

1. For the database system to be usable, it must retrieve data efficiently. The need of efficiency has led designers to use complex data structures to represent data in the database. Developers hide this complexity from the database system users through several levels of abstraction. Explain those levels of abstraction in detail with example.

In a database system, various levels of abstraction are used to simplify interaction and ensure efficient data management. These levels abstract away the complexities of how data is stored and manipulated internally, allowing users and developers to interact with the system more easily. The three primary levels of abstraction are:

1. Physical Level (Storage Level)

Description: The physical level, also known as the storage level, deals with how data is physically stored on the storage medium (e.g., hard drives, SSDs). It includes details about file structures, indexing, and data access methods.

Example: At this level, the database might use a particular file format or storage structure to store data. For instance, a database might use B-trees or hash tables for indexing, manage data blocks or pages, and handle low-level file organization. This level also considers aspects like data compression, encryption, and how data is retrieved from disk storage.

Key Points:

- Physical storage details are not visible to the end-users.
- Concerned with optimizing storage and retrieval operations.
- Examples include file formats, index structures, and data access algorithms.

2. Logical Level (Conceptual Level)

Description: The logical level, or conceptual level, describes what data is stored in the database and the relationships among that data. This level focuses on the structure and organization of data, such as tables, schemas, and relationships between entities, without specifying how data is physically stored.

Example: At this level, a database might be organized into tables (e.g., Customers, Orders, Products) and define relationships between these tables (e.g., foreign key constraints). For instance, a Customer table might have fields such as CustomerID, Name, and Address, while an Orders table might reference CustomerID to indicate which customer made the order.

Key Points:

- Defines the database schema and relationships.
- Abstracts away physical storage details.
- Example components include tables, views, and relationships.

3. View Level (External Level)

Description: The view level, or external level, is the highest level of abstraction. It defines how users interact with the database through various views or interfaces. Each user or user group may see a different subset of the data, tailored to their specific needs or permissions. This level hides the complexity of the underlying data structures and allows users to interact with the data in a way that is relevant to them.

Example: A database system might offer different views or user interfaces for different roles. For instance:

- A sales representative might have a view that only includes customer contact details and their orders.

- An **inventory manager** might have a view that focuses on product stock levels and suppliers.

Each view might present a subset of the database's information or aggregate data in a way that is meaningful for the user, such as generating reports or summaries.

Key Points:

- Provides tailored views of the data to different users or applications.
- Ensures data access is user-friendly and relevant.
- Examples include user-specific dashboards, query results, and custom reports.

Summary

In essence, these levels of abstraction work together to manage data effectively:

- **Physical Level:** Focuses on how data is stored and accessed.
- **Logical Level:** Defines the structure and relationships of the data.
- **View Level:** Provides tailored access to the data based on user needs.

This layered approach simplifies database interaction and improves efficiency by separating concerns, ensuring that users and developers do not need to deal with the complexities of data storage and organization directly.

2. Explain the concept of candidate key and primary key, foreign key. Identify above listed key for the following schema:

Person (driver_id, name, address, contactno)

Car(licence, model, year)

Owns (driver_id, licence)

In relational databases, keys are critical for maintaining data integrity and establishing relationships between tables. Here's a detailed explanation of candidate keys, primary keys, and foreign keys, followed by their identification in the provided schema.

Key Concepts

1. Candidate Key

Definition: A candidate key is a minimal set of attributes (columns) that uniquely identifies each record in a table. There can be multiple candidate keys in a table, and each one can serve as a unique identifier.

Characteristics:

- Each candidate key must be unique for each record in the table.
- A candidate key must be minimal, meaning no subset of the key can uniquely identify records.
- Candidate keys are potential choices for the primary key.

Example: In a table with columns A, B, and C, where:

- A and B together can uniquely identify records.
- B alone can also uniquely identify records.

Both A and B or B alone are candidate keys if they uniquely identify each record.

2. Primary Key

Definition: A primary key is a specific candidate key chosen to uniquely identify records in a table. It is selected from among the candidate keys and must have a unique value for every record.

Characteristics:

- There can only be one primary key per table.
- It enforces uniqueness and ensures that no two records have the same primary key value.
- It cannot contain NULL values.

Example: In a table with columns A, B, and C, if A is chosen as the primary key, A must be unique for all records and cannot be NULL.

3. Foreign Key

Definition: A foreign key is a column or set of columns in one table that refers to the primary key of another table. It establishes a relationship between the two tables, ensuring referential integrity.

Characteristics:

- A foreign key in a child table refers to the primary key in the parent table.
- It may contain duplicate values and NULL values.
- It helps maintain consistent data across related tables.

Example: In a table Orders, if CustomerID is a foreign key that references CustomerID in the Customers table, CustomerID in Orders must match a valid CustomerID in Customers.

Identification in the Given Schema

Consider the schema:

1. **Person** (driver_id, name, address, contactno)
2. **Car** (licence, model, year)
3. **Owns** (driver_id, licence)

1. Candidate Key(s):

- **Person Table:**
 - driver_id is a candidate key because it uniquely identifies each person in the table. Given that driver_id is unique for each person and not a subset of any other key, it is the candidate key.
- **Car Table:**
 - licence is a candidate key because it uniquely identifies each car. Similar to driver_id in the Person table, licence is unique and minimal.
- **Owns Table:**
 - The combination of driver_id and licence is a candidate key because this pair uniquely identifies each record in the Owns table. Neither driver_id nor licence alone can uniquely identify a record in Owns since multiple drivers can own multiple cars, and multiple cars can be owned by the same driver.

2. Primary Key(s):

- **Person Table:**
 - driver_id is chosen as the primary key because it uniquely identifies each record.
- **Car Table:**
 - licence is chosen as the primary key because it uniquely identifies each record.
- **Owns Table:**
 - The primary key is the combination of driver_id and licence. This composite key ensures uniqueness in the Owns table.

