The **Relational Model** is a foundational concept in database systems that represents data as a structured collection of **relations**, similar to tables. Here's a breakdown of its core elements:

1. **Relation (Table)**:
    - A relation represents a table within the database, with rows and columns. Each table has a unique name, which gives context to the data it holds.
2. **Tuple (Row)**:
    - Each row in the table is a **tuple**. It represents a single record or entity instance, containing a set of related values that describe some entity or relationship. For example, in a "STUDENT" table, each row could represent data about one specific student.
3. **Attribute (Column)**:
    - Each column header in a table is known as an **attribute**. Attributes have a name, which describes the data they contain, and every attribute in a table is of a specific data type. For example, in a "STUDENT" table, attributes could be "Name," "Student_number," "Class," and "Major."
4. **Domain**:
    - Each attribute has an associated **domain**, which defines the allowable values for that attribute. For example, the "Class" attribute might have a domain that restricts values to "Freshman," "Sophomore," "Junior," or "Senior."

Key Differences from Files

While a relational model is similar to a flat file (since both store data linearly), it has important structural and functional differences. In the relational model:

- **Data Integrity**: Relations enforce data integrity rules (like domains and unique keys).
- **Structured Queries**: The model is designed for complex querying with SQL.
- **Data Independence**: Relations abstract the data from its storage, making it more adaptable to changes in the structure without impacting how it's accessed.

These concepts are foundational in relational databases, where data consistency, integrity, and accessibility are prioritized.

DOMAIN

In the relational model, a **domain** is essentially a set of valid values that an attribute (column) can have. Think of it as a "data type with constraints." Here's how it works in simpler terms:

1. **Data Type**: A domain includes the type of data allowed in a column, like numbers, text, or dates.
2. **Allowed Values**: A domain can further restrict values to only certain options. For example, a column for a student's grade might only allow values like "A," "B," "C," "D," or "F."
3. **Example Domains**:

- A "BirthDate" column in a table might have a **date** domain, meaning it only accepts dates.
- A "Student_age" column could have a **positive integer** domain, only allowing whole numbers greater than zero.
- A "Department" column might have a **text** domain but restrict values to "Science," "Arts," "Commerce," etc.

In short, a domain defines **what kind of data** and **what range of values** are allowed in a column, ensuring consistency and accuracy in the database.

```
CREATE TABLE Students (
    StudentID INT,            -- Only integer values allowed
    Name VARCHAR(50),         -- Text values up to 50 characters
    BirthDate DATE,           -- Only valid dates allowed
    Grade CHAR(1) CHECK (Grade IN ('A', 'B', 'C', 'D', 'F')),  -- Restrict
    Age INT CHECK (Age > 0)   -- Positive integers only
);
```

Characteristics of Relations

The passage you provided delves into the characteristics of relations in the context of the relational model in databases. Let's break down the key concepts for better understanding:

1. Ordering of Tuples

- **No Order in Relations**: A relation is defined as a set of tuples (rows), and in set theory, elements have no specific order. This means the tuples in a relation can appear in any order and are considered equal regardless of how they are arranged.
- **Display Order**: When displaying a relation (like in a table), we can choose any order to present the tuples (e.g., sorting by Name or Age). However, this display does not affect the fundamental nature of the relation.

2. Ordering of Values within a Tuple

- **Ordered Values**: Each tuple is an ordered list of values corresponding to attributes (columns). While the order of the tuples does not matter, the order of the attributes and their values within a tuple is important.

3. Atomic Values and NULLs

- **Atomic Values**: Each value in a tuple is atomic, meaning it cannot be subdivided further within the relational model. This principle leads to

the requirement that composite or multi-valued attributes must be represented in separate relations.

- **NULL Values**: NULL is a special marker used to indicate that a value is unknown or does not apply to a particular tuple. NULL can have several interpretations, such as:
    - The value is unknown.
    - The value exists but is not available.
    - The attribute does not apply to the tuple.

Handling NULL values can complicate queries and operations, so it's often recommended to minimize their use.

4. Interpretation of a Relation

- **Schema Interpretation**: A relation schema declares the attributes of an entity (e.g., a STUDENT relation with attributes like Name and Age). Each tuple represents a specific instance of this schema.
- **Entity and Relationship Representation**: Relations can represent both entities (like students) and relationships (like majors). This duality can sometimes create confusion about whether a relation represents an entity or a relationship.

