Storage and Basic File Structures

3.1.6. File Operations

- When several file records meet a search criterion, the first record in the physical sequence of file records is identified and assigned as the **Current Record**. Following search operations will start with this record and find the next record in the file that meets the criterion.

- The actual procedures for identifying and retrieving file records differ from one system to the next.

3.1 Disk Storage and Basic File Structures

3.1.6. File Operation

The following are the File Access Operations.

Operation	Description
Open	Allows to read or write to a file. Sets the file pointer to the file's beginning.
Reset	Sets the file pointer of an open file to the beginning of the file.
Find (Locate)	The first record that meets a search criterion is found. The block holding that record is transferred to a main memory buffer. The file pointer is set to the buffer record, which becomes the current record.

3.1 Disk Storage and Basic File Structures

3.1.6. File Operations

Operation	Description
Read (Get)	Copies the current record from the buffer to a user-defined program variable. The current record pointer may also be advanced to the next record in the file using this command.
FindNext	Searches the file for the next entry that meets the search criteria. The block holding that record is transferred to a main memory buffer.
Delete	The current record is deleted, and the file on disk is updated to reflect the deletion.

3.1 Disk Storage and Basic File Structures

3.1.6. File Operations

Operation	Description
Modify	Modifies some field values for the current record and the file on disk is updated to reflect the modification.
Insert	Locates the block where the record is to be inserted and transfers that block into a main memory buffer to insert a new record in the file and the file on disk is updated to reflect the insertion.
Close	Releases the buffers and does any other necessary cleaning actions to complete the file access.

3.1 Disk Storage and Basic File Structures

3.1.6. File Operations

- The following is called “Record at a time” operation since it is applied to a single record.

Operation	Description
Scan	Scan returns the initial record if the file has just been opened or reset; otherwise, it returns the next record.

3.1 Disk Storage and Basic File Structures

3.1.6. File Operations

- The following are called “Set at a time” operations since they are applied to the file in full.

Operation	Description
FindAll	Locates all the records in the file that satisfy a search condition.
FindOrdered	Locates all the records in the file in a specified order condition.
Reorganize	Starts the reorganization process. (In cases such as ordering the records)

3.1 Disk Storage and Basic File Structures

3.1.6. File Operations

- **File Organization** - The way a file's data is organized into records, blocks, and access structures, including how records and blocks are put on the storage media and interconnected.

- **Access Methods** - A set of operations that may be applied to a file is provided. In general, a file structured using a specific organization can be accessed via a variety of techniques.

3.1 Disk Storage and Basic File Structures

3.1.6. File Operations

- **Static Files -** The files on which modifications are rarely done.

- **Dynamic Files -** The files on which modifications are frequently done.

- **Read Only File** - A file where modifications cannot be done by the end user.

Activity

Match the following descriptions with the relevant file operation out of the following.

[Find, Reset, Close, Scan, FindAll]

1. Returns the initial record if the file has just been opened or reset; otherwise, returns the next record.
2. Releases the buffers and does any other necessary cleaning actions
3. Sets the file pointer of an open file to the beginning of the file
4. The first record that meets a search criterion is found
5. Locates all the records in the file that satisfy a search condition.

3.1 Disk Storage and Basic File Structures

3.1.7. Files of unordered records (Heap Files) and ordered records (Sorted Files)

Files of Unordered Records (Heap Files)

- Records are entered into the file in the order in which they are received, thus new records are placed at the end.
- Inserting a new record is quick and efficient. The file's last disk block is transferred into a buffer, where the new record is inserted before the block is overwritten to disk. Then the final file block's address is saved in the file header.
- Searching for a record is done by the **Linear Search**.

3.1 Disk Storage and Basic File Structures

3.1.7. Files of unordered records (Heap Files) and ordered records (Sorted Files)

Files of Unordered Records (Heap Files)

- If just one record meets the search criteria, the program will typically read into memory and search half of the file blocks before finding the record. Here, on average, searching (***b/2***) blocks for a file of b blocks is required.
- If the search criteria is not satisfied by any records or there are many records, the program must read and search all ***b*** blocks in the file.

3.1 Disk Storage and Basic File Structures

3.1.7. Files of unordered records (Heap Files) and ordered records (Sorted Files)

Files of Unordered Records (Heap Files)

- Deleting a Record.
  - A program must first locate its block, copy the block into a buffer, remove the record from the buffer, and then rewrite the block back to the disk to **delete** a record.
  - This method of deleting a large number of data results in waste of storage space.

3.1 Disk Storage and Basic File Structures

3.1.7. Files of unordered records (Heap Files) and ordered records (Sorted Files)

Files of Unordered Records (Heap Files)

- Deleting a Record cont.
  - **Deletion Marker** - An extra byte or bit stored with every record whereas the deletion marker will get a certain value when the record is deleted. This value is not similar to the value that the deletion marker holds when there is data available in the record.
  - Using the space of deleted records to store data can also be used. But it includes additional work.

3.1 Disk Storage and Basic File Structures

3.1.7. Files of unordered records (Heap Files) and ordered records (Sorted Files)

Files of Unordered Records (Heap Files)

- Modifying a Record.
    - Because the updated record may not fit in its former space on disk, modifying a variable-length record may require removing the old record and inserting the modified record.

3.1 Disk Storage and Basic File Structures

3.1.7. Files of unordered records (Heap Files) and ordered records (Sorted Files)

<u>Files of Unordered Records (Heap Files)</u>

- Reading a Record.
  - A sorted copy of the file is produced to read all entries in order of the values of some field. Because sorting a huge disk file is a costly task, specific approaches for external sorting are employed.

3.1 Disk Storage and Basic File Structures

3.1.7. Files of unordered records (Heap Files) and ordered records (Sorted Files)

Files of Ordered Records (Sorted Files)

- The values of one of the fields of a file's records, called the **Ordering Field** can be used to physically order the data on disk. It will generate an ordered or sequential file.
- Ordered records offer a few benefits over files that are unordered.
- The benefits are listed in the next slide.

3.1 Disk Storage and Basic File Structures

3.1.7. Files of unordered records (Heap Files) and ordered records (Sorted Files)

Files of Ordered Records (Sorted Files)

- Benefits of Ordered records:
  - Because no sorting is necessary, reading the records in order of the ordering key values becomes highly efficient.
  - Because the next item is in the same block as the current one, locating it in order of the ordering key typically does not need any extra block visits.When the binary search approach is employed, a search criterion based on the value of an ordering key field results in quicker access.

3.1 Disk Storage and Basic File Structures

3.1.8. Hashing techniques for storing database records: Internal hashing, external hashing

- Another kind of primary file structure is **Hashing**, which allows for extremely quick access to information under specific search conditions.
- The equality requirement on a single field, termed the **Hash Field**, must be used as the search condition.
- The hash field is usually also a key field of the file, in which case it is referred to as the hash key.
- The concept behind hashing is to offer a function h, also known as a **Hash Function** or randomizing function, that is applied to a record's hash field value and returns the address of the disk block where the record is stored.

Activity

Fill in the blanks with the correct technical term.

1. The _____________________ is an extra byte that is stored with a record which will get updated when a record is deleted.

2. A field which can generate an ordered or sequential file by physically ordering the records is called ______________.

3. The function which calculates the Hash value of a field is called ______________.

4. Searching for a record in a Heap file is done by the ____________.

3.1 Disk Storage and Basic File Structures

3.1.8. Hashing techniques for storing database records: Internal hashing, external hashing

- Internal Hashing.

  When it comes to internal files, hashing is usually done with a **Hash Table** and an array of records.

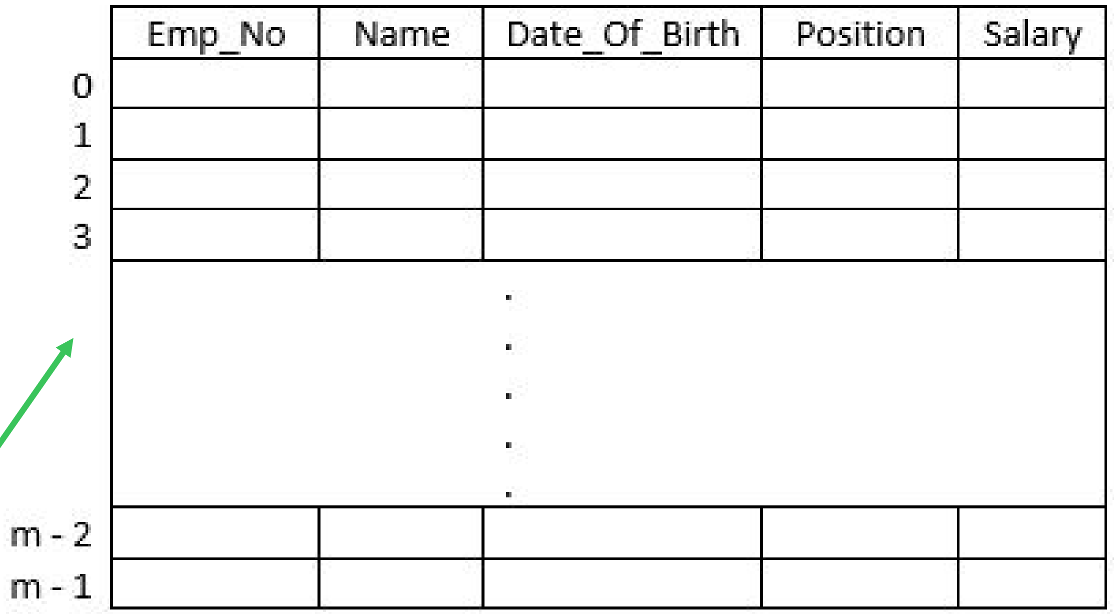
- Method 1 for Internal Hashing
  - If the array index range is 0 to m – 1, there are m slots with addresses that correspond to the array indexes.
  - Then a hash function is selected that converts the value of the hash field into an integer between 0 and m-1.
  - The record address is then calculated using the given function.
    - h(K) = Hash Function of K Value
    - K = Field Value

h(K) = K Mod m

3.1 Disk Storage and Basic File Structures

3.1.8. Hashing techniques for storing database records: Internal hashing, external hashing

- <u>**Internal Hashing.**</u>



	Emp_No	Name	Date_Of_Birth	Position	Salary
0					
1					
2					
3					
			. . . . .		
m - 2					
m - 1					

Internal Hashing Data Structure - Array of m positions to use in internal hashing

3.1 Disk Storage and Basic File Structures

3.1.8. Hashing techniques for storing database records: Internal hashing, external hashing

- **<u>Internal Hashing.</u>**

- <u>Method 2 for Internal Hashing</u>
- By using algorithms that calculate the Hash Function

```
temp ← 1;
for i ← 1 to 20 do temp ← temp * code(K[i ] ) mod M ;
hash_address ← temp mod M;
```

Hashing Algorithm in applying the mod hash function to a character string K.

