# MODULE 3

## Question 1: Differentiate DDL and DML with suitable example.

### Data Definition Language (DDL)

DDL defines or modifies data structures and schemas in a database. Examples include:

- **CREATE**: Creates new tables or databases.
- **ALTER**: Modifies an existing database object.
- **DROP**: Deletes tables or databases.

CREATE TABLE Courses (CourseID int, CourseName varchar(255), Credits int);

ALTER TABLE Courses ADD Department varchar(255);

DROP TABLE Courses;

### Data Manipulation Language (DML)

DML is used for managing data within existing structures. It includes:

- **INSERT**: Adds new records.
- **UPDATE**: Modifies existing records.
- **DELETE**: Removes records.
- **SELECT**: Retrieves data.

INSERT INTO Courses (CourseID, CourseName, Credits) VALUES (101, 'Database Systems', 4);

UPDATE Courses SET Credits = 3 WHERE CourseID = 101;

DELETE FROM Courses WHERE CourseID = 101;

SELECT * FROM Courses;

## Question 2: Differentiate DELETE and DROP command.

The **DELETE** command removes specific rows from a table, while the **DROP** command completely removes an entire table or database schema along with its data.

## Question 3: How many ways INSERT is possible.

1. **Inserting a Single Row:** Adds one row of data by specifying each column's value.

   INSERT INTO Employee (EmployeeID, Name, Department) VALUES (1, 'John Doe', 'Finance');

2. **Inserting Multiple Rows**: Adds multiple rows in a single SQL command by specifying values for each row.

   INSERT INTO Employee (EmployeeID, Name, Department)
   VALUES (2, 'Jane Smith', 'HR'),
   (3, 'Alice Johnson', 'IT'),
   (4, 'Mike Brown', 'Marketing');

3. **Inserting Results of a Query**: Adds rows from the result of another SQL select query (assuming there's another table with relevant data).

   INSERT INTO Employee (EmployeeID, Name, Department) SELECT StaffID, StaffName, StaffDept FROM TempStaff WHERE Department = 'HR';

4. **Inserting Default Values**: Inserts a row with default values for all columns (assuming defaults are set in the table schema).

   INSERT INTO Employee DEFAULT VALUES;

5. **Bulk Insert Using Database Tools** such as MySQL's `LOAD DATA INFILE from xml/csy/excel/textfile`:

```
LOAD DATA INFILE '/path_to_file/employees.csv' INTO TABLE Employee
FIELDS TERMINATED BY ',' LINES TERMINATED BY '\n'
(EmployeeID, Name, Department);
```

# Question 4: what are Nested queries (correlated and non-correlated), explain with examples ?

Nested queries, also known as subqueries, are SQL queries placed inside another SQL query. There are two main types of nested queries: **correlated** and **non-correlated**.

**Non-Correlated Subqueries**: The inner query can run independently of the outer query. It's executed once and its result is used by the outer query.

**Example**: Find employees whose salary is above the average salary.

```
SELECT Name
FROM Employee
WHERE Salary > (SELECT AVG(Salary) FROM Employee);
```

**Correlated Subqueries**: The inner query depends on data from the outer query, running once for each row processed by the outer query.

**Example**: List employees who earn the highest salary in their department.

```
SELECT Name FROM Employee e1
WHERE Salary = (
   SELECT MAX(Salary)
   FROM Employee e2
   WHERE e1.Department = e2.Department);
```

# Question 5: what is Aggregation and Grouping in SQL?

**Aggregation in SQL**

Aggregation uses functions to compute a single result from a set of rows. Common functions include `SUM()`, `AVG()`, `MAX()`, `MIN()`, and `COUNT()`.

**Example**: Calculate the total and average salaries of all employees.

```
SELECT SUM(Salary) AS TotalSalary, AVG(Salary) AS AverageSalary
FROM Employee;
```

.

**Grouping in SQL**

Grouping organizes rows into groups with the `GROUP BY` clause, often used with aggregation to perform calculations on each group.

**Example**: Find the average salary by **each** department.

```
SELECT Department, AVG(Salary) AS AverageSalary
FROM Employee
GROUP BY Department;
```

# Question 6: Differentiate where and having clause, explain with simple example ?

**WHERE Clause**

The **WHERE** clause is used to filter rows before grouping or aggregating data. It applies conditions to individual records in a database.