5. Predicate Interpretation

- **Logical Interpretation**: A relation schema can also be seen as a predicate. For example, the predicate STUDENT(Name, Ssn, …) is considered true for the tuples that match this schema. This view is useful in contexts like logical programming, where it helps express facts and queries.

    - A **predicate** is a condition or assertion that can be evaluated as true or false concerning the data in a relational database.
    - It helps express relationships and conditions regarding the data, and is fundamental in querying databases and defining constraints.

    Predicates are also used in database queries, particularly in the **WHERE** clause to filter records based on specific conditions. For instance:

    SELECT * FROM STUDENT WHERE Age > 18;

Relational Model Constraints and relational database schemas

In the context of the relational model, **constraints** and **schemas** are fundamental concepts that ensure the integrity and structure of the data stored in relational databases. Let's break down each concept:

1. Relational Database Schema

A **relational database schema** is a blueprint that defines the structure of a relational database. It describes *how data is organized and how the relations (tables) among the various entities are set up.*

Components of a Relational Database Schema:

- **Tables (Relations)**: Each table represents a relation, which consists of rows (tuples) and columns (attributes).
- **Attributes**: Each table has attributes (columns) that define the properties of the entities represented in that table. Each attribute has a data type (e.g., integer, string, date).
- **Primary Key**: A primary key is a unique identifier for each row in a table. It ensures that each record can be uniquely identified. For example, in a STUDENT table, the Student_ID might be a primary key.
- **Foreign Key**: A foreign key is an attribute that creates a link between two tables. It refers to the primary key of another table, establishing a relationship between the two. For example, if there's a COURSE table with a Course_ID, the STUDENT table might include a Course_ID as a foreign key.
- **Constraints**: Constraints are rules that define certain properties of the data. They help ensure data integrity.

2. Relational Model Constraints

Constraints are rules applied to the data in the relational model to ensure its accuracy and reliability. They enforce data integrity and maintain the correctness of the relationships between tables.

Common Types of Constraints:

1. **Domain Constraints**:
    - Specify the allowable values for an attribute. For example, if an attribute is defined as an integer, then it cannot store string values.
    - Example:

```
CREATE TABLE STUDENT (
  Student_ID INT,
```

```
    Name VARCHAR(100),
    Age INT CHECK (Age >= 0)  -- Domain constraint for Age
);
```

2. **Key Constraints**:
    - Define rules regarding primary keys and unique constraints. A primary key must contain unique values and cannot be null.
    - Example:

```
CREATE TABLE STUDENT (
    Student_ID INT PRIMARY KEY,  -- Key constraint
    Name VARCHAR(100) NOT NULL
);
```

3. **Foreign Key Constraints**:
    - Establish a relationship between two tables by enforcing that a value in one table must exist in another table.
    - Example:

```
CREATE TABLE COURSE (
    Course_ID INT PRIMARY KEY,
    Course_Name VARCHAR(100)
);

CREATE TABLE STUDENT (
    Student_ID INT PRIMARY KEY,
    Name VARCHAR(100) NOT NULL,
    Course_ID INT,
    FOREIGN KEY (Course_ID) REFERENCES COURSE(Course_ID)  -- Foreign key
constraint
);
```

4. **Not Null Constraints**:
    - Ensure that a column cannot have a null value, meaning a value must always be provided.
    - Example:

```
sql
Copy code
CREATE TABLE STUDENT (
    Student_ID INT PRIMARY KEY,
    Name VARCHAR(100) NOT NULL  -- Not Null constraint
);
```

5. **Check Constraints**:
    - Specify a condition that must be met for each row in a table. If the condition fails, the operation will be rejected.

- Example:

sql
Copy code
```
CREATE TABLE STUDENT (
   Student_ID INT PRIMARY KEY,
   Age INT CHECK (Age >= 18)  -- Check constraint for Age
);
```

Update Operations, Transactions, and Dealing with Constraint Violations

1. Basic Update Operations (2 marks)

- **INSERT**: This operation allows new tuples (rows) to be added to a relation (table) in the database. It's used to expand data within tables, crucial for maintaining up-to-date information.
    - **Example**: In an employee table, an INSERT operation can add a new employee's record with details like EmployeeID, Name, DepartmentID, etc.

```
INSERT INTO EMPLOYEE (EmployeeID, Name, DepartmentID)
VALUES (123, 'John Doe', 4);
```

DELETE: This operation removes specified tuples from a relation, used to delete outdated or incorrect data.

