Chapter Six

Relational Query Languages

In addition to the structural component of any data model equally important is the manipulation mechanism. This component of any data model is called the "query language".

- ***Query languages:*** Allow manipulation and retrieval of data from a database.
- Query Languages! **=** programming languages!
  - QLs not intended to be used for complex calculations.
  - QLs support easy, efficient access to large data sets.
- Relational model supports simple, powerful query languages.

Formal Relational Query Languages

- There are varieties of Query languages used by relational DBMS for manipulating relations.

- Some of them are **procedural**
  - User tells the system exactly ***what*** and ***how*** to manipulate the data
- Others are **non-procedural**
  - User states ***what*** data is needed rather than ***how*** it is to be retrieved.

Two mathematical Query Languages form the basis for Relational Query Languages

- *Relational Algebra*:
- *Relational Calculus*:

- We may describe the ***relational algebra as procedural language***: it can be used to tell the DBMS how to build a new relation from one or more relations in the database.
- We may describe ***relational calculus as a non procedural language***: it can be used to formulate the definition of a relation in terms of one or more database relations.
- Formally the relational algebra and relational calculus are equivalent to each other. ***For every expression in the algebra, there is an equivalent expression in the calculus.***
- Both are non-user friendly languages. They have been used as the basis for other, higher-level data manipulation languages for relational databases.

A query is applied to relation instances, and the result of a query is also a relation instance.

- *Schemas* of input relations for a query are fixed
- The schema for the *result* of a given query is also fixed! Determined by definition of query language constructs.

Relational Algebra

The basic set of operations for the relational model is known as the relational algebra. These operations enable a user to specify basic retrieval requests.

The result of the retrieval is a new relation, which may have been formed from one or more relations. The **algebra operations** thus produce new relations, which can be further manipulated using operations of the same algebra.

A sequence of relational algebra operations forms a **relational algebra expression**, whose result will also be a relation that represents the result of a database query (or retrieval request).

- Relational algebra is a theoretical language with operations that work on one or more relations to define another relation without changing the original relation.
- The output from one operation can become the input to another operation (nesting is possible)

- **There are different basic operations that could be applied on relations on a database based on the requirement.**
  - ***Selection*** **(±)** Selects a subset of rows from a relation.
  - ***Projection (®)*** Deletes unwanted columns from a relation.
  - ***Renaming:*** assigning intermediate relation for a single operation
  - ***Cross-Product (x)*** Allows to concatenate a tuple from one relation with all the tuples from the other relation.
  - ***Set-Difference (-)*** Tuples in relation R_1, but not in relation R_2.
  - ***Union*** (Ć) Tuples in relation R_1, or in relation R_2.
  - ***Intersection*** (ą) Tuples in relation R_1 and in relation R_1
  - ***Join*** ⋈ Tuples joined from two relations based on a condition

Join and intersection are derivable from the rest.

- Using these, we can build up sophisticated database queries.

Table1:

Sample table used to illustrate different kinds of relational operations. The relation contains information about employees, IT skills they have and the school where they attend each skill.

Employee

EmpID	***FName***	***LName***	***SkillID***	***Skill***	***SkillType***	***School***	***SchoolAdd***	***SkillLevel***
12	Abebe	Mekuria	2	SQL	Database	AAU	Sidist_Kilo	5
16	Lemma	Alemu	5	C++	Programming	Unity	Gerji	6
28	Chane	Kebede	2	SQL	Database	AAU	Sidist_Kilo	10
25	Abera	Taye	6	VB6	Programming	Helico	Piazza	8
65	Almaz	Belay	2	SQL	Database	Helico	Piazza	9
24	Dereje	Tamiru	8	Oracle	Database	Unity	Gerji	5
51	Selam	Belay	4	Prolog	Programming	Jimma	Jimma City	8
94	Alem	Kebede	3	Cisco	Networking	AAU	Sidist_Kilo	7
18	Girma	Dereje	1	IP	Programming	Jimma	Jimma City	4
13	Yared	Gizaw	7	Java	Programming	AAU	Sidist_Kilo	6