**Example**:

```
SELECT Name, Department
FROM Employee
WHERE Department = 'Marketing';
```

This filters employees who work in the Marketing department.

**HAVING Clause**

The **HAVING** clause is used to filter data after grouping or aggregating. It applies conditions to groups formed by the **GROUP BY** clause, typically involving an aggregate function.

**Example**:

```
SELECT Department, AVG(Salary) AS AvgSalary
FROM Employee
GROUP BY Department
HAVING AVG(Salary) > 50000;
```

# Question 7: What is the importance of views in sql? Explain with suitable example ?

A **view** in SQL is a **virtual table** based on the result-set of an SQL statement. It contains rows and columns, just like a real table. A view can be created from a single table or multiple tables.

- Updates in the view will reflect in parent table, and vice-versa, if view is created from single parent table.
- If parent table is deleted, view is automatically deleted.
- All DML applicable to parent table are also applicable to view

**Importance of Views in SQL**

1. **Simplifying Complex Queries**: They allow users to perform complex queries easily.

2. **Enhancing Security**: Views can restrict access to specific data, improving database security.
3. **Data Abstraction**: Views provide a way to change the database schema without affecting how users see the data.
4. **Logical Data Independence**: They protect users from changes in the database structure.

### Creating the View

You can create a view called `StudentMajors` that includes just the `Name` and `Major` of each student:

```
CREATE VIEW StudentMajors AS
SELECT Name, Major
FROM Students;
```

Query View : SELECT * FROM StudentMajors;

# Question 8: With the help of an example, illustrate the use of SQL TRIGGER

## What is an SQL Trigger?

An SQL trigger is a set of SQL statements that automatically "fires" or executes when a specific database **event** occurs, such as an **update, insert, or delete** operation on a specified table.

## Why It's Used:

Triggers are used to automatically perform a procedure or enforce rules at the database level, often for maintaining data integrity and enforcing business logic.

## Example:

Suppose we want to automatically update a column `LastModified` in the `Employee` table to record when a row was last updated.

### Example of Trigger Usage:

`Update PCM automatically while inserting maks`

```
CREATE TRIGGER UpdateTotalScore
BEFORE INSERT ON Students
FOR EACH ROW
SET PCM = (Physics + Chemistry + Maths));
```

### Invoke trigger

INSERT INTO Students ( RollNo, Name, Physics, Chemistry, Maths) VAUES (10, ‘Alice’ 50,60,70);

The event in trigger is **BEFORE INSERT**.
So on insert, the trigger will automatically fire and updates PCM from the trigger.

# Question 9: Illustrate use of assertions with an example

## Assertion in SQL:

An assertion is a database **constraint** that ensures specific conditions are always met across tables or the entire database, enforcing data integrity and business rules.

Assertion will execute always whenever update/insert/delete on the table specified in assertion definition.

## Why It's Used:

Assertions are used to enforce global constraints that involve multiple tables or the entire database, ensuring consistent data integrity and enforcing business rules beyond what local constraints can achieve.

## How It's Used:

salary of Manager in staff table must not be less than Project Manager

CREATE ASSERTION salary_staff
CHECK( NOT EXISTS
(SELECT * FROM staff WHERE position='MANAGER' AND
salary<(SELECT salary FROM staff WHERE position ='Project Manager')));

### How assertion differ from trigger

They differ from triggers in that they are declarative constraints, stating what should always be true, while triggers are procedural code that execute in response to specific events.

# Question 10: Define terms:
# physical and logical records,
# blocking factor,
# pinned and unpinned organization.
# Heap files, Indexing

1. **Physical and Logical Records**:
   **Physical Record**: Represents data stored on a disk. Example: a block of 4096 bytes.
   **Logical Record**: Represents a single entry in a database. Example: a row in a table.
2. **Blocking Factor (bfr)**: Number of logical records stored in a single physical block. Let block sie = B, and record size = R, bfr = floor function (B/R)
3. **Pinned and Unpinned Organization**:

   **Pinned Organization**: Records stored in fixed order. Example: an alphabetically sorted list of names.
   **Unpinned Organization**: Records not stored in a fixed order. Example: records in a heap file.

