UNIT II

RELATIONAL MODEL

Rule 1: Information rule

This rule states that all information (data), which is stored in the database, must be a value of some table cell. Everything in a database must be stored in table formats. This information can be user data or meta-data.

Rule 2: Guaranteed Access rule

This rule states that every single data element (value) is guaranteed to be accessible logically with combination of table-name, primary-key (row value) and attribute-name (column value). No other means, such as pointers, can be used to access data.

Rule 3: Systematic Treatment of NULL values

This rule states the NULL values in the database must be given a systematic treatment. As a NULL may have several meanings, i.e. NULL can be interpreted as one the following: data is missing, data is not known, data is not applicable etc.

Rule 4: Active online catalog

This rule states that the structure description of whole database must be stored in an online catalog, i.e. data dictionary, which can be accessed by the authorized users. Users can use the same query language to access the catalog which they use to access the database itself.

Rule 5: Comprehensive data sub-language rule

This rule states that a database must have a support for a language which has linear syntax which is capable of data definition, data manipulation and transaction management operations. Database can be accessed by means of this language only, either directly or by means of some application. If the database can be accessed or manipulated in some way without any help of this language, it is then a violation.

Rule 6: View updating rule

This rule states that all views of database, which can theoretically be updated, must also be updatable by the system.

Rule 7: High-level insert, update and delete rule

This rule states the database must employ support high-level insertion, updation and deletion. This must not be limited to a single row that is, it must also support union, intersection and minus operations to yield sets of data records.

Rule 8: Physical data independence

This rule states that the application should not have any concern about how the data is physically stored. Also, any change in its physical structure must not have any impact on application.

Rule 9: Logical data independence

This rule states that the logical data must be independent of its user's view (application). Any change in logical data must not imply any change in the application using it. For example, if two tables are merged or one is split into two different tables, there should be no impact the change on user application. This is one of the most difficult rule to apply.

Rule 10: Integrity independence

This rule states that the database must be independent of the application using it. All its integrity constraints can be independently modified without the need of any change in the application. This rule makes database independent of the front-end application and its interface.

Rule 11: Distribution independence

This rule states that the end user must not be able to see that the data is distributed over various locations. User must also see that data is located at one site only. This rule has been proven as a foundation of distributed database systems.

Rule 12: Non-subversion rule

This rule states that if a system has an interface that provides access to low level records, this interface then must not be able to subvert the system and bypass security and integrity constraints.

Relational data model is the primary data model, which is used widely around the world for data storage and processing. This model is simple and have all the properties and capabilities required to process data with storage efficiency.

Concepts

Tables: In relation data model, relations are saved in the format of Tables. This format stores the relation among entities. A table has rows and columns, where rows represent records and columns represents the attributes.

Tuple: A single row of a table, which contains a single record for that relation is called a tuple.

Relation instance: A finite set of tuples in the relational database system represents relation instance. Relation instances do not have duplicate tuples.

Relation schema: This describes the relation name (table name), attributes and their names.

Relation key: Each row has one or more attributes which can identify the row in the relation (table) uniquely, is called the relation key.

Attribute domain: Every attribute has some pre-defined value scope, known as attribute domain.

Constraints

Every relation has some conditions that must hold for it to be a valid relation. These conditions are called Relational Integrity Constraints. There are three main integrity constraints.

- Key Constraints
- Domain constraints
- Referential integrity constraints

Key Constraints:

There must be at least one minimal subset of attributes in the relation, which can identify a tuple uniquely. This minimal subset of attributes is called **key** for that relation. If there are more than one such minimal subsets, these are called ***candidate keys***.

Key constraints forces that:

- in a relation with a key attribute, no two tuples can have identical value for key attributes.
- key attribute can not have NULL values.

Key constrains are also referred to as Entity Constraints.

Domain constraints

Attributes have specific values in real-world scenario. For example, age can only be positive integer. The same constraints has been tried to employ on the attributes of a relation. Every attribute is bound to have a specific range of values. For example, age can not be less than zero and telephone number can not be a outside 0-9.

Referential integrity constraints

This integrity constraints works on the concept of Foreign Key. A key attribute of a relation can be referred in other relation, where it is called ***foreign key***.

Referential integrity constraint states that if a relation refers to an key attribute of a different or same relation, that key element must exists.

Relational database systems are expected to be equipped by a query language that can assist its user to query the database instances. This way its user empowers itself and can populate the results as required. There are two kinds of query languages, relational algebra and relational calculus.

Relational algebra

Relational algebra is a procedural query language, which takes instances of relations as input and yields instances of relations as output. It uses operators to perform queries. An operator can be either unary or binary. They accept relations as their input and yields relations as their output. Relational algebra is performed recursively on a relation and intermediate results are also considered relations.

Fundamental operations of Relational algebra:

- Select
- Project
- Union
- Set different
- Cartesian product
- Rename

These are defined briefly as follows:

Select Operation (σ)

Selects tuples that satisfy the given predicate from a relation.

Notation $\sigma_p(r)$

Where *p* stands for selection predicate and r stands for relation. *p* is prepositional logic formulae which may use connectors like and, or and not. These terms may use relational operators like: $=, \neq, \geq, <, >, \leq$.

For example:

$\sigma_{subject="database"}$(Books)

Output : Selects tuples from books where subject is 'database'.

$\sigma_{\text{subject="database" and price="450"}}$(Books)

Output : Selects tuples from books where subject is 'database' and 'price' is 450.

$\sigma_{\text{subject="database" and price < "450" or year > "2010"}}$(Books)

Output : Selects tuples from books where subject is 'database' and 'price' is 450 or the publication year is greater than 2010, that is published after 2010.

Project Operation (∏)

Projects column(s) that satisfy given predicate.

Notation: $\prod_{A1, A2, An} (r)$

Where a_1, a_2 , a_n are attribute names of relation r.

Duplicate rows are automatically eliminated, as relation is a set.

for example:

```
∏subject, author (Books)
```

Selects and projects columns named as subject and author from relation Books.

Union Operation (∪)

Union operation performs binary union between two given relations and is defined as:

```
r ∪ s = { t | t ∈ r or t ∈ s}
```

Notion: r U s

Where r and s are either database relations or relation result set (temporary relation).

For a union operation to be valid, the following conditions must hold:

- r, s must have same number of attributes.
- Attribute domains must be compatible.

Duplicate tuples are automatically eliminated.

```
∏ author (Books) ∪ ∏ author (Articles)
```

Output : Projects the name of author who has either written a book or an article or both.

Set Difference (−)

The result of set difference operation is tuples which present in one relation but are not in the second relation.

Notation: r − s

Finds all tuples that are present in r but not s.

```
∏ author (Books) − ∏ author (Articles)
```

Output: Results the name of authors who has written books but not articles.

Cartesian Product (X)

Combines information of two different relations into one.

Notation: r X s

Where r and s are relations and there output will be defined as:

r X s = { q t | q ∈ r and t ∈ s}

$\prod_{\text{author = 'tutorialspoint'}}$(Books X Articles)

Output : yields a relation as result which shows all books and articles written by tutorialspoint.

Rename operation (ρ)

Results of relational algebra are also relations but without any name. The rename operation allows us to rename the output relation. rename operation is denoted with small greek letter rho ρ

Notation: ρ_x (E)

Where the result of expression E is saved with name of x.

Additional operations are:

- Set intersection
- Assignment
- Natural join

Relational Calculus

In contrast with Relational Algebra, Relational Calculus is non-procedural query language, that is, it tells what to do but never explains the way, how to do it.

Relational calculus exists in two forms:

Tuple relational calculus (TRC)

Filtering variable ranges over tuples

Notation: { T | Condition }

Returns all tuples T that satisfies condition.

For Example:

```
{ T.name |  Author(T) AND T.article = 'database' }
```

Output: returns tuples with 'name' from Author who has written article on 'database'.

TRC can be quantified also. We can use Existential (∃)and Universal Quantifiers (∀).

