Unit -2

Relational Models: Structure of relational databases, Domains, Relations, Relational algebra – fundamental operators and syntax, selection, Projection, relational algebra queries, tuple relational calculus, set operations, renaming, Joins, Division, syntax. Operators, grouping and ungrouping, relational comparison. Codd's rules, Relational Schemas,

Introduction to UML

Relational database model: Logical view of data, keys, integrity rules

The Relational Model

- The first database systems were based on the **network** and **hierarchical** models.

- The relational model was first proposed by E.F. Codd in 1970 and

- the first such systems (notably INGRES and System/R) was developed in 1970s.

- The relational model is now the dominant model for commercial data processing applications.

- Note: **Attribute Name Abbreviations**The text uses fairly long attribute names which are abbreviated in the notes as follows.
  - *customer-name* becomes *cname*
  - *customer-city* becomes *ccity*
  - *branch-city* becomes *bcity*
  - *branch-name* becomes *bname*
  - *account-number* becomes *account#*
  - *loan-number* becomes *loan#*
  - *banker-name* becomes *banker*
-

Structure of Relational Database

- A relational database consists of a collection of **tables**, each having a unique **name**.
- A **row(tuple)** in a table represents a **relationship** among a set of values.
- Thus a table represents a **collection of relationships**.
- **There is a direct correspondence between the concept of a table and the mathematical concept of a relation.**
- A substantial theory has been developed for relational databases.

- **Basic Structure**
- Figure 3.1 shows the *deposit* and *customer* tables for our banking example.
  Figure 3.1: The *deposit* and *customer* relations.

  – It has four attributes.
  – For each attribute there is a permitted set of values, called the **domain** of that attribute.
  – E.g. the domain of *bname* is the set of all branch names.

Figure 3.1 shows the *deposit* and *customer* tables for our banking example.

bname	account#	cname	balance
Downtown	101	Johnson	500
Longheed Mall	215	Smith	700
SFU	102	Hayes	400
SFU	301	Adams	1300

cname	street	city
Johnson	Pender	Vancouver
Smith	North	Burnaby
Hayes	Curtis	Burnaby
Adams	No.3 Road	Richmond
Jones	Oak	Vancouver

Figure 3.1: The *deposit* and *customer* relations.

- Let denote the domain of *bname*, and , and the remaining attributes' domains respectively.Then, any row of *deposit* consists of a four-tuple where
- In general, *deposit* contains a **subset** of the set of all possible rows.
- That is, *deposit* is a subset of
- In general, a table of n columns must be a subset of
- Mathematicians define a relation to be a subset of a Cartesian product of a list of domains. You can see the correspondence with our tables.We will use the terms **relation** and **tuple** in place of **table** and **row** from now on.

- Some more formalities:
  - let the tuple variable refer to a tuple of the relation .
  - We say to denote that the tuple is in relation .
  - Then [*bname*] = [1] = the value of on the *bname* attribute.
  - So [*bname*] = [1] = ``Downtown'',
  - and [*cname*] = [3] = ``Johnson''.

- We'll also require that the domains of all attributes be **indivisible units.**
  - A domain is **atomic** if its elements are indivisible units.
  - For example, the set of integers is an atomic domain.
  - The set of all sets of integers is not.
  - Why? Integers do not have subparts, but sets do - the integers comprising them.
  - We could consider integers non-atomic if we thought of them as ordered lists of digits.

Database Scheme

- We distinguish between a **database scheme** (logical design) and a **database instance** (data in the database at a point in time).
- A **relation scheme** is a list of attributes and their corresponding domains.

- The text uses the following conventions:
  - italics for all names
  - lowercase names for relations and attributes
  - names beginning with an uppercase for relation schemes
- These notes will do the same.For example, the relation scheme for the *deposit* relation:
  - *Deposit-scheme = (bname, account#, cname, balance)*

- We may state that *deposit* is a relation on scheme *Deposit-scheme* by writing *deposit(Deposit-scheme)*.
- If we wish to specify domains, we can write:
  - (*bname*: string, *account#*: integer, *cname*: string, *balance*: integer).
- Note that customers are identified by name. In the real world, this would not be allowed, as two or more customers might share the same name.

