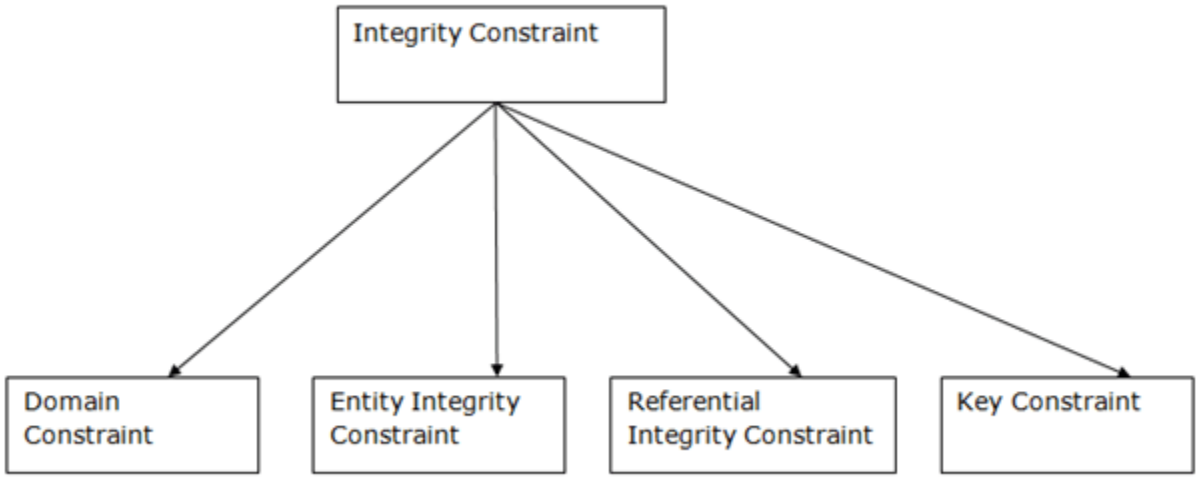


1. Domain constraints

- 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

D_No — Foreign key

18 — 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_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

(or)

RELATIONAL CONSTRAINTS:

There are three types of constraints on relational database that include

- DOMAIN CONSTRAINTS
- KEY CONSTRAINTS
- INTEGRITY CONSTRAINTS

DOMAIN CONSTRAINTS:

It specifies that each attribute in a relation an atomic value from the corresponding domains. The data types associated with commercial RDBMS domains include:

- Standard numeric data types for integer
- Real numbers
- Characters
- Fixed length strings and variable length strings

Thus, domain constraints specifies the condition that we to put on each instance of the relation. So the values that appear in each column must be drawn from the domain associated with that column.

Rollno	Name	City	Age
101	Sujit	Bam	23
102	Kunal	bbsr	22

Key constraints:

This constraints states that the key attribute value in each tuple msut be unique .i.e, no two tuples contain the same value for the key attribute.(null values can allowed)

Emp(empcode,name,address) . here empcode can be unique

Integrity constraints:

There are two types of integrity constraints:

- Entity integrity constraints
- Referential integrity constraints

Entity integrity constraints:

It states that no primary key value can be null and unique. This is because the primary key is used to identify individual tuple in the relation. So we will not be able to identify the records uniquely containing null values for the primary key attributes. This constraint is specified on one individual relation.

Referential integrity constraints:

It states that the tuple in one relation that refers to another relation must refer to an existing tuple in that relation. This constraints is specified on two relations .
If a column is declared as foreign key that must be primary key of another table.

Department(deptcode,dname)
Here the deptcode is the primary key.
Emp(*empcode*,name,city,*deptcode*).
Here the deptcode is foreign key.

CODD'S RULES

Rule 1 : The information Rule.

"All information in a relational data base is represented explicitly at the logical level and in exactly one way - by values in tables."

Everything within the database exists in tables and is accessed via table access routines.

Rule 2 : Guaranteed access Rule.

"Each and every datum (atomic value) in a relational data base is guaranteed to be logically accessible by resorting to a combination of table name, primary key value and column name."

To access any data-item you specify which column within which table it exists, there is no reading of characters 10 to 20 of a 255 byte string.

Rule 3 : Systematic treatment of null values.

"Null values (distinct from the empty character string or a string of blank characters and distinct from zero or any other number) are supported in fully relational DBMS for representing missing information and inapplicable information in a systematic way, independent of data type."