For example:

```
{ R| ∃T   ∈ Authors(T.article='database' AND R.name=T.name)}
```

Output : the query will yield the same result as the previous one.

Domain relational calculus (DRC)

In DRC the filtering variable uses domain of attributes instead of entire tuple values (as done in TRC, mentioned above).

Notation:

$\{ a_1, a_2, a_3, ..., a_n \mid P (a_1, a_2, a_3, ... ,a_n)\}$

where a1, a2 are attributes and P stands for formulae built by inner attributes.

For example:

```
{< article, page, subject > |  ∈ TutorialsPoint ∧ subject = 'database'}
```

Output: Yields Article, Page and Subject from relation TutorialsPoint where Subject is database.

Just like TRC, DRC also can be written using existential and universal quantifiers. DRC also involves relational operators.

Expression power of Tuple relation calculus and Domain relation calculus is equivalent to Relational Algebra.

ER Model when conceptualized into diagrams gives a good overview of entity-relationship, which is easier to understand. ER diagrams can be mapped to Relational schema that is, it is possible to create relational schema using ER diagram. Though we cannot import all the ER constraints into Relational model but an approximate schema can be generated.

There are more than one processes and algorithms available to convert ER Diagrams into Relational Schema. Some of them are automated and some of them are manual process. We may focus here on the mapping diagram contents to relational basics.

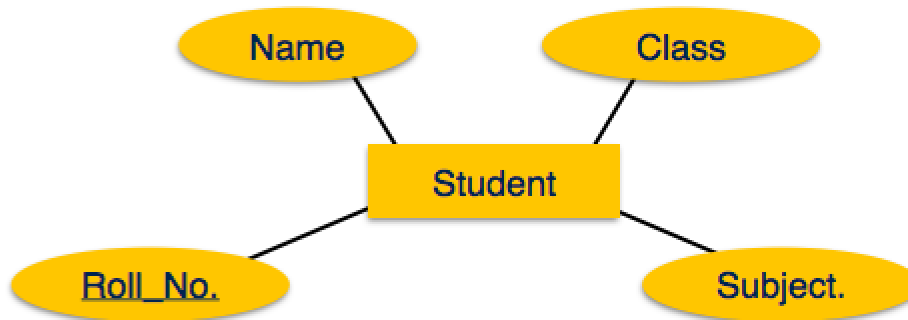
ER Diagrams mainly comprised of:

- Entity and its attributes
- Relationship, which is association among entities.

Mapping Entity

An entity is a real world object with some attributes.

Mapping Process (Algorithm):



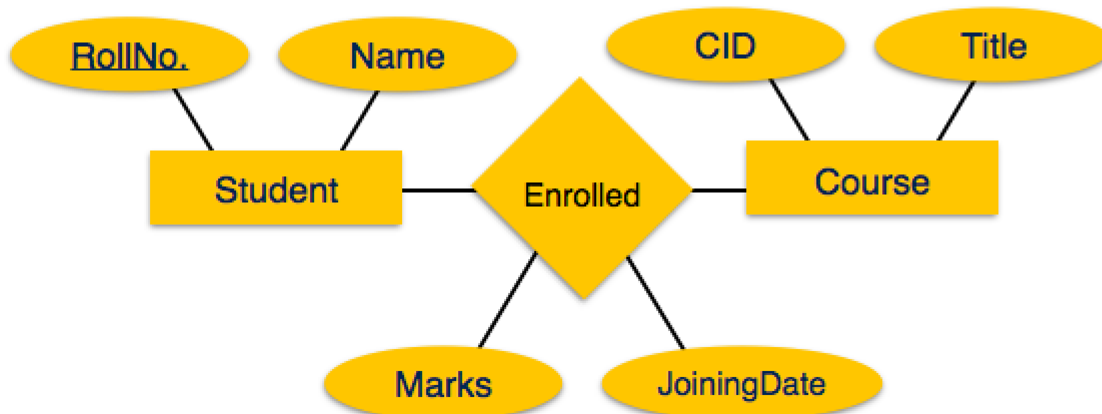
[Image: Mapping Entity]

- Create table for each entity
- Entity's attributes should become fields of tables with their respective data types.
- Declare primary key

Mapping relationship

A relationship is association among entities.

Mapping process (Algorithm):



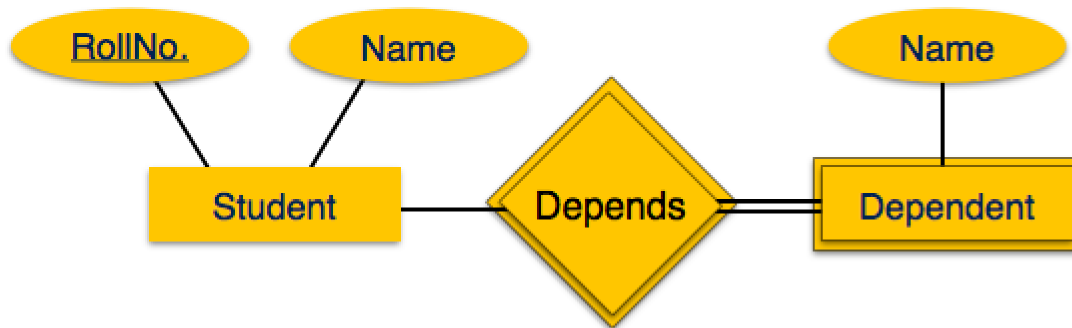
[Image: Mapping relationship]

- Create table for a relationship
- Add the primary keys of all participating Entities as fields of table with their respective data types.
- If relationship has any attribute, add each attribute as field of table.
- Declare a primary key composing all the primary keys of participating entities.
- Declare all foreign key constraints.

Mapping Weak Entity Sets

A weak entity sets is one which does not have any primary key associated with it.

Mapping process (Algorithm):



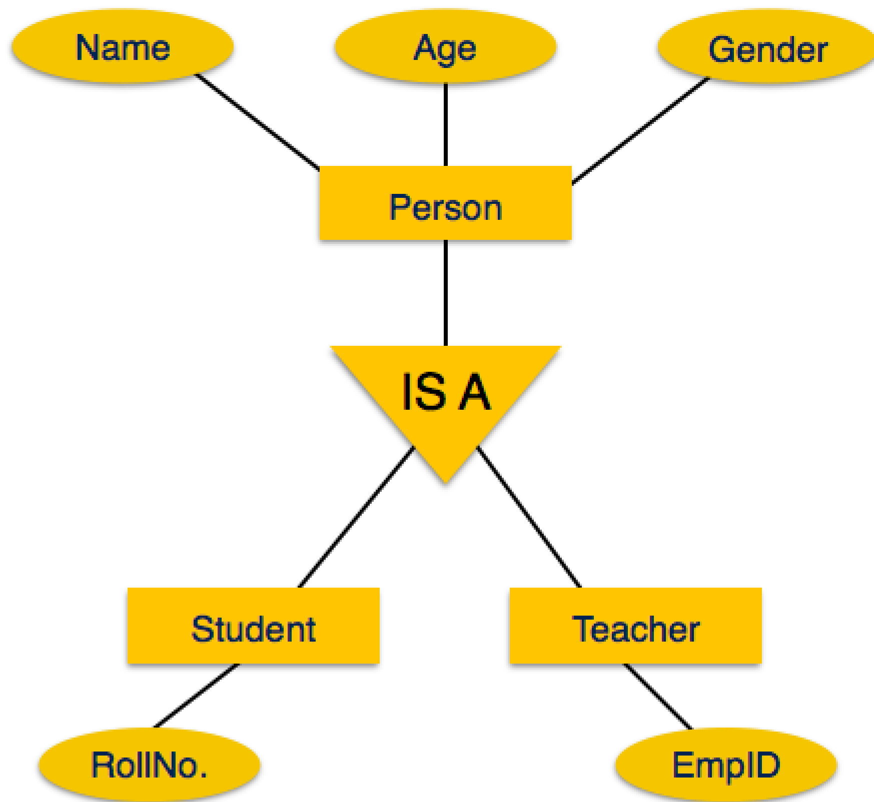
[Image: Mapping Weak Entity Sets]

- Create table for weak entity set
- Add all its attributes to table as field
- Add the primary key of identifying entity set
- Declare all foreign key constraints

Mapping hierarchical entities

ER specialization or generalization comes in the form of hierarchical entity sets.