Figure 3.2 shows the E-R diagram for a banking enterprise.

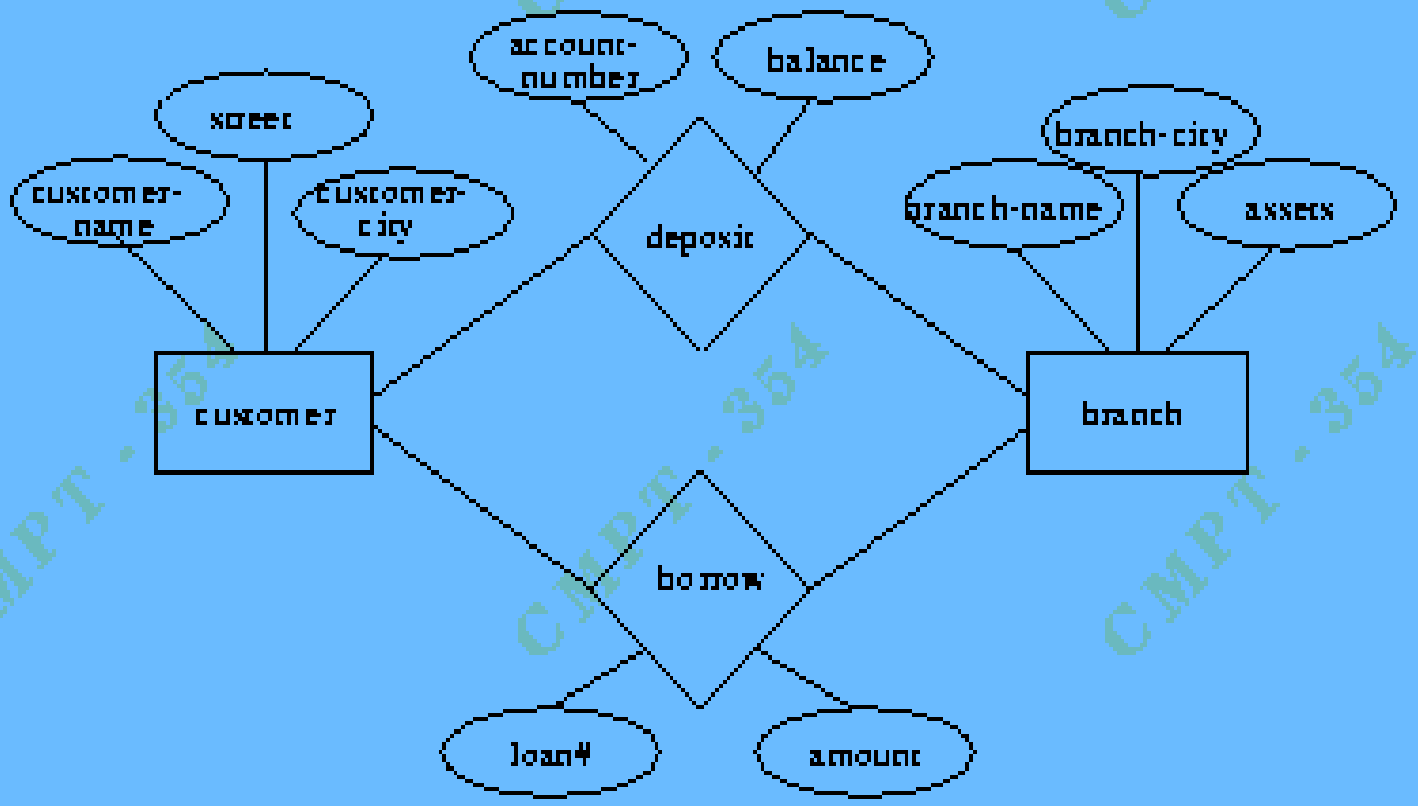

Figure 3.2: E-R diagram for the banking enterprise

- The relation schemes for the banking example used throughout the text are:
  - *Branch-scheme = (bname, assets, bcity)*
  - *Customer-scheme = (cname, street, ccity)*
  - *Deposit-scheme = (bname, account#, cname, balance)*
  - *Borrow-scheme = (bname, loan#, cname, amount)*
- **Note:** some attributes appear in several relation schemes (e.g. *bname, cname*). This is legal, and provides a way of **relating** tuples of distinct relations.

