**WEEK-3**

**3.1 DISCUSS AND DOCUMENT CODD RULES (12 RULES)**

**CODD RULES:**

**Edgar F. Codd** wrote a paper in 1985 defining rules for [Relational Database Management Systems](http://olap.com/learn-bi-olap/olap-bi-definitions/rdbms/) (RDBMS), which revolutionized the IT industry. So the CODD Rules consist of 12 rules that are:

**1st Rule: Information Rule:**

All data in the database should be stored in tables with rows and columns. Each row should represent a unique record, and each column should represent a specific attribute. This makes it easy to organize and manage data.

**2nd Rule: Guaranteed Access Rule:**

Every piece of data should be accessible using a combination of table name, primary key value, and column name. This ensures that data can be retrieved efficiently without any ambiguity.

**3rd Rule: Systematic Treatment of Null Values:**

Missing or unknown data should be represented using NULL values, which helps in dealing with incomplete information in a consistent and reliable manner.

**4th Rule: Dynamic Online Catalog Based on the Relational Model:**

The database should have a dynamic catalog that describes its structure and organization, providing users with the necessary information about the data and its relationships.

**5th Rule: Comprehensive Data Sublanguage Rule:**

The system should provide a comprehensive language for defining, manipulating, and controlling the data. This enables users to interact with the database using standardized commands and functions.

**6th Rule: View Updating Rule:**

Any data that can be retrieved by authorized users can also be updated by the same users.

**7th Rule: High-level Insert, Update, and Delete Rule:**

The system should support high-level insertion, updating, and deletion operations, allowing users to work with data at a conceptual level without worrying about the specific implementation details.

**8th Rule: Physical Data Independence:**

Changes in the physical data storage structures and devices should not affect the logical data structure. This enables the system to be flexible and adaptable to changes in the physical environment.

**9th Rule: Logical Data Independence:**

Changes in the logical data structure, such as modifications to tables or views, should not affect the application programs. This ensures that the database can evolve without disrupting the existing applications.

**10th Rule: Integrity Independence:**

The integrity constraints of the database should be separate from the application programs and stored in the catalog. This guarantees that the integrity rules remain consistent regardless of changes in the application programs.

**11th Rule: Distribution Independence:**

The distribution of data across different locations or networks should not be visible to users. This allows users to perceive the database as a single, centralized system, even if the data is distributed across multiple locations.

**12th Rule: Non-subversion Rule:**

The system should not allow users to bypass the integrity rules and security constraints, ensuring that the data remains secure and consistent. This prevents any unauthorized access or manipulation of the data.

**3.2 IDENTIFY STEPS TO CREATE**

**LOGICAL DATABASE DESIGN**

**DATABASE DESIGN:**

Creating a Logical Database Design Involves several Steps to ensure that the Database Structure Supports the Information needs of your Application. Here are the steps to be followed according to the Logical Database Design:

1. **Entity-Relationship Diagram (ERD)**
2. **Normalization of the Data (Normalization)**
3. **Define Data Types**
4. **Identify Keys**
5. **Establish Relationship**
6. **Create Tables**
7. **Constraints and Triggers**
8. **Document the Design**
9. **Review and Iterate**
10. **Implement the Design**
11. **Test and Validate**

**1. Entity-Relationship Diagram (ERD):**

ER Diagram stands for Entity-Relationship is based on Notion of a Real-World Entities and Relationship among them. It is Based on:

* **Entity:** An Entity in an ER Model is a Real World Object/Entity having Properties called attributes.
* **Relationship:** The Logical Association among Entities is called Relationship. They are mapped with entities in various way. Mapping Cardinalities define the number of association between two entities.

**2. Normalization of the Data (Normalization):**

Normalization is the process of Organizing the Data in the Database. Normalization is used to minimize the Redundancy and dependency from a set of relation. It is also used to eliminate undesirable characteristics like Insertion, Updation, Deletion Anomalies. It is used to Reduce Redundancy from the Database Table.