If data does not exist or does not apply then a value of NULL is applied, this is understood by the RDBMS as meaning non-applicable data.

Rule 4 : Dynamic on-line catalog based on the relational model.

"The data base description is represented at the logical level in the same way as ordinary data, so that authorized users can apply the same relational language to its interrogation as they apply to the regular data."

The Data Dictionary is held within the RDBMS, thus there is no-need for off-line volumes to tell you the structure of the database.

Rule 5 : Comprehensive data sub-language Rule.

"A relational system may support several languages and various modes of terminal use (for example, the fill-in-the-blanks mode). However, there must be at least one language whose statements are expressible, per some well-defined syntax, as character strings and that is comprehensive in supporting all the following items

- Data Definition
- View Definition
- Data Manipulation (Interactive and by program).
- Integrity Constraints
- Authorization.

Every RDBMS should provide a language to allow the user to query the contents of the RDBMS and also manipulate the contents of the RDBMS.

Rule 6 : .View updating Rule

"All views that are theoretically updateable are also updateable by the system."

Not only can the user modify data, but so can the RDBMS when the user is not logged-in.

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

"The capability of handling a base relation or a derived relation as a single operand applies not only to the retrieval of data but also to the insertion, update and deletion of data."

The user should be able to modify several tables by modifying the view to which they act as base tables.

Rule 8 : Physical data independence.

"Application programs and terminal activities remain logically unimpaired whenever any changes are made in either storage representations or access methods."

The user should not be aware of where or upon which media data-files are

stored Rule 9 : Logical data independence.

"Application programs and terminal activities remain logically unimpaired when information-preserving changes of any kind that theoretically permit un-impairment are made to the base tables."

User programs and the user should not be aware of any changes to the structure of the tables (such as the addition of extra columns).

Rule 10 : Integrity independence.

"Integrity constraints specific to a particular relational data base must be definable in the relational data sub-language and storable in the catalog, not in the application programs."

If a column only accepts certain values, then it is the RDBMS which enforces these constraints and not the user program, this means that an invalid value can never be entered into this column, whilst if the constraints were enforced via programs there is always a chance that a buggy program might allow incorrect values into the system.

Rule 11 : Distribution independence.

"A relational DBMS has distribution independence."

The RDBMS may spread across more than one system and across several networks, however to the end-user the tables should appear no different to those that are local.

Rule 12 : Non-subversion Rule.

"If a relational system has a low-level (single-record-at -a-time) language, that low level cannot be used to subvert or bypass the integrity Rules and constraints expressed in the higher level relational language (multiple-records-at-a-time)."

RELATION ALGEBRA:

Relational algebra is a set of basic operations used to manipulate the data in relational model. These operations enable the user to specify basic retrieval request. The result of retrieval is anew relation, formed from one or more relation. These operation can be classified in two categories.

- **Basic Set Operation**
  - Union
  - Intersection
  - Set difference
  - Cartesian product
- **Relational operations**
  - Select
  - Project
  - Join
  - Division

Basic set operation:

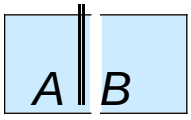
These are the binary operations; i.e, each is applied to two sets or relations. These two relations should be union compatible except in case of Cartesian product.

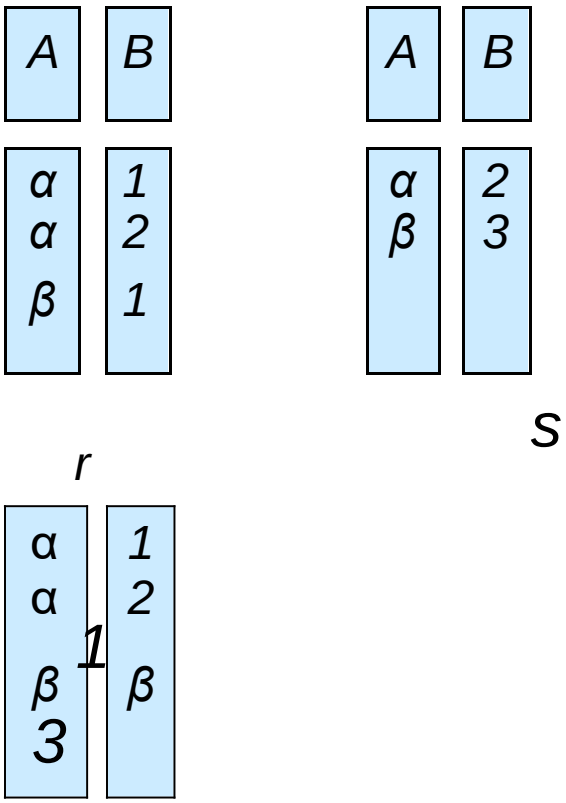
Two relations R(A1,A2…,AN) and S(B1,B2…BN) are said to be union compatible if they have the same degree n and domains of the corresponding attributes are also the same; domain(Ai)=Domain(bI) for 1<=i<=n.