Mapping process (Algorithm):



[*Image: Mapping hierarchical entities*]

- Create tables for all higher level entities
- Create tables for lower level entities
- Add primary keys of higher level entities in the table of lower level entities
- In lower level tables, add all other attributes of lower entities.
- Declare primary key of higher level table the primary key for lower level table
- Declare foreign key constraints.

SQL Overview

SQL is a programming language for Relational Databases. It is designed over relational algebra and tuple relational calculus. SQL comes as a package with all major distributions of RDBMS.

SQL comprises both data definition and data manipulation languages. Using the data definition properties of SQL, one can design and modify database schema whereas data manipulation properties allows SQL to store and retrieve data from database.

Data definition Language

SQL uses the following set of commands to define database schema:

CREATE

Creates new databases, tables and views from RDBMS

For example:
```
Create database tutorialspoint;
Create table article;
Create view for_students;
```

DROP

Drop commands deletes views, tables and databases from RDBMS
```
Drop object_type object_name;
Drop database tutorialspoint;
Drop table article;
Drop view for_students;
```

ALTER

Modifies database schema.
```
Alter object_type object_name parameters;
```

for example:
```
Alter table article add subject varchar;
```
This command adds an attribute in relation article with name subject of string type.

Data Manipulation Language

SQL is equipped with data manipulation language. DML modifies the database instance by inserting, updating and deleting its data. DML is responsible for all data modification in databases. SQL contains the following set of command in DML section:

- SELECT/FROM/WHERE
- INSERT INTO/VALUES
- UPDATE/SET/WHERE
- DELETE FROM/WHERE

These basic constructs allows database programmers and users to enter data and information into the database and retrieve efficiently using a number of filter options.

SELECT/FROM/WHERE

- **SELECT**

  This is one of the fundamental query command of SQL. It is similar to projection operation of relational algebra. It selects the attributes based on the condition described by WHERE clause.

- **FROM**

This clause takes a relation name as an argument from which attributes are to be selected/projected. In case more than one relation names are given this clause corresponds to cartesian product.

- **WHERE**

  This clause defines predicate or conditions which must match in order to qualify the attributes to be projected.

  For example:
  ```
  Select author_name
  From book_author
  Where age > 50;
  ```

  This command will project names of author's from book_author relation whose age is greater than 50.

INSERT INTO/VALUES

This command is used for inserting values into rows of table (relation).

yntax is
```
INSERT INTO table (column1 [, column2, column3 ... ]) VALUES (value1 [,
value2, value3 ... ])
```

Or
```
INSERT INTO table VALUES (value1, [value2, ... ])
```

For Example:
```
INSERT INTO tutorialspoint (Author, Subject) VALUES ("anonymous",
"computers");
```

UPDATE/SET/WHERE

This command is used for updating or modifying values of columns of table (relation).

Syntax is
```
UPDATE table_name SET column_name = value [, column_name = value ...] [WHERE
condition]
```

For example:
```
UPDATE tutorialspoint SET Author="webmaster" WHERE Author="anonymous";
```

DELETE/FROM/WHERE

This command is used for removing one or more rows from table (relation).

Syntax is

```
DELETE FROM table_name [WHERE condition];
```

For example:

```
DELETE FROM tutorialspoints
  WHERE Author="unknown";
```

For in-depth and practical knowledge of SQL.

UNIT IV

DBMS Normalization

Functional Dependency

Functional dependency (FD) is set of constraints between two attributes in a relation. Functional dependency says that if two tuples have same values for attributes A1, A2,..., An then those two tuples must have to have same values for attributes B1, B2, ..., Bn.

Functional dependency is represented by arrow sign (→), that is X→Y, where X functionally determines Y. The left hand side attributes determines the values of attributes at right hand side.

Armstrong's Axioms

If F is set of functional dependencies then the closure of F, denoted as F^+, is the set of all functional dependencies logically implied by F. Armstrong's Axioms are set of rules, when applied repeatedly generates closure of functional dependencies.

- **Reflexive rule:** If alpha is a set of attributes and beta is_subset_of alpha, then alpha holds beta.
- **Augmentation rule:** if a → b holds and y is attribute set, then ay → by also holds. That is adding attributes in dependencies, does not change the basic dependencies.
- **Transitivity rule:** Same as transitive rule in algebra, if a → b holds and b → c holds then a → c also hold. a → b is called as a functionally determines b.

Trivial Functional Dependency

- **Trivial:** If an FD X → Y holds where Y subset of X, then it is called a trivial FD. Trivial FDs are always hold.
- **Non-trivial:** If an FD X → Y holds where Y is not subset of X, then it is called non-trivial FD.
- **Completely non-trivial:** If an FD X → Y holds where x intersect Y = Φ, is said to be completely non-trivial FD.

Normalization

If a database design is not perfect it may contain anomalies, which are like a bad dream for database itself. Managing a database with anomalies is next to impossible.

- **Update anomalies:** if data items are scattered and are not linked to each other properly, then there may be instances when we try to update one data item that has copies of it scattered at several places, few instances of it get updated properly while few are left with there old values. This leaves database in an inconsistent state.
- **Deletion anomalies:** we tried to delete a record, but parts of it left undeleted because of unawareness, the data is also saved somewhere else.
- **Insert anomalies:** we tried to insert data in a record that does not exist at all.

Normalization is a method to remove all these anomalies and bring database to consistent state and free from any kinds of anomalies.

First Normal Form:

This is defined in the definition of relations (tables) itself. This rule defines that all the attributes in a relation must have atomic domains. Values in atomic domain are indivisible units.

Course	Content
Programming	Java, c++
Web	HTML, PHP, ASP

[*Image: Unorganized relation*]

We re-arrange the relation (table) as below, to convert it to First Normal Form

Course	Content
Programming	Java
Programming	c++
Web	HTML
Web	PHP
Web	ASP

[*Image: Relation in 1NF*]

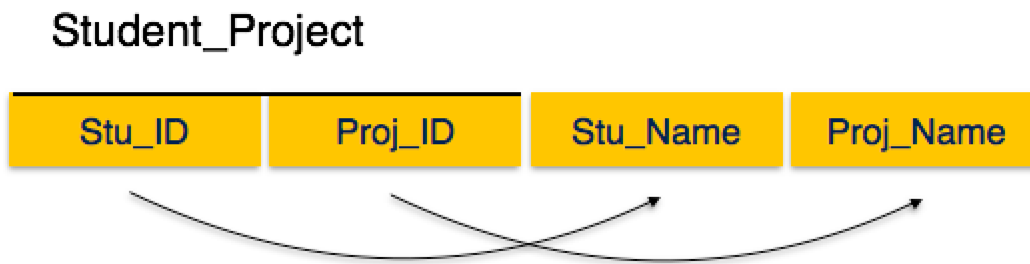
Each attribute must contain only single value from its pre-defined domain.

Second Normal Form:

Before we learn about second normal form, we need to understand the following:

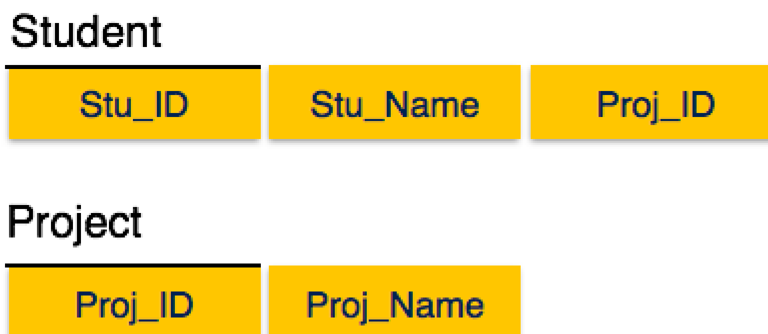
- **Prime attribute:** an attribute, which is part of prime-key, is prime attribute.
- **Non-prime attribute:** an attribute, which is not a part of prime-key, is said to be a non-prime attribute.