4. **Heap Files**: Unsorted files where records are appended. Example: a log file where new entries are added at the end.
5. **Indexing**:Technique to speed up searches by creating a structure (index) with key-value pairs. Example: an index on a book's chapters for quick reference
6. **Spanned and Unspanned**

   **Spanned:** Data records that can span multiple physical blocks. Example: A large record that has record size > block size that is B > R.

   **Unspanned**: Data records that fit within a single physical block. Example: A small record that fits entirely within one block that is has record size < block size that is B < R.

# Question 11: What are different types of indexing:

TYPES OF INDEXING are

1) Single level ordered indexes
   - 1.1) Primary Index
   - 1.2) Clustering index
   - 1.3) Secondary index
2) Multilevel index
3) Dynamic Multilevel Indexes Using B-Trees

**Primary Index:**

**data file**

- **O**rdered (sorted) file on primary key
- no duplicate key, as indexing on primary key
- records are saved in blocks in physical memory

**index file**

- index file has two fields 1. key value, 2. block pointer
- key value holds primary key of first record of the block
- block ointer points to the block on which key is saved.
- **sparse** file organization, since index file **does not** save all primary-keys becaue of sorted data file.
- number of records in **index file** = number of **blocks** in **data file**

*Draw diagram from notes.*

Efficient for point queries: select * from EMP where ssn = ‘100-1000101’;

**Clustering Index:**

**data file**

- **O**rdered (sorted) file on **non-primary** key
- **can have duplicate key,** as indexing on non-primary key
- records are saved in blocks in physical memory

**index file**

- index file has two fields 1. key value, 2. block pointer
- key value holds non-primary key
- block ointer points to the block on which the first occurance of key is saved.
- **dense** file organization, index file saves all non-primary keys without duplication.

*Draw diagram from notes*

Beneficial for range queries and table scans: select * from EMP where dno= 5;

**Secondary Index:**

**data file**

- **Not-O**rdered (sorted) file on **non-key** key , or CK
- **can have duplicate key,** as indexing on non-key,
- no duplicate if indexing on CK
- records are saved in blocks in physical memory

**index file**

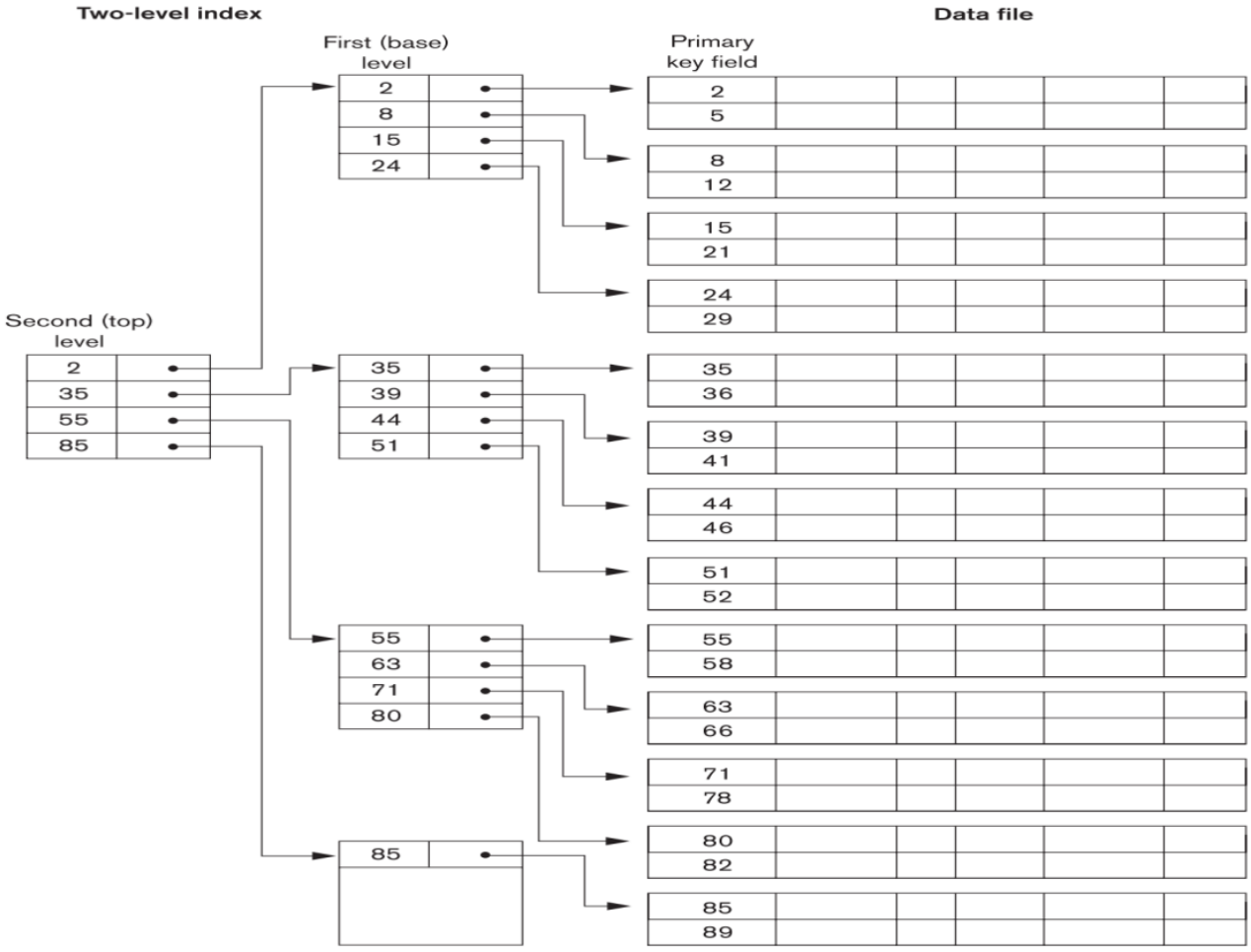
- index file has two fields 1. key value, 2. block pointer
- key value holds non-key or CK as per indeing scheme.
- block ointer points to the block on which non-key is saved.
- **dense** file organization, index file saves all non-keys or CKs
- number of records in **index file** = number of **records** in **data file**