Union Operation – Example

Relations *r, s:*

r ∪ s:





UNION OPERATION:

Notation: $r \cup s$

Defined as:

$$r \cup s = \{t \mid t \in r \text{ or } t \in s\}$$

For $r \cup s$ **to be valid.**

***r, s* must have the *same arity* (same number of attributes)**

2. The attribute domains must be *compatible* (e.g., 2nd column of *r* deals with the same type of values as does the 2nd column of *s*)

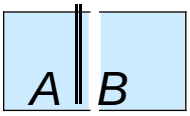
E.g. to find all customers with either an account or a loan $\Pi_{customer\text{-}name}$ (*depositor*) $\cup$ $\Pi_{customer\text{-}name}$ (*borrower*)

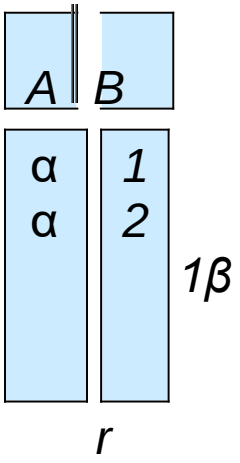
Set Difference

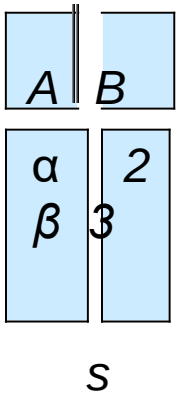
Operation – Example

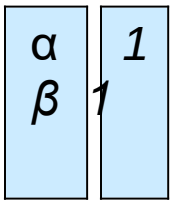
Relations *r, s:*

***r – s*:**









Set Difference Operation

- **Notation *r – s***
- **Defined as:**
  - ***r – s* = {*t* | *t* ∈ *r* and t ∉ *s*}**
- **Set differences must be taken between *compatible* relations.**
  - ***r* and *s* must have the *same arity***
  - **attribute domains of *r* and *s* must be compatible**

Cartesian product – Example

A	B
α	1
β	2

r

C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

r x *s*:

Relation *r*:

A	B
α	10
α	20
β	30
β	40

C
1
1
1
2

Notation *r* x *s*

Defined as:

$\prod_{A,C}(r)$

$r \times s = \{t\, q \mid t \in r \text{ and } q \in s\}$

Notation:

$\prod_{A1, A2, \ldots, Ak}(r)$

where A_1, A_2 are attribute names and r is a relation name.

A	C
α	1
α	1
β	2

=

A	C
α	1
β	2

used.

The result is defined as the relation of *k* columns obtained by erasing the columns that are not listed

Duplicate rows removed from result, since relations are sets

E.g. To eliminate the *branch-name* attribute of *account*

$\prod_{account\text{-}number,\ balance}(account)$

Select Operation – Example

- Relation ***r***

A	B	C	D
α	α	1	7
α	β	5	7
β	β	12	3
β	β	23	10

- $\sigma_{A=B \wedge D>5}(r)$

A	B	C	D
α	α	1	7
β	β	23	10

Notation: $\sigma_p(r)$

***p* is called the selection predicate**

Defined as:

$$\sigma_p(r) = \{t \mid t \in r \text{ and } p(t)\}$$

Where *p* is a formula in propositional calculus consisting of terms connected by : ∧ (and), ∨ (or), ¬ (not) Each term is one of:

<attribute> *op* <attribute> or <constant>

where *op* is one of: =, ≠, >, ≥. <. ≤

Example of selection:

Q: Display the account details belonging to the branch "perryridge". $\sigma_{branch\text{-}name=\text{“}Perryridge\text{”}}(account)$

Rename Operation

Allows us to name, and therefore to refer to, the results of relational-algebra expressions.
Allows us to refer to a relation by more than one name.
Example:

$$\rho_{x}(E)$$

returns the expression *E* under the name *X*
If a relational-algebra expression *E* has arity *n*, then

$$\rho_{x(A1, A2, \ldots, An)}(E)$$

returns the result of expression *E* under the name *X*, and with the attributes renamed to *A1, A2, …., An*.