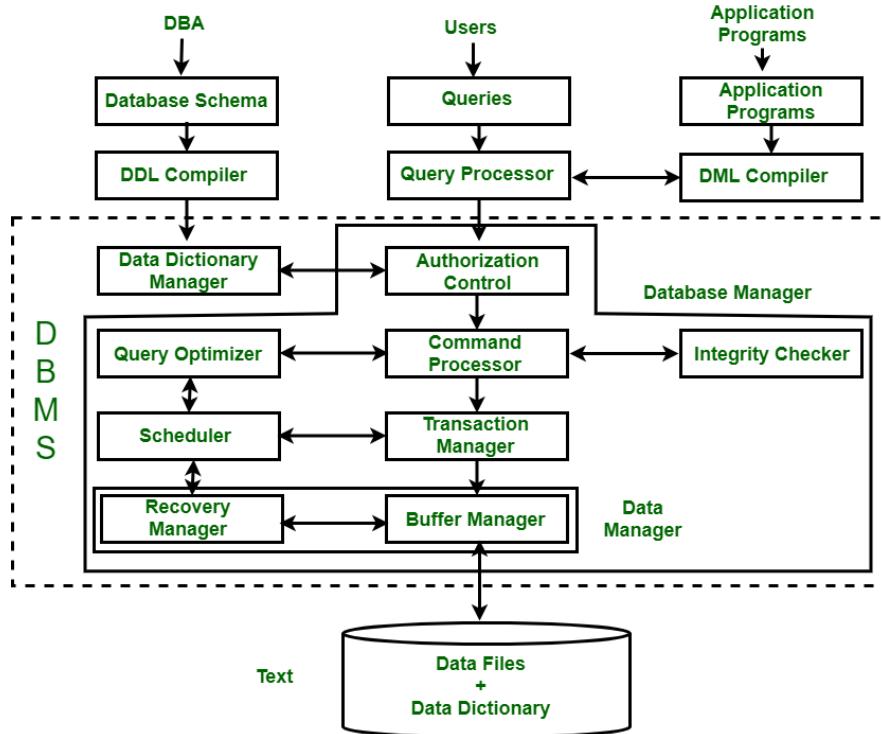
3. Foreign Key(s):

- **Owns Table:**
 - driver_id is a foreign key that references driver_id in the Person table. It establishes a relationship between Owns and Person.
 - licence is a foreign key that references licence in the Car table. It establishes a relationship between Owns and Car.

Summary:

- **Person Table:**
 - Candidate Key: driver_id
 - Primary Key: driver_id
- **Car Table:**
 - Candidate Key: licence
 - Primary Key: licence
- **Owns Table:**
 - Candidate Key: (driver_id, licence)
 - Primary Key: (driver_id, licence)

- Foreign Keys: driver_id (references Person), licence (references Car)
3. Draw architecture of DBMS system and explain function of following components:
- i) Storage manager
 - ii) Query Processor



1. Query Processor: It interprets the requests (queries) received from end user via an application program into instructions. It also executes the user request which is received from the DML compiler. Query Processor contains the following components –

- **DML Compiler:** It processes the DML statements into low level instruction (machine language), so that they can be executed.
- **DDL Interpreter:** It processes the DDL statements into a set of table containing meta data (data about data).
- **Embedded DML Pre-compiler:** It processes DML statements embedded in an application program into procedural calls.
- **Query Optimizer:** It executes the instruction generated by DML Compiler.

2. Storage Manager: Storage Manager is a program that provides an interface between the data stored in the database and the queries received. It is also known as Database Control System. It maintains the consistency and integrity of the database by applying the constraints and executing the DCL statements. It is responsible for updating, storing, deleting, and retrieving data in the database.

It contains the following components –

- **Authorization Manager:** It ensures role-based access control, i.e., checks whether the particular person is privileged to perform the requested operation or not.
- **Integrity Manager:** It checks the integrity constraints when the database is modified.
- **Transaction Manager:** It controls concurrent access by performing the operations in a scheduled way that it receives the transaction. Thus, it ensures that the database remains in the consistent state before and after

the execution of a transaction.

- **File Manager:** It manages the file space and the data structure used to represent information in the database.
- **Buffer Manager:** It is responsible for cache memory and the transfer of data between the secondary storage and main memory.

3. Disk Storage: It contains the following components –

- **Data Files:** It stores the data.
- **Data Dictionary:** It contains the information about the structure of any database object. It is the repository of information that governs the metadata.
- **Indices:** It provides faster retrieval of data item.

The structure of a Database Management System (DBMS) can be divided into three main components: the Internal Level, the Conceptual Level, and the External Level.

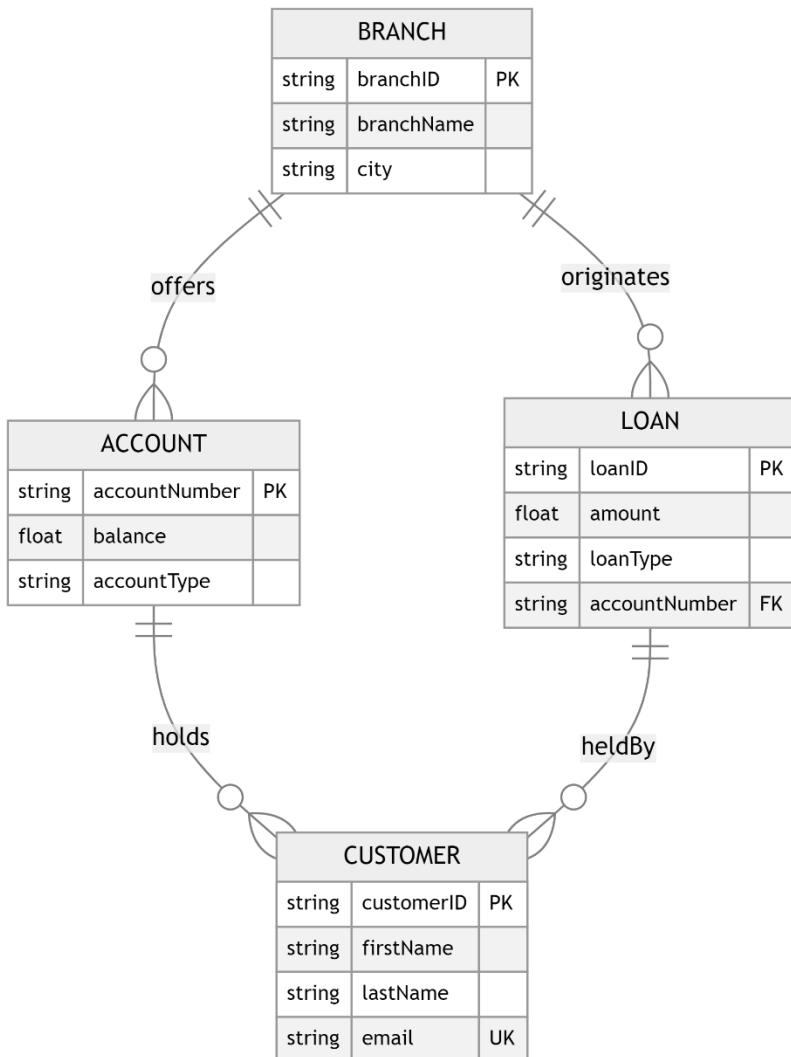
1. Internal Level: This level represents the physical storage of data in the database. It is responsible for storing and retrieving data from the storage devices, such as hard drives or solid-state drives. It deals with low-level implementation details such as data compression, indexing, and storage allocation.
2. Conceptual Level: This level represents the logical view of the database. It deals with the overall organization of data in the database and the relationships between them. It defines the data schema, which includes tables, attributes, and their relationships. The conceptual level is independent of any specific DBMS and can be implemented using different DBMSs.
3. External Level: This level represents the user's view of the database. It deals with how users access the data in the database. It allows users to view data in a way that makes sense to them, without worrying about the underlying implementation details. The external level provides a set of views or interfaces to the database, which are tailored to meet the needs of specific user groups.