Second normal form says, that every non-prime attribute should be fully functionally dependent on prime key attribute. That is, if X → A holds, then there should not be any proper subset Y of X, for that Y → A also holds.



[Image: Relation not in 2NF]

We see here in Student_Project relation that the prime key attributes are Stu_ID and Proj_ID. According to the rule, non-key attributes, i.e. Stu_Name and Proj_Name must be dependent upon both and not on any of the prime key attribute individually. But we find that Stu_Name can be identified by Stu_ID and Proj_Name can be identified by Proj_ID independently. This is called partial dependency, which is not allowed in Second Normal Form.



[Image: Relation in 2NF]

We broke the relation in two as depicted in the above picture. So there exists no partial dependency.

Third Normal Form:

For a relation to be in Third Normal Form, it must be in Second Normal form and the following must satisfy:

- No non-prime attribute is transitively dependent on prime key attribute
- For any non-trivial functional dependency, X → A, then either
- X is a superkey or,
- A is prime attribute.

[Image: Relation not in 3NF]

We find that in above depicted Student_detail relation, Stu_ID is key and only prime key attribute. We find that City can be identified by Stu_ID as well as Zip itself. Neither Zip is a superkey nor City is a prime attribute. Additionally, Stu_ID → Zip → City, so there exists ***transitive dependency***.

[Image: Relation in 3NF]

We broke the relation as above depicted two relations to bring it into 3NF.

Boyce-Codd Normal Form:

BCNF is an extension of Third Normal Form in strict way. BCNF states that

- For any non-trivial functional dependency, X → A, then X must be a super-key.

In the above depicted picture, Stu_ID is super-key in Student_Detail relation and Zip is super-key in ZipCodes relation. So,

Stu_ID → Stu_Name, Zip

And

Zip → City

Confirms, that both relations are in BCNF.

DBMS Joins

We understand the benefits of Cartesian product of two relation, which gives us all the possible tuples that are paired together. But Cartesian product might not be feasible for huge relations where number of tuples are in thousands and the attributes of both relations are considerable large.

Join is combination of Cartesian product followed by selection process. Join operation pairs two tuples from different relations if and only if the given join condition is satisfied.

Following section should describe briefly about join types:

Theta (θ) join

θ in Theta join is the join condition. Theta joins combines tuples from different relations provided they satisfy the theta condition.

Notation:

$R1 \bowtie_{\theta} R2$

R1 and R2 are relations with their attributes (A1, A2, .., An) and (B1, B2,.. ,Bn) such that no attribute matches that is R1 ∩ R2 = Φ Here θ is condition in form of set of conditions C.

Theta join can use all kinds of comparison operators.

Student

SID	Name	Std
101	Alex	10
102	Maria	11

[*Table: Student Relation*]

Subjects

Class	Subject
10	Math
10	English

11	Music
11	Sports

[*Table: Subjects Relation*]

Student_Detail =
STUDENT $\bowtie_{Student.Std = Subject.Class}$ SUBJECT

Student_detail

SID	Name	Std	Class	Subject
101	Alex	10	10	Math
101	Alex	10	10	English
102	Maria	11	11	Music
102	Maria	11	11	Sports

[*Table: Output of theta join*]

Equi-Join

When Theta join uses only **equality** comparison operator it is said to be Equi-Join. The above example conrresponds to equi-join

Natural Join (⋈)

Natural join does not use any comparison operator. It does not concatenate the way Cartesian product does. Instead, Natural Join can only be performed if the there is at least one common attribute exists between relation. Those attributes must have same name and domain.

Natural join acts on those matching attributes where the values of attributes in both relation is same.

Courses

CID	Course	Dept
CS01	Database	CS
ME01	Mechanics	ME
EE01	Electronics	EE

[*Table: Relation Courses*]

HoD

Dept	Head
CS	Alex
ME	Maya
EE	Mira

[*Table: Relation HoD*]

Courses ⋈ HoD

Dept	CID	Course	Head

CS	CS01	Database	Alex
ME	ME01	Mechanics	Maya
EE	EE01	Electronics	Mira

[*Table: Relation Courses ⋈ HoD*]

Outer Joins

All joins mentioned above, that is Theta Join, Equi Join and Natural Join are called inner-joins. An inner-join process includes only tuples with matching attributes, rest are discarded in resulting relation. There exists methods by which all tuples of any relation are included in the resulting relation.

There are three kinds of outer joins:

Left outer join (R S)

All tuples of Left relation, R, are included in the resulting relation and if there exists tuples in R without any matching tuple in S then the S-attributes of resulting relation are made NULL.

Left	
A	**B**
100	Database
101	Mechanics
102	Electronics

[*Table: Left Relation*]

Right	
A	**B**
100	Alex
102	Maya
104	Mira

[*Table: Right Relation*]

Courses		HoD	
A	**B**	**C**	**D**
100	Database	100	Alex
101	Mechanics	---	---
102	Electronics	102	Maya

[*Table: Left outer join output*]

Right outer join: (R S)

All tuples of the Right relation, S, are included in the resulting relation and if there exists tuples in S without any matching tuple in R then the R-attributes of resulting relation are made NULL.

Courses		HoD	
A	**B**	**C**	**D**
100	Database	100	Alex
102	Electronics	102	Maya
---	---	104	Mira

[*Table: Right outer join output*]

Full outer join: (R S)

All tuples of both participating relations are included in the resulting relation and if there no matching tuples for both relations, their respective unmatched attributes are made NULL.

Courses		HoD	
A	**B**	**C**	**D**
100	Database	100	Alex
101	Mechanics	---	---
102	Electronics	102	Maya
---	---	104	Mira

[*Table: Full outer join output*]

Complex SQL Queries

1. Employees Table

This table is used in many examples, such as `GROUP BY`, `HAVING`, `SELECT INTO`, etc.

employee_id	employee_name	department	salary
1	John Doe	HR	60000
2	Jane Smith	IT	75000
3	Sam Brown	IT	85000
4	Lisa White	HR	45000
5	James Black	Finance	70000
6	Emily Green	IT	95000
7	Michael Blue	HR	40000
8	Sarah Gray	Finance	65000

2. Department Table

This is used for examples of `EXISTS` and some subqueries.

department_id	department_name
1	HR
2	IT
3	Finance

3. Employees Backup Table

This table might be created via `SELECT INTO` and `INSERT INTO SELECT`.

employee_name	salary
John Doe	60000
Jane Smith	75000
Sam Brown	85000
Lisa White	45000
James Black	70000
Emily Green	95000
Michael Blue	40000
Sarah Gray	65000

4. Employee Salary Range Query Result (using `CASE`)

employee_name	salary_range
John Doe	High
Jane Smith	High
Sam Brown	High
Lisa White	Medium
James Black	High
Emily Green	High
Michael Blue	Low
Sarah Gray	Medium

5. Employees Table after NULL Handling (using `COALESCE`)

Assume that `salary` has some `NULL` values in certain rows.

employee_id	employee_name	department	salary
1	John Doe	HR	60000
2	Jane Smith	IT	NULL
3	Sam Brown	IT	85000
4	Lisa White	HR	45000
5	James Black	Finance	NULL
6	Emily Green	IT	95000
7	Michael Blue	HR	40000
8	Sarah Gray	Finance	65000

Using the `COALESCE()` function to replace `NULL` with 0 for salaries would give the following output:

employee_name	salary
John Doe	60000
Jane Smith	0
Sam Brown	85000
Lisa White	45000
James Black	0
Emily Green	95000
Michael Blue	40000
Sarah Gray	65000

6. Stored Procedure Query Result (Get Employees by Department)

If you execute the following stored procedure:

```
EXEC GetEmployeesByDepartment 'IT';
```

It would return:

employee_name
Jane Smith
Sam Brown
Emily Green

7. Employees Table with a Subquery (using `EXISTS`)

When checking if employees belong to any department (using the `EXISTS` operator):

employee_name
John Doe
Jane Smith
Sam Brown
Lisa White
James Black
Emily Green
Michael Blue
Sarah Gray

(The query would return all employees who belong to a department, so all rows are returned because every employee is assigned to a department.)