**3. Define Data Types:**

- Assign appropriate data types to each attribute (e.g., integer, string, date).

- Consider data constraints (e.g., unique keys, foreign keys) to enforce data integrity.

**4. Identify Keys:**

- Determine primary keys for each entity to uniquely identify records.

- Define foreign keys to establish relationships between entities.

**5. Establish Relationships:**

- Specify the cardinality and participation constraints for relationships (e.g., one-to-one, one-to- many).

- Use referential integrity constraints to maintain data consistency.

**6. Create Tables:**

- Translate the entities based on ER Diagram into database tables.

- Define the structure of each table, including columns, data types, and constraints.

**7. Constraints and Triggers:**

- Implement any additional constraints or triggers needed to enforce business rules and maintain

data integrity.

**8. Document the Design:**

- Create documentation that includes the schema, data dictionary, and any relevant information

about the design decisions.

**9. Review and Iterate:**

- Review the logical design with stakeholders and make necessary revisions based on feedback.

- Iterate through the design process as needed to refine and improve the structure.

**10. Implement the Design:**

- Translate the logical database design into SQL statements to create the actual database.

**11. Test and Validate:**

- Perform thorough testing to ensure that the database design meets the functional and

performance requirements.

**WEEK-4**

**4.1 NORMALIZATION, FORMS & ITS CONCEPTS**

**Normalization:**

Normalization is the process of Organizing the Data in the Database.

Normalization is used to minimize the Redundancy and dependency from a set of relation.

It is also used to eliminate undesirable characteristics like Insertion, Updation, Deletion Anomalies.

It is used to Reduce Redundancy from the Database Table.

**Normal Forms:**

The Normal Forms consist of 6 types of Normal Forms that are:

1. **4th Normal Form (4NF).**
2. **5th Normal Form (5NF).**
3. **Boyce Codd Normal Form (BCNF).**
4. **4th Normal Form (4NF):**

         A Relation ‘R’ are said to be in 4NF if it satisfies the Following conditions are:

* A relation will be in 4NF if it is in Boyce Codd normal form and has no multi-valued dependency.
* For a dependency A → B, if for a single value of A, multiple values of B exists, then the relation will be a multi-valued dependency.
* **STUDENT\_Table**

|  |  |  |
| --- | --- | --- |
| **STU\_ID** | **COURSE\_Name** | **HOBBY** |
| 21 | Computer | Dancing |
| 21 | Math | Singing |
| 34 | Chemistry | Dancing |
| 74 | Biology | Cricket |
| 59 | Physics | Hockey |

* The given STUDENT table is in 3NF, but the COURSE and HOBBY are two independent entity. Hence, there is no relationship between COURSE and HOBBY.
* In the STUDENT relation, a student with STU\_ID, **21** contains two courses, **Computer** and **Math** and two hobbies, **Dancing** and **Singing**. So there is a Multi-valued dependency on STU\_ID, which leads to unnecessary repetition of data.
* So to make the above table into 4NF, we can decompose it into two tables:

* **STUDENT\_COURSE** :

|  |  |
| --- | --- |
| **STU\_ID** | **COURSE** |
| 21 | Computer |
| 21 | Math |
| 34 | Chemistry |
| 74 | Biology |
| 59 | Physics |

* **STUDENT\_HOBBY** :

|  |  |
| --- | --- |
| **STU\_ID** | **HOBBY** |
| 21 | Dancing |
| 21 | Singing |
| 34 | Dancing |
| 74 | Cricket |
| 59 | Hockey |

1. **5th Normal Form (5NF):**

A Relation ‘R’ is said to be in 5NF if it satisfies the Following Conditions:

* A relation is in 5NF if it is in 4NF and not contains any join dependency and joining should be lossless.
* 5NF is satisfied when all the tables are broken into as many tables as possible in order to avoid redundancy.
* 5NF is also known as Project-join normal form (PJ/NF).