Why not put all attributes in one relation?

- Suppose we use one large relation instead of *customer* and *deposit*:
  - *Account-scheme = (bname, account#, cname, balance, street, ccity)*
  - If a customer has several accounts, we must duplicate her or his address for each account.
  - If a customer has an account but no current address, we cannot build a tuple, as we have no values for the address.
  - We would have to use **null values** for these fields.
  - Null values cause difficulties in the database.
  - By using two separate relations, we can do this without using null values

- **Keys**
- The notions of **superkey**, **candidate key** and **primary key** all apply to the relational model.
- For example, in *Branch-scheme*,
  - {*bname*} is a superkey.
  - {*bname, bcity*} is a superkey.
  - {*bname, bcity*} is not a candidate key, as the superkey {*bname*} is contained in it.
  - {*bname*} is a candidate key.
  - {*bcity*} is not a superkey, as branches may be in the same city.
  - We will use {*bname*} as our primary key.

- The primary key for *Customer-scheme* is {*cname*}.
- More formally, if we say that a subset of is a *superkey* for , we are restricting consideration to relations in which no two distinct tuples have the same values on all attributes in .

Query Languages

- A **query language** is a language in which a user requests information from a database. These are typically higher-level than programming languages.They may be one of:
  - **Procedural**, where the user instructs the system to perform a sequence of operations on the database. This will compute the desired information.
  - **Nonprocedural**, where the user specifies the information desired without giving a procedure for obtaining the information.
- A complete query language also contains facilities to insert and delete tuples as well as to modify parts of existing tuples.
-

The Relational Algebra

- The **relational algebra** is a procedural query language.
  - Six fundamental operations:
    - select (unary)
    - project (unary)
    - rename (unary)
    - cartesian product (binary)
    - union (binary)
    - set-difference (binary)
  - Several other operations, defined in terms of the fundamental operations:
    - set-intersection
    - natural join
    - division
    - assignment
  - Operations produce a new relation as a result.

Fundamental Operations

- **The Select OperationSelect** selects tuples that satisfy a given predicate.
- Select is denoted by a lowercase Greek sigma , with the predicate appearing as a subscript.

- The argument relation is given in parentheses following the sigma
- For example, to select tuples (rows) of the *borrow* relation where the branch is ``SFU'', we would write

$$\sigma_{bname = \text{"SFU"}}(borrow)$$

Let Figure 3.3 be the *borrow* and *branch* relations in the banking example.

bname	loan#	cname	amount
Downtown	17	Jones	1000
Lougheed_Mall	23	Smith	2000
SFU	15	Hayes	1500

bname	assets	bcity
Downtown	9,000,000	Vancouver
Lougheed_Mall	21,000,000	Burnaby
SFU	17,000,000	Burnaby

Figure 3.3: The *borrow* and *branch* relations.

- The new relation created as the result of this operation consists of one tuple: .
- We allow comparisons using =, , <, , > and in the selection predicate.
- We also allow the logical connectives (or) and (and). For example:

$$\sigma_{bname="Downtown" \wedge amount > 1200}(borrow)$$

cname	banker
Hayes	Jones
Johnson	Johnson

Figure 3.4: The *client* relation.

Suppose there is one more relation, *client*, shown in Figure 3.4, with the scheme

$$Client\_scheme = (cname, banker)$$

we might write

$$\sigma_{cname=banker}(client)$$

to find clients who have the same name as their banker.

2. **The Project Operation**

Project copies its argument relation for the specified attributes only. Since a relation is a **set**, duplicate rows are eliminated.

Projection is denoted by the Greek capital letter pi (Π). The attributes to be copied appear as subscripts.

For example, to obtain a relation showing customers and branches, but ignoring amount and loan#, we write

$$\Pi_{bname,cname}(borrow)$$

We can perform these operations on the relations resulting from other operations.

To get the names of customers having the same name as their bankers,

$$\Pi_{cname}(\sigma_{cname=banker}(client))$$

- The **cartesian product** of two relations is denoted by a cross (X), written
- The result of is a new relation with a tuple for each possible **pairing** of tuples from and .