Set-Intersection Operation

Notation: $r \cap s$

Defined as:

$r \cap s = \{ t \mid t \in r \text{ and } t \in s \}$

Assume:
r, *s* have the *same arity*
attributes of r and s are compatible

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

Set-Intersection Operation - Example

- Relation r, s:

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

- r ∩ s

A	B
α	2

Natural-Join Operation

Let *r* and *s* be relations on schemas *R* and *S* respectively.

Notation: r ⨯ s

Then, r ⨯ s is a relation on schema $R \cup S$ obtained as follows:

Consider each pair of tuples t_r from *r* and t_s from *s*.

If t_r and t_s have the same value on each of the attributes in $R \cap S$, add a tuple *t* to the result, where

t has the same value as t_r on *r* *t*

has the same value as t_s on *s*

Example:

$R=(A,B,C,D)$

$S=(E,B,D)$

Result schema = (A, B, C, D, E)

$r \bowtie s$ is defined as:

$$\Pi_{r.A,\ r.B,\ r.C,\ r.D,\ s.E} (\sigma_{r.B = s.B \ \wedge \ r.D = s.D} (r \times s))$$

Natural Join Operation – Example

Relations r, s:

r

A	B	C	D
α	1	α	a
β	2	γ	a
γ	4	β	b
α	1	γ	a
δ	2	β	b

s

B	D	E
1	a	α
3	a	β
1	b	γ
2	b	δ
3		∈

$r \bowtie s$

A	B	C	D	E
α	1	α	a	α
α	1	γ	a	γ
α	1	γ	a	α
α	1	β	b	γ
δ	2			δ

Division Operation

$$r \div s$$

Suited to queries that include the phrase "for all".
Let *r* and *s* be relations on schemas R and S respectively where

$R = (A1, \ldots, Am, B1, \ldots, Bn)$

$S = (B1, \ldots, Bn)$

The result of r ÷ s is a relation on schema

$R - S = (A1, \ldots, Am)$

$$r \div s = \{ t \mid t \in \prod_{R-S}(r) \wedge \forall u \in s (tu \in r) \}$$

Division Operation – Example

Relations *r*, *s*:

A	B
α	1
α	2
α	3
β	1
γ	1
δ	1
δ	3
δ	4
∈	6
∈	1
β	2

r

B
1
2

s

r ÷ *s*:

A
α
β

Example Queries

- Find all customers who have an account from at least the “Downtown” and the Uptown” branches.

Query 1

$\prod$**CN(**σ*BN*=“Downtown”(*depositor* ⋈*account***)**) ∩

$\prod$*CN*(**σ***BN*=**“Uptown”**(*depositor* ⋈*account*)**)**

where *CN* denotes customer-name and *BN* denotes *branch-name*.

Query 2

$$\prod_{\textit{customer-name, branch-name}} (\textit{depositor} \bowtie \textit{account}) \div \rho\textit{temp(branch-name}) (\{(\text{“Downtown”}), (\text{“Uptown”})\})$$

Example Queries

- Find all customers who have an account at all branches located in Brooklyn city.

$$\prod_{\textit{customer-name, branch-name}} (\textit{depositor} \bowtie \textit{account}) \div \prod_{\textit{branch-name}} (\sigma_{\textit{branch-city} = \text{“Brooklyn”}} (\textit{branch}))$$

Banking Example

branch (branch-name, branch-city, assets)

customer (customer-name, customer-street, customer-only)

account (account-number, branch-name, balance)

loan (loan-number, branch-name, amount)

depositor (customer-name, account-number)

borrower (customer-name, loan-number)

Example Queries

- Find all loans of over $1200

$$\sigma_{amount > 1200}(loan)$$

- Find the loan number for each loan of an amount greater than $1200

$$\prod_{loan\text{-}number}(\sigma_{amount > 1200}(loan))$$

Example Queries

- Find the names of all customers who have a loan, an account, or both, from the bank

$$\prod_{customer\text{-}name}(borrower) \cup \prod_{customer\text{-}name}(depositor)$$

- Find the names of all customers who have a loan and an account at bank.