- **Example**: If an employee leaves the company, the DELETE operation removes that employee's record.

```
DELETE FROM EMPLOYEE
WHERE EmployeeID = 123;
```

UPDATE (MODIFY): The UPDATE operation modifies the attribute values in existing tuples without affecting other rows.

- **Example**: Updating an employee's department or salary.

```
UPDATE EMPLOYEE
SET Salary = Salary + 5000
WHERE EmployeeID = 123;
```

2. Constraint Types and Violations (2 marks)

Constraints ensure that the data within the database remains valid and consistent. When an update operation (INSERT, DELETE, or UPDATE) violates any of these constraints, the DBMS may reject the operation. Key types include:

- **Domain Constraints**: Ensure that values in each column meet the predefined data type or value range.
    - **Example**: If a Salary column is restricted to positive values, an INSERT with a negative value would violate this constraint.
- **Key Constraints**: Ensure that unique identifiers, like primary keys, are not duplicated or NULL.
    - **Example**: An INSERT or UPDATE attempting to use an existing EmployeeID would violate the primary key constraint.
- **Entity Integrity**: Ensures that primary key columns are unique and cannot contain NULL values, maintaining each row's identity.
    - **Example**: Inserting a record without a primary key would violate entity integrity.
- **Referential Integrity**: Ensures that foreign keys in one table must match primary keys in the referenced table.
    - **Example**: An INSERT or UPDATE where DepartmentID in EMPLOYEE doesn't match an existing department in DEPARTMENT would violate this constraint.

3. Constraint Violation Handling (2 marks)

When constraint violations occur, DBMS offers three main strategies to handle them:

- **RESTRICT**: The operation is rejected if it violates any constraint, ensuring data integrity is preserved.
    - **Example**: Trying to delete a DEPARTMENT that has employees may be restricted if the EMPLOYEE table references it.

- **CASCADE**: Propagates changes to any related tuples in referencing tables.
    - **Example**: If an employee is deleted, all dependent records, like work hours in a WORKS_ON table, are automatically deleted.
- **SET NULL/DEFAULT**: Sets referencing attributes to NULL or a default value to maintain integrity.
    - **Example**: If a manager is removed, the ManagerID field in other records could be set to NULL or a default ID, allowing deletion without integrity issues.

-- Deletes an employee and cascades deletion to related records in WORKS_ON

DELETE FROM EMPLOYEE

WHERE EmployeeID = 123 CASCADE;

Setting as null , if deleted

```
CREATE TABLE DEPARTMENT (
    DepartmentID INT PRIMARY KEY,
    DepartmentName VARCHAR(100)
);

CREATE TABLE EMPLOYEE (
    EmployeeID INT PRIMARY KEY,
    EmployeeName VARCHAR(100),
    DepartmentID INT,
    FOREIGN KEY (DepartmentID) REFERENCES DEPARTMENT(DepartmentID)
    ON DELETE SET NULL
);
```

EmployeeID	EmployeeName	DepartmentID
101	Alice	NULL
102	Bob	2
103	Charlie	NULL

Relational Algebra

Relational Algebra is a formal language for querying and manipulating relations (tables) in a database. It provides both **unary** and **binary operations** for performing complex queries. In addition to basic operations, it includes **aggregate functions** and **grouping** for summarizing data.

1. Unary Relational Operations

Unary operations are applied to a single relation and include **Selection**, **Projection**, **Renaming**, **Aggregation**, and **Grouping**.

a. Selection (σ)

The **Selection** operation filters rows based on a condition. It's represented as **σ_{condition}(R)**, where R is the relation and condition is the filtering criterion.

Example: Retrieve employees with a salary greater than 50,000 from an EMPLOYEE relation.

Algebraic Notation:

$$\sigma_{Salary>50000}(\text{EMPLOYEE})$$

Result:

EmployeeID	Name	Salary
101	Alice	60000
102	Bob	70000

With OR

- **Example**: Retrieve employees from the `EMPLOYEE` table who are either in the **Sales** department or earn **more than 50,000**.