**Example:**

|  |  |  |
| --- | --- | --- |
| **SUBJECT** | **LECTURER** | **SEMESTER** |
| Computer | Jessica | 1st Semester |
| Computer | John | 2nd Semester |
| Math | John | 3rd Semester |
| Math | Robin | 4th Semester |
| Chemistry | Natasha | 5th Semester |

In the above table, John takes both Computer and Math class for Semester 1 but he doesn't take Math class for Semester 2. In this case, combination of all these fields required to identify a valid data.

Suppose we add a new Semester as Semester 3 but do not know about the subject and who will be taking that subject so we leave Lecturer and Subject as NULL. But all three columns together acts as a primary key, so we can't leave other two columns blank.

So to make the above table into 5NF, we can decompose it into three relations P1, P2 & P3:

**P1:**

|  |  |
| --- | --- |
| **SEMESTER** | **SUBJECT** |
| 1st Semester | Computer |
| 2nd Semester | Math |
| 3rd Semester | Chemistry |
| 4th Semester | Math |

**P2:**

|  |  |
| --- | --- |
| **SUBJECT** | **LECTURER** |
| Computer | Jessica |
| Computer | John |
| Math | John |
| Math | Robin |
| Chemistry | Natasha |

**P3:**

|  |  |
| --- | --- |
| **SEMSTER** | **LECTURER** |
| Semester 1 | Jessica |
| Semester 1 | John |
| Semester 1 | John |
| Semester 2 | Robin |
| Semester 1 | Natasha |

1. **Boyce Codd Normal Form (BCNF):**

* A BCNF is the Advanced Version of 3NF, it is stricter than 3NF.
* A Table is in the BCNF. If every function dependency X🡪 Y, X is the Super Key of the Table.
* The Table should be in 3NF, and for every FD, LHS is the Super Key.

**Example:- Let’s Assume there is a Company where employee work in more than one Department.**

**EMPLOYEE\_TABLE:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| EMP\_ID | **EMP\_COUNTRY** | **EMP\_DEPT\_NO** | **DEPT\_TYPE** | **EMP\_DEPT\_NO** |
| 264 | India | Designing | D394 | 283 |
| 264 | India | Testing | D394 | 300 |
| 364 | UK | Stores | D283 | 232 |
| 364 | UK | Developing | D283 | 549 |

In this Above Table,

**Functional Dependency are as follows:**

* 1. EMP🡪EMP\_COUNTRY
  2. EMP\_DEPT🡪{DEPT\_TYPE,EMP\_DEPT\_NO}

**Candidate Key: {EMP\_ID,EMP\_DEPT}**

* The Table Is not in BCNF, Because neither EMP\_DEPT nor EMP\_ID alone are Keys.
* To Convert the Given Table into BCNF, we decompose it into Three Table.

EMP\_COUNTRY:

|  |  |
| --- | --- |
| **EMP\_ID** | **EMP\_COUNTRY** |
| 264 | India |
| 264 | India |

**EMP\_DEPT table:**

|  |  |  |
| --- | --- | --- |
| **EMP\_DEPT** | **DEPT\_TYPE** | **EMP\_DEPT\_NO** |
| Designing | D394 | 283 |
| Testing | D394 | 300 |
| Stores | D283 | 232 |
| Developing | D283 | 549 |

**EMP\_DEPT\_MAPPING table:**

|  |  |
| --- | --- |
| **EMP\_ID** | **EMP\_DEPT** |
| D394 | 283 |
| D394 | 300 |
| D283 | 232 |
| D283 | 549 |

**WEEK=5**

**5. 1 DISCUSS WORKING OF SQL OPTIMIZER**

The job of the SQL optimizer is to determine the most efficient way to execute a given SQL query. When you submit a SQL statement to the database, the optimizer looks at the query and available indexes, statistics, and other factors to come up with a query plan that will return the results as quickly as possible.