3.1 Disk Storage and Basic File Structures

3.1.8. Hashing techniques for storing database records: Internal hashing, external hashing

- **Internal Hashing.**

- Method 3 for Internal Hashing
- **Folding** - To compute the hash address, an arithmetic function such as addition or a logical function such as Exclusive OR (XOR) is applied to distinct sections of the hash field value.

3.1 Disk Storage and Basic File Structures

3.1.8. Hashing techniques for storing database records: Internal hashing, external hashing

- **Internal Hashing.**
- **Collision** - When the hash field value of a record that is being inserted hashes to an address that already holds another record.
- Because the hash address is already taken, the new record must be moved to a different location.
- **Collision Resolution** - The process of finding another location.
- There are several methods for collision resolution.

3.1 Disk Storage and Basic File Structures

3.1.8. Hashing techniques for storing database records: Internal hashing, external hashing

- **Internal Hashing.**
- Methods of Collision Resolution
  - **Open Addressing** - The program scans the subsequent locations in order until an unused (empty) position is discovered, starting with the occupied position indicated by the hash address.
  - **Chaining** - Changing the pointer of the occupied hash address location to the address of the new record in an unused overflow location and putting the new record in an unused overflow location.
  - **Multiple Hashing** - If the first hash function fails, the program uses a second hash function. If a new collision occurs, the program will utilize open addressing or a third hash function, followed by open addressing if required.

3.1 Disk Storage and Basic File Structures

3.1.8. Hashing techniques for storing database records: Internal hashing, external hashing

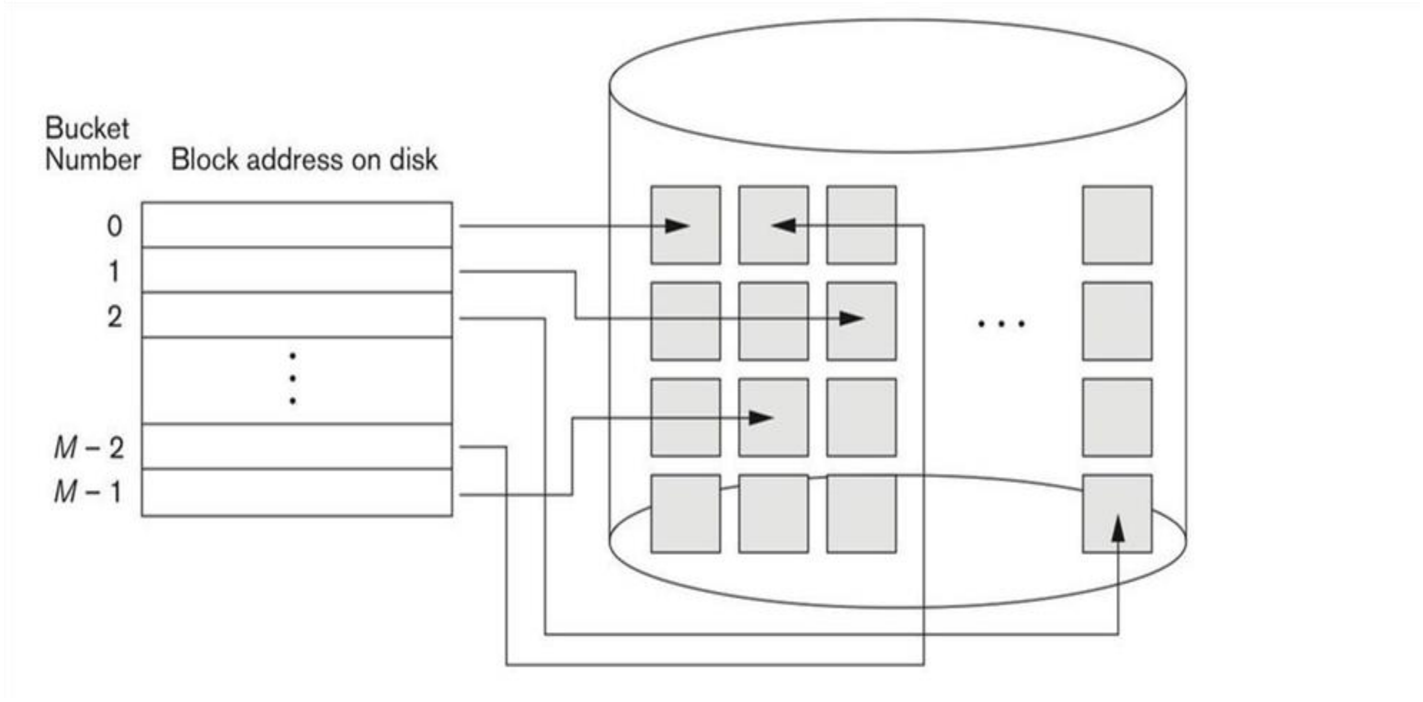
- <u>**External Hashing.**</u>
- Hashing for disk files is named as **External Hashing**.
- The target address space is built up of **Buckets**, each of which stores many records, to match the properties of disk storage.
- A bucket is a continuous group of disk blocks or a single disk block.
- Rather than allocating an absolute block address to the bucket, the hashing function translates a key to a relative bucket number.
- The bucket number is converted into the matching disk block address via a table in the file header.

3.1 Disk Storage and Basic File Structures

3.1.8. Hashing techniques for storing database records: Internal hashing, external hashing

- **External Hashing.**

  The following diagram shows matching bucket numbers (0 to M -1) to disk block addresses.



3.1 Disk Storage and Basic File Structures

3.1.8. Hashing techniques for storing database records: Internal hashing, external hashing

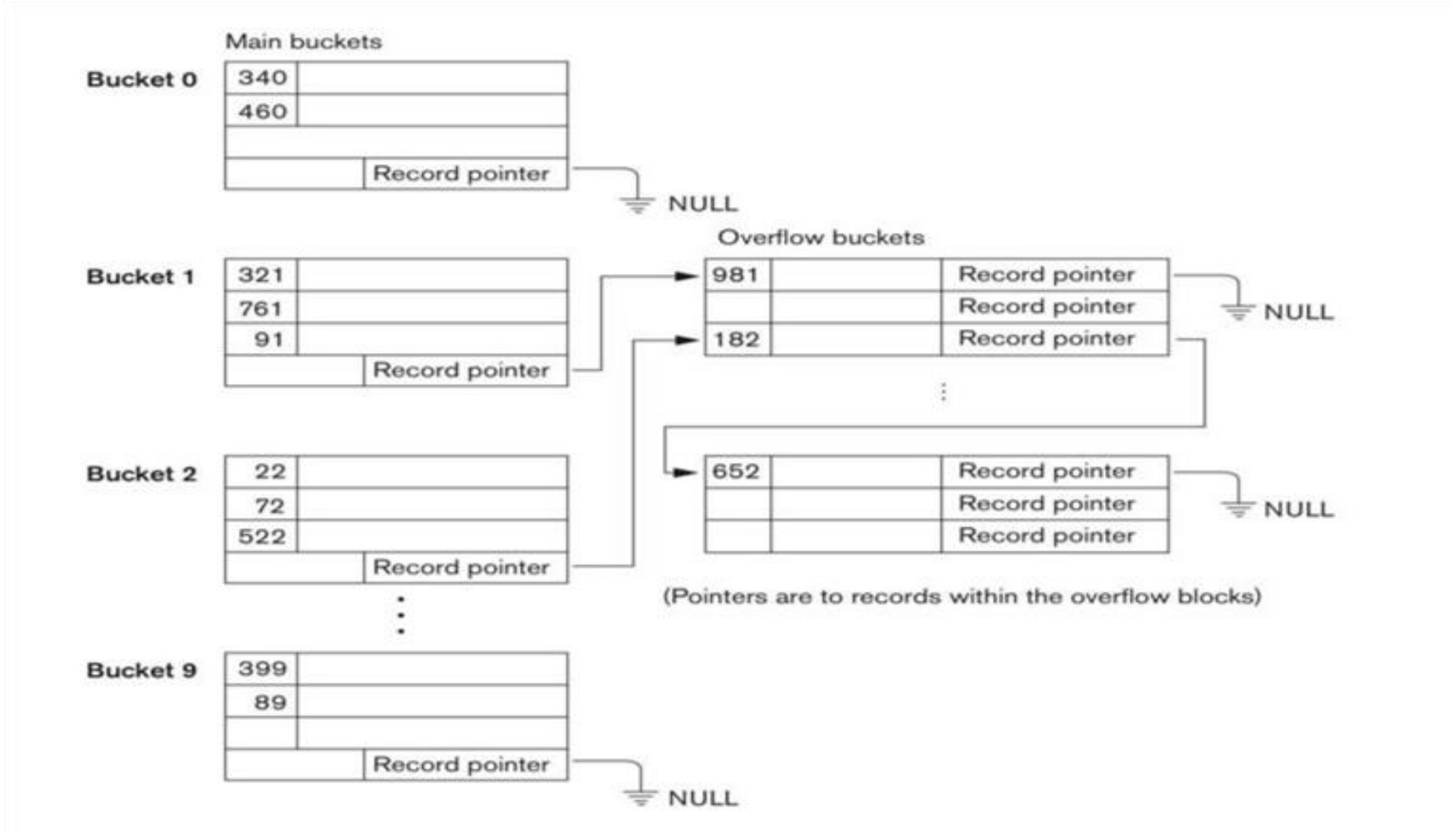
- **External Hashing.**
- Since many records will fit in a bucket can hash to the same bucket without generating issues, the collision problem is less severe with buckets.
- When a bucket is full to capacity and a new record is entered, a variant of chaining can be used in which a pointer to a linked list of overflow records for the bucket is stored in each bucket.
- Here, the linked list pointers should be **Record Pointers**, which comprise a block address as well as a relative record position inside the block.