The three levels are connected through a schema mapping process that translates data from one level to another. The schema mapping process ensures that changes made at one level are reflected in the other levels.

In addition to these three levels, a DBMS also includes a Database Administrator (DBA) component, which is responsible for managing the database system. The DBA is responsible for tasks such as database design, security management, backup and recovery, and performance tuning.

Overall, the structure of a DBMS is designed to provide a high level of abstraction to users, while still allowing low-level implementation details to be managed effectively. This allows users to focus on the logical organization of data in the database, without worrying about the physical storage or implementation details.

4. Draw an ER diagram for the banking system. Assume the banking requirements are as given below.
 - The bank is organized into branches. Each branch is located in a particular city.
 - The bank offers two types of accounts: saving and current. Accounts can be held by more than one customer and a customer can have more than one account.
 - A loan originates at a particular branch and can be held by one or more customersIdentify the relationship among the entities along with the mapping cardinalities, keys in the E.R. diagram. Construct appropriate tables for E-R diagram designed with above requirements.



5. Define DBMS. Explain advantages of DBMS over file system.

Definition of DBMS

A **Database Management System (DBMS)** is a software system designed to manage databases. It provides an interface for users and applications to interact with the data stored in a database, enabling the efficient organization, retrieval, and manipulation of data. A DBMS abstracts the complexities of data management, offering a structured way to store, query, update, and secure data.

Key Functions of a DBMS:

- **Data Storage Management:** Handles how data is stored on disk.
- **Data Retrieval and Querying:** Provides tools and languages for querying and retrieving data.
- **Data Manipulation:** Allows for the modification, insertion, and deletion of data.
- **Data Security:** Manages user access and enforces permissions.
- **Transaction Management:** Ensures data consistency and supports atomic transactions.
- **Backup and Recovery:** Protects data from loss or corruption and supports recovery processes.

Advantages of DBMS Over File Systems

A DBMS offers several advantages over traditional file systems. Here's a comparison highlighting the key benefits:

1. Data Redundancy and Inconsistency Control

- **File System:** In a traditional file system, data redundancy is common, where the same data might be stored in multiple places, leading to inconsistencies and increased storage costs.
- **DBMS:** A DBMS minimizes redundancy through normalization and centralizes data storage. It ensures consistency by maintaining a single source of truth and providing mechanisms to update data in one place, reflecting changes throughout the system.

2. Data Integrity and Accuracy

- **File System:** Ensuring data integrity and accuracy in a file system is challenging. There are limited built-in mechanisms to enforce constraints or validation rules, leading to potential data anomalies.
- **DBMS:** DBMSs enforce data integrity through constraints such as primary keys, foreign keys, and check constraints. These rules ensure that data adheres to predefined standards and relationships, reducing errors and maintaining accuracy.

3. Efficient Data Access

- **File System:** Searching and accessing data can be inefficient and slow, especially as the volume of data grows. File systems lack advanced indexing and querying capabilities.
- **DBMS:** DBMSs provide efficient data access through indexing, query optimization, and sophisticated search algorithms. This allows for fast retrieval of data and supports complex queries.

4. Data Security

- **File System:** File systems offer limited security features, typically relying on operating system-level access controls. There is little granular control over data access within files.
- **DBMS:** DBMSs provide robust security mechanisms, including user authentication, authorization, and access control. Administrators can define fine-grained permissions, ensuring that only authorized users can access or modify specific data.

5. Data Independence

- **File System:** Changes to the data structure in a file system can require significant changes to application programs. There is a tight coupling between the application and the data.
- **DBMS:** DBMSs offer data independence by separating the logical view of data from the physical storage. Changes to the database schema can often be made without affecting the applications that use the data.

6. Concurrent Access

- **File System:** Managing concurrent access to files can be complex, often requiring manual handling of file locks and leading to potential data conflicts and inconsistencies.
- **DBMS:** DBMSs handle concurrent access through transaction management and concurrency control mechanisms. They ensure that multiple users can access and modify data simultaneously without conflicts or data corruption.

7. Backup and Recovery

- **File System:** Implementing backup and recovery processes for file systems can be cumbersome and often requires manual intervention. Data recovery might be limited and less reliable.

- **DBMS:** DBMSs include built-in backup and recovery features, enabling automated, consistent, and reliable data backup. They offer tools for recovering data in case of failures or crashes, ensuring business continuity.

8. Data Management and Administration

- **File System:** Managing data in a file system often involves manual handling of data files, directories, and user permissions. It lacks centralized management tools.
- **DBMS:** DBMSs provide comprehensive data management tools and administrative functions, including monitoring, reporting, and tuning. These tools simplify database administration and enhance overall efficiency.

Summary

In summary, a **Database Management System (DBMS)** offers a structured and efficient approach to managing data compared to traditional file systems. It addresses issues related to data redundancy, integrity, security, and management, and provides advanced capabilities for data retrieval, manipulation, and concurrency control.

6. Different components of database management systems like query processor, storage manager, transaction manager etc. are functional for processing the query submitted by user. Explain the functions of each component in view of getting query output.

When a user submits a query to a Database Management System (DBMS), several components work together to process the query and return the desired output. Each component plays a specific role in ensuring that the query is executed efficiently and accurately. Here's an overview of the key components involved in query processing and their functions:

1. Query Processor

Function: The Query Processor is responsible for interpreting, optimizing, and executing the query submitted by the user. It involves several sub-components:

- **Query Interpreter:**
 - **Function:** Parses the SQL query to ensure it is syntactically and semantically correct. It converts the high-level query into an internal representation, often referred to as a query tree or query plan.
 - **Process:**
 1. **Syntax Checking:** Verifies that the query is written correctly according to the SQL language rules.
 2. **Semantic Checking:** Ensures that the query makes sense in the context of the database schema (e.g., checking if columns and tables exist).
- **Query Optimizer:**
 - **Function:** Analyzes the query and determines the most efficient way to execute it. The optimizer generates different execution plans and selects the one with the lowest estimated cost, balancing factors such as execution time and resource usage.
 - **Process:**
 1. **Cost Estimation:** Estimates the cost of different query execution plans based on factors like index usage, join methods, and data distribution.
 2. **Plan Generation:** Creates multiple execution strategies and selects the optimal plan.
- **Query Executor:**

- **Function:** Executes the optimized query plan, interacting with the Storage Manager to retrieve or modify data as specified in the query.
- **Process:**
 1. **Execution:** Retrieves data from storage, applies filters, joins tables if needed, and performs other operations specified in the query.
 2. **Result Generation:** Formats and returns the query result to the user.