Some key aspects of SQL optimization:

SQL OPTIMIZER

Feedback & Statistics Update

Plan Execution and Monitoring

Evaluating

Different Plans

Choosing

The Plan

Parsing the Query

Acessing

Database

Statistics

* **1. Parsing the Query:**

The optimizer first parses the SQL statement to understand the logical operations that need to be performed (joins, aggregations, filters, etc).

* **2. Accessing Database Statistics:**

The optimizer looks at statistics about the database, such as the number of rows in each table, data distribution, and index information. These provide important clues for determining the optimal plan.

* **3. Evaluating Different Plans:**

The optimizer generates multiple possible query plans, each with a cost estimate based on the stats. Different join orders, join types, access paths are evaluated.

* **4. Choosing the Plan:**

The plan with the lowest cost is chosen as the optimal plan to execute. The costs are calculated based on estimated resource usage such as I/O, CPU usage, memory, etc.

* **5. Plan Execution and Monitoring:**

Once the optimizer selects the best execution plan, the DBMS executes the query accordingly. During the execution phase, the system monitors the resource usage and query performance to ensure that the selected plan is indeed the most efficient. If necessary, the optimizer may dynamically adjust the execution plan based on real-time performance data.

* **6. Feedback and Statistics Update:**

SQL optimizers often rely on query performance feedback and statistics to continuously refine their optimization strategies. They gather information about query execution and update the statistics related to tables and indexes, enabling them to make more informed decisions in the future.

**5. 2 OPTIMIZATION TECHNIQUES**

Here are some key optimization techniques used by database management systems to improve query performance:

* **Join Order Optimization :**

Evaluating different join orders between tables and choosing the most efficient order based on statistics. Joining tables in a suboptimal order can significantly slow down queries.

* **Access Path Selection:**

Choosing whether to use an index or full table scan to retrieve rows based on factors like selectivity, index availability, etc.

* **Partition Pruning:**

Only accessing partitions of data relevant to the query instead of full tables or indexes. This technique is useful in partitioned databases.

* **Parallel Execution:**

Breaking up query processing over multiple CPUs/cores to speed up long-running queries like large aggregations.

* **Materialized Views:**

Pre-computing and storing aggregated tables to avoid complex processing for certain query patterns.

* **Index Hints:**

Using hints like INDEX, FORCE INDEX to influence index usage. Helpful when optimizer chooses a bad plan.

* **Caching frequently accessed data:**

Keeping commonly accessed data in memory caches to reduce disk I/O. Useful for small lookup tables.

* **Statistics management:**

Ensuring statistics used by the optimizer are up-to-date through periodic analysis and gathering of fresh statistics.

* **Query rewrites:**

Transforming syntax or logic of queries internally for better performance without changing semantics.

* **Avoiding expensive operators:**

Minimizing costly operators like DISTINCT, ORDER BY, UNION if possible.

The goal of all these techniques is to reduce query execution time by minimizing disk I/O, optimizing memory and CPU usage, and avoiding suboptimal plans.

**WEEK=9**

Stored procedures and functions are important features in a database management system (DBMS) that allow users to execute a set of instructions or operations within the database. It is essential to carefully weigh these advantages and disadvantages when deciding whether to use stored procedures and functions in a particular database management system. Proper planning and implementation strategies can help mitigate the potential drawbacks and maximize the benefits of using stored procedures and functions.

Below are the advantages and disadvantages of using stored procedures and functions in a DBMS:

Advantages of Stored Procedures and Functions:

1. Improved Performance: Stored procedures and functions can enhance performance by reducing the amount of data transmitted between the database and the application, thereby minimizing network traffic and latency.

2. Security: By allowing controlled access to data, stored procedures and functions can help prevent unauthorized access and protect sensitive information, improving overall data security.

3. Code Reusability: Stored procedures and functions facilitate code reusability, as they can be called multiple times from different parts of an application, reducing redundancy and promoting efficient coding practices.