3.1 Disk Storage and Basic File Structures

3.1.8. Hashing techniques for storing database records: Internal hashing, external hashing

- **External Hashing.**

Handling overflow for buckets by chaining



Activity

Match the description with the correct term.

1. Applying an arithmetic function such as addition or a logical function such as Exclusive OR (XOR) to distinct sections of the hash field value.
2. The technique used for hashing where the program uses a second hash function if first hash function fails.
3. The instance when the value of the hash field of a newly inserted record hashes to an address that already contains another record.
4. Starting with the occupied place given by the hash address, the program examines the succeeding locations in succession until an unused (empty) spot is located when a collision has occurred.
5. A continuous group of disk blocks or a single disk block which is comprising of the target address space.

3.2 Introduction to indexing

- **Indexes** are used to speed up record retrieval in response to specific search criteria.
- The index structures are extra files on disk that provide secondary access pathways, allowing users to access records in different ways without changing the physical location of records in the original data file on disk.
- They make it possible to quickly access records using the **Indexing Fields** that were used to create the index.
- Any field in the file can be used to generate an index, and the same file can have numerous indexes on separate fields as well as indexes on multiple fields.

3.2 Introduction to indexing

- **Some Commonly used Types of Indexes**
  - Single Level Ordered Indexes
    - Primary Index
    - Secondary Index
    - Clustering Index
  - Multi Level Tree Structured Indexes
    - B Trees
    - B+ Trees
  - Hash Indexes
  - Logical Indexes
  - Multi Key Indexes
  - Bitmap Indexes

3.3 Types of Indexes

- **Single Level Indexes: Primary, Clustering and Secondary indexes**
  - Primary, Clustering and Secondary index are types of single level ordered indexes.
  - In some books, the last pages have ordered list of words, which are categorized from A-Z. In each category they have put the word, as well as the page numbers where that particular word exactly appears. These list of words are known as index.
  - If a reader needs to find about a particular term, he/she can go to the index and find the pages where the term appears first and then can go through the particular pages.
  - Otherwise readers have to go through the whole book, searching the term, which is similar to the linear search.

3.3 Types of Indexes

- **Single Level Indexes: Primary, Clustering and Secondary indexes**
  - **Primary Index -** defined for an ordered file of records using the ordering key field.
  - File records on a disk are physically ordered by the *ordering key field.* This ordering key field holds unique values for each record.
  - Clustering index is applied when multiple records in the file have same value for the ordering field; here the ordering field is a non key field. In this scenario, data file is referred as clustered file.

3.3 Types of Indexes

- **Single Level Indexes: Primary, Clustering and Secondary indexes**
  - A file can have maximum of one physical ordering field. Therefore, a file can have one primary index or one clustering index. However, it cannot hold both primary index and clustered index at once.
  - Unlike the primary indexes, a file can have few secondary indexes additional to the primary index.

3.3 Types of Indexes

- **Single Level Indexes: Primary indexes**
  - Primary indexes are access structures that used to increase the efficiency of searching and accessing the data records in a data file.
  - An ordered file which consists **two fields** and limited length records is known as a primary index file.
  - One field is the **ordering key field.** Ordering key field of the index file and the primary key of the data file have same data type.
  - The other field contains **pointers** to the disk blocks.
  - Hence, the index file contains one **index entry(**a.k.a index record**)** for each block in the data file.

3.3 Types of Indexes

- **Single Level Indexes: Primary indexes**
  - As mentioned before, index entry consist of two values.

    i. Primary key field value of the first record in a data block.

    i. Pointer to the data block which contains above primary key field.

    for index entry *i*, two field values can be referred as,

    <K(*i*) P(*i*)>

3.3 Types of Indexes

- **Single Level Indexes: Primary indexes**
    - Ex: Assuming that “name” is a unique field and the “name” has been used to order the data file, we can create index file as follows.

      <K(1) = (Aaron, Ed), P(1) = address of block 1>

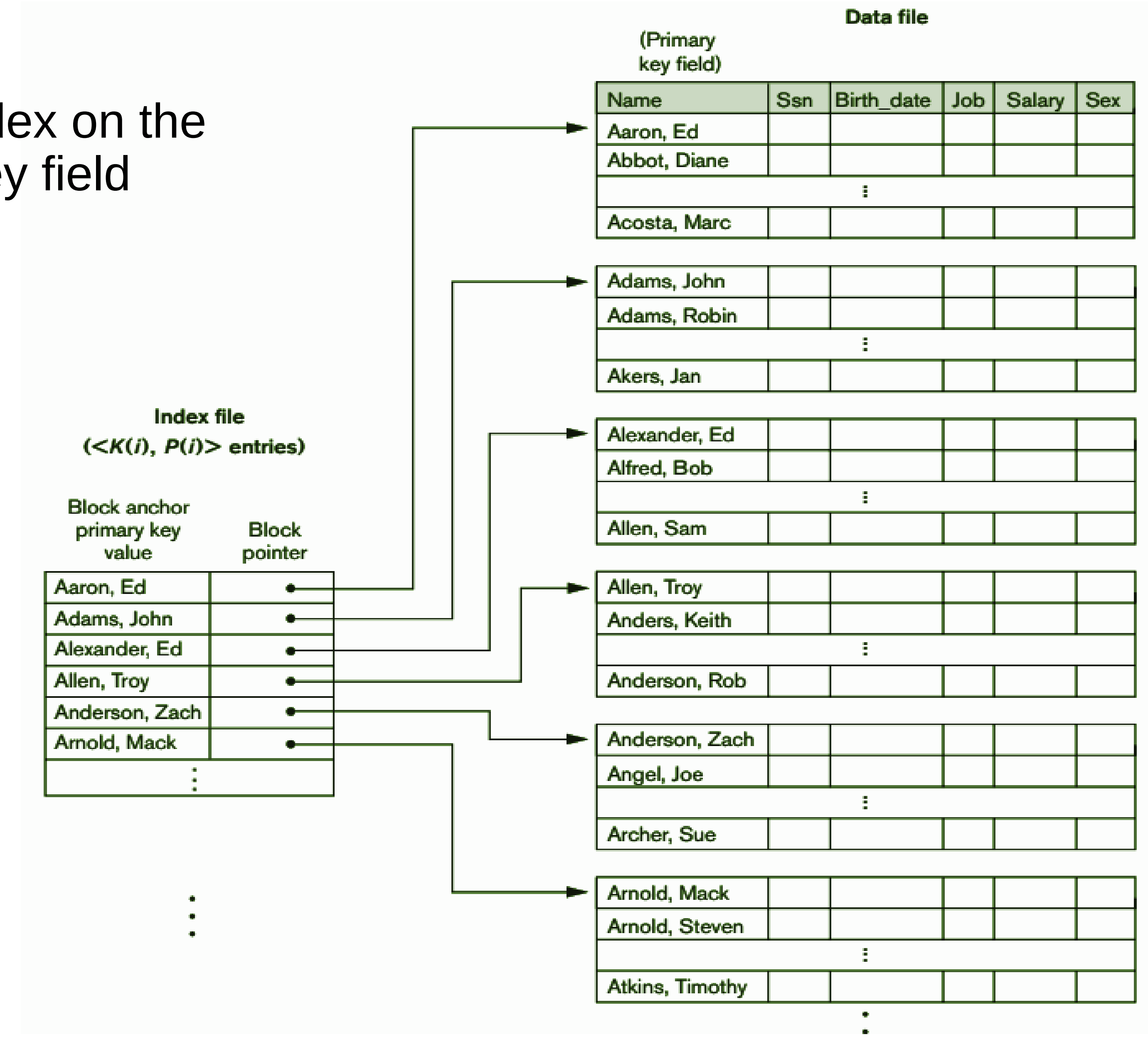
      <K(2) = (Adams, John), P(2) = address of block 2>

      <K(3) = (Alexander, Ed), P(3) = address of block 3>

The image given in the next slide illustrates the index file and respective block pointers to the data file.