8. Employees Table (with Salary Comparison using ANY or ALL)

Using ANY:

If we check if an employee's salary is greater than any salary in the 'HR' department:

```
SELECT employee_name FROM employees WHERE salary > ANY (SELECT salary FROM
employees WHERE department = 'HR');
```

This query would return employees whose salary is greater than any HR employee's salary.

employee_name
Jane Smith
Sam Brown
Emily Green

Using ALL:

If we check if an employee's salary is greater than all salaries in the 'HR' department:

```
SELECT employee_name FROM employees WHERE salary > ALL (SELECT salary FROM
employees WHERE department = 'HR');
```

This query would return employees whose salary is greater than all HR employees' salaries.

employee_name
Sam Brown
Emily Green

1. SQL GROUP BY Statement

- **Purpose**: The `GROUP BY` statement groups rows that have the same values into summary rows, such as total, average, count, etc.

Query:

```
SELECT department, COUNT(*) AS employee_count
FROM employees
GROUP BY department;
```

Result Table:

department	employee_count
HR	3
IT	3
Finance	2

Explanation: Groups employees by department and counts how many employees are in each department.

2. SQL HAVING Clause

- **Purpose**: The `HAVING` clause is used to filter records after grouping with `GROUP BY`. It works with aggregate functions.

Query:

```sql
SELECT department, COUNT(*) AS employee_count
FROM employees
GROUP BY department
HAVING COUNT(*) > 2;
```

Result Table:

department	employee_count
IT	3

Explanation: Filters out departments with less than or equal to 2 employees, leaving only the "IT" department.

3. SQL EXISTS Operator

- **Purpose**: The `EXISTS` operator checks if a subquery returns any rows. It returns `TRUE` if the subquery returns one or more records.

Query:

```sql
SELECT employee_name
FROM employees e
WHERE EXISTS (
    SELECT 1
    FROM department d
    WHERE d.department_id = e.department_id
);
```

Result Table:

employee_name
John Doe
Jane Smith
Sam Brown
Lisa White
James Black
Emily Green
Michael Blue
Sarah Gray

Explanation: This query checks if each employee has an associated department in the `department` table. It will return all employees since all employees belong to a department.

4. SQL ANY and ALL Operators

- **Purpose**: `ANY` and `ALL` are used to compare a value with a set of results from a subquery.
- **`ANY`**: Checks if the condition is met for **any** of the values returned by the subquery.
- **`ALL`**: Checks if the condition is met for **all** of the values returned by the subquery.

Query for `ANY`:

```
sql

SELECT employee_name
FROM employees
WHERE salary > ANY (
    SELECT salary
    FROM employees
    WHERE department = 'HR'
);
```

Result Table for `ANY`:

employee_name
Jane Smith
Sam Brown
Emily Green

Explanation: This query returns employees whose salary is greater than any salary in the HR department.

Query for `ALL`:

```
sql

SELECT employee_name
FROM employees
WHERE salary > ALL (
    SELECT salary
    FROM employees
    WHERE department = 'HR'
);
```

Result Table for `ALL`:

employee_name
Sam Brown
Emily Green

Explanation: This query returns employees whose salary is greater than all salaries in the HR department.

5. SQL SELECT INTO Statement

- **Purpose**: The `SELECT INTO` statement creates a new table and inserts the result set from a query into it.

```
SELECT employee_name, salary INTO employees_backup
FROM employees;
```

Result Table (`employees_backup`):

employee_name	salary
John Doe	60000
Jane Smith	75000
Sam Brown	85000
Lisa White	45000
James Black	70000
Emily Green	95000
Michael Blue	40000
Sarah Gray	65000

Explanation: This query creates a new table (`employees_backup`) and inserts data from the `employees` table into it.

6. SQL INSERT INTO SELECT Statement

- **Purpose**: The `INSERT INTO SELECT` statement allows you to insert data into a table from the result of a query.

```
INSERT INTO employees_backup (employee_name, salary)
SELECT employee_name, salary
FROM employees
WHERE department = 'HR';
```

Result Table (`employees_backup`):

employee_name	salary
John Doe	60000
Jane Smith	75000
Sam Brown	85000
Lisa White	45000
James Black	70000
Emily Green	95000
Michael Blue	40000
Sarah Gray	65000
John Doe	60000
Lisa White	45000
Michael Blue	40000

Explanation: This query inserts employees from the HR department into the `employees_backup` table. These rows are added again, making it a duplicate.

7. SQL CASE Expression

- **Purpose**: The `CASE` expression allows you to return a value based on conditions. It can be used in SELECT, UPDATE, or DELETE statements.

Query:

```sql
SELECT employee_name,
       CASE
           WHEN salary > 50000 THEN 'High'
           WHEN salary BETWEEN 30000 AND 50000 THEN 'Medium'
           ELSE 'Low'
       END AS salary_range
FROM employees;
```

Result Table:

employee_name	salary_range
John Doe	High
Jane Smith	High
Sam Brown	High
Lisa White	Medium
James Black	High
Emily Green	High
Michael Blue	Low
Sarah Gray	Medium

Explanation: The query categorizes employees into "High", "Medium", and "Low" salary ranges.

8. SQL NULL Functions

- **Purpose**: Functions like `COALESCE()` or `ISNULL()` are used to handle NULL values in SQL queries.

Query (Using `COALESCE`):

```sql
SELECT employee_name, COALESCE(salary, 0) AS salary
FROM employees;
```

Result Table:

employee_name	salary
John Doe	60000
Jane Smith	0
Sam Brown	85000
Lisa White	45000
James Black	0
Emily Green	95000
Michael Blue	40000
Sarah Gray	65000

Explanation: Replaces `NULL` values in the `salary` column with 0 using `COALESCE()`.

9. SQL Stored Procedures for SQL Server

- **Purpose**: A stored procedure is a set of SQL statements that can be executed as a single unit to perform a specific task.

Example Stored Procedure:

```
sql

CREATE PROCEDURE GetEmployeesByDepartment
@department_name VARCHAR(50)
AS
BEGIN
    SELECT employee_name, department
    FROM employees
    WHERE department = @department_name;
END;
```

Calling the Stored Procedure:

```
sql

EXEC GetEmployeesByDepartment 'IT';
```

Result Table:

employee_name
Jane Smith
Sam Brown
Emily Green

Explanation: The stored procedure `GetEmployeesByDepartment` takes a department name as input and returns employees in that department.

10. SQL Comments

- **Purpose**: Comments are used to document SQL code and improve readability. Comments are ignored during execution.

Example:

```
sql

-- This query retrieves all employees from HR
SELECT employee_name
FROM employees
WHERE department = 'HR';
```

Result Table:

employee_name
John Doe
Lisa White
Michael Blue

Explanation: The query is the same as shown earlier for selecting employees from the HR department. The comment is ignored during query execution.

11. SQL Operators

- **Purpose**: Operators in SQL are used to perform operations on data, such as comparison, logical operations, etc.

Common SQL Operators:

- `=, >, <, >=, <=, <>` (Comparison Operators)
- `AND`, `OR`, `NOT` (Logical Operators)
- `IN`, `BETWEEN`, `LIKE` (Range and Pattern Matching)

Example (Using `IN` Operator):

```sql
sql

SELECT employee_name
FROM employees
WHERE department IN ('HR', 'IT');
```

Result Table:

employee_name
John Doe
Jane Smith
Sam Brown
Lisa White
Emily Green

Explanation: This query returns employees working in the 'HR' or 'IT' department using the `IN` operator.