  Algebraic Notation:

$$\sigma_{(\text{Department} = \text{'Sales'}) \text{ OR } (\text{Salary} > 50000)}(\text{EMPLOYEE})$$

WITH AND

- **Alternative with AND**: Retrieve employees in the **Sales department** who earn **more than 50,000**.

$$\sigma_{(\text{Department} = \text{'Sales'}) \text{ AND } (\text{Salary} > 50000)}(\text{EMPLOYEE})$$

b. Projection (π)

The **Projection** operation extracts specific columns (attributes) from a relation. It's represented as

$$\pi_{<\text{attribute list}>}(R)$$

- **Example**: List only the EmployeeID and Name of employees.

Algebraic Notation:

$$\pi_{\text{EmployeeID, Name}}(\text{EMPLOYEE})$$

Result:

EmployeeID	Name
101	Alice
102	Bob

c. Renaming (ρ)

Renaming changes the attribute names of a relation or gives it a new alias. It's represented as **ρ_{newName}(R)**.

- **Example**: Rename the EMPLOYEE relation to STAFF.

Algebraic Notation:

$$\rho_{\text{STAFF}}(\text{EMPLOYEE})$$

Relational Algebra Operations from Set Theory

Relational Algebra Operations from Set Theory

1. UNION (∪)

Combines tuples from two relations and eliminates duplicates.

Requirements:

- Both relations must have same number of attributes
- Corresponding attributes must have same domain

Example:

Consider two relations:

```
STUDENTS_CS
ID  | NAME  | DEPT
------------------
101 | Alice | CS
102 | Bob   | CS

STUDENTS_IT
ID  | NAME  | DEPT
------------------
103 | Carol | IT
104 | David | IT
```

STUDENTS_CS ∪ STUDENTS_IT would give:

```
ID  | NAME  | DEPT
------------------
101 | Alice | CS
102 | Bob   | CS
103 | Carol | IT
104 | David | IT
```

2. INTERSECTION (∩)

Returns only tuples that appear in both relations.

Example:

```
CLASS_A
ID  | NAME  | GRADE
-------------------
101 | Alice | A
102 | Bob   | B
103 | Carol | A

CLASS_B
ID  | NAME  | GRADE
-------------------
101 | Alice | A
```

103 | Carol | A

104 | David | B

```

CLASS_A ∩ CLASS_B would give:

```

ID | NAME | GRADE

----

101 | Alice | A

103 | Carol | A

```

## 3. SET DIFFERENCE (-)

Returns tuples that appear in first relation but not in second.

**Example:**

CLASS_A - CLASS_B would give:

```

ID | NAME | GRADE

----

102 | Bob | B

```

## 4. CARTESIAN PRODUCT (×)

Combines each tuple of first relation with every tuple of second relation.
```

Example:

```
COURSES
CID | COURSE
-------------
C1  | Math
C2  | Physics

PROFESSORS
PID | NAME
------------
P1  | Smith
P2  | Jones
```

COURSES × PROFESSORS gives:

```
CID | COURSE  | PID | NAME
------------------------
C1  | Math    | P1  | Smith
C1  | Math    | P2  | Jones
C2  | Physics | P1  | Smith
C2  | Physics | P2  | Jones
```

Practical Applications

1. **UNION:**

- Combining semester registration data from multiple departments

- Merging customer lists from different regions

2. **INTERSECTION:**

- Finding students enrolled in both CS and Math courses

- Identifying customers who bought products in both online and physical stores

3. **SET DIFFERENCE:**

- Finding students who enrolled but didn't complete a course

- Identifying products in stock but not sold

4. **CARTESIAN PRODUCT:**

- Generating all possible course-professor combinations

- Creating all possible product-color combinations for an inventory

d. Cartesian Product (×) with Condition Filtering

Cartesian Product pairs each tuple in one relation with every tuple in another relation, resulting in all possible combinations. It's often followed by a **Selection** to filter meaningful rows.

- **Example**: Combine `EMPLOYEE` and `DEPARTMENT` data, and show employees who are in **Department 5**.

  Algebraic Notation:

$$\sigma_{\text{DepartmentID} = 5}(\text{EMPLOYEE} \times \text{DEPARTMENT})$$

1. Aggregate Functions

Aggregate functions perform calculations on a set of values to return a single value. Common aggregate functions in Relational Algebra include:

- **SUM**: Adds up all values in a set.
- **COUNT**: Counts the number of values.
- **AVG**: Calculates the average of values.
- **MIN**: Finds the minimum value.
- **MAX**: Finds the maximum value.