3.3 Types of Indexes

Primary index on the ordering key field



3.3 Types of Indexes

- **Single Level Indexes: Primary indexes**
  - In the given illustration of the previous slide, **number of index entries in the index file is equal to the number of disk blocks in the data file**.
  - **Anchor record/Block anchor**: for a given block in an ordered data file, the first record in that block is known as anchor record. Each block has an anchor record.

3.3 Types of Indexes

- **Single Level Indexes: Primary indexes**
  - **Dense index and Sparse index**
    1. Indexes that contain index entries for each record in the data file (or each search key value) referred to as **dense index.**
    2. Indexes that contains index entries for some records in the data file referred as **sparse index.**
  - Therefore, by definition, primary index falls into the sparse (or the non - dense) index type since it does not keep index entries for every record in the data file. Instead, primary index keep index entries for anchor records for each block which contains data file.

3.3 Types of Indexes

- **Single Level Indexes: Primary indexes**
  - Generally, a primary indexing file takes smaller space compared to the datafile due to two reasons.
    - i. Number of index entries are smaller than the number of records in the data file.
    - ii. Index entry holds two fields which are comparatively very short in size.
  - Hence, performing a binary search on an index file results in less number of block accesses when compared to the binary search performed on a data file.

3.3 Types of Indexes

- **Single Level Indexes: Primary indexes**
  - Block accesses for an ordered file with *b* blocks can be calculated by using $\log_2 b$.
  - Let's assume that we want to access a record with primary key value is K , which resides on the block address is P(i), where $K(i) \leq K < K(i + 1)$.
  - Since the physical ordering of the data file is depends on the primary key, all records of K(i) resides in the the *i*th block.
  - Therefore, to retrieve the record corresponding to given K value, a binary search is performed on the index file to find the index entry for *i.*
  - Then we can get the block address for the P (i) and retrieve the record.

3.3 Types of Indexes

- **Single Level Indexes: Primary indexes**

Ex: Let's say we have an ordered file with its key field. File records are of fixed size and are unspanned. Following details are given and we are going to calculate the block accesses require when performing a binary search,

number of records **r** = 300,000
block size **B** = 4,096 bytes
record length **R** = 100 bytes

We can calculate the blocking factor,

bfr = (**B/R**)= floor(4,096/100) = 40 records per block

Hence, the number of blocks needed to store all records

b = (**r/bfr**) = ceiling(300,000/40)= 7,500 blocks.

Block accesses required = $\log_2 b$

$= \text{ceiling}(\log_2 7{,}500) = 13$

3.3 Types of Indexes

- **Single Level Indexes: Primary indexes**

Ex: For the previous scenario given, if we have a primary index file with 9 bytes long ordering key field (V) and 6 bytes long block pointer (P), the required block accesses can be calculated as follows.

number of records **r** = 300,000
block size **B** = 4,096 bytes
index entry length $\mathbf{R}_i$ = (**V+P**)= 15

We can calculate the blocking factor for index,
bfr = (**B/R**)= floor(4,096/15) = 273 records per block

Number of index entries required is equal to number of blocks required for data file.
Hence, number of blocks needed for index file,
$\mathbf{b}_i$ = (**r/bfr**) = ceiling(7,500/273)= 28 blocks.

Go to the next slide for the rest of the calculation →

3.3 Types of Indexes

block accesses required

= $\mathbf{log_2\ b_{\mathit{i}}}$= ceiling($\log_2 28$)= 5

However to access the record using the index, we have to perform binary search on the index file plus one additional access to retrieve the record.

- Therefore the formula for total number of block access to access the record should be,

  $\log_2 b_i + 1$ accesses = 6 block accesses.

3.3 Types of Indexes

- **Single Level Indexes: Primary indexes**
  - Primary indexing has problems when we add new records to or delete existing records from an ordered file.
  - If a new record is inserted to its correct position according to the order, existing records in the data file might subject to change their index in order to spare some space for the new record.
  - Sometimes this change result in change of anchor records as well.
  - Deletion of records also has the same issue as insertion.

3.3 Types of Indexes

- **Single Level Indexes: Primary indexes**
  - An unordered overflow file, can be used to scale down this problem.
  - Adding a linked list of overflow records for each block in the data file is another way to address this issue.
  - Deletion markers can be used to manage the issues with record deletion.

3.3 Types of Indexes

- **Single Level Indexes: Clustering indexes**
  - When a datafile is ordered using a non-key field which does not consist of unique values, such file are known as **clustered files**. The field which is used to order the file is known as **clustering field**.
  - Clustering index accelerate the retrieval of all records whose clustering field (field that is used to order the data file) has same value.
  - In primary index the the ordering field consist of distinct values unlike the clustering index.

3.3 Types of Indexes

- **Single Level Indexes: Clustering indexes**
  - Clustering index also consists of two fields. One is for the clustering field of the data file and the second one is for block pointers.
  - In the index file, there is only one entry for distinct values in the clustering field with a pointer to the **first block** where the record corresponding to the clustering field appear.

3.3 Types of Indexes

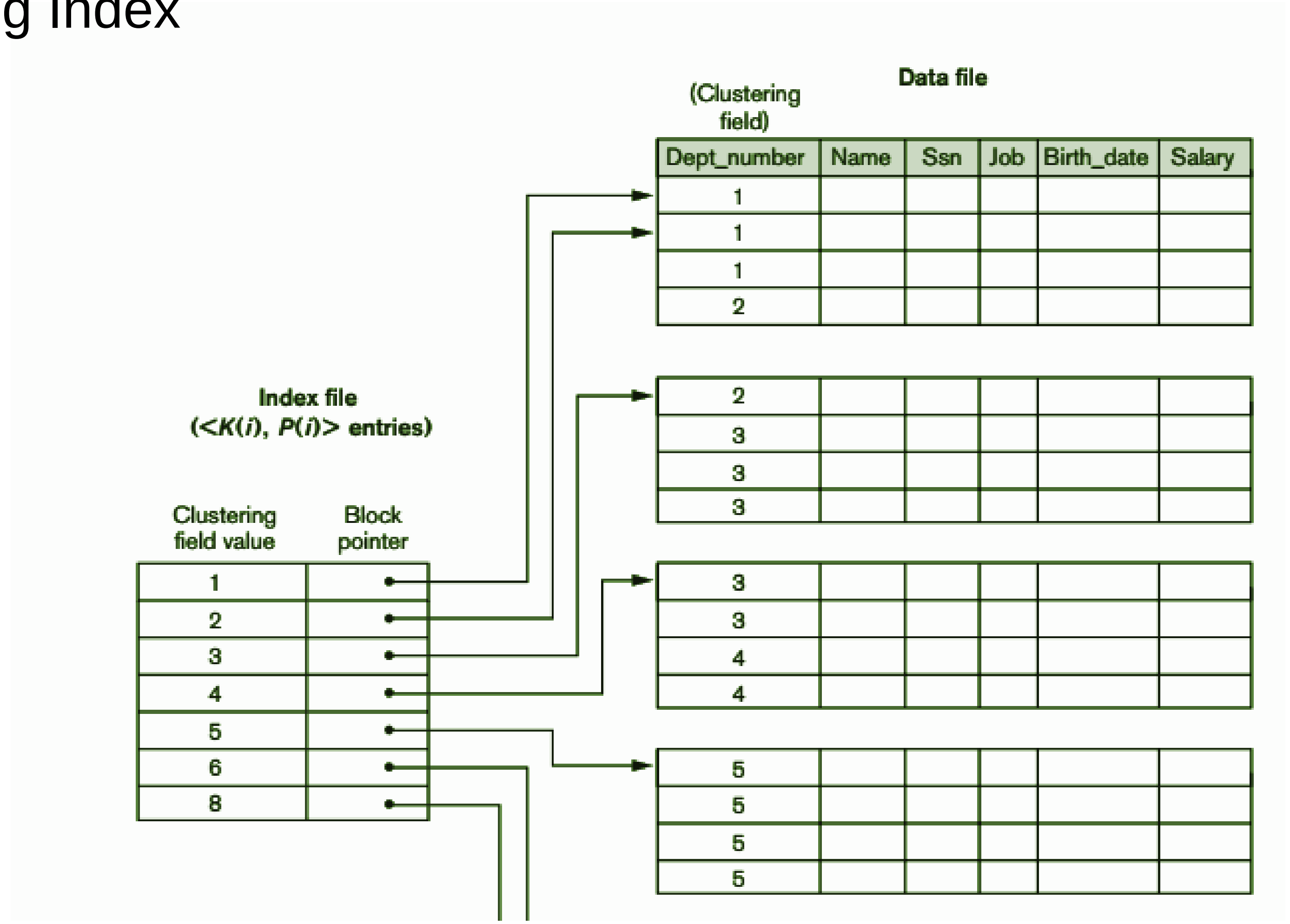
- **Single Level Indexes: Clustering indexes**
  - Since the data file is ordered, entering and deleting records still causes problems in the clustering index as well.
  - A common method to address this problem is to assign an entire block (or a set of neighbouring blocks) for each value in the clustering field.
  - All records that have similar clustering field will be stored in that allocated block.
  - This method ease the insertion and deletion of the records.

3.3 Types of Indexes

- **Single Level Indexes: Clustering indexes**
  - This problem can be scaled down using an unordered overflow file.
  - Adding a linked list of overflow records for each block in the data file is another way to address this issue.
  - Deletion markers can be used to manage the issues with record deletion.
  - Clustering index also falls into the sparse index type since the index field contains entries for distinct values of the ordering key field in the data file, rather than each and every record in the ordering key field.