- In order to avoid ambiguity, the attribute names have attached to them the name of the relation from which they came. If no ambiguity will result, we drop the relation name.

- The result is a very large relation. If has tuples, and has tuples, then will have tuples.

- The resulting scheme is the concatenation of the schemes of and , with relation names added as mentioned.

- To find the clients of banker Johnson and the city in which they live, we need information in both *client* and *customer* relations. We can get this by writing

$$\sigma_{banker="Johnson"}(client \times customer)$$

However, the *customer.cname* column contains customers of bankers other than Johnson. (Why?)

We want rows where *client.cname* = *customer.cname*. So we can write

$$\sigma_{client.cname=customer.cname}(\sigma_{banker="Johnson"}(client \times customer))$$

to get just these tuples.

Finally, to get just the customer's name and city, we need a projection:

$$\Pi_{client.cname,ccity}(\sigma_{client.cname=customer.cname}(\sigma_{banker="Johnson"}(client \times customer)))$$

- **The Rename Operation**
- The **rename** operation solves the problems that occurs with naming when performing the cartesian product of a relation with itself.
- Suppose we want to find the names of all the customers who live on the same street and in the same city as Smith.
- We can get the street and city of Smith by writing
-

$$\Pi_{street,ccity}(\sigma_{cname="Smith"}(customer))$$

To find other customers with the same information, we need to reference the *customer* relation again:

$$\sigma_P(customer \times (\Pi_{street,ccity}\ (\sigma_{cname="Smith"}(customer))))$$

where P is a selection predicate requiring *street* and *ccity* values to be equal.

Problem: how do we distinguish between the two street values appearing in the Cartesian product, as both come from a *customer* re
Solution: use the rename operator, denoted by the Greek letter rho (ρ).

We write

$$\rho_x(r)$$

to get the relation r under the name of x.

If we use this to rename one of the two **customer** relations we are using, the ambiguities will disappear.

$$\Pi_{customer.cname}(\sigma_{cust2.street=customer.street \wedge cust2.ccity=customer.ccity}$$
$$(customer \times (\Pi_{street,ccity}(\sigma_{cname="Smith"}(\rho_{cust2}(customer))))))$$

5. The Union Operation

The **union** operation is denoted $\cup$ as in set theory. It returns the union (set union) of two compatible relations.

For a union operation $r \cup s$ to be legal, we require that

- r and s must have the same number of attributes.
- The domains of the corresponding attributes must be the same.

To find all customers of the SFU branch, we must find everyone who has a loan or an account or both at the branch.

We need both *borrow* and *deposit* relations for this:

$$\Pi_{cname}(\sigma_{bname="SFU"}(borrow)) \cup \Pi_{cname}(\sigma_{bname="SFU"}(deposit))$$

As in all set operations, duplicates are eliminated, giving the relation of Figure 3.5(a).

(a)

cname
Hayes
Adams

(b)

cname
Adams

Figure 3.5: The *union* and *set-difference* operations.

6. The Set Difference Operation

Set difference is denoted by the minus sign (_). It finds tuples that are in one relation, but not in another.

Thus $r - s$ results in a relation containing tuples that are in r but not in s.

To find customers of the SFU branch who have an account there but no loan, we write

$$\Pi_{cname}(\sigma_{bname="SFU"}(deposit)) - \Pi_{cname}(\sigma_{bname="SFU"}(borrow))$$

The result is shown in Figure 3.5(b).

We can do more with this operation. Suppose we want to find the largest account balance in the bank. Strategy:

- Find a relation r containing the balances **not** the largest.
- Compute the set difference of r and the *deposit* relation.

To find r, we write

$$\Pi_{deposit.balance}(\sigma_{deposit.balance < d.balance}(deposit \times \rho_d(deposit)))$$

This resulting relation contains all balances except the largest one. (See Figure 3.6(a)).

Now we can finish our query by taking the set difference:

$$\Pi_{balance}(deposit) - \Pi_{deposit.balance}(\sigma_{deposit.balance < d.balance}(deposit \times \rho_d(deposit)))$$

Figure 3.6(b) shows the result.