$$\prod_{customer\text{-}name}(borrower) \cap \prod_{customer\text{-}name}(depositor)$$

Example Queries

Find the names of all customers who have a loan at the Perryridge branch.

$$\prod_{customer\text{-}name}(\sigma_{branch\text{-}name=\text{“}Perryridge\text{”}}(\sigma_{borrower.loan\text{-}number = loan.loan\text{-}number}(borrower \times loan)))$$

Find the names of all customers who have a loan at the Perryridge branch but do not have an account at any branch of the bank.

$$\prod_{customer\text{-}name}(\sigma_{branch\text{-}name = \text{“}Perryridge\text{”}}(\sigma_{borrower.loan\text{-}number = loan.loan\text{-}number}(\text{borrower x loan}))) - \prod_{customer\text{-}name}(\text{depositor})$$

Example Queries

- Find the names of all customers who have a loan at the Perryridge branch.
  - Query 1

$$\prod_{\text{customer-name}}(\sigma_{\text{branch-name = “Perryridge”}}(\sigma_{\text{borrower.loan-number = loan.loan-number}}(\text{borrower x loan})))$$

  - Query 2

$$\prod_{\text{customer-name}}(\sigma_{\text{loan.loan-number = borrower.loan-number}}((\sigma_{\text{branch-name = “Perryridge”}}(\text{loan})) \times \text{borrower}))$$

Example Queries

Find the largest account balance

- Rename *account* relation as *d*
- The query is:

$$\prod_{balance}(account) - \prod_{account.balance}(\sigma_{account.balance < d.balance}(account \times \rho_{d}(account)))$$

Aggregate functions:

Aggregation function takes a collection of values and returns a single value as a result.

avg: average value
min: minimum value
max: maximum value
sum: sum of values
count: number of values

Aggregate operation in relational algebra

G1, G2, …, Gn g F1(A1), F2(A2),…, Fn(An) (*E*)

E is any relational-algebra expression
*G*1, *G*2 …, *G*n is a list of attributes on which to group (can be empty)
Each *Fi* is an aggregate function.
Each *Ai* is an attribute name

Find sum of sal fo all emp records
$\mathcal{G}_{sum(sal)}(emp)$

Find maximum salalry from emp table
$\mathcal{G}_{max(salary)}(emp)$

Find branch name and maximum salalry from emp table $Branch\_name\ \mathcal{G}_{,max(salary)}(emp)$

Aggregate Operation – Example

- Relation *r*:

A	B	C
α	α	7
α	β	7
β	β	3
β	β	10

$g_{sum(c)}(r)$

sum-C
27

OUTER JOIN:

The outer join operation is an extension of the join operation to deal with missing information.

There are three forms of outer join

- left outer join
- right outer join
- full outer join

employee:

Empname	Street	City
Coyote	Toon	Hollywood
Rabbit	Tunnel	carrot
Smith	Revolver	Death valley
William	Seaview	Seattle

Ft_works:

Empname	Branch name	Salary
Coyote	Mesa	1500
Rabbit	Mesa	1300
Gates	Redmond	5300
William	Redmond	1500

Employee $\bowtie$ **ft_works**

Empname	Street	City	Branch name	Salary
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	carrot	Mesa	1300
William	Seaview	Seattle	Redmond	1500

Left outer join:

It takes all tuples in the left relation that did not match with any tuple in the right relation, pads the tuples with null values for all other attributes. The right relation and adds them to the result of the natural join. In tuple (smith, Revolcer, Death valley, null, null) is such a tuple. All information from the left relation is present in the result of the left outer join.

Empname	Street	City	Branch name	Salary
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	carrot	Mesa	1300
William	Seaview	Seattle	Redmond	1500
Smith	Revolver	Death valley	Null	null

Result of Employee ⟕ ft_works

Right outer join:

It is symmetric with the left outer join. It pads tuples from the right relation that did not match any from the left relation with nulls and adds them to the result of the natural join. tuple(Gates,null,null,Redmond,5300) is such a tuple. Thus, all information from the right relation is present in the result of the right outer join.

Empname	Street	City	Branch name	Salary
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	carrot	Mesa	1300
William	Seaview	Seattle	Redmond	1500
gates	Null	null	Redmond	5300

Full outer join:

It does both of those operations, padding tuples from the left relation that did not match any from the right relation, as well as tuples from the right relation that did not match any from the left relation, and adding them to the result of the join. Figure 3.35 shows the result of a full outer join.