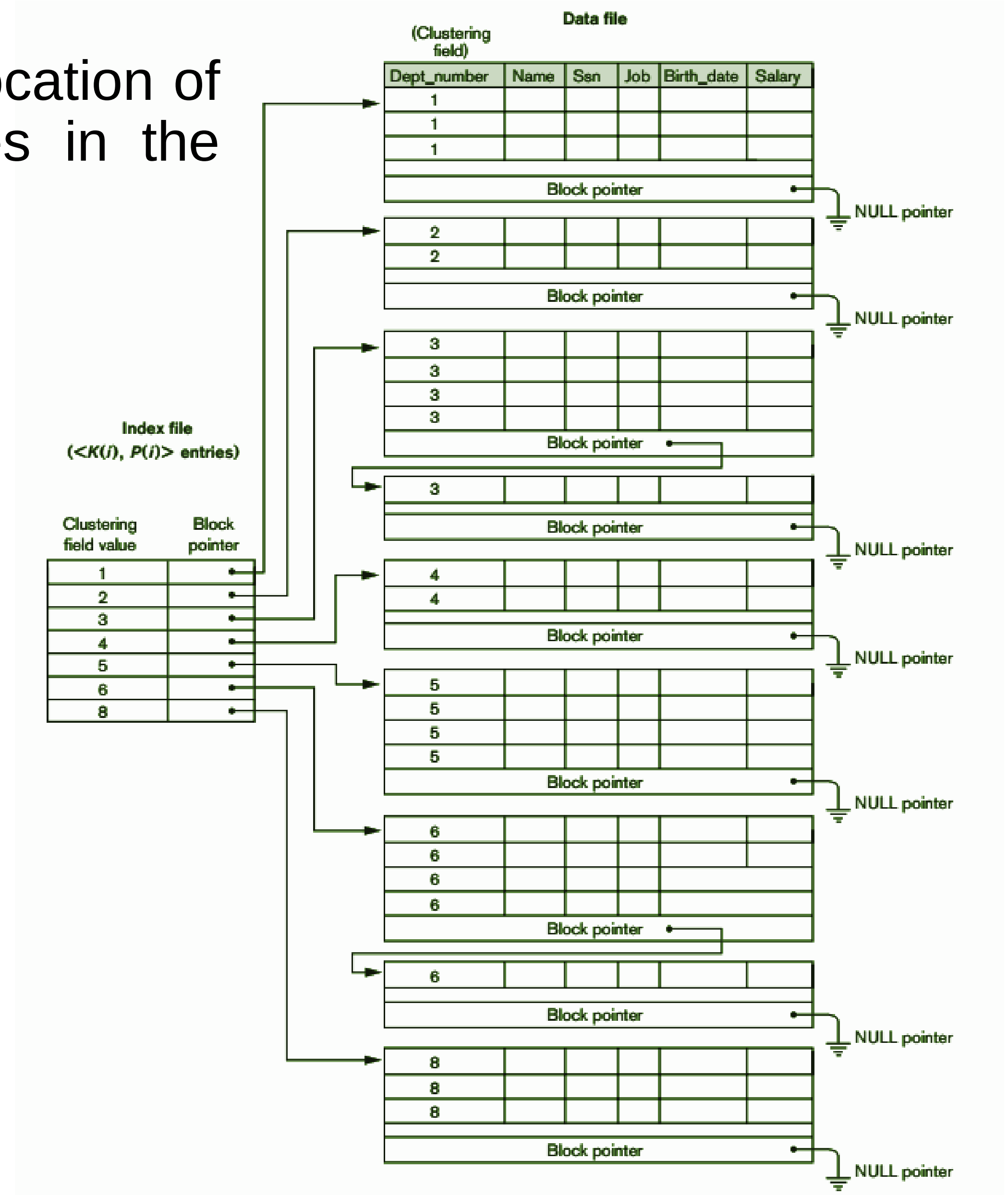
3.3 Types of Indexes

Clustering Index



3.3 Types of Indexes

Clustering Index with allocation of blocks for distinct values in the ordered key field.



3.3 Types of Indexes

- **Single Level Indexes: Clustering indexes**

Ex: For the same ordered file with r = 300,000, B = 4,096 bytes, let's say we have used a field "Zip code" which is non key field, to order the data file.

Assumption: Each Zip Code has equal number of records and there are 1000 distinct values for Zip Codes ($\mathbf{r_i}$). Index entries consist of 5-byte long Zip Code and 6-byte long block pointer.

Size of the record $\mathbf{R_i}$ **=** 5+6 = 11 bytes
Blocking factor $\mathbf{bfr_i}$ = $\mathbf{B/R_i}$ = floor(4,096/11)
= 372 index entries per block

Hence, number of blocks needed $\mathbf{b_i}$ = $\mathbf{R_i/bfr_i}$

= ceiling(1,000/372) = **3 blocks**.

Block accesses to perform a binary search,

= $\mathbf{\log_2 (b_i)}$ = **ceiling**($\mathbf{\log_2 (3)}$)= **2**

3.3 Types of Indexes

- **Single Level Indexes: Secondary indexes**
  - A secondary index provides an additional medium for accessing a data file which already has a primary access.
  - Data file records can be ordered, not ordered or hashed. Yet, the index file is ordered.
  - A candidate key which has unique values for every record or a non - key value which holds redundant values can be use as the indexing field to define the secondary indexing.
  - The first field of the index file has the same data type as the non-ordering field in the data file, which is an indexing field.
  - A block pointer or a record pointer is put in the second field.

3.3 Types of Indexes

- **Single Level Indexes: Secondary indexes**
  - For a single file, few secondary Indexes (and therefore indexing fields) can be created - Each of these serves as an additional method of accessing that file based on a specific field.
  - For a secondary index created on a candidate key (unique key/ primary key), which has unique values for every record in the file, the secondary index will get entries for every record in the data file.
  - The reason to have entries for every record is, the key attribute which is used to create secondary index has distinct values for each and every record.
  - In such scenarios, the secondary index will create a dense index which holds key value and block pointer for each record in the data file.

3.3 Types of Indexes

- **Single Level Indexes: Secondary indexes**
  - Same as before in primary index, here also two fields of index entries are referred as <K (i), P (i)>.
  - Since the order of the data file is based on the value of K (i), a binary search can be performed.
  - However, block anchors cannot be used since the records of the data file is not physically ordered by the values of the secondary key field.
  - This is the reason for creating an index entry for each record of data instead of using block anchors like in primary index.

3.3 Types of Indexes

- **Single Level Indexes: Secondary indexes**
  - Due to the huge number of entries, a secondary index requires much storage capacity when compared to the primary index.
  - But, on the other hand, secondary indexing gives greater improvement in the search time for an arbitrary record.
  - Secondary index is more important because we have to do a linear search of the data file, If there was no secondary index.
  - For a primary index, a binary search can be performed in the main file even if the index is not present.

3.3 Types of Indexes

- **Single Level Indexes: Secondary indexes**

Ex: If we take the same example in primary index and assume we search for a non-ordering key field V = 9 bytes long,in a file with 300,000 records with a fixed length of 100 bytes. And given block size B =4,096 bytes.

We can calculate the blocking factor,

bfr = (B/R)= floor(4,096/100) = 40 records per block

Hence, the number of blocks needed,

b = (r/bfr) = ceiling(300,000/40)= **7,500** blocks.

- If we perform a **linear search** on this file, the required number of block access **= b/2**

  = 7,500/2

  = 3,750 block accesses

3.3 Types of Indexes

- **Single Level Indexes: Secondary indexes**

However, if we have a secondary indexing on that non - ordering keyfield, with entries for the block pointers P= 6 bytes long,

Length of the index entry $\mathbf{R_i}$ $= \mathbf{V+P}$

$= \mathbf{9+6 = 15}$

Blocking factor $\mathbf{bfr_i}$ $= \mathbf{B/R_i}$

$= \mathbf{floor(4{,}096/15) = 273}$

Since the secondary index is dense, the number of index entries $(\mathbf{r_i})$ are same as the number of records (300,000) in the file.

- Therefore, number of blocks required for secondary index is,

$\mathbf{b_i}$ $= \mathbf{r_i / bfr_i}$
$\mathbf{=ceiling(300{,}000 / 273)}$
$\mathbf{= 1{,}099}$

3.3 Types of Indexes

- **Single Level Indexes: Secondary indexes**
  - If we perform a binary search on this secondary index the required number of block accesses can be calculated as follows,

    $(\log_2 b_i)$ = ceiling($\log_2 1{,}099$)

    = 11 block accesses.
  - Since we need additional block access to find the record in the data file using the index, the total number of block accesses required is,

    11 + 1 = 12 block accesses.

3.3 Types of Indexes

- **Single Level Indexes: Secondary indexes**
  - Comparing to the **linear search**, that required **3,750 block** accesses,the secondary index shows a big improvement with **12 block** accesses. But it is slightly worse than the **primary index** which needed only **6 block accesses.**
  - This difference is a result of the size of the primary index. The primary index is sparse index, therefore, it has only 28 blocks.
  - While the secondary index which is dense, require length of 1,099 blocks. This is longer when compared to the primary index.

3.3 Types of Indexes

- **Single Level Indexes: Secondary indexes**
  - Secondary index retrieves the records in the order of the index field that we considered to create the secondary index, because secondary indexing gives a **logical ordering** of the records.
  - However, in primary and clustering index, it assumes that, **physical ordering** of the file is similar to the order of the indexing field.