*Draw diagram from notes*

# Question 12: What is Multi level indexing ?

**What are the advantage ?**

**hen do you prefer multilevel index over single level index?**




For the database is very large, single indexing will lead to

1. increased disk I/O,
2. slower query performance

So need multi level indexing for efficient file organization.

**Multi-level indexing**

- A multi-level index is a hierarchical indexing structure where index entries point to another level of indexes.
- Organized in a tree structure
- The first-level index will be ordered (sorted) and pointing to database file
- first-level index can be  either
  - primary index – if database in ordered (sorted) on primary key
    - number of records in **index file** = number of **blocks** in **data file**
  - secondary – if database in not-ordered (not-sorted)
    - number of records in **index file** = number of **records** in **data file**
- The second-level index will be indexing the first level.
  - The second-level indexing will be primary index, as the first level is sorted.
    - number of records in second level **index file** = number of **blocks** needed to store first level index  **file**
- This will repeat like third level index, indexing second-level and so on
- from second level indexing onward, the the indexing scheme will be primary

**Advantage**

Multilevel indexing will reduce the number of block access considerably.

Total number of block access = number of levels + 1

# Question 13: What are different hashing methods ?

**Hashing categorized as  below**

1) Internal
2) External
    2.1. Static
    2.2. Dynamic
        2.2.1. Extendible
        2.2.1. Dynamic

**Internal hashing:**

Data stored in RAM, Hash table has fixed/limited size.

**External hashing:**

Data stored in Hard disk, Hash table size is not fixed.

**<u>1.) Internal Hashing</u>**

M – Hash Table Size – fixed in internal hashing

Hash Function : h = MOD

h(102) = 102%100 = 2

| | eno | ename | dno |
|---|---|---|---|
| 0 | 100 | Alice | 4 |
| 1 | 101 | Bob | 5 |
| 2 | 102 | Cindy | 6 |
| | | | 4 |
| M-2 | 198 | Sam | 5 |
| M-1 | 199 | Eric | 4 |

**Collision Resolution in internal Hashing**

**Address Space**

| | eno | ename | dno | Overflow buffers pointer |
|---|---|---|---|---|
| 0 | 100 | Alice | 4 | |
| 1 | 101 | Bob | 5 | M |
| | 102 | Cindy | 6 | |
| M-2 | | | 4 | |
| M-1 | 199 | Sam | 5 | |