2. Storage Manager

Function: The Storage Manager is responsible for managing the physical storage of data and handling data retrieval and modification operations. It interfaces with the Query Processor to fetch or update data as needed.

- **File Manager:**
 - **Function:** Manages the organization of data on disk. It handles file creation, deletion, and access.
 - **Process:**
 1. **Data Retrieval:** Reads data from files based on the query's requirements.
 2. **Data Storage:** Writes new or updated data back to the files.
- **Buffer Manager:**
 - **Function:** Manages the in-memory cache (buffer) where data pages are temporarily stored. This reduces the number of disk accesses by keeping frequently accessed data in memory.
 - **Process:**
 1. **Page Fetching:** Retrieves data pages from disk into memory.
 2. **Page Replacement:** Decides which pages to keep in memory and which to evict based on caching policies.
- **Index Manager:**
 - **Function:** Manages indexes that speed up data retrieval by providing quick access paths to the data.
 - **Process:**
 1. **Index Lookup:** Uses indexes to quickly locate the data that matches the query conditions.
 2. **Index Maintenance:** Updates indexes when data is inserted, deleted, or modified.
- **Log Manager:**
 - **Function:** Maintains logs of all changes to the database to support transaction management, recovery, and consistency.
 - **Process:**
 1. **Logging:** Records modifications made during query execution.
 2. **Recovery:** Uses logs to restore data in case of a failure.

3. Transaction Manager

Function: The Transaction Manager ensures that transactions are processed reliably and adhere to the ACID (Atomicity, Consistency, Isolation, Durability) properties. It handles the execution of queries within transactions and manages concurrency control.

- **Transaction Management:**
 - **Function:** Ensures that all operations within a transaction are completed successfully or none are applied (atomicity). It also maintains database consistency and manages concurrent access to prevent conflicts.
 - **Process:**
 1. **Transaction Control:** Starts, commits, or rolls back transactions based on query execution.
 2. **Concurrency Control:** Manages simultaneous transactions to avoid conflicts and ensure isolation.

Query Processing Workflow

1. **User submits query:** The query is received by the Query Processor.
2. **Query Interpreter:** Parses and validates the query.
3. **Query Optimizer:** Generates and selects the most efficient execution plan.
4. **Query Executor:** Executes the plan, interacting with the Storage Manager.
5. **Storage Manager:** Retrieves or updates data using the File Manager, Buffer Manager, Index Manager, and Log Manager.
6. **Transaction Manager:** Ensures that operations within transactions are handled correctly, maintaining data consistency and managing concurrency.
7. **Results returned to user:** The final results are formatted and sent back to the user.

Summary

In summary, the DBMS architecture involves several components that work together to process user queries efficiently:

- **Query Processor:** Handles the interpretation, optimization, and execution of queries.
- **Storage Manager:** Manages data storage, retrieval, and indexing.
- **Transaction Manager:** Ensures reliable transaction processing and concurrency control.

Each component has a specific role, contributing to the overall efficiency, reliability, and integrity of the database system.

7. Explain with example what is physical data independence. Also explain its importance.

Physical Data Independence is a concept in database management systems (DBMS) that ensures changes in the physical storage of data do not affect the application's ability to interact with the data. Essentially, it separates the physical level of data storage from the logical level of data representation. This means that modifications to how data is stored on disk (e.g., changes in file structure or storage format) do not impact the structure or operations of the database schema as seen by the user or application.

Example

Scenario:

Imagine you have a database for an online retail store with a Products table. The logical schema for this table might look like this:

Products

+-----+-----+

| Column | Description |

+-----+-----+

| ProductID | Integer, Primary Key |

| Name | Text |

| Price | Decimal |

| Category | Text |

+-----+-----+

Initial Physical Storage:

1. Initial Setup:

- Data for the Products table is stored in a simple, flat file format on a hard disk. Each record is stored sequentially.

Change in Physical Storage:

2. Physical Storage Optimization:

- To improve performance and manageability, you decide to optimize the physical storage. This could involve:
 - **Adding Indexes:** Create an index on the Price column to speed up queries filtering by price.
 - **Changing Storage Format:** Migrate from a flat file to a more advanced storage system like a B-tree index or a columnar store to improve query performance.

Impact on Logical Schema:

3. Logical Schema Remains Unchanged:

- Even after these physical changes, the logical schema of the Products table remains the same. Queries written in SQL, like:

SELECT Name, Price FROM Products WHERE Category = 'Electronics';

- Continue to function without modification. Applications and users interact with the Products table as if no changes were made to the underlying storage.

Importance of Physical Data Independence

1. Flexibility in Data Storage:

- **Adaptability:** You can modify the physical storage mechanism to optimize performance or manage storage capacity without altering the logical schema. This allows the database to adapt to new storage technologies or optimizations as needed.
- **Ease of Upgrades:** Upgrading storage hardware or file formats can be done without impacting how applications access or use the data.

2. Separation of Concerns:

- **Modularity:** It separates the database's physical storage details from its logical organization. This separation allows database administrators to manage and optimize storage without affecting application development.

- **Simplified Application Design:** Developers can design applications based on the logical schema, without worrying about physical storage details.

3. Improved Performance:

- **Optimization:** Changes to the physical storage, such as adding indexes or reorganizing data, can enhance performance without altering the logical schema. This leads to more efficient query processing and faster data access.

4. Reduced Risk of Disruption:

- **Stability:** Changes to physical storage structures (e.g., changing file formats or adding indexing) do not disrupt existing applications. This minimizes the risk of introducing bugs or requiring extensive testing.

5. Support for Evolving Technologies:

- **Technological Advancements:** As storage technologies evolve, physical data storage can be updated or optimized without needing changes to the database schema or application logic. This ensures that the database can benefit from new technologies without extensive redevelopment.

6. Consistent User Experience:

- **Data Accessibility:** Users and applications interact with the data through a consistent logical schema. Regardless of changes in how the data is physically stored, the interface and query mechanisms remain stable and predictable.

8. What is view and how to create it? Can you update view? If yes, how? If not, why not?

A **view** in a database is a virtual table that provides a specific representation of data from one or more tables. It is essentially a stored query that can be used to present data in a specific format or subset without altering the actual tables. Views are used to simplify complex queries, restrict access to certain data, or present data in a way that is easier for users to understand.

Characteristics of a View:

- **Virtual Table:** A view does not store data physically; it dynamically retrieves data from the underlying tables when queried.
- **Simplification:** Views can simplify complex queries by encapsulating them into a single table-like structure.
- **Security:** Views can restrict access to sensitive data by exposing only a subset of columns or rows.
- **Customization:** Views can present data in a customized format or aggregation that suits specific reporting needs.