3.3.2 Multilevel indexes: Overview of multilevel indexes

- Considering a single-level index is an ordered file, we can create a primary index to the index file itself.
- Here the original index file is called as the first-level index and the index file created to the original index is called as the second-level index.
- We can repeat the process, creating a third, fourth, ..., top level until all entries of the top level fit in one disk block.
- A multi-level index can be created for any type of first level index (primary, secondary, clustering) as long as the first-level index consists of more than one disk block.

3.3.2 Multilevel indexes: Overview of multilevel indexes

- As we have discussed in topic 3.3, an ordered index file is associated to the primary, clustered and secondary indexing schemes.
- Binary search is used to find indexes and the algorithm continues to reduces the part of the index file that search, by factor 2 in each step. Hence we use the log function to the base 2. **($\log_2 b_i$)**
- Multilevel indexing is used to faster this search by reducing the search space if the blocking factor of the index is greater than 2.
- In multilevel indexing the blocking factor of the index ***bfr_i*** referred to as **fan-out** which is symbolized as ***fo.***
- In multilevel indexing, the number of block accesses required is (approximately) ***$\log_{fo} b_i$.***

3.3.2 Multilevel indexes: Overview of multilevel indexes

- If the first level index has r_1 entries, blocking factor for the first level ***bfr₁ = fo.***
- The number of blocks required for the first level is given by, (r_1 / fo).
- Therefore, the number of records in the second level index $r_2 = (r_1 / fo)$.
- Similarly, $r_3 = (r_2 / fo)$.
- However, we need to have second level only if the first level requires more than 1 block. Likewise, we consider for a next level only if the current level requires more than 1 block.
- If the top level is ***t,***

$$t = \lceil (\log_{fo}(r1)) \rceil$$

3.4 Indexes on Multiple Keys

- If a certain combination of attributes is used frequently, we can set up a key value on those combination of attributes for efficient access.
- For an example, let's say we have a file for *students*, containing student_id, name, age, gpa, department_id and department_name.
- If we want to find students whose department id = 1 and gpa is 3.5, we can have the search strategies specified in the next slide.

3.4 Indexes on Multiple Keys

1. By assuming only the *department_id has an index*, we can access the records with department_id = 1 using the index and then find the records that has 3.5 gpa.
2. Alternatively, we can assume only the *gpa has an index* and not the department_id, we can access the records with gpa = 3.5 using the index and then find the records that has department_id = 1.
3. If both of this *department_id and gpa fields have indexes,* we can get the records that meets the given individual condition (depadrmrnt_id = 1 and gpa = 3.5) and then take the intersection of those records.

3.4 Indexes on Multiple Keys

- All of the mentioned methods will eventually give the same set of records as the result.
- However, the number of individual records which meet one of the specified conditions (either department_id= 1 or gpa= 3.5) are larger than the records that satisfy both conditions (department_id= 1 and gpa= 3.5).
- Hence, none of the above three methods is efficient for searching records we required.
- Having a multiple key index on department_id and gpa would be more efficient in this case, because we can search for the records which meets given requirements just by accessing the index file.
- We refer to keys containing multiple attributes as **composite keys**.

3.4 Indexes on Multiple Keys

- **Ordered Index on Multiple Attributes**
  - We can create a key field for previously discussed file as <department_id,gpa>.
  - Search key is also a pair of values. For the previous example this will be <1,3.5>
  - In general, if an index is created on attributes $<A_1,A_2,A_3 \ldots. ,A_n>$, the search key values are tuples with n values ; $<v_1,v_2,v_3 \ldots. ,v_n>$.
  - A lexicographic (alphabetical) ordering of these tuple values establishes an order on this composite search keys.
  - For example, all the composite keys with 1 for department_id will precede those for department_id 2.
  - When the department_id is the same, the composite keys will be sorted in ascending order of the gpa.

3.4 Indexes on Multiple Keys

- **Partitioned Hashing**
  - Partitioned hashing is an extension of static external hashing (when a search-key value is provided, the hash function always computes the same address) which allows access on multiple keys.
  - This is suitable only for equality comparisons. It doesn’t support range queries.
  - For a key consisting *n* attributes, *n* separate hash addresses are generated. The **bucket address** is a concatenation of these *n* addresses.
  - Then it is possible to search for composite key by looking up the appropriate buckets that match the parts of the address in which we are interested.

3.4 Indexes on Multiple Keys

- **Partitioned Hashing**
  - For example, consider the composite search key <department_id,gpa>
  - If *department_id* and *gpa* are hashed into 2-bit and 6-bit address respectively, we get an 8 bit bucket address.
  - If department_id = 1 hashed to **01** and gpa = 3.5 hashed to **100011** then the bucket address is **01100011.**
  - To search for students with 3.5 gpa, we can search for buckets **00100011** , **01100011**, **10100011**, **11100011**

3.4 Indexes on Multiple Keys

- **Partitioned Hashing**
  - Advantages of partitioned hashing:
    1. Ease of extending for any number of attributes.
    2. Ability to design the bucket addresses in a way that frequently accessed attributes get higher-order bits in the address. (Higher-order bits are the left most bits)
    3. There is no need to maintain a separate access structure for individual attributes.

3.4 Indexes on Multiple Keys

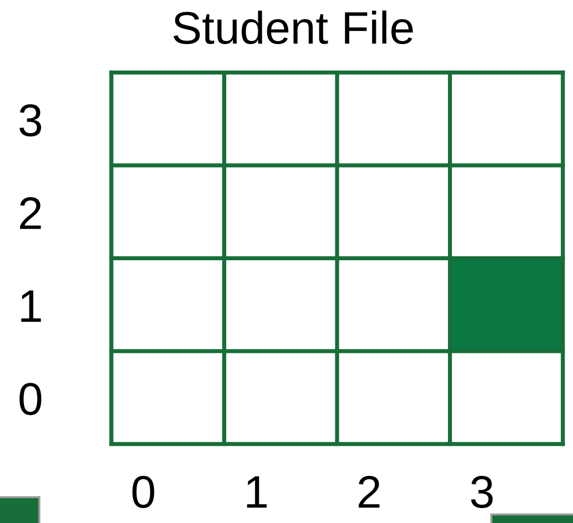
- **Partitioned Hashing**
  - Disadvantages of partitioned hashing:
    1. Inability to handle range queries on any of the component attributes.
    2. Most of the time, records are not maintained by the order of the key which was used for the hash function. Hence, using lexicographic order of combination of attributes as a key (eg: <department_id,gpa>) to access the records would not be straightforward or efficient.

3.4 Indexes on Multiple Keys

- **Grid Files**
  - Constructed using a grid array with one linear scale (or dimension) for each of the search attributes.
  - For the previous example of students file, we can construct a linear scale for department_id and another for gpa.
  - These linear scales are created to preserve the uniform distribution of that particular attributes that are considered as index.
  - Each cell points to some bucket address where the records corresponding to that cell are stored.

3.4 Indexes on Multiple Keys

Following illustration shows a grid array for the Student file with one linear scale for department_id and another for the gpa attribute



department_id	Linear scale
0	0
1	**1**
2	2
3	3

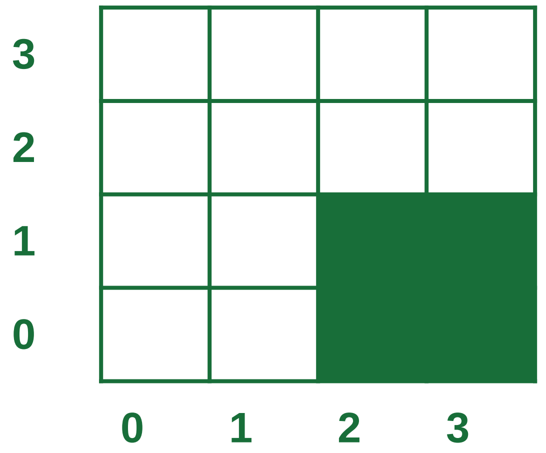
Linear scale for department_id

gpa	Linear scale
< 0.9	0
1.0 - 1.9	1
2.0 - 2.9	2
> 3.0	**3**

Linear scale for gpa

3.4 Indexes on Multiple Keys

- **Grid Files**
  - When we query for deparment_id = 1 and gpa =3.5, it maps to cell (1,3) as highlighted in the previous slide.
  - Records for this combination can be found in the corresponding bucket.
  - Due to nature of this indexing, we can perform range queries.
  - As an example, for range query gpa > 2.0 and department_id < 2 , following bucket pool can be selected.



3.4 Indexes on Multiple Keys

- **Grid Files**
  - Grid files can be applied to any number of search keys.
  - If we have ***n*** number of search keys, we'll get a grid array of ***n*** dimensions.
  - Hence it is possible to partition the file along the dimensions of the search key attributes.
  - Thus, grid files provide an access by combinations of values along dimensions of grid array.
  - Space overhead and additional maintenance cost for reorganization of the dynamic files are some drawbacks of grid files.