(a)

balance
100
500
700

(b)

balance
1300

Formal Definition of Relational Algebra

1. A basic expression consists of either
   - A relation in the database.
   - A constant relation.
2. General expressions are formed out of smaller subexpressions using
   - $\sigma_p(E_1)$ select (p a predicate)
   - $\Pi_s(E_1)$ project (s a list of attributes)
   - $\rho_x(E_1)$ rename (x a relation name)
   - $E_1 \cup E_2$ union
   - $E_1 - E_2$ set difference
   - $E_1 \times E_2$ cartesian product

Additional Operations

1. Additional operations are defined in terms of the fundamental operations. They do not add power to the algebra, but are useful to simplify common queries.

2. **The Set Intersection Operation**

   Set intersection is denoted by $\cap$, and returns a relation that contains tuples that are in **both** of its argument relations.

   It does not add any power as

$$r \cap s = r - (r - s)$$

   To find all customers having both a loan and an account at the SFU branch, we write

$$\Pi_{cname}(\sigma_{bname="SFU"}(borrow)) \cap \Pi_{cname}(\sigma_{bname="SFU"}(deposit))$$

ies on a cartesian product.

ers having a loan at the bank and the cities in which they live, we need *borrow* and *customer* relations:

$$\pi_{borrow.cname,\ customer.city}(\sigma_{borrow.cname=customer.cname}(borrow \times customer))$$

only those tuples pertaining to only one *cname*.

common, so we have the **natural join**, denoted by a $\bowtie$ sign. Natural join combines a cartesian product and a selection into one op
t appear in both relation schemes. Duplicates are removed as in all relation operations.

e previous query as

$$\pi_{cname,\ city}(borrow \bowtie customer)$$

The resulting relation is shown in Figure 3.7.

cname	city
Smith	Burnaby
Hayes	Burnaby
Jones	Vancouver

Figure 3.7: Joining *borrow* and *customer* relations.

We can now make a more formal definition of natural join.

- Consider R and S to be **sets** of attributes.
- We denote attributes appearing in **both** relations by $R \cap S$.
- We denote attributes in **either or both** relations by $R \cup S$.
- Consider two relations $r(R)$ and $s(S)$.
- The natural join of r and s, denoted by $r \bowtie s$ is a relation on scheme $R \cup S$.
- It is a projection onto $R \cup S$ of a selection on $r \times s$ where the predicate requires $r.A = s.A$ for each attribute A in $R \cap S$.

- **The Division Operation**Division, denoted , is suited to queries that include the phrase ``for all''.
- Suppose we want to find all the customers who have an account at **all** branches located in Brooklyn.
- Strategy: think of it as three steps.
- We can obtain the names of all branches located in Brooklyn by

$$r_1 = \Pi_{bname}(\sigma_{bcity="Brooklyn"}(branch))$$

Figure 3.19 in the textbook shows the result.

We can also find all *cname, bname* pairs for which the customer has an account by

$$r_2 = \Pi_{cname,bname}(deposit)$$

Figure 3.20 in the textbook shows the result.

Now we need to find all customers who appear in r_2 with **every** branch name in r_1.

The divide operation provides exactly those customers:

$$\Pi_{cname,bname}(deposit) \div \Pi_{bname}(\sigma_{bcity="Brooklyn"}(branch))$$

which is simply $r_2 \div r_1$.

Formally,

- Let $r(R)$ and $s(S)$ be relations.
- Let $S \subseteq R$.
- The relation $r \div s$ is a relation on scheme $R - S$.
- A tuple t is in $r \div s$ if for every tuple t_s in s there is a tuple t_r in r satisfying both of the following:

$$t_r[S] = t_s[S] \quad (3.2.1)$$

$$t_r[R - S] = t[R - S] \quad (3.2.2)$$

- These conditions say that R-S portion of a tuple t is in r÷s if and only if there are tuples with the r-s portion and the S portion in r for every value of the S portion in relation S.

- The division operation can be defined in terms of the fundamental operations

5. The Assignment Operation

Sometimes it is useful to be able to write a relational algebra expression in parts using a temporary relation variable (as we did with r_1 and r_2 in the division example).