How to Create a View

In SQL, a view is created using the CREATE VIEW statement. Here's the general syntax:

```
CREATE VIEW view_name AS
```

```
SELECT column1, column2, ...
```

```
FROM table_name
```

```
WHERE condition;
```

Example:

Suppose you have a table named Employees:

Employees

ID	Name	Position	Salary
1	Alice	Manager	80000
2	Bob	Developer	60000
3	Carol	Analyst	55000

To create a view that shows only employees with a salary greater than 60,000, you can use the following SQL statement:

```
CREATE VIEW HighEarners AS
```

```
SELECT Name, Position, Salary
```

```
FROM Employees
```

```
WHERE Salary > 60000;
```

This creates a view named HighEarners that presents the Name, Position, and Salary of employees who earn more than 60,000.

Can You Update a View?

Updating a View:

Yes, it is possible to update a view, but whether you can update a view depends on several factors:

1. Updatable Views:

- A view is updatable if it meets certain criteria. Generally, the view must be based on a single table, and it must not include aggregate functions, group by clauses, or joins that create ambiguity in how rows should be updated.
- **Example:** If the view HighEarners is created as shown earlier, and you want to update the salary of an employee via this view, it can be done if the underlying query meets the update criteria.

```
UPDATE HighEarners
```

```
SET Salary = 85000
```

```
WHERE Name = 'Alice';
```

2. Non-Updatable Views:

- Views that involve complex queries, multiple tables, or aggregate functions may be non-updatable. In these cases, attempting to perform update operations directly on the view will result in an error.
- **Example:** A view that includes a join or aggregate function:

```
CREATE VIEW EmployeeStats AS
```

```
SELECT Position, AVG(Salary) AS AvgSalary
```

```
FROM Employees
```

```
GROUP BY Position;
```

Updating EmployeeStats directly is not possible because it involves aggregated data and multiple rows from the underlying Employees table.

Why Some Views Are Not Updatable

1. Complex Queries:

- Views that involve joins, unions, or subqueries might create ambiguity about how to propagate changes to the underlying tables. For instance, if a view involves multiple tables, it's unclear which table should be updated.

2. Aggregations and Groupings:

- Views with aggregate functions like SUM, AVG, or COUNT cannot be updated directly because the results are derived from calculations rather than direct data in a single table.

3. Distinct or Set Operations:

- Views that use DISTINCT, UNION, or other set operations can result in data that isn't straightforward to update, as the view might combine or filter out rows from the underlying tables.

Summary

- **View:** A virtual table that presents data from one or more tables based on a query.
- **Creating a View:** Use the CREATE VIEW statement to define a view with a specific query.
- **Updating a View:** Some views can be updated if they meet certain criteria (e.g., based on a single table without aggregation). Complex or derived views may be non-updatable due to ambiguity in data updates.

9. Defined stored procedure. Explain the creating and calling stored procedure with example.

A **stored procedure** is a precompiled collection of SQL statements and optional control-flow logic that is stored in the database and can be executed as a single unit. Stored procedures are used to encapsulate repetitive tasks, business logic, and complex queries, making them reusable, maintainable, and efficient. They help in improving performance by reducing the amount of SQL sent to the database and provide a way to implement application logic directly in the database.

Creating a Stored Procedure

Syntax for Creating a Stored Procedure:

The syntax for creating a stored procedure can vary slightly depending on the database system (e.g., MySQL, SQL Server, PostgreSQL). Below is a general syntax:

```
CREATE PROCEDURE procedure_name (parameters)
```

```
BEGIN
```

```
    -- SQL statements and logic
```

```
END;
```

Example:

Suppose you have a table Employees and you want to create a stored procedure to retrieve employee details based on their EmployeeID. Here's how you can create it:

```
CREATE PROCEDURE GetEmployeeDetails (IN empID INT)
```

```
BEGIN  
    SELECT * FROM Employees  
    WHERE EmployeeID = empID;  
END;

- CREATE PROCEDURE GetEmployeeDetails: Defines a new stored procedure named GetEmployeeDetails.
- IN empID INT: Defines an input parameter empID of type INT.
- BEGIN ... END: Encapsulates the SQL statements that form the body of the stored procedure.

```

Calling a Stored Procedure

Syntax for Calling a Stored Procedure:

To execute or call a stored procedure, you use a specific command that also varies slightly depending on the database system:

```
CALL procedure_name (parameters);
```

Example:

To call the GetEmployeeDetails stored procedure you created, you would use:

```
CALL GetEmployeeDetails(123);
```

Here, 123 is the value passed as the empID parameter.

Example in Different Database Systems

1. MySQL:

- **Creating a Stored Procedure:**

```
CREATE PROCEDURE GetEmployeeDetails (IN empID INT)
```

```
BEGIN
```

```
    SELECT * FROM Employees
```

```
    WHERE EmployeeID = empID;
```

```
END;
```

- **Calling a Stored Procedure:**

```
CALL GetEmployeeDetails(123);
```

2. SQL Server:

- **Creating a Stored Procedure:**

```
CREATE PROCEDURE GetEmployeeDetails
```

```
    @empID INT
```

```
AS
```

```
BEGIN
```

```
    SELECT * FROM Employees
```

```
WHERE EmployeeID = @empID;
```

```
END;
```

- **Calling a Stored Procedure:**

```
EXEC GetEmployeeDetails @empID = 123;
```

3. PostgreSQL:

- **Creating a Stored Procedure:**

```
CREATE OR REPLACE FUNCTION GetEmployeeDetails(empID INT)
```

```
RETURNS TABLE (EmployeeID INT, Name TEXT, Position TEXT, Salary NUMERIC) AS
```

```
$$
```

```
BEGIN
```

```
    RETURN QUERY
```

```
    SELECT * FROM Employees
```

```
    WHERE EmployeeID = empID;
```

```
END;
```

```
$$
```

```
LANGUAGE plpgsql;
```

- **Calling a Stored Procedure:**

```
SELECT * FROM GetEmployeeDetails(123);
```

Importance of Stored Procedures

1. Performance Improvement:

- Stored procedures are precompiled, which means they execute faster than dynamic SQL queries because they avoid repeated parsing and compilation.

2. Code Reusability:

- They allow you to encapsulate complex logic in one place, which can be reused across multiple applications or queries, reducing code duplication.

3. Maintainability:

- Changes to business logic can be made in the stored procedure without altering the application code, making it easier to manage and update logic.

4. Security:

- Stored procedures can help to enhance security by restricting direct access to the underlying tables. Users can execute procedures without having direct permissions on the tables.

5. Reduced Network Traffic:

- By executing multiple SQL statements in a single call to the stored procedure, network traffic is reduced compared to sending individual SQL statements from the application.

Summary

- **Stored Procedure:** A precompiled set of SQL statements stored in the database for reuse.
- **Creating a Stored Procedure:** Defined using CREATE PROCEDURE with SQL statements encapsulated between BEGIN and END.
- **Calling a Stored Procedure:** Executed using the CALL or EXEC statement, depending on the DBMS.