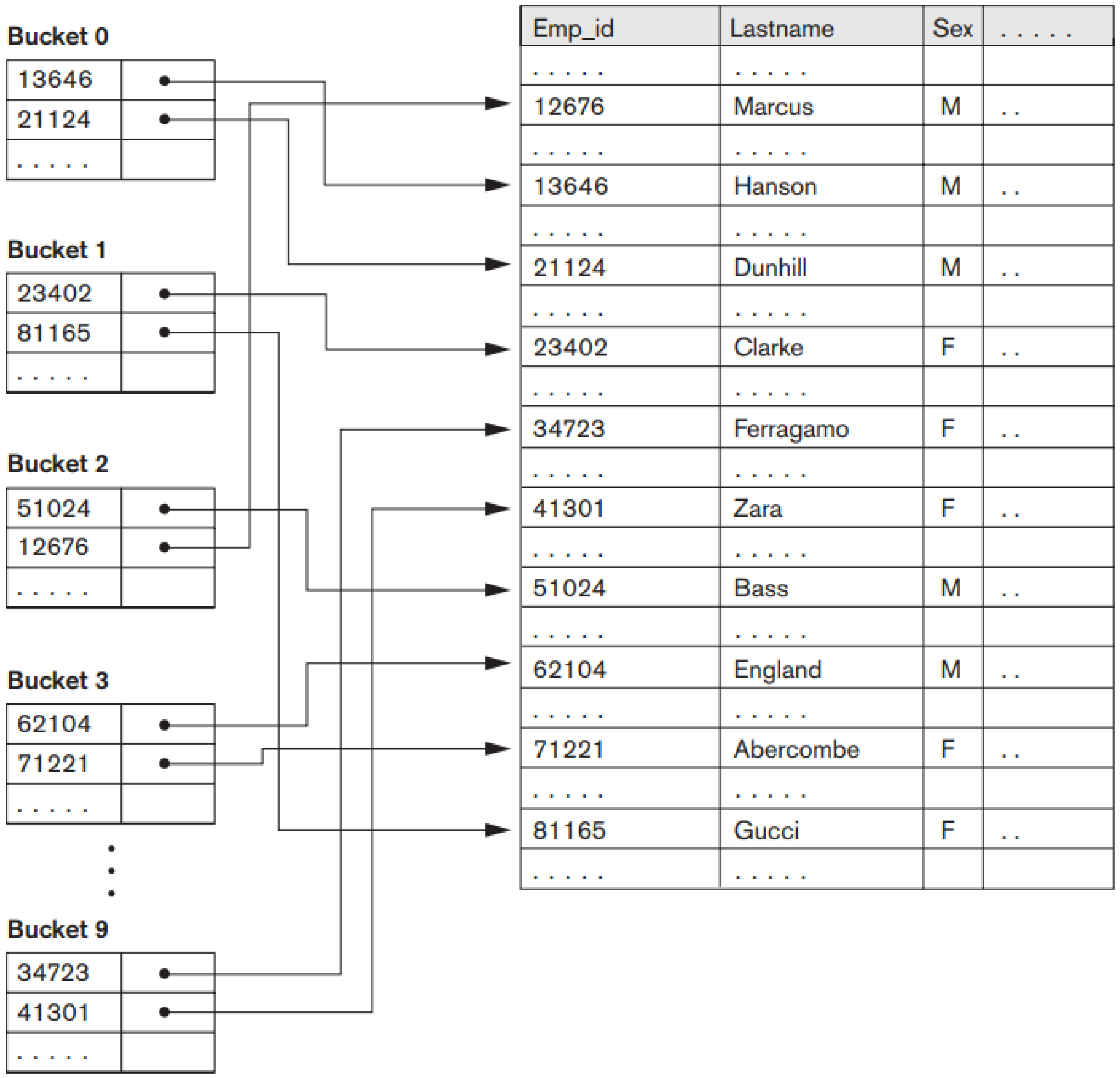
3.5 Other types of Indexes

- **Hash Indexes**
  - The hash index is a secondary structure that allows access to the file using hashing.
  - The search key is defined on an attribute except the one used for organizing the primary data file.
  - Index entries consist of the hashed key and the pointer to the record which is corresponding to the key.
  - The index files with hash index could be arranged as dynamically expandable hash file.

3.5 Other types of Indexes

- **Hash Indexes**

Hash-based indexing.



Emp_id	Lastname	Sex	. . . . .
. . . . .	. . . . .		
12676	Marcus	M	. .
. . . . .	. . . . .		
13646	Hanson	M	. .
. . . . .	. . . . .		
21124	Dunhill	M	. .
. . . . .	. . . . .		
23402	Clarke	F	. .
. . . . .	. . . . .		
34723	Ferragamo	F	. .
. . . . .	. . . . .		
41301	Zara	F	. .
. . . . .	. . . . .		
51024	Bass	M	. .
. . . . .	. . . . .		
62104	England	M	. .
. . . . .	. . . . .		
71221	Abercombe	F	. .
. . . . .	. . . . .		
81165	Gucci	F	. .
. . . . .	. . . . .		

3.5 Other types of Indexes

- **Bitmap Indexes**
  - Bitmap index is commonly used for querying on multiple keys.
  - Generally, this is used for relations which are consist of large number of rows.
  - Bitmap index can be created for every value or range of values in single or multiple columns.
  - However, those columns used to create bitmap index have quite less number of unique values.

3.5 Other types of Indexes

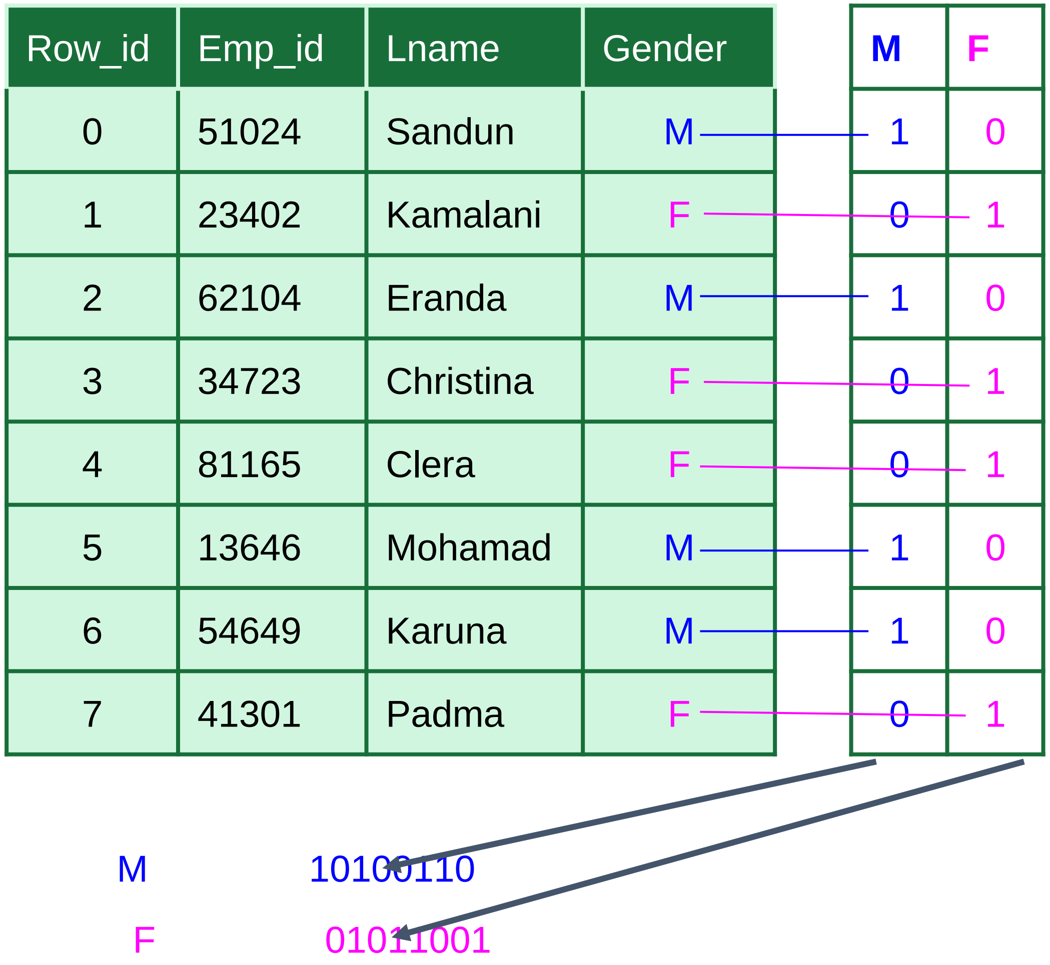
- **Bitmap Indexes**

  - Consider we are creating a bitmap index on column C, for a particular value V and we have *n* number of records.
  - Therefore, the index contains *n* number of bits.
  - For a given record with record number *i*, if that record has the value V in column C, the i^{th} bit will be given 1, otherwise it will be 0.

3.5 Other types of Indexes

- **Bitmap Indexes**

- In the given table we have a column for record the gender of the employee.
- The bitmap index for the values are an array of bits as shown.



Row_id	Emp_id	Lname	Gender
0	51024	Sandun	M
1	23402	Kamalani	F
2	62104	Eranda	M
3	34723	Christina	F
4	81165	Clera	F
5	13646	Mohamad	M
6	54649	Karuna	M
7	41301	Padma	F

M	F
1	0
0	1
1	0
0	1
0	1
1	0
1	0
0	1

M	10100110
F	01011001

3.5 Other types of Indexes

- **Bitmap Indexes**
- According to the example given in the previous slide,
  - If we consider value F in column gender, 1st, 3rd, 4th and 7th bits are marked as “1” because record ids of 1,3,4, and 7 have value F init. But the record ids of 0,2,5 and 6 set to “0”.
  - Bitmap index is created on a set of records which are numbered from 0 to *n* with a record id or row id that can be mapped to a physical address.
  - This physical address is created with block number and record offset within the block.