Here are some complex SQL queries along with their answers using example data and explanations:

1. Find the Employees with Highest Salary in Each Department

Problem:
You have a table `Employees` with columns `employee_id`, `employee_name`, `salary`, and `department_id`. You want to find the employee with the highest salary in each department.

SQL Query:
```
SELECT department_id, employee_name, salary
FROM (
    SELECT department_id, employee_name, salary,
           RANK() OVER (PARTITION BY department_id ORDER BY salary DESC) AS
rank
    FROM Employees
) AS ranked_employees
WHERE rank = 1;
```

Explanation:

- `RANK()` is a window function that ranks employees based on their salary within each department (`PARTITION BY department_id`).
- The result is filtered by `rank = 1`, which returns the highest-paid employee in each department.

Sample Data:

employee_id	employee_name	salary	department_id
1	Alice	80000	1
2	Bob	95000	1
3	Charlie	70000	2
4	David	75000	2
5	Eve	60000	3

Result:

department_id	employee_name	salary
1	Bob	95000
2	David	75000
3	Eve	60000

---

2. Find the Total Sales for Each Product and Identify Top-Selling Products

Problem:
You have two tables:

- `Sales` table with columns `product_id`, `quantity_sold`, and `sale_date`.
- `Products` table with columns `product_id` and `product_name`.
  You need to find the total sales (`quantity_sold`) for each product and identify products that sold more than 100 units.

SQL Query:

```
SELECT p.product_name, SUM(s.quantity_sold) AS total_sales
FROM Sales s
JOIN Products p ON s.product_id = p.product_id
GROUP BY p.product_name
HAVING SUM(s.quantity_sold) > 100;
```

Explanation:

- The query joins `Sales` and `Products` on `product_id`.
- The `GROUP BY` clause groups the sales data by `product_name`, and `SUM()` calculates the total sales for each product.
- The `HAVING` clause filters products with total sales greater than 100 units.

Sample Data:

`Sales` table:

product_id	quantity_sold	sale_date
1	50	2024-11-01
2	120	2024-11-01
1	30	2024-11-02
3	60	2024-11-02
2	40	2024-11-03

`Products` table:

product_id	product_name
1	Product A
2	Product B
3	Product C

Result:

product_name	total_sales
Product B	160

3. Find Customers Who Have Not Placed Any Orders

Problem:
You have two tables:

- `Customers` table with columns `customer_id` and `customer_name`.
- `Orders` table with columns `order_id`, `customer_id`, and `order_date`.
  You need to find customers who have never placed an order.

SQL Query:
```
SELECT customer_name
FROM Customers c
WHERE NOT EXISTS (
    SELECT 1
    FROM Orders o
    WHERE o.customer_id = c.customer_id
);
```

Explanation:

- The `NOT EXISTS` subquery checks for customers who do not have matching records in the `Orders` table.
- If no orders exist for a customer, the `customer_name` is returned.

Sample Data:
`Customers` table:

customer_id	customer_name
1	Alice
2	Bob
3	Charlie

`Orders` table:

order_id	customer_id	order_date
1	1	2024-11-01
2	2	2024-11-03

Result:

customer_name
Charlie

---

4. Find the Department with the Highest Average Salary

Problem:
You have a table `Employees` with columns `employee_id`, `employee_name`, `salary`, and `department_id`. You need to find the department with the highest average salary.

SQL Query:

```
sql
Copy code
SELECT department_id, AVG(salary) AS avg_salary
FROM Employees
GROUP BY department_id
HAVING AVG(salary) = (
    SELECT MAX(avg_salary)
    FROM (
        SELECT AVG(salary) AS avg_salary
        FROM Employees
        GROUP BY department_id
    ) AS avg_salaries
);
```

Explanation:

- The inner query calculates the average salary for each department.
- The outer query finds the department(s) with the maximum average salary by comparing the `AVG(salary)` with the maximum of all department averages.

Sample Data:

employee_id	employee_name	salary	department_id
1	Alice	80000	1
2	Bob	95000	1
3	Charlie	70000	2
4	David	75000	2
5	Eve	60000	3

Result:

department_id	avg_salary
1	87500

---

5. Find the Second Highest Salary

Problem:
You need to find the second-highest salary from the `Employees` table.

SQL Query:

```
SELECT MAX(salary) AS second_highest_salary
FROM Employees
WHERE salary < (SELECT MAX(salary) FROM Employees);
```

Explanation:

- The inner query finds the highest salary.
- The outer query finds the maximum salary that is less than the highest salary, which is the second-highest.

Sample Data:

employee_id	employee_name	salary
1	Alice	80000
2	Bob	95000
3	Charlie	70000
4	David	75000
5	Eve	60000

Result:

second_highest_salary
80000

---

6. Find Products That Have Never Been Sold

Problem:
You have two tables:

- `Products` table with columns `product_id` and `product_name`.
- `Sales` table with columns `product_id`, `quantity_sold`, and `sale_date`.
  You need to find products that have never been sold.

SQL Query:

```
SELECT product_name
FROM Products p
WHERE NOT EXISTS (
    SELECT 1
    FROM Sales s
    WHERE s.product_id = p.product_id
);
```

Explanation:

- The `NOT EXISTS` subquery checks if a product exists in the `Sales` table.
- If no sales exist for a product, the product is returned.

Sample Data:

`Products` table:

product_id	product_name
1	Product A
2	Product B
3	Product C

`Sales` table:

product_id	quantity_sold	sale_date
1	50	2024-11-01
2	30	2024-11-02

Result:

product_name
Product C

These complex SQL queries demonstrate how to perform advanced operations like subqueries, joins, window functions, and aggregation. Each query tackles different challenges and real-world scenarios.

Views in DBMS:

In Database Management Systems (DBMS), a **view** is a virtual table that is derived from one or more base tables or other views. A view doesn't store the data itself; rather, it stores a query that, when executed, retrieves the data dynamically from the underlying tables. Views are often used to simplify complex queries, enhance security by restricting access to specific columns or rows, and provide a way to present data in a customized manner.

Key Features of Views:

1. **Virtual Table:**
   A view behaves like a table, but it doesn't store data physically. The data in a view is dynamically retrieved from the base tables whenever the view is queried.
2. **Simplicity:**
   Views can simplify complex queries by encapsulating them into a single object. Users can query the view as if it were a regular table.
3. **Security:**
   Views can restrict access to sensitive data. By creating views that exclude certain columns or rows, you can control the information that different users can access.
4. **Data Abstraction:**
   Views provide an abstraction layer between users and the database schema, allowing you to change the underlying structure (e.g., table names or joins) without affecting user queries.
5. **Updatable Views:**
   Some views are updatable, meaning you can perform `INSERT`, `UPDATE`, or `DELETE` operations on the base tables through the view. However, the view must meet certain criteria to be updatable (e.g., it must be a simple view with no aggregate functions or joins).

Types of Views:

1. **Simple Views:**
   A simple view is based on a single table and doesn't contain complex calculations, aggregate functions, or joins. It is usually updatable.
   Example:

   ```
   CREATE VIEW EmployeeNames AS
   SELECT employee_name, department_id
   FROM Employees;
   ```

   In this example, the view `EmployeeNames` selects the `employee_name` and `department_id` from the `Employees` table.
2. **Complex Views:**
   A complex view is based on multiple tables and may include joins, aggregate functions, or subqueries. These views are often used to provide summarized or combined data from various tables.
   Example:

   ```
   CREATE VIEW DepartmentSalarySummary AS
   SELECT department_id, AVG(salary) AS avg_salary
   FROM Employees
   GROUP BY department_id;
   ```

This view `DepartmentSalarySummary` calculates the average salary for each department, combining data from the `Employees` table.

3. **Materialized Views (in some DBMS):**
   Unlike regular views, materialized views store the actual data. They are used to improve performance for complex queries by precomputing the result set and storing it. Materialized views can be refreshed periodically to reflect changes in the underlying tables.