1. Selection

- Selects subset of tuples/rows in a relation that satisfy ***selection condition***.
- Selection operation is a unary operator (it is applied to a single relation)
- The Selection operation is applied to each tuple individually
- The degree of the resulting relation is the same as the original relation but the cardinality (no. of tuples) is less than or equal to the original relation.
- The Selection operator is commutative.
- Set of conditions can be combined using Boolean operations (ė**(AND)**, Ę**(OR)**, and ~**(NOT)**)
- No duplicates in result!
- *Schema* of result identical to schema of (only) input relation.
- *Result* relation can be the *input* for another relational algebra operation! (*Operator composition.*)
- It is a filter that keeps only those tuples that satisfy a qualifying condition (those satisfying the condition are selected while others are discarded.)

Notation:

σ *<Selection Condition>* *<Relation Name>*

Example: Find all Employees with skill type of Database.

σ *< SkillType ="Database">* *(Employee)*

This query will extract every tuple from a relation called Employee with all the attributes where the SkillType attribute with a value of "Database".

The resulting relation will be the following.

EmpID	*FName*	*LName*	*SkillID*	*Skill*	*SkillType*	*School*	*SchoolAdd*	*SkillLevel*
12	Abebe	Mekuria	2	SQL	Database	AAU	Sidist_Kilo	5
28	Chane	Kebede	2	SQL	Database	AAU	Sidist_Kilo	10
65	Almaz	Belay	2	SQL	Database	Helico	Piazza	9
24	Dereje	Tamiru	8	Oracle	Database	Unity	Gerji	5

If the query is all employees with a SkillType *Database* and School *Unity* the relational algebra operation and the resulting relation will be as follows.

σ *< SkillType ="Database" AND School="Unity">* *(Employee)*

EmpID	*FName*	*LName*	*SkillID*	*Skill*	*SkillType*	*School*	*SchoolAdd*	*SkillLevel*
24	Dereje	Tamiru	8	Oracle	Database	Unity	Gerji	5

2. Projection

- Selects certain attributes while discarding the other from the base relation.
- The PROJECT creates a vertical partitioning – one with the needed columns (attributes) containing results of the operation and other containing the discarded Columns.
- Deletes attributes that are not in *projection list*.
- *Schema* of result contains exactly the fields in the projection list, with the same names that they had in the (only) input relation.
- Projection operator has to eliminate *duplicates*!
  - Note: real systems typically don't do duplicate elimination unless the user explicitly asks for it.
- If the Primary Key is in the *projection list*, then duplication will not occur
- Duplication removal is necessary to insure that the resulting table is also a relation.

Notation:

π *<Selected Attributes>* ***<Relation Name>***

Example: To display Name, Skill, and Skill Level of an employee, the query and the resulting relation will be:

π *<FName, LName, Skill, Skill_Level>* ***(Employee)***

FName	*LName*	*Skill*	*SkillLevel*
Abebe	Mekuria	SQL	5
Lemma	Alemu	C++	6
Chane	Kebede	SQL	10
Abera	Taye	VB6	8
Almaz	Belay	SQL	9
Dereje	Tamiru	Oracle	5
Selam	Belay	Prolog	8
Alem	Kebede	Cisco	7
Girma	Dereje	IP	4
Yared	Gizaw	Java	6

If we want to have the Name, Skill, and Skill Level of an employee with Skill SQL and SkillLevel greater than 5 the query will be:

π*<FName, LName, Skill, Skill_Level>* ***(***σ*<Skill="SQL" è SkillLevel>5>****(Employee))***

FName	*LName*	*Skill*	*SkillLevel*
Chane	Kebede	SQL	10
Almaz	Belay	SQL	9

3. *Rename Operation*

- We may want to apply several relational algebra operations one after the other. The query could be written in two different forms:
    1. Write the operations as a single relational algebra expression by nesting the operations.
    2. Apply one operation at a time and create intermediate result relations. In the latter case, we must give names to the relations that hold the intermediate results➔Rename Operation

If we want to have the Name, Skill, and Skill Level of an employee with salary greater than 1500 and working for department 5, we can write the expression for this query using the two alternatives:

1. A single algebraic expression:
   The above used query is using a single algebra operation, which is:

$$\pi_{<FName,\ LName,\ Skill,\ Skill\_Level>}\ (\sigma_{<Skill="SQL"\ \text{è}\ SkillLevel>5>}(Employee))$$

2. Using an intermediate relation by the Rename Operation:

Step1: $Result1 \leftarrow \sigma_{<DeptNo=5\ \text{è}\ Salary>1500>}\ (Employee)$

Step2: $Result \leftarrow \pi_{<FName,\ LName,\ Skill,\ Skill\_Level>}(Result1)$

Then Result will be equivalent with the relation we get using the first alternative.