10. Consider following schema.

`Student_fee_details (rollno, name, fee_deposited, date)`

Write a trigger to preserve old values of student fee details before updating in the table.

To preserve old values of student fee details before updating in the table, you can create a trigger that:

1. **Creates a temporary table:** This table will store the old values of the row being updated.
2. **Copies the old values:** Before the update, the values of the row being updated are copied to the temporary table.
3. **Performs the update:** The update operation proceeds as normal.

Here's a SQL example for a trigger that achieves this:

SQL

```
CREATE TRIGGER tr_student_fee_details_update
BEFORE UPDATE ON student_fee_details
FOR EACH ROW
BEGIN
    -- Create a temporary table to store old values
    CREATE TEMPORARY TABLE old_student_fee_details (
        rollno INT,
        name VARCHAR(50),
        fee_deposited DECIMAL(10,2),
        date DATE
    );
```

```
-- Insert old values into the temporary table
INSERT INTO old_student_fee_details
VALUES (OLD.rollno, OLD.name, OLD.fee_deposited, OLD.date);
END;
```

This trigger will create a temporary table named `old_student_fee_details` and insert the old values of the row being updated into it before the update occurs. You can then query this temporary table to access the old values if needed.

Note: This trigger assumes that the `student_fee_details` table has the specified columns and data types. You may need to adjust the trigger based on your specific table structure

11. Consider the following schemes Supplier(SNO, Sname, Status, City) Parts (PNO, Pname, Color, Weight, City) Shipments(SNO,PNO,QTY) Write SQL queries for the following: i) Find shipment information (SNO, Sname, PNO, Pname, QTY) for those having quantity less than 157. ii) List SNO, Sname, PNO, Pname for those suppliers who made shipments of parts whose quantity is larger than the average quantity iii) Find aggregate quantity of PNO 1692 of color green for which shipments made by supplier number who residing Mumbai.

SQL Queries for Supplier, Parts, and Shipments

i) Find shipment information for those having quantity less than 157:

SQL

```
SELECT S.SNO, S.Sname, P.PNO, P.Pname, Sh.QTY
FROM Shipments Sh
INNER JOIN Supplier S ON Sh.SNO = S.SNO
INNER JOIN Parts P ON Sh.PNO = P.PNO
WHERE Sh.QTY < 157;
```

ii) List SNO, Sname, PNO, Pname for suppliers who made shipments of parts whose quantity is larger than the average quantity:

SQL

```
SELECT S.SNO, S.Sname, P.PNO, P.Pname
FROM Shipments Sh
INNER JOIN Supplier S ON Sh.SNO = S.SNO
INNER JOIN Parts P ON Sh.PNO = P.PNO
WHERE Sh.QTY > (SELECT AVG(QTY) FROM Shipments);
```

iii) Find aggregate quantity of PNO 1692 of color green for which shipments made by supplier number who residing Mumbai:

SQL

```
SELECT SUM(Sh.QTY) AS TotalQuantity
FROM Shipments Sh
INNER JOIN Supplier S ON Sh.SNO = S.SNO
INNER JOIN Parts P ON Sh.PNO = P.PNO
WHERE P.PNO = 1692 AND P.Color = 'Green' AND S.City = 'Mumbai';
```

These queries effectively combine data from the Supplier, Parts, and Shipments tables to provide the requested information.

12. What is a trigger? How to create it? Discuss various types of triggers.

Triggers in SQL

What is a Trigger?

A trigger in SQL is a special type of stored procedure that is automatically executed in response to a specific data modification event, such as an INSERT, UPDATE, or DELETE statement. It provides a way to enforce business rules, maintain data integrity, and automate tasks based on changes to database data.

How to Create a Trigger

To create a trigger, you typically use the CREATE TRIGGER statement. The syntax for creating a trigger generally looks like this:

SQL

```
CREATE TRIGGER trigger_name
BEFORE | AFTER | INSTEAD OF event_name
```

```
ON table_name  
FOR EACH ROW  
BEGIN  
    -- Trigger body (SQL statements)  
END;
```

- **trigger_name:** The name you want to give to the trigger.
- **BEFORE | AFTER | INSTEAD OF:** Specifies when the trigger should execute relative to the event.
- **event_name:** The type of event that triggers the execution (e.g., INSERT, UPDATE, DELETE).
- **table_name:** The table on which the event occurs.
- **FOR EACH ROW:** Specifies that the trigger should execute for each row affected by the event.
- **BEGIN ... END:** Encloses the SQL statements that will be executed when the trigger is activated.

Types of Triggers

1. **Row-Level Triggers:** These triggers are executed for each row affected by the event. They can be used to perform actions on individual rows, such as updating related tables or validating data.
2. **Statement-Level Triggers:** These triggers are executed once for the entire statement, regardless of the number of rows affected. They are useful for tasks that need to be performed only once per operation, such as sending notifications or auditing changes.
3. **BEFORE Triggers:** These triggers execute before the event occurs. They can be used to validate data, prevent invalid changes, or modify data before it is inserted or updated.
4. **AFTER Triggers:** These triggers execute after the event occurs. They can be used to log changes, update related tables, or perform other actions after the data has been modified.
5. **INSTEAD OF Triggers:** These triggers replace the default behavior of the event. They are often used to implement custom logic for specific operations, such as virtual tables or audit trails.

Example:

Here's a simple example of an AFTER trigger that logs changes to a table:

SQL

```
CREATE TRIGGER audit_changes  
AFTER UPDATE ON customers  
FOR EACH ROW  
BEGIN  
    INSERT INTO customer_audit (customer_id, old_name, new_name)  
    VALUES (OLD.customer_id, OLD.name, NEW.name);  
END;
```

This trigger inserts a record into an customer_audit table whenever a customer's name is updated, logging the old and new values.

13. Draw the neat diagram of Database System Structure and explain its components in detail.

Refer Q3

14. Construct an ER Diagram for Company having following details :

- Company organized into DEPARTMENT. Each department has unique name and a particular employee who manages the department. Start date for the manager is recorded. Department may have several locations.
- A department controls a number of PROJECT. Projects have a unique name, number and a single location.
- Company's EMPLOYEE name, ssno, address, salary, sex and birth date are recorded. An employee is assigned to one department, but may work for several projects (not necessarily controlled by her dept). Number of hours/week an employee works on each project is recorded; The immediate supervisor for the employee.
- Employee's DEPENDENT are tracked for health insurance purposes (dependent name, birthdate, relationship to employee).

Identify the relationship among the entities along with the mapping cardinalities, keys in the E.R. diagram.

15. A post office has few postmen who go every day to distribute letter. Every morning post office receives a large number of registered letters. The post office intends to create a database to keep track of these letters.