Creating a View:

The basic syntax for creating a view is:

```
CREATE VIEW view_name AS
SELECT columns
FROM tables
WHERE conditions;
```

Example:

```
CREATE VIEW HighSalaryEmployees AS
SELECT employee_name, salary
FROM Employees
WHERE salary > 80000;
```

This view `HighSalaryEmployees` will contain a list of employees who earn more than 80,000.

Viewing the Definition of a View:

You can check the definition of a view using a `SHOW` or `DESCRIBE` statement, depending on the DBMS.

```
SHOW CREATE VIEW HighSalaryEmployees;
```

Querying a View:

Once a view is created, you can query it just like a regular table:

```
SELECT * FROM HighSalaryEmployees;
```

This will retrieve all the rows from the `HighSalaryEmployees` view.

Updating Data Through Views:

Some views allow you to perform `INSERT`, `UPDATE`, or `DELETE` operations on the underlying tables. However, only views that meet certain conditions are updatable:

- The view must reference only one table.
- The view must not include aggregate functions (e.g., `COUNT()`, `SUM()`).
- The view must not include `DISTINCT`, `GROUP BY`, or `HAVING` clauses.

Example of an Updatable View:

```
CREATE VIEW EmployeeDetails AS
SELECT employee_id, employee_name, salary
FROM Employees;
```

Now, you can perform `UPDATE` operations on this view, and it will modify the data in the base `Employees` table.

```
UPDATE EmployeeDetails
SET salary = 95000
WHERE employee_id = 2;
```

Dropping a View:

If you no longer need a view, you can delete it using the `DROP VIEW` statement:

```
DROP VIEW HighSalaryEmployees;
```

Benefits of Using Views:

1. **Simplify Complex Queries:**
   Views can simplify repetitive complex queries, making them easier to reuse.
2. **Encapsulate Business Logic:**
   Views can encapsulate business rules, filters, and logic that apply to the data. This helps ensure consistency across different users and applications.
3. **Security Control:**
   Views allow you to control access to sensitive data by exposing only specific columns or rows to different users.
4. **Data Integrity:**
   By using views to aggregate or summarize data, you can ensure that data integrity is maintained across different applications.

Limitations of Views:

1. **Performance:**
   Since views are virtual and do not store data, querying complex views can sometimes lead to performance issues, especially if they involve joins or aggregations.
2. **Updatability:**
   Not all views are updatable, especially those involving complex queries with joins, aggregations, or multiple tables.
3. **Materialized Views:**
   While materialized views store data and provide better performance, they require additional storage space and need to be periodically refreshed to ensure they reflect changes in the underlying tables.

Examples of View Usage:

Example 1: View for Employee Salaries by Department

```
CREATE VIEW DepartmentSalary AS
SELECT department_id, AVG(salary) AS avg_salary
FROM Employees
GROUP BY department_id;
```

This view calculates the average salary for each department.

Example 2: View for Sales Performance

```
CREATE VIEW SalesPerformance AS
SELECT salesperson_id, SUM(sales_amount) AS total_sales
FROM Sales
GROUP BY salesperson_id;
```

This view summarizes the total sales for each salesperson.

Conclusion:

Views in DBMS provide an efficient way to simplify complex queries, abstract data, and enhance security. They can be used for reporting, summarization, or controlling access to sensitive data. However, it's essential to understand the underlying limitations, especially concerning performance and updatability.

Constraints on Relational Database Model

In modeling the design of the relational database we can put some restrictions like what values are allowed to be inserted in the relation, and what kind of modifications and deletions are allowed in the relation. These are the restrictions we impose on the relational database.

In models like Entity-Relationship models, we did not have such features. Database Constraints can be categorized into 3 main categories:

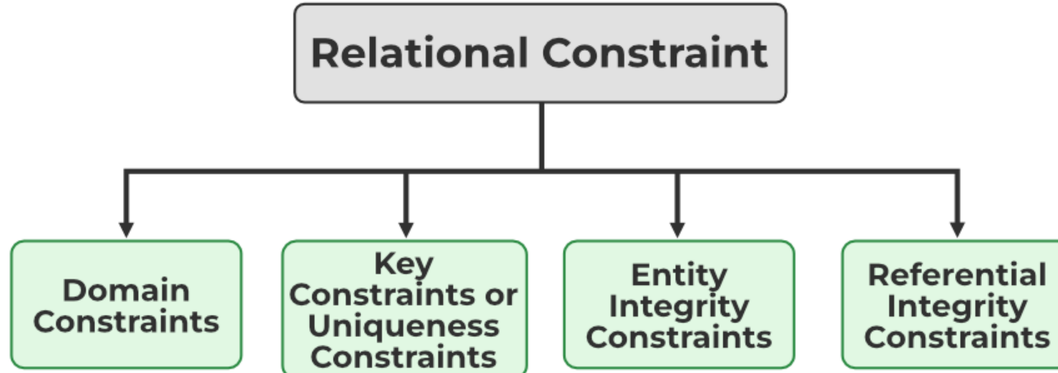
1. Constraints that are applied in the data model are called **Implicit Constraints**.
2. Constraints that are directly applied in the schemas of the data model, by specifying them in the DDL(Data Definition Language). These are called **Schema-Based Constraints or Explicit Constraints**.
3. Constraints that cannot be directly applied in the schemas of the data model. We call these Application-based or **Semantic Constraints**.

So here we are going to deal with **Implicit constraints**.

Relational Constraints

These are the restrictions or sets of rules imposed on the database contents. It validates the quality of the database. It validates the various operations like data insertion, updation, and other processes that have to be performed without affecting the integrity of the data. It protects us against threats/damages to the database. Mainly Constraints on the relational database are of 4 types

- Domain constraints
- Key constraints or Uniqueness Constraints
- Entity Integrity constraints
- Referential integrity constraints



Let's discuss each of the above constraints in detail.

1. Domain Constraints

- Every domain must contain atomic values(smallest indivisible units) which means composite and multi-valued attributes are not allowed.
- We perform a datatype check here, which means when we assign a data type to a column we limit the values that it can contain. Eg. If we assign the datatype of attribute age as int, we can't give it values other than int datatype.

Example:

EID	Name	Phone
01	Bikash Dutta	123456789 234456678

Explanation: In the above relation, Name is a composite attribute and Phone is a multi-values attribute, so it is violating domain constraint.

2. Key Constraints or Uniqueness Constraints

- These are called uniqueness constraints since it ensures that every tuple in the relation should be unique.
- A relation can have multiple keys or candidate keys(minimal superkey), out of which we choose one of the keys as the primary key, we don't have any restriction on choosing the primary key out of candidate keys, but it is suggested to go with the candidate key with less number of attributes.
- Null values are not allowed in the primary key, hence Not Null constraint is also part of the key constraint.

Example:

EID	Name	Phone
01	Bikash	6000000009
02	Paul	9000090009
01	Tuhin	9234567892

Explanation: In the above table, EID is the primary key, and the first and the last tuple have the same value in EID ie 01, so it is violating the key constraint.

3. Entity Integrity Constraints

- Entity Integrity constraints say that no primary key can take a NULL value, since using the primary key we identify each tuple uniquely in a relation.

Example:

EID	Name	Phone
01	Bikash	9000900099
02	Paul	600000009
NULL	Sony	9234567892

Explanation: In the above relation, EID is made the primary key, and the primary key can't take NULL values but in the third tuple, the primary key is null, so it is violating Entity Integrity constraints.

4. Referential Integrity Constraints

- The Referential integrity constraint is specified between two relations or tables and used to maintain the consistency among the tuples in two relations.
- This constraint is enforced through a foreign key, when an attribute in the foreign key of relation R1 has the same domain(s) as the primary key of relation R2, then the foreign key of R1 is said to reference or refer to the primary key of relation R2.
- The values of the foreign key in a tuple of relation R1 can either take the values of the primary key for some tuple in relation R2, or can take NULL values, but can't be empty.