4. Set Operations

The three main set operations are the Union, Intersection and Set Difference. The properties of these set operations are similar with the concept we have in mathematical set theory. The difference is that, in database context, the elements of each set, which is a Relation in Database, will be tuples. The set operations are Binary operations which demand the two operand Relations to have type compatibility feature.

Type Compatibility

Two relations R_1 and R_2 are said to be Type Compatible if:

1. The operand relations $R_1(A_1, A_2, ..., A_n)$ and $R_2(B_1, B_2, ..., B_n)$ have the same number of attributes, and
2. The domains of corresponding attributes must be compatible; that is, $Dom(A_i)=Dom(B_i)$ for i=1, 2, ..., n.

To illustrate the three set operations, we will make use of the following two tables:

Employee

EmpID	***FName***	***LName***	***SkillID***	***Skill***	***SkillType***	***School***	***SkillLevel***
12	Abebe	Mekuria	2	SQL	Database	AAU	5
16	Lemma	Alemu	5	C++	Programming	Unity	6
28	Chane	Kebede	2	SQL	Database	AAU	10
25	Abera	Taye	6	VB6	Programming	Helico	8
65	Almaz	Belay	2	SQL	Database	Helico	9
24	Dereje	Tamiru	8	Oracle	Database	Unity	5
51	Selam	Belay	4	Prolog	Programming	Jimma	8
94	Alem	Kebede	3	Cisco	Networking	AAU	7
18	Girma	Dereje	1	IP	Programming	Jimma	4
13	Yared	Gizaw	7	Java	Programming	AAU	6

RelationOne: *Employees who attend Database Course*

EmpID	***FName***	***LName***	***SkillID***	***Skill***	***SkillType***	***School***	***SkillLevel***
12	Abebe	Mekuria	2	SQL	Database	AAU	5
28	Chane	Kebede	2	SQL	Database	AAU	10
65	Almaz	Belay	2	SQL	Database	Helico	9
24	Dereje	Tamiru	8	Oracle	Database	Unity	5

RelationTwo : *Employees who attend a course in AAU*

EmpID	***FName***	***LName***	***SkillID***	***Skill***	***SkillType***	***School***	***SkillLevel***
12	Abebe	Mekuria	2	SQL	Database	AAU	5
94	Alem	Kebede	3	Cisco	Networking	AAU	7
28	Chane	Kebede	2	SQL	Database	AAU	10
13	Yared	Gizaw	7	Java	Programming	AAU	6

a. UNION Operation

The result of this operation, denoted by R U S, is a relation that includes all tuples that are either in R or in S or in both R and S. Duplicate tuple is eliminated.

The two operands must be "*type compatible*"

Eg: RelationOne U RelationTwo

Employees who attend Database in any School or who attend any course at AAU

EmpID	***FName***	***LName***	***SkillID***	***Skill***	***SkillType***	***School***	***SkillLevel***
12	Abebe	Mekuria	2	SQL	Database	AAU	5
28	Chane	Kebede	2	SQL	Database	AAU	10
65	Almaz	Belay	2	SQL	Database	Helico	9
24	Dereje	Tamiru	8	Oracle	Database	Unity	5
94	Alem	Kebede	3	Cisco	Networking	AAU	7
13	Yared	Gizaw	7	Java	Programming	AAU	6

b. INTERSECTION Operation

The result of this operation, denoted by R ∩ S, is a relation that includes all tuples that are in both R and S. The two operands must be "*type compatible*"

Eg: RelationOne ∩ RelationTwo

Employees who attend Database Course at AAU

EmpID	***FName***	***LName***	***SkillID***	***Skill***	***SkillType***	***School***	***SkillLevel***
12	Abebe	Mekuria	2	SQL	Database	AAU	5
28	Chane	Kebede	2	SQL	Database	AAU	10

c. Set Difference (or MINUS) Operation

The result of this operation, denoted by R - S, is a relation that includes all tuples that are in R but not in S.