Since outer join operations may generate results containing null values, we need to specify how the different relation-algebra operations deal with null values. It is interesting to note that the outer join operations can be expressed by the basic relational algebra operations. For instance the left outer join operation

Employee ⟗ ft_works

Empname	Street	City	Branch name	Salary
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	carrot	Mesa	1300
William	Seaview	Seattle	Redmond	1500
gates	Null	null	Redmond	5300

Tuple Relational Calculus

The tuple relational calculus is a non procedural query language. It describes the desired information with out giving a specific procedure for obtaining that information.

A query in the tuple relational calculus is expressed as:

{t|P(t)}

where P is a formula. Several tuple variables may appear in a formula. A tuple variable is said to be a free variable unless it is quantified by a ∃ or ∀.

n A nonprocedural query langua**g**e, where each query is of the form

Bankin Example

Example{|()} **Queries**

t P t

branch (branch-name, branch-city, assets)

nn FindItis the *loan*setof*-number,*alltuples*branch*tsuch*-name,*that predicate^and *amount*Pis^for true^loans for ^of *t*

*customer*over$1200*(customer-name, customer-street, customer-city)*

account (account-number, branch-name, balance)

n *t* is a *tuple variable*{*t*|*t*∈, *loan*t[*A*]denotes∧[*amount*the] value>1200}of tuple *t* on attribute *A*

loan (loan-number, branch -name, amount)

depositor (customer-name, account-number)

n *t* ∈ *r* denotes that tuple *t* is in relation *r*

*borrower (customer***Predicate***-name,loan***Ca***-number)***culus Formula**

n Find the loan number for each loan of an amount greater than $1200

1. *Set of attributes and constants*

2. n *Set**P* is*of* a*comparison**formula** operators:*similarto*(e*that*.g.,*<of,≤ ,the=,≠ ,predicate>,≥)* calculus

{*t* | ∃ *s* ∈*connectives:*loan([*loan-number*] = *s*[*loan-number*] ∧ *s* [*amount*] > 1200)}

3. *Set of* *and (∧), or (v), not (¬)*

4. *Implication (⇒): x ⇒ y, if x if true, then y is true*

x ⇒ y ≡ ¬x v y

5. Set of quantifiers:

Notice that a relation on schema [*loan-number*] is implicitly defined

- ∃ *t* ∈ *r (Q(t))* ≡ ”there exists” a tuple in *t* in relation *r*

by the query such that predicate *Q(t)* is true

- ∀*t* ∈ *r* (*Q*(*t*)) ≡ *Q* is true “for all” tuples *t* in relation *r*

Example Queries

Find the names of **Example** allcustomers

having **Queries** aloan, an account, or both at the bank

Find

thethe namesnames ofof allallcustomerscustomershavinghavinga aloan,loananat account,the

{ *t* | ∃*s* ∈ *borrower(t*[*customer-name*] = *s*[*customer-name*]) ∨ ∃*u* ∈ Perryridge branch

or both at the bank

depositor(*t*[*customer-name*] = *u*[*customer-name*])

{*t* | ∃*s* ∈ [-] = *s*[- ∧

{*t* | ∃*s* ∈borrower(trower(tcustomer[-name] = s[customer name]) ∨

∃*u* ∈ *loan*(*u*[*branch-name*] = “Perryridge” ∧

∃u ∈ depositor (t[customer- name] = u [customer-name])

Find the names of all customers who have a loan and an account

u[*loan-number*] = *s*[*loan-number*]))}

at the bank

Safety of Expressions

Find the names of all customers who have a loan and an

{*t* | ∃*s* ∈ *borrower(t*[*customer-name*] = *s*[*customer-name*])

account ∧ ∃*u* at the bank

∈ *depositor*(*t*[*customer -name*] = *u*[*customer-name*])

It is possible to write tuple calculus expressions that generate

infinite{| ∃*s* relations ∈*borrower*(. *t*[*customer-name*] = *s*[*customer-*

Domain Relational Calculus

A nonprocedural query language equivalent in power to the tuple relational calculus