Example:

EID	Name	DNO
01	Divine	12
02	Dino	22
04	Vivian	14

DNO	Place
12	Jaipur
13	Mumbai
14	Delhi

Explanation: In the above tables, the DNO of Table 1 is the foreign key, and DNO in Table 2 is the primary key. DNO = 22 in the foreign key of Table 1 is not allowed because DNO = 22 is not defined in the primary key of table 2. Therefore, Referential integrity constraints are violated here.

Advantages of Relational Database Model

- It is simpler than the hierarchical model and network model.
- It is easy and simple to understand.
- Its structure can be changed anytime upon requirement.
- **Data Integrity:** The relational database model enforces data integrity through various constraints such as primary keys, foreign keys, and unique constraints. This ensures that the data in the database is accurate, consistent, and valid.
- **Flexibility:** The relational database model is highly flexible and can handle a wide range of data types and structures. It also allows for easy modification and updating of the data without affecting other parts of the database.

- **Scalability:** The relational database model can scale to handle large amounts of data by adding more tables, indexes, or partitions to the database. This allows for better performance and faster query response times.
- **Security:** The relational database model provides robust security features to protect the data in the database. These include user authentication, authorization, and encryption of sensitive data.
- **Data consistency:** The relational database model ensures that the data in the database is consistent across all tables. This means that if a change is made to one table, the corresponding changes will be made to all related tables.
- **Query Optimization:** The relational database model provides a query optimizer that can analyze and optimize SQL queries to improve their performance. This allows for faster query response times and better scalability.

Disadvantages of the Relational Model

- Few database relations have certain limits which can't be expanded further.
- It can be complex and it becomes hard to use.
- **Complexity:** The relational model can be complex and difficult to understand, particularly for users who are not familiar with SQL and database design principles. This can make it challenging to set up and maintain a relational database.
- **Performance:** The relational model can suffer from performance issues when dealing with large data sets or complex queries. In particular, joins between tables can be slow, and indexing strategies can be difficult to optimize.
- **Scalability:** While the relational model is generally scalable, it can become difficult to manage as the database grows in size. Adding new tables or indexes can be time-consuming, and managing relationships between tables can become complex.
- **Cost:** Relational databases can be expensive to license and maintain, particularly for large-scale deployments. Additionally, relational databases often require dedicated hardware and specialized software to run, which can add to the cost.
- **Limited flexibility:** The relational model is designed to work with tables that have predefined structures and relationships. This can make it difficult to work with data that does not fit neatly into a table-based format, such as unstructured or semi-structured data.
- **Data redundancy:** In some cases, the relational model can lead to data redundancy, where the same data is stored in multiple tables. This can lead to inefficiencies and can make it difficult to ensure data consistency across the database.

Conclusion

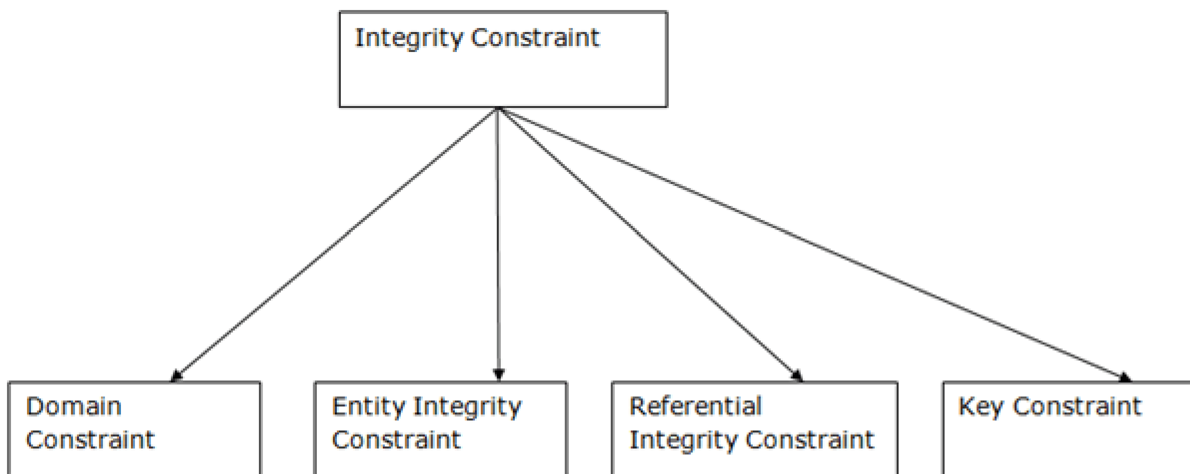
Relational database constraints are rules in a database model that help maintain the integrity and consistency of data. These rules include primary key constraints, unique constraints, foreign key constraints, check constraints, default constraints, not null constraints, multi-column constraints, etc. Relational database constraints help keep data accurate, maintain relationships, and avoid the insertion of wrong or inconsistent data.

More Examples

Integrity Constraints

- Integrity constraints are a set of rules. It is used to maintain the quality of information.
- Integrity constraints ensure that the data insertion, updating, and other processes have to be performed in such a way that data integrity is not affected.
- Thus, integrity constraint is used to guard against accidental damage to the database.

Types of Integrity Constraint



1. Domain constraints

- Domain constraints can be defined as the definition of a valid set of values for an attribute.
- The data type of domain includes string, character, integer, time, date, currency, etc. The value of the attribute must be available in the corresponding domain.

Example:

ID	NAME	SEMENSTER	AGE
1000	Tom	1st	17
1001	Johnson	2nd	24
1002	Leonardo	5th	21
1003	Kate	3rd	19
1004	Morgan	8th	A

Not allowed. Because AGE is an integer attribute

2. Entity integrity constraints

- The entity integrity constraint states that primary key value can't be null.
- This is because the primary key value is used to identify individual rows in relation and if the primary key has a null value, then we can't identify those rows.
- A table can contain a null value other than the primary key field.

Example:

EMPLOYEE

EMP_ID	EMP_NAME	SALARY
123	Jack	30000
142	Harry	60000
164	John	20000
	Jackson	27000

Not allowed as primary key can't contain a NULL value

3. Referential Integrity Constraints

- A referential integrity constraint is specified between two tables.
- In the Referential integrity constraints, if a foreign key in Table 1 refers to the Primary Key of Table 2, then every value of the Foreign Key in Table 1 must be null or be available in Table 2.

Example:

(Table 1)

EMP_NAME	NAME	AGE	D_No
1	Jack	20	11
2	Harry	40	24
3	John	27	18
4	Devil	38	13

Foreign key

Not allowed as D_No 18 is not defined as a Primary key of table 2 and In table 1, D_No is a foreign key defined

Relationships

(Table 2)

Primary Key

D_No	D_Location
11	Mumbai
24	Delhi
13	Noida

4. Key constraints

- Keys are the entity set that is used to identify an entity within its entity set uniquely.
- An entity set can have multiple keys, but out of which one key will be the primary key. A primary key can contain a unique and null value in the relational table.

Example:

ID	NAME	SEMENSTER	AGE
1000	Tom	1st	17
1001	Johnson	2nd	24
1002	Leonardo	5th	21
1003	Kate	3rd	19
1002	Morgan	8th	22

Not allowed. Because all row must be unique

SQL Constraints

SQL constraints are used to specify rules for the data in a table.
Constraints are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the table. If there is any violation between the constraint and the data action, the action is aborted.
Constraints can be column level or table level. Column level constraints apply to a column, and table level constraints apply to the whole table.
The following constraints are commonly used in SQL:

- NOT NULL - Ensures that a column cannot have a NULL value
- UNIQUE - Ensures that all values in a column are different
- PRIMARY KEY - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table
- FOREIGN KEY - Prevents actions that would destroy links between tables
- CHECK - Ensures that the values in a column satisfies a specific condition
- DEFAULT - Sets a default value for a column if no value is specified
- CREATE INDEX - Used to create and retrieve data from the database very quickly