Now if a new record comes as below,
201 David 5

H(201) = 201%100 = 1, index 1 is already filled in address space, so allocate above record in overflow buffer at available free slot, say at M. Keep pointer of M at address space at index 1

**Overflow Buffer**

| | eno | ename | dno | Overflow buffers |
|---|---|---|---|---|
| M | 201 | David | 5 | |
| M+1 | | | | |
| M+2 | | | | |
| | | | | |
| N | | | | |

**2.) Static Hashing**

Hash table size is fixed.
Records of database are stored in blocks.
Uses the concept of Bucket.
Bucket contain single block or mora than one contiguous blocks.

**Limitation**

M – Number of buckets
m – number of records per bucket
Total number of record = M*m

**underflow**

when total number of records in database < M*n

**overflow**

when total number of records in database > M*n

**Solution:** create overflow buckets as given below.
- Add a record pointer in each bucket, pointing to overflow buffer
- chain overflow buffer as required, else keep record pointer to null

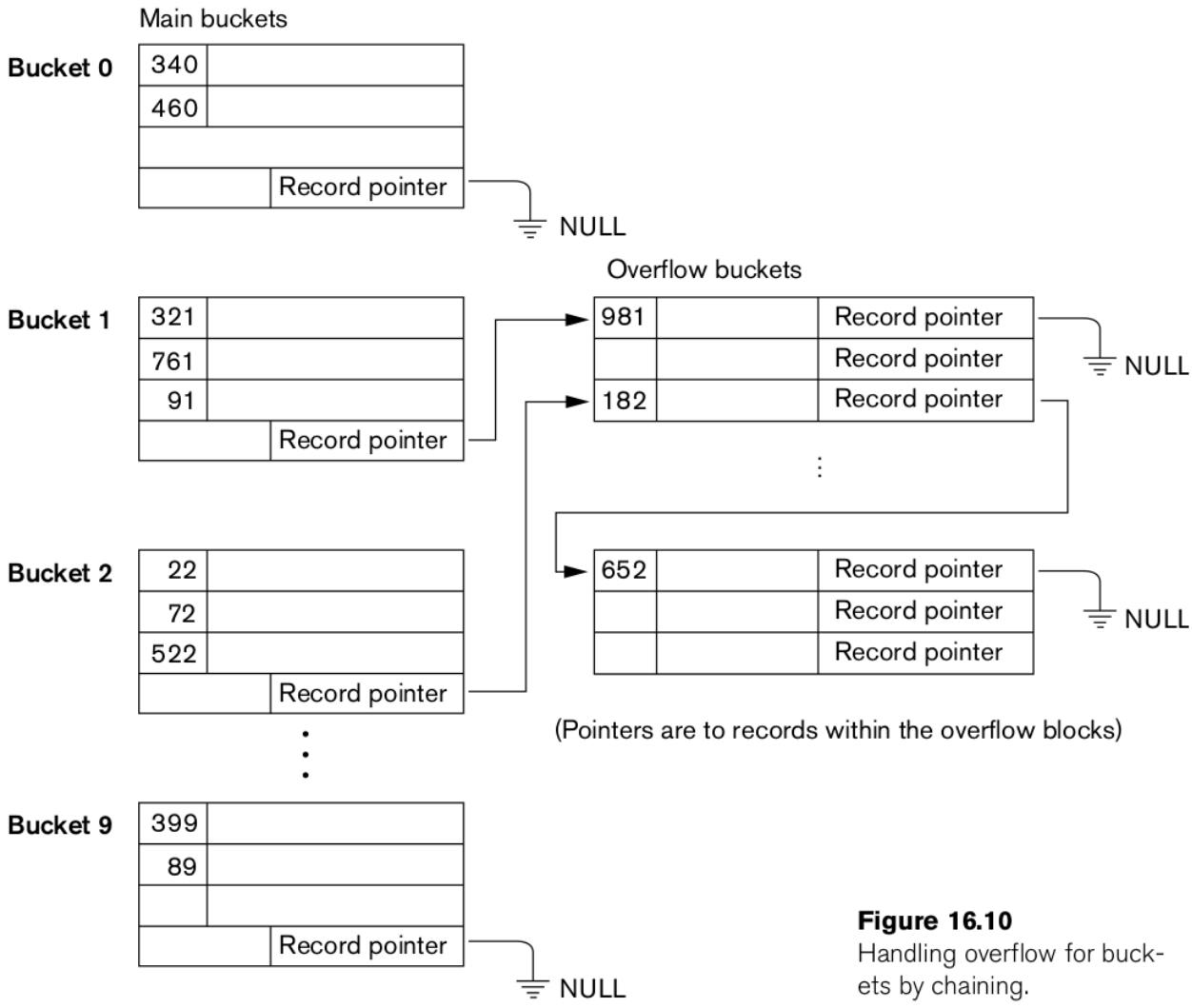



**Figure 16.10**
Handling overflow for buckets by chaining.

**3). Extendible Hashing.**

Hash table is not fixed.