The assignment operation, denoted $\leftarrow$, works like assignment in a programming language.

We could rewrite our division definition as

$$temp \leftarrow \Pi_{R-S}(r)$$
$$temp - \Pi_{R-S}((temp \times s) - r)$$

No extra relation is added to the database, but the relation variable created can be used in subsequent expressions. Assignment to a permanent relation would constitute a modification to the database.

The Tuple Relational Calculus

1. The tuple relational calculus is a nonprocedural language. (The relational algebra was procedural.)

   We must provide a formal description of the information desired.

2. A query in the tuple relational calculus is expressed as

$$\{t \mid P(t)\}$$

   i.e. the set of tuples t for which predicate P is true.

3. We also use the notation
   - $t[a]$ to indicate the value of tuple t on attribute a.
   - $t \in r$ to show that tuple t is in relation r.

Formal Definitions

1. A tuple relational calculus expression is of the form

$$\{t \mid P(t)\}$$

where P is a **formula**.

Several tuple variables may appear in a formula.

2. A tuple variable is said to be a **free variable** unless it is quantified by a $\exists$ or a $\forall$. Then it is said to be a **bound variable**.

3. A formula is built of **atoms**. An atom is one of the following forms:
   - $s \in r$, where s is a tuple variable, and r is a relation ($\notin$ is not allowed).
   - $s[x] \ \Theta \ u[y]$, where s and u are tuple variables, and x and y are attributes, and Θ is a comparison operator ($<, \leq, =, \neq, >, \geq$).
   - $s[x] \ \Theta \ c$, where c is a constant in the domain of attribute x.
4. **Formulae** are built up from atoms using the following rules:
   - An atom is a formula.
   - If P is a formula, then so are $\neg P$ and (P).
   - If P_1 and P_2 are formulae, then so are $P_1 \vee P_2$, $P_1 \wedge P_2$ and $P_1 \Rightarrow P_2$.
   - If $P(s)$ is a formula containing a free tuple variable s, then

$$\exists s \in r(P(s)) \text{ and } \forall s \in r(P(s))$$

   are formulae also.

5. Note some equivalences:
   - $P_1 \wedge P_2 = \neg(\neg P_1 \vee \neg P_2)$
   - $\forall t \in r(P(t)) = \neg \exists t \in r(\neg P(t))$
   - $P_1 \Rightarrow P_2 = \neg P_1 \vee P_2$

Safety of Expressions

1. A tuple relational calculus expression may generate an infinite expression, e.g.

$$\{t \mid \neg(t \in borrow\}$$

2. There are an infinite number of tuples that are not in *borrow*! Most of these tuples contain values that do not appear in the database.

3. **Safe Tuple Expressions**

We need to restrict the relational calculus a bit.

- The domain of a formula P, denoted dom(P), is the set of all values referenced in P.
- These include values mentioned in P as well as values that appear in a tuple of a relation mentioned in P.
- So, the domain of P is the set of all values explicitly appearing in P or that appear in relations mentioned in P.
- $dom(t \in borrow \wedge t[amount] < 1200)$ is the set of all values appearing in *borrow*.
- $dom(t \mid \neg(t \in borrow))$ is the set of all values appearing in *borrow*.

We may say an expression $\{t \mid P(t)\}$ is **safe** if all values that appear in the **result** are values from dom(P).

4. A **safe** expression yields a finite number of tuples as its result. Otherwise, it is called **unsafe**.

Codd's Rule for Relational DBMS

- E.F Codd was a Computer Scientist who invented the **Relational model** for Database management.
- Based on relational model, the **Relational database** was created.
- Codd proposed 13 rules popularly known as **Codd's 12 rules** to test DBMS's concept against his relational model.
- Codd's rule actualy define what quality a DBMS requires in order to become a Relational Database Management System(RDBMS).
- Till now, there is hardly any commercial product that follows all the 13 Codd's rules. Even **Oracle** follows only eight and half(8.5) out of 13. The Codd's 12 rules are as follows.

- Rule zero
- This rule states that for a system to qualify as an **RDBMS**, it must be able to manage database entirely through the relational capabilities.
-

- Rule 1: Information rule
- All information(including metadata) is to be represented as stored data in cells of tables. The rows and columns have to be strictly unordered.