The two operands must be "*type compatible*"

Eg: RelationOne - RelationTwo

Employees who attend Database Course but didn't take any course at AAU

EmpID	***FName***	***LName***	***SkillID***	***Skill***	***SkillType***	***School***	***SkillLevel***
65	Almaz	Belay	2	SQL	Database	Helico	9
24	Dereje	Tamiru	8	Oracle	Database	Unity	5

Eg: RelationTwo – RelationOne

Employees who attend Database Course but didn't take any course at AAU

EmpID	***FName***	***LName***	***SkillID***	***Skill***	***SkillType***	***School***	***SkillLevel***
12	Abebe	Mekuria	2	SQL	Database	AAU	5
94	Alem	Kebede	3	Cisco	Networking	AAU	7
28	Chane	Kebede	2	SQL	Database	AAU	10
13	Yared	Gizaw	7	Java	Programming	AAU	6

The resulting relation for; R1 ∪ R2, R1 ∩ R2, or R1-R2 has the same attribute names as the first operand relation R1 (by convention).

Some Properties of the Set Operators

Notice that both union and intersection are commutative operations; that is

$R \cup S = S \cup R$, *and* $R \cap S = S \cap R$

Both union and intersection can be treated as n-nary operations applicable to any number of relations as both are associative operations; that is

$R \cup (S \cup T) = (R \cup S) \cup T$, *and* $(R \cap S) \cap T = R \cap (S \cap T)$

The minus operation is not commutative; that is, in general

$R - S \neq S - R$

5. CARTESIAN (cross product) Operation

This operation is used to combine tuples from two relations in a combinatorial fashion. That means, every tuple in Relation (R) will be related with every other tuple in Relation (S).

- In general, the result of $R(A_1, A_2, \ldots, A_n) \times S(B_1, B_2, \ldots, B_m)$ is a relation Q with degree $n + m$ attributes $Q(A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m)$, in that order.
- Where R has n attributes and S has m attributes.
- The resulting relation Q has one tuple for each combination of tuples—one from R and one from S.
- Hence, if R has n tuples, and S has m tuples, then | R x S | will have $n * m$ tuples.

Example:

Employee

ID	FName	LName
123	Abebe	Lemma
567	Belay	Taye
822	Kefle	Kebede

Dept

DeptID	DeptName	MangID
2	Finance	567
3	Personnel	123

Then the Cartesian product between Employee and Dept relations will be of the form:

Employee X Dept:

ID	FName	LName	DeptID	DeptName	MangID
123	Abebe	Lemma	2	Finance	567
123	Abebe	Lemma	3	Personnel	123
567	Belay	Taye	2	Finance	567
567	Belay	Taye	3	Personnel	123
822	Kefle	Kebede	2	Finance	567
822	Kefle	Kebede	3	Personnel	123

Basically, even though it is very important in query processing, the Cartesian Product is not useful by itself since it relates every tuple in the First Relation with every other tuple in the Second Relation. Thus, to make use of the Cartesian Product, one has to use it with the Selection Operation, which discriminate tuples of a relation by testing whether each will satisfy the selection condition.
In our example, to extract employee information about managers of the departments (Managers of each department), the algebra query and the resulting relation will be.

$\pi_{<ID,\ FName,\ LName,\ DeptName>}(\sigma_{<ID=MangID>}(Employee\ X\ Dept))$

ID	FName	LName	DeptName
123	Abebe	Lemma	Personnel
567	Belay	Taye	Finance

6. JOIN Operation

The sequence of Cartesian product followed by select is used quite commonly to identify and select related tuples from two relations, a special operation, called ***JOIN***. Thus in JOIN operation, the Cartesian Operation and the Selection Operations are used together.

JOIN Operation is denoted by a ⋈