Its dynamic, it can expand on overflow and shrink on underflow

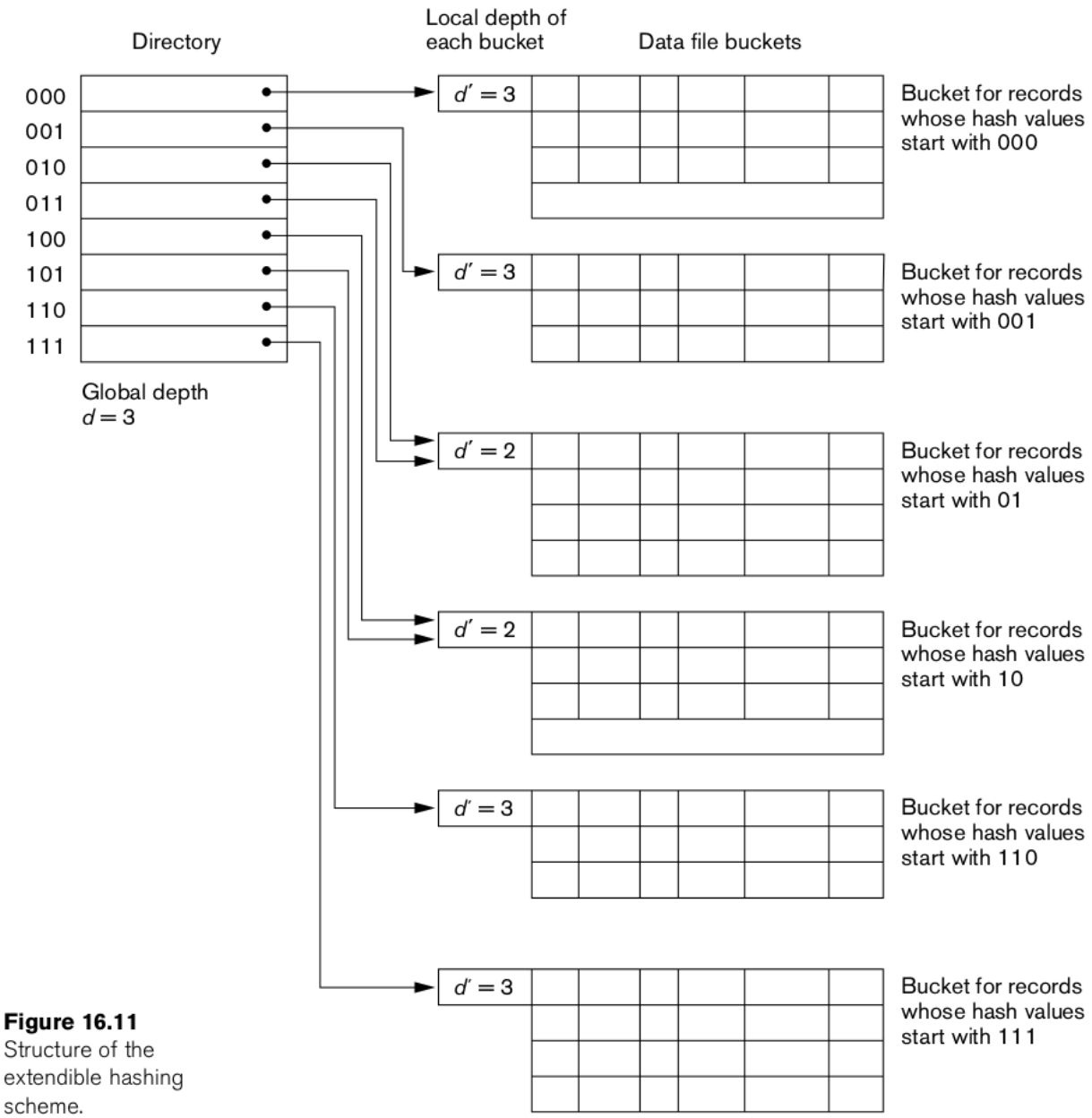



**Figure 16.11**
Structure of the extendible hashing scheme.

**Concepts used are**

1. Directory : an array of $2^d$ bucket addresses—is maintained
   -array elements points to buckets
   -directory index are binary value of $2^d$ where d = global depth
2. global depth - d
   -Number of bucket = $2^d$
3. Local depth – d' (d-dash)
   -Associated with each bucket that controls expansion and shrinking of buckets.
   -The size of d' indicate how many starting bits of directory will be used to point bucket
   eg, d' = 2 , means the first two bits will the starting bits to point bucket.

**Expanding buckets:**

If bucket exceeds bucket size , split the bucket by increasing local depth d' = d' +1

**Shrinking buckets:**

if buckets are under used, combine the buckets , that is shrink by decreasing global depth d' = d'-1