3.5 Other types of Indexes

- **Function based indexing**
  - This methods was introduced by commercial DBMS products like Oracle relational DBMS.
  - In function based indexing, a function is applied on one or more columns and the resulting value is the key to the index.
  - For an example, we can create an index on uppercase of the last_name field as follows;

```
CREATE INDEX upper_lname

ON Employee (UPPER(Lname));
```

  - "UPPER" function is applied on "Lname" field to create index called "upper_lname".

3.5 Other types of Indexes

- **Function based indexing**
  - If we apply following query, DBMS will use the index created on last_name rather than searching the entire table.

```
SELECT Emp_id, Lname
FROM Employee
WHERE UPPER(last_name)= "Sandun"
```

3.6 Index Creation and Tuning

- **Index Creation**
  - An index is not an essential part of a data file. However, we can create and remove index dynamically.
  - Usually, index is known as access structures. We can create index based on the frequently used search requirements.
  - The physical ordering of the data file is disregarded by creating a secondary index.
  - Secondary index can be created in conjunction with virtually any primary record organization.
  - Secondary index can be used in addition to the primary index such as ordering, hashing or mixed files.

3.6 Index Creation and Tuning

- **Index Creation**
  - Following command is a general way of creating an index in RDBMS;

    CREATE [UNIQUE] INDEX <index name>

    ON <table name> (<column name> [<order>] { , <column name> [<order>] })

    [CLUSTER] ;

  - Keywords in green square brackets are optional.
  - [Cluster] → sort records in the datafile on the indexing attribute.
  - <order> → ASC/DESC (default- ASC)

3.6 Index Creation and Tuning

- **Tuning Indexes**
  - The indexes that we have created, may require modifications due to following reasons,
    1. Long run time of the queries due to deficiency of an index.
    2. Index may not get utilized.
    3. Attributes that are used to create the index might subject to frequent changes.
  - DBMS provide options to view the execution order of the queries. The indexes used, number of disk accesses are include in this view and it is known as query plan.
  - With the query plan, we can identify if the above problems are taking place and hence update or remove index accordingly.

3.6 Index Creation and Tuning

- **Tuning Indexes**
  - Database tuning takes place with the goal of meeting best overall performance. The requirements are dynamically evaluated and the organization of the index and files are changed accordingly.
  - Change nonclustered index into a clustered index or change clustered index into a nonclustered index , creating or dropping index are some ways of improving performance.
  - Rebuild operation of index might help to improve the performance by claiming the wasted space due to many deletions.

3.7 Physical Database Design in Relational Databases

- **Analyzing the Database queries and transactions**
  - Before design the physical structure, we should have a thorough idea of intended use of the database and abstract knowledge about the queries that will be used.
  - Physical design of a database should provide the appropriate structure to store data and at the same time it should facilitate better performance.
  - The mix of queries, transactions and applications that are expected to be run on the database are some factors that database designer should consider before design the physical structure.
  - Let’s discuss about each factor in detail.

3.7 Physical Database Design in Relational Databases

- **Analyzing the Database queries and transactions**

  - For **retrieval query,** the information given below will be important
    1. The relations that will be access by the query
    2. The attributes specified for the selection condition
    3. Type of selection condition (equal, unequal, range etc)
    4. Attributes help in linking multiple tables (join conditions)
    5. Attributes retrieved by the query
  - ii and iv are candidates for index creation.

3.7 Physical Database Design in Relational Databases

- **Analyzing the Database queries and transactions**

  - When it comes to the **update operation** or **update transaction**, we should consider,
    i. Files subject to update
    ii. Whether it is an insert, delete or update operation
    iii. Which attributes are specified in the selection condition, to update or delete.
    iv. Attributes whose values are subject to change by the update query.
  - Attributes in iii are useful when creating an index.

3.7 Physical Database Design in Relational Databases

- **Analyzing the Expected Frequency of Invocation of Queries and Transactions**
  - We must consider how frequently we expect to call/ invoke a particular query.
  - An aggregated list of expected frequencies for all the queries and transactions along with their attributes is prepared.

3.7 Physical Database Design in Relational Databases

- **Analyzing the Time Constraints of Queries and Transactions.**
    - Some queries and transactions have rigid time constraints. For an example, if we take a stock exchange system, some of the queries required to be completed within milliseconds.
    - Generally, primary access structures provides the most effective way of locating a record in a file. Hence, selection attributes in queries with time constraints should be given a high priority when creating primary access structures.

3.7 Physical Database Design in Relational Databases

- **Analyzing the Expected Frequency of the update queries**

  - Updating the access paths for a record itself slow down the operations. Therefore, least amount of access paths should be specified for the file that are subject to frequent updates.

3.7 Physical Database Design in Relational Databases

- **Analyzing the Uniqueness constraint on attributes**
  - Primary key of a file or the unique attributes of a file that are candidate keys, should have access paths defined.
  - Having index (or the access path) defined will make it easy to search on the index when checking for uniqueness.
  - This will help to check the uniqueness when inserting new records because if the value is already exist, database will reject that record since it violates the uniqueness.

3.7 Physical Database Design in Relational Databases

- **Design Decisions about indexing**
  - **Whether to index an attribute:**
  - In general, indexes are created on the attributes which are used as the unique key of the file or the attributes which are used in selection conditions or in join conditions in queries.
  - Multiple indexes are defined to process operation just by scanning the index rather than accessing data files.

3.7 Physical Database Design in Relational Databases

- **Design Decisions about indexing**
  - **What attribute or attributes to index on:**
  - An index can be defined on a single attribute or it could be a composite index created on multiple attributes. In composite index, the order of the attributes should be match with their order in the respective queries.

    Ex: If we have a composite key with (department, subject) it assumes the queries are based on subjects within a department.

3.7 Physical Database Design in Relational Databases

- **Design Decisions about indexing**
  - **Whether to set up a clustered index:**
  - We cannot have both primary and clustering index on the same file because the data file is physically ordered accordingly in both scenarios. We can apply clustered index, if it supports answer the queries just by accessing index. Otherwise, there is no use of making a clustered index. If multiple queries require clustering on different attributes, we should evaluate the gain of each and decide on which attribute to use.
  - **Whether to use dynamic hashing for the file:**
  - Dynamic hashing would be suitable for files which are subject to frequent expansion and shrinking.

Activity

1. What are the types of single level ordered indexes?
   a. _________
   b. _________
   c. _________

Activity

Fill in the blanks

1. A file can have _____ physical ordering field.
2. Primary, Clustering and Secondary index are types of _______ level _____ indexes.
3. ________ search is possible on the index field since it has ________ values.
4. Indexing access structure is established on ______ ____.
5. Index file is usually _____ than the datafile.

Activity

Mark whether the given statement is true false.

1. An unordered file which consists two fields and limited length records is known as a primary index file. (t/f)
2. for a given block in an ordered data file, the first record in that block is known as anchor record. (t/f)
3. Indexes that contains index entries for some records in the data file referred as non - dense index. (t/f)
4. Ordering key field of the index file and the primary key of the data file have same data type. (t/f)
5. In primary index, index file contains one index entry(a.k.a index record) for each record in the data file.(t/f)

Activity

1. You have a file with 600,000 records (r), which is ordered by its key field and each record of this file is fixed length and unspanned. Record length (R) is 100 bytes and block size(B) is 4096.
   a. What is the blocking factor?
   b. How many blocks required to store this file?
   c. Calculate the number of block accesses required when performing a binary search on this file and access data.

Activity

1. You have a file with 400,000 records (r), which is ordered by its key field and each record of this file is fixed length and unspanned. Record length (R) is 100 bytes and block size(B) is 4096.
   a. What is the blocking factor?
   b. How many blocks required to store this file?
   c. Calculate the number of block accesses required when performing a binary search on this file and access data.

Activity

1. You have a file with 400,000 records (r), which is ordered by its key field and each record of this file is fixed length and unspanned. Record length (R) is 100 bytes and block size(B) is 4096. If you have created a primary index file with 9 bytes long ordering key field (v) and 6 bytes long block pointer (p),
   a. What is the blocking factor for index ?
   b. How many blocks required for the index file?
   c. Calculate the number of block accesses required when performing a binary search on index fileand access data.

Activity

1. A data file with 400,000 records (r) is ordered by a non-key field called “product_category”. The product_category field has 750 distinct values. Record length (R) is 100 bytes and block size(B) is 4096. If you have created a primary index file on this non-key field with 9 bytes long ordering key field (v) and 6 bytes long block pointer (p),
   a. What is the blocking factor for index ?
   b. How many blocks required to store the index file?
   c. Calculate the number of block accesses required when performing a binary search on index file and access data.

Activity

1. Assume we search for a non-ordering key field V = 9 bytes long,in a file with 600,000 records with a fixed length of 100 bytes. And given block size B =8,192 bytes.
   a. What is the blocking factor?
   b. What is the required number of blocks?
   c. How many block accesses required for a linear search?

Activity

1. Assume we create a secondary index on non-ordering key field V = 9 bytes long, with entries for the block pointers P=6, in a file with 600,000 records with a fixed length of 100 bytes. And given block size B =8,192 bytes.
   a. What is the blocking factor for index ?
   b. How many blocks required to store the index file?
   c. Calculate the number of block accesses required when performing a binary search on index file and access data.

Activity

1. Assume we have a file with multi-level indexing. In the first level, number of blocks b_1 = 1099 and blocking factor (bfr_i) = 273.
   a. Calculate the number of blocks required for second level.
   b. Calculate the number of blocks required for third level.
   c. what is the top level index(*t*)?
   d. How many block accesses required to access a record using this multi-level index?