Each query is an expression of the form:

$$\{ < x_1, x_2, \ldots, x_n > \mid P(x_1, x_2, \ldots, x_n)\}$$

P represents a formula similar to that of the predicate calculus

Find the names of all customers who have a loan of over $1200

- A graphical query language which is based (roughly) on the domain relational calculus

$$\{< c > \mid \exists\ l, b, (< c, l > \in borrower \wedge < l, b, a > \in loan \wedge a > 1200)\}$$

- Two dimensional syntax – system creates templates of relations that are requested by users

Perryridge branch and the loan amount:

- Queries are expressed as

$$\{< c, a > \mid \exists l (< c, l > \in borrower \wedge \exists b (< l, b, a > \in loan \wedge b = \text{“Perryridge”}))\}$$

or

$$\{< c, a > \mid \exists\ l\ (< c, l > \in borrower \wedge < l, \text{“Perryridge”}, a > \in loan)\}$$

(or)

Relational Calculus

- Relational calculus is a non-procedural query language. In the non-procedural query language, the user is concerned with the details of how to obtain the end results.
- The relational calculus tells what to do but never explains how to do.

Types of Relational calculus:

1. Tuple Relational Calculus (TRC)

- The tuple relational calculus is specified to select the tuples in a relation. In TRC, filtering variable uses the tuples of a relation.
- The result of the relation can have one or more tuples.

Notation:

1. {T | P (T)} or {T | Condition (T)}

Where

T is the resulting tuples

P(T) is the condition used to fetch T.

For example:

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

OUTPUT: This query selects the tuples from the AUTHOR relation. It returns a tuple with 'name' from Author who has written an article on 'database'.

TRC (tuple relation calculus) can be quantified. In TRC, we can use Existential (∃) and Universal Quantifiers (∀).

For example:

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

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

2. Domain Relational Calculus (DRC)

- The second form of relation is known as Domain relational calculus. In domain relational calculus, filtering variable uses the domain of attributes.
- Domain relational calculus uses the same operators as tuple calculus. It uses logical connectives ∧ (and), ∨ (or) and ¬ (not).
- It uses Existential (∃) and Universal Quantifiers (∀) to bind the variable.

Notation:

1. { a1, a2, a3, ..., an | P (a1, a2, a3, ... ,an)}

Where

a1, a2 are attributes
P stands for formula built by inner attributes

For example:

1. {< article, page, subject > | ∈ javatpoint ∧ subject = 'database'}

Output: This query will yield the article, page, and subject from the relational javatpoint, where the subject is a database.

RELATIONAL DATABASE DEGIN

Data base design is a process in which you create a logical data model for a database, which store data of a company. It is performed after initial database study phase in the database life cycle. You use normalization technique to create the logical data model for a database and eliminate data redundancy. Normalization also allows you to organize data efficiently in a data base and reduce anomalies during data operation. Various normal forms, such as first, second and third can be applied to create a logical data model for a database. The second and third normal forms are based on partial dependency and transitivity dependency. Partial dependency occurs when a row of table is uniquely identified by one column that is a part of a primary key. A transitivity dependency ours when a non key column is uniquely identified by values in another non-key column of a table.

Data base design process:

We can identify six main phases of the database design process:

1. Requirement collection and analysis
2. Conceptual data base design
3. Choice of a DBMS
4. Data model mapping(logical database design)
5. physical data base design
6. database system implementation and tuning

1. **Requirement collection and analysis**

   Before we can effectively design a data base we must know and analyze the expectation of the users and the intended uses of the database in as much as detail.

2. **Conceptual data base design**

   The goal for this phase I s to produce a conceptual schema for the database that is independent of a specific DBMS.

   - We often use a high level data model such er-model during this phase
   - We specify as many of known database application on transactions as possible using a notation the is independent of any specific dbms.
   - Often the dbms choice is already made for the organization the intent of conceptual design still to keep , it as free as possible from implementation consideration.

3. **Choice of a DBMS**

   The choice of dbms is governed by a no. of factors some technical other economic and still other concerned with the politics of the organization.

   The economics and organizational factors that offer the choice of the dbms are:

   Software cost, maintenance cost, hardware cost, database creation and conversion cost, personnel cost, training cost, operating cost.

4. **Data model mapping (logical database design)**
   During this phase, we map the conceptual schema from the high level data model used on phase 2 into a data model of the choice dbms.
5. **Physical databse design**
   During this phase we design the specification for the database in terms of physical storage structure ,record placement and indexes.
6. **Database system implementation and tuning**
   During this phase, the database and application programs are implemented, tested and eventually deployed for service.