4. Simplified Maintenance: Using stored procedures and functions can simplify the maintenance process, as any updates or modifications made to the stored code are automatically applied to all instances where the procedure or function is invoked.

5. Data Integrity: By encapsulating data manipulation operations within stored procedures and functions, data integrity can be maintained more effectively, ensuring consistent and accurate data updates.

Disadvantages of Stored Procedures and Functions:

1. Complexity: Creating and managing stored procedures and functions can add complexity to the database, especially for individuals who are not familiar with the specific syntax or language used in the DBMS.

2. Vendor Lock-in: Using proprietary languages or syntax for stored procedures and functions can potentially lead to vendor lock-in, limiting the flexibility to migrate to a different DBMS without significant changes to the code.

3. Debugging Challenges: Debugging stored procedures and functions can be more challenging compared to debugging standard application code, as they often execute within the database environment and may not provide detailed debugging capabilities.

4. Scalability Issues: Poorly designed or inefficiently implemented stored procedures and functions can lead to scalability issues, hindering the overall performance of the database as the volume of data and the number of users increase.

5. Version Control: Managing version control for stored procedures and functions can be complex, especially in environments where multiple developers are working on the same database, potentially leading to conflicts and inconsistencies in the codebase.

WEEK-11

Sure, here is a brief documentation on variables, data types, and control statements in PL/SQL:

1. Variables:

- In PL/SQL, variables are used to store data that can be manipulated and processed within the program.

- They must be declared before they can be used and can be of various data types.

- Variables can be declared using the `DECLARE` keyword followed by the variable name and its data type.

Example:

```sql

DECLARE

var\_name VARCHAR2(50);

BEGIN

var\_name := 'Hello, World!';

DBMS\_OUTPUT.PUT\_LINE(var\_name);

END;

```

2. \*\*Data Types:\*\*

- PL/SQL supports various data types, including:

- Numeric data types (INTEGER, NUMBER, etc.)

- Character data types (CHAR, VARCHAR2, etc.)

- Date and time data types (DATE, TIMESTAMP, etc.)

- Boolean data type

- Composite data types (RECORD, TABLE, etc.)

Example:

```sql

DECLARE

num\_val NUMBER := 10;

str\_val VARCHAR2(50) := 'Sample Text';

dt\_val DATE := SYSDATE;

BEGIN

-- Code implementation

END;

```

3. \*\*Control Statements:\*\*

- PL/SQL provides several control statements to manage the flow of execution in a program.

- Common control statements include `IF-THEN-ELSE`, `CASE`, loops (`FOR LOOP`, `WHILE LOOP`), and `EXIT` or `CONTINUE` statements.

Example of IF-THEN-ELSE:

```sql

DECLARE

num\_val NUMBER := 10;

BEGIN

IF num\_val > 0 THEN

DBMS\_OUTPUT.PUT\_LINE('Positive number');

ELSIF num\_val < 0 THEN

DBMS\_OUTPUT.PUT\_LINE('Negative number');

ELSE

DBMS\_OUTPUT.PUT\_LINE('Zero');

END IF;

END;

```

Example of FOR LOOP:

```sql

DECLARE

i NUMBER;

BEGIN

FOR i IN 1..5 LOOP

DBMS\_OUTPUT.PUT\_LINE('Iteration: ' || i);

END LOOP;

END;

```

Example of WHILE LOOP:

```sql

DECLARE

i NUMBER := 1;

BEGIN

WHILE i <= 5 LOOP

DBMS\_OUTPUT.PUT\_LINE('Iteration: ' || i);

i := i + 1;

END LOOP;

END;

```

PL/SQL provides various other features for error handling, exception handling, and modular programming, making it a powerful language for database programming and development.