- Every letter has a sender, an origin post office from where it was sent, a destination post office to which it is to be sent, a date of registration, date of arrival at destination post office, receiver and a status.
- Every sender has a name, an address.
- Every receiver has a name and an address. iv)
- Every postman has a designated area where he delivers letters. v)
- The area consists of a set of streets under the jurisdiction of the post office. vi) Every street consists of a set of buildings.
- Every building has number and may be name. It may be housing more than one family.
- The status of the letter can be not yet taken for delivery, delivered, address not available, address not known, addressee did not accept the letter, redirected to the address of address and sent to the sender.

Identify the relationship among the entities along with the mapping cardinalities, keys in the E.R. diagram. Construct appropriate tables for E-R diagram designed with above requirements.

16. List the main characteristics of the database approach and explain how it differs from the traditional file system.

Refer Q5

17. Consider the following schemas

Emp(Emp_no, Emp_name, Dept_no)

Dept(Dept_no, Dept_name)

Address(Dept_name, Dept_location)

Write SQL queries for the following

- Display the location of department where employee 'Ram' is working.
- Create a view to store total no of employees working in each department in ascending order.
- Find the name of the department in which no employee is working.

1. Schemas:

- **Emp:** (Emp_no, Emp_name, Dept_no)
- **Dept:** (Dept_no, Dept_name)
- **Address:** (Dept_name, Dept_location)

i) Display the location of the department where employee 'Ram' is working

To find the location of the department where employee 'Ram' is working, you need to join the Emp, Dept, and Address tables.

SQL Query:

```
SELECT A.Dept_location
FROM Emp E
JOIN Dept D ON E.Dept_no = D.Dept_no
JOIN Address A ON D.Dept_name = A.Dept_name
WHERE E.Emp_name = 'Ram';
```

Explanation:

- Join Emp and Dept on Dept_no to get the department name for employee 'Ram'.
- Join Dept and Address on Dept_name to get the location of the department.
- Filter the result where Emp_name is 'Ram'.

ii) Create a view to store the total number of employees working in each department in ascending order

To create a view that shows the total number of employees in each department, sorted in ascending order, follow these steps:

SQL Query:

```
CREATE VIEW EmployeeCountByDept AS
SELECT D.Dept_name, COUNT(E.Emp_no) AS Total_Employees
FROM Emp E
JOIN Dept D ON E.Dept_no = D.Dept_no
GROUP BY D.Dept_name
ORDER BY Total_Employees ASC;
```

Explanation:

- Join Emp and Dept on Dept_no to group employees by department.
- Use COUNT(E.Emp_no) to get the number of employees in each department.
- Group by Dept_name and order the results by Total_Employees in ascending order.
- Create a view named EmployeeCountByDept to store this information.

iii) Find the name of the department in which no employee is working

To find departments with no employees, you need to find departments that do not appear in the Emp table.

SQL Query:

```

SELECT D.Dept_name
FROM Dept D
LEFT JOIN Emp E ON D.Dept_no = E.Dept_no
WHERE E.Emp_no IS NULL;

```

Explanation:

- Use a LEFT JOIN to join Dept with Emp on Dept_no.
- The LEFT JOIN ensures that all departments are included, even if there are no matching records in Emp.
- Filter for departments where E.Emp_no is NULL, indicating no employees are associated with that department.

Summary of SQL Queries:

1. Department Location for Employee 'Ram':

- Joins Emp, Dept, and Address to find the department location for the specified employee.

2. View of Total Employees per Department:

- Creates a view that counts employees per department and sorts the result by the number of employees in ascending order.

3. Departments with No Employees:

- Uses a LEFT JOIN to identify departments with no corresponding employees in the Emp table.

These queries help retrieve relevant information about departments and employees from the provided schema

18. Consider following schema Student_fee_details (rollno, name, fee_deposited, date) Write a trigger to preserve old values of student fee details before updating in the table.

Creating a Trigger to Preserve Old Student Fee Details

Understanding the Task: We need to create a trigger that will capture the old values of a student's fee details before they are updated in the Student_fee_details table. This will allow us to track changes and potentially revert to previous values if necessary.

Trigger Creation:

SQL

```

CREATE TRIGGER tr_student_fee_details_update
BEFORE UPDATE ON Student_fee_details
FOR EACH ROW
BEGIN
    -- Create a temporary table to store old values
    CREATE TEMPORARY TABLE old_student_fee_details (
        rollno INT,
        name VARCHAR(50),

```

```

fee_deposited DECIMAL(10,2),
date DATE
);

-- Insert old values into the temporary table
INSERT INTO old_student_fee_details
VALUES (OLD.rollno, OLD.name, OLD.fee_deposited, OLD.date);

END;

```

Explanation:

1. **CREATE TRIGGER tr_student_fee_details_update:** Creates a trigger named tr_student_fee_details_update.
2. **BEFORE UPDATE ON Student_fee_details:** Specifies that the trigger will execute before an update operation on the Student_fee_details table.
3. **FOR EACH ROW:** Indicates that the trigger will be executed for each row that is updated.
4. **CREATE TEMPORARY TABLE old_student_fee_details:** Creates a temporary table to store the old values.
5. **INSERT INTO old_student_fee_details:** Inserts the old values from the OLD pseudo-table into the temporary table.

How it Works: When a student's fee details are updated, this trigger will:

1. Create a temporary table to store the old values of the row being updated.
2. Insert the old values into the temporary table.
3. Allow the update to proceed as normal.

Accessing Old Values: To access the old values of a student's fee details after an update, you can query the old_student_fee_details temporary table. For example:

SQL

```
SELECT * FROM old_student_fee_details WHERE rollno = 123;
```

This would return the old values for the student with roll number 123.

Note: The specific column names and data types in the Student_fee_details table may vary, so adjust them accordingly in the trigger and temporary table definition.

19. Explain the concept of Referential and Entity Integrity constraint with example.

Referential Integrity and **Entity Integrity** are two fundamental concepts in relational database design that ensure the accuracy and consistency of the data within a database. Here's an explanation of each concept, including examples to illustrate how they work.

1. Entity Integrity Constraint

Definition: Entity integrity ensures that each table in a database has a primary key and that the primary key values are unique and not null. This constraint guarantees that each record within a table can be uniquely identified.

Key Points:

- **Primary Key:** Each table should have a primary key column or a set of columns that uniquely identifies each row in the table.
- **Uniqueness:** The values in the primary key column(s) must be unique for each row.
- **Non-null:** The primary key column(s) cannot contain NULL values because NULLs do not uniquely identify a row.

Example: Consider a table Employee with the following schema:

```
CREATE TABLE Employee (
```

```
    Emp_ID INT PRIMARY KEY,
```

```
    Emp_Name VARCHAR(100),
```

```
    Dept_ID INT
```

```
);
```

- **Emp_ID** is the primary key.
- Each Emp_ID must be unique and cannot be NULL.
- This ensures that every employee record can be uniquely identified by their Emp_ID.