- Rule 2: Guaranted Access
- Each unique piece of data(atomic value) should be accesible by : **Table Name + Primary Key(Row) + Attribute(column)**.
- **NOTE:** Ability to directly access via POINTER is a violation of this rule.

- Rule 3: Systematic treatment of NULL
- Null has several meanings, it can mean missing data, not applicable or no value. It should be handled consistently. Also, Primary key must not be null, ever. Expression on NULL must give null.
-

- Rule 4: Active Online Catalog
- Database dictionary(catalog) is the structure description of the complete **Database** and it must be stored online. The Catalog must be governed by same rules as rest of the database. The same query language should be used on catalog as used to query database.

- Rule 5: Powerful and Well-Structured Language
- One well structured language must be there to provide all manners of access to the data stored in the database. Example: **SQL**, etc. If the database allows access to the data without the use of this language, then that is a violation.
-

- Rule 6: View Updation Rule
- All the view that are theoretically updatable should be updatable by the system as well.
-

- Rule 7: Relational Level Operation
- There must be Insert, Delete, Update operations at each level of relations. Set operation like Union, Intersection and minus should also be supported.
-

- Rule 8: Physical Data Independence
- The physical storage of data should not matter to the system. If say, some file supporting table is renamed or moved from one disk to another, it should not effect the application.
-

- Rule 9: Logical Data Independence
- If there is change in the logical structure(table structures) of the database the user view of data should not change. Say, if a table is split into two tables, a new view should give result as the join of the two tables. This rule is most difficult to satisfy.

- Rule 10: Integrity Independence
- The database should be able to enforce its own integrity rather than using other programs. Key and Check constraints, trigger etc, should be stored in Data Dictionary. This also make **RDBMS** independent of front-end.
-

- Rule 11: Distribution Independence
- A database should work properly regardless of its distribution across a network. Even if a database is geographically distributed, with data stored in pieces, the end user should get an impression that it is stored at the same place. This lays the foundation of **distributed database**.
-

- Rule 12: Nonsubversion Rule
- If low level access is allowed to a system it should not be able to subvert or bypass integrity rules to change the data. This can be achieved by some sort of looking or encryption.
-

- **Relation schema:** A set of attributes is called a relation schema (or relation scheme).
- A relation schema is also known as table schema (or table scheme). A relation schema can be thought of as the basic information describing a table or relation.
- It is the logical definition of a table. Relation schema defines what the name of the table is. This includes a set of column names, the data types associated with each column.

- **Relational schema** may also refer to as database schema.
  A *database schema* is the collection of relation schemas for a whole database. **Relational or** Database schema is a collection of meta-data. **Database schema** describes the structure and constraints of data representing in a particular domain.
- A **Relational schema** can be described a blueprint of a database that outlines the way data is organized into tables. This blueprint will not contain any type of data. In a relational schema, each tuple is divided into fields called **Domains**.

- There are different kinds of database schemas:
- **Conceptual schema**
- **Logical schema**
- **Physical schema**

- A physical schema can be defined as the design of a database at its physical level. In this level, it is expressed how data is stored in blocks of storage.

- A logical schema can be defined as the design of the database at its logical level. In this level, the programmers as well as the database administrator (DBA) work.
- At this level, data can be described as certain types of data records which can be stored in the form of data structures. However, the internal details (such as an implementation of data structure) will be remaining hidden at this level.

Conceptual schema

- View schema can be defined as the design of the database at view level which generally describes end-user interaction with database systems.
- For example: Let suppose, you are storing students' information on a student's table. At the physical level, these records are described as chunks of storage (in bytes, gigabytes, terabytes or higher) in memory, and these elements often remain hidden from the programmers.
- Then comes the logical level; here in logical level these records can be illustrated as fields and attributes along with their data type(s), their relationship with each other can be logically implemented.
- Programmers generally work at this level because they are aware of such things about database systems. At view level, a user can able to interact with the system, with the help of GUI and enter the details on the screen. The users are not aware of the fact how the data is stored and what data is stored; such details are hidden from them.