Certainly! Here is an example of a stored procedure and a function in PL/SQL:

* 1. **Stored Procedure::**
* **SYNTAX CODE :**

```sql

CREATE OR REPLACE PROCEDURE procedure\_name (parameter1 datatype, parameter2 datatype, ...) IS

-- Declarations

BEGIN

-- Procedure logic

NULL;

END procedure\_name;

```

* **SYNTAX EXAMPLE:**

```sql

CREATE OR REPLACE PROCEDURE greet\_user (name\_in VARCHAR2) IS

BEGIN

DBMS\_OUTPUT.PUT\_LINE('Hello, ' || name\_in || '! Welcome to the system.');

END greet\_user;

```

* 1. Function
* **SYNTAX CODE:**

```sql

CREATE OR REPLACE FUNCTION function\_name (parameter1 datatype, parameter2 datatype, ...) RETURN return\_datatype IS

-- Declarations

BEGIN

-- Function logic

RETURN value;

END function\_name;

```

* **SYNTAX EXAMPLE:**

```sql

CREATE OR REPLACE FUNCTION calculate\_area (length\_in NUMBER, width\_in NUMBER) RETURN NUMBER IS

area\_out NUMBER;

BEGIN

area\_out := length\_in \* width\_in;

RETURN area\_out;

END calculate\_area;

```

In the examples above, the stored procedure `greet\_user` takes a `name\_in` parameter and prints a greeting message to the user. The function `calculate\_area` takes two parameters, `length\_in` and `width\_in`, and calculates the area based on the provided length and width, returning the calculated area.

The ACID properties are the four key properties that are essential for ensuring the reliability and consistency of transactions within a database. These ACID properties are crucial for maintaining data integrity, consistency, and reliability in database systems, thereby ensuring the successful execution and management of transactions. They form the foundation for robust and secure database operations, allowing users to rely on the database for critical data storage and retrieval operations.

These properties are fundamental for maintaining data integrity and ensuring the successful execution of transactions in database management systems. The ACID acronym stands for:

* + **ATOMICITY:**

Atomicity ensures that a transaction is treated as a single unit, which either occurs in its entirety or not at all. In other words, if any part of a transaction fails, the entire transaction is rolled back, and the database is restored to its previous state. This property guarantees that the database remains consistent even in the event of a system failure, ensuring that incomplete transactions do not affect the overall integrity of the data.

* + **CONSISTENCY**:

Consistency ensures that the database remains in a valid state before and after the execution of a transaction. This means that any transaction must adhere to all defined rules and constraints, preserving the consistency of the data. When a transaction is executed, the database moves from one consistent state to another consistent state, ensuring that all data constraints and integrity rules are maintained.

* + **ISOLATION:**

Isolation ensures that the concurrent execution of multiple transactions does not result in interference or data inconsistency. Transactions should be isolated from each other, and the execution of one transaction should not affect the concurrent execution of other transactions. Isolation prevents issues such as dirty reads, non-repeatable reads, and phantom reads, ensuring that the transactions are executed in a reliable and predictable manner.

* + **Durability:**

Durability ensures that once a transaction is committed, the changes made by the transaction are permanent and persist even in the event of system failures, crashes, or power outages. The changes are saved to the database and cannot be lost, ensuring that the data remains intact and consistent even after the system recovers from any failure. This property guarantees the reliability of the data and ensures that the effects of committed transactions are not lost.

Certainly! The life cycle of a transaction in a database management system (DBMS) typically follows a standard process. Here is a simplified representation of the transaction life cycle with a diagram:

* 1. Active:

The transaction begins and executes various operations. It can read and write data from the database during this phase.

* 1. Partially Committed:

The transaction reaches a point where it has executed all operations, and it is ready to be committed. However, it has not been fully committed yet.

* 1. Committed:

The transaction is successfully completed, and all changes made by the transaction are now permanently saved to the database. The changes become visible and accessible to other transactions.

* 1. Aborted:

If an issue occurs during the transaction, it may be aborted, and any changes made by the transaction are rolled back, returning the database to its state before the transaction began.

The transaction life cycle is crucial for maintaining data integrity and consistency in a DBMS. It ensures that the data remains accurate and reliable, even in the event of failures or errors during the execution of transactions.