Example Table: EMPLOYEE

EmployeeID	Name	DepartmentID	Salary	Age
1	Alice	1	50000	30
2	Bob	2	60000	40
3	Charlie	1	55000	25
4	David	3	48000	35
5	Eve	2	62000	29

1. **Total Salary** of all employees (using **SUM**):

$$\text{SUM}_{\text{Salary}}(\text{EMPLOYEE})$$

   Result: 50000 + 60000 + 55000 + 48000 + 62000 = **275000**

2. **Average Salary** of employees (using **AVG**):

$$\text{AVG}_{\text{Salary}}(\text{EMPLOYEE})$$

   Result: (50000 + 60000 + 55000 + 48000 + 62000) / 5 = **55000**

3. **Maximum Salary** (using **MAX**):

$$\text{MAX}_{\text{Salary}}(\text{EMPLOYEE})$$

   Result: **62000**

4. **Count** of employees (using **COUNT**):

$$\text{COUNT}_{*}(\text{EMPLOYEE})$$

   Result: **5**

2. Grouping with Aggregate Functions

The **Grouping** operation, often combined with aggregate functions, groups rows that have the same values in specified columns and performs an aggregate function on each group.

Example: Find the **total salary** per **DepartmentID**.

Grouping Query:

$$\gamma_{\text{DepartmentID},\text{SUM}_{\text{Salary}}}(\text{EMPLOYEE})$$

Result:

DepartmentID	SUM(Salary)
1	105000
2	122000
3	48000

JOIN operations

JOIN operations allow us to combine related tuples from two different relations. There are multiple types of joins, each serving a specific purpose.

1. Theta Join (θ-join)

Theta Join (θ) is the most general form of join, allowing any condition as a predicate. In a theta join, two relations are combined based on a condition that can use any operator, such as =, >, <, !=, etc.

- Notation: $R \bowtie_\theta S$

- **Condition (θ)**: Any conditional expression between attributes of R and S.

Example Tables

STUDENT

StudentID	Name	Age
1	Alice	20
2	Bob	22
3	Charlie	23

COURSE

CourseID	StudentID	Score
C1	1	85
C2	2	78
C3	3	90

Theta Join Query: Find all students with scores greater than 80.

$$\text{STUDENT} \bowtie_{\text{COURSE.Score}>80} \text{COURSE}$$

Result:

StudentID	Name	Age	CourseID	Score
1	Alice	20	C1	85
3	Charlie	23	C3	90

This join combines rows from **STUDENT** and **COURSE** where the **Score** in **COURSE** is greater than 80.

2. Equi Join

Equi Join is a specific type of theta join where the condition is equality (=) between attributes of two relations. In an equi join, we retain all attributes of both relations, including the joining columns, which may appear duplicated.

- **Notation**: $R \bowtie_{\text{R.attr}=\text{S.attr}} S$

Example: Let's join STUDENT and COURSE on **StudentID**.

Equi Join Query:

$$\text{STUDENT} \bowtie_{\text{STUDENT.StudentID}=\text{COURSE.StudentID}} \text{COURSE}$$

Result:

StudentID	Name	Age	CourseID	StudentID	Score
1	Alice	20	C1	1	85
2	Bob	22	C2	2	78
3	Charlie	23	C3	3	90

Here, **StudentID** appears twice because equi join retains both attributes in the output.

3. Natural Join (⋈)

Natural Join is a specialized form of equi join where we automatically join on all common attributes with the same name and eliminate duplicate columns in the result.

- Notation: $R \bowtie S$

Natural Join Query:

$$\text{STUDENT} \bowtie \text{COURSE}$$

Result:

StudentID	Name	Age	CourseID	Score
1	Alice	20	C1	85
2	Bob	22	C2	78
3	Charlie	23	C3	90

In **Natural Join**, **StudentID** is not duplicated, as it is automatically combined.

DIVISION Operation (÷)

The **Division** operation is used to find tuples in one relation that are associated with **all** tuples in another relation. It is typically applied in queries involving "for all" conditions.

Example Scenario

Suppose we want to find **students** who have enrolled in **all courses** offered.

Example Tables

1. **STUDENT_COURSE** (enrollment information)

StudentID	CourseID
1	C1
1	C2
2	C1
2	C2
3	C1

2. **COURSE_LIST** (all offered courses)

CourseID
C1
C2

STUDENT_COURSE ÷ CORSE_LIST

Result:

StudentID
1
2