**4). Dynamic Hashing:**

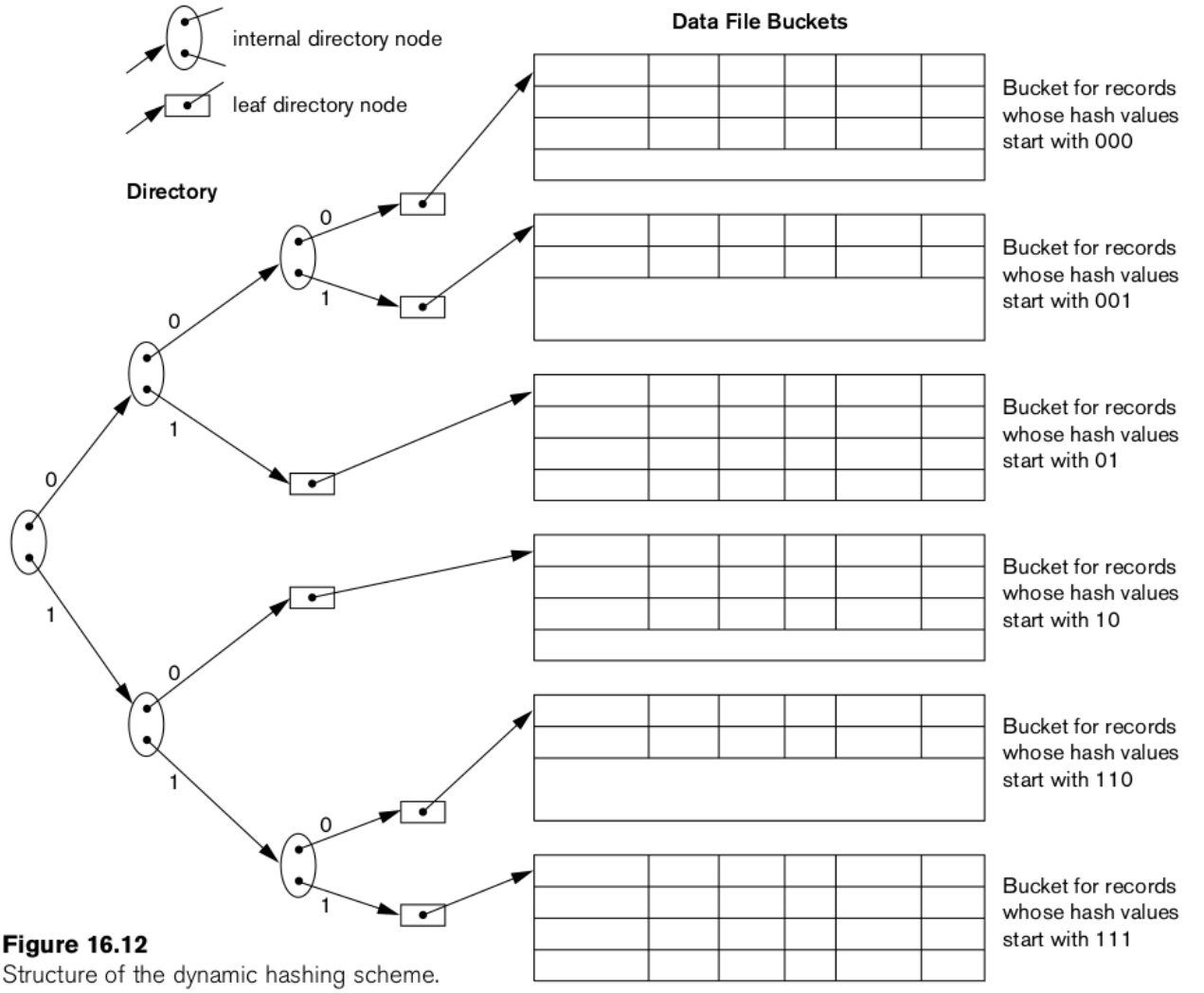



**Figure 16.12**
Structure of the dynamic hashing scheme.

- Use tree like structures
- **Not using** global depth, local depth, directory.
- Leaf nodes are the buckets
- intermediate node has left pointer for 0 and right pointer for 1, simulating the address scheme of directory.

# Question 14: What are GRID files ?

**select * from EMP where age >40 and Dno <= 5;**

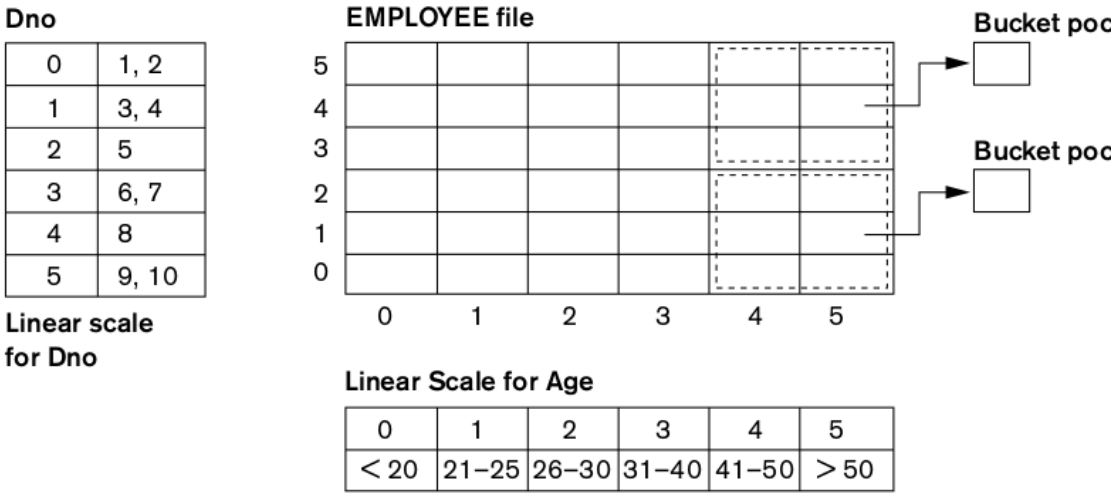
Here we want to access a file on two keys, say Dno and Age.
To process this efficiently, we can construct a grid array.
The cells in GRID represents buckets.

GRID looks like a 2D matrix, hence for indexing the cell (cell[i][j], we have to use linear scale of attributes.

It uses a linear scale of Age and Dno ( our attribute in this example)

- Age is a continuous value, hence use a range in linear scale
- Dno is a finite value, hence use integer values in linear scale

**Refer Diagram below for GRID and Linear Scale**




**Example 1**

**select * from EMP where age >40 and Dno <= 5;**

Ans:

Find linear scale for age >40, this comes index 4 and 5

Find linear scale for Dno <=5, this comes index 0,1,2

→ so it comes to cells on **<(age 4,5), (dno, 0,1,2)>**

**So the data for the above query will be available in grids <(age 4,5), (dno, 0,1,2)>** as marked on rectangle right bottom corner.