Violation Example: If you try to insert a new record with an Emp_ID that already exists in the table, it violates entity integrity:

```
INSERT INTO Employee (Emp_ID, Emp_Name, Dept_ID)
```

```
VALUES (1, 'John Doe', 101); -- Assuming an employee with Emp_ID 1 already exists
```

2. Referential Integrity Constraint

Definition: Referential integrity ensures that relationships between tables remain consistent. It does this by ensuring that a foreign key value in one table matches a primary key value in another table or is NULL. This constraint maintains valid links between related tables.

Key Points:

- **Foreign Key:** A column or a set of columns in one table that refers to the primary key of another table.
- **Consistency:** The foreign key value must match an existing primary key value in the referenced table or be NULL.
- **Cascade Actions:** Options like CASCADE, SET NULL, or RESTRICT can be defined for how changes to the referenced primary key affect the foreign key.

Example: Consider the following tables Department and Employee:

```
CREATE TABLE Department (
```

```
    Dept_ID INT PRIMARY KEY,
```

```
    Dept_Name VARCHAR(100)
```

```
);
```

```

CREATE TABLE Employee (
    Emp_ID INT PRIMARY KEY,
    Emp_Name VARCHAR(100),
    Dept_ID INT,
    FOREIGN KEY (Dept_ID) REFERENCES Department(Dept_ID)
);

```

- **Dept_ID** in Employee is a foreign key that references the primary key **Dept_ID** in Department.
- **Referential Integrity:** Each **Dept_ID** in the Employee table must exist in the Department table.

Violation Example: If you try to insert an employee with a **Dept_ID** that does not exist in the Department table, it violates referential integrity:

```
INSERT INTO Employee (Emp_ID, Emp_Name, Dept_ID)
```

```
VALUES (1, 'Jane Smith', 999); -- Assuming Dept_ID 999 does not exist in the Department table
```

Summary

- **Entity Integrity:**
 - Ensures each table has a unique, non-null primary key.
 - Guarantees that each row in a table can be uniquely identified.
- **Referential Integrity:**
 - Ensures foreign key values in one table match primary key values in the referenced table or are NULL.
 - Maintains consistency in relationships between tables.

20. Write a PL/SQL block of code which accepts the rollno from user. The attendance of rollno entered by user will be checked in student_attendance(RollNo, Attendance) table and display on the screen.What is the importance of creating constraints on the table? Explain with example any 4 constraints that can be specified when a database table is created

1. PL/SQL Block to Check and Display Attendance

Here's a PL/SQL block that accepts a roll number from the user, checks the attendance for that roll number in the student_attendance table, and displays it on the screen.

Assumptions:

- You are using Oracle Database.
- You have the student_attendance table with the following schema:
student_attendance (RollNo, Attendance).

PL/SQL Block:

```

DECLARE
    v_rollno      student_attendance.RollNo%TYPE;
    v_attendance student_attendance.Attendance%TYPE;
BEGIN

```

```

-- Prompt the user to enter the roll number
v_rollno := &Enter_RollNo; -- `&Enter_RollNo` is a placeholder for user
input

-- Fetch the attendance for the given roll number
SELECT Attendance
INTO v_attendance
FROM student_attendance
WHERE RollNo = v_rollno;

-- Display the attendance
DBMS_OUTPUT.PUT_LINE('Roll Number: ' || v_rollno || ', Attendance: ' || v_attendance);

EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('No attendance record found for Roll Number: ' || v_rollno);
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('An error occurred: ' || SQLERRM);
END;
/

```

Explanation:

- **DECLARE:** Declares variables for storing user input and query results.
- **v_rollno and v_attendance:** Variables to hold the roll number and attendance data.
- **v_rollno := &Enter_RollNo;**: Placeholder for user input. In actual use, you might prompt for input differently depending on the environment.
- **SELECT INTO:** Fetches attendance data for the given roll number.
- **DBMS_OUTPUT.PUT_LINE:** Outputs the attendance data or an error message.
- **EXCEPTION:** Handles cases where no data is found or other errors occur.

2. Importance of Creating Constraints

Constraints are rules applied to table columns to enforce data integrity and consistency. They help maintain the accuracy and reliability of the database.

Importance:

- **Data Integrity:** Ensures that data entered into the database is accurate and reliable.
- **Consistency:** Maintains uniformity across the database by enforcing specific rules and relationships.
- **Error Prevention:** Prevents the entry of invalid data that could lead to errors or inconsistencies.

Types of Constraints and Examples

1. PRIMARY KEY Constraint:

- **Purpose:** Ensures that each row in a table has a unique identifier and no NULL values.
- **Example:**

```

CREATE TABLE Student (
    RollNo INT PRIMARY KEY,
    Name VARCHAR(100)
);

```

- **Explanation:** The RollNo column is a primary key, meaning each roll number must be unique and cannot be NULL.

2. FOREIGN KEY Constraint:

- **Purpose:** Ensures that values in a column (or columns) in one table must match values in a column of another table, maintaining referential integrity.
- **Example:**

```
CREATE TABLE Enrollment (
    RollNo INT,
    CourseID INT,
    FOREIGN KEY (RollNo) REFERENCES Student(RollNo),
    FOREIGN KEY (CourseID) REFERENCES Course(CourseID)
);
```

- **Explanation:** The RollNo in the Enrollment table must exist in the Student table, and CourseID must exist in the Course table.

3. UNIQUE Constraint:

- **Purpose:** Ensures that all values in a column (or a group of columns) are unique across the table.
- **Example:**

```
CREATE TABLE Teacher (
    TeacherID INT UNIQUE,
    Email VARCHAR(100) UNIQUE
);
```

- **Explanation:** The TeacherID and Email columns must have unique values across all rows.

4. CHECK Constraint:

- **Purpose:** Ensures that all values in a column meet a specific condition.
- **Example:**

```
CREATE TABLE Employee (
    EmpID INT,
    Salary DECIMAL(10, 2),
    CONSTRAINT salary_check CHECK (Salary > 0)
);
```

- **Explanation:** The Salary column must have values greater than 0. This prevents the entry of negative or zero salaries.

5. NOT NULL Constraint:

- **Purpose:** Ensures that a column cannot have NULL values.
- **Example:**

```
CREATE TABLE Course (
    CourseID INT PRIMARY KEY,
    CourseName VARCHAR(100) NOT NULL
);
```

- **Explanation:** The CourseName column must always have a value; it cannot be NULL.

Summary

- **PL/SQL Block:** Provides a mechanism to interact with the database, perform checks, and handle data, including user input and error handling.
- **Constraints:** Crucial for maintaining data integrity, consistency, and preventing errors. Constraints such as PRIMARY KEY, FOREIGN KEY, UNIQUE, CHECK, and NOT NULL enforce rules on data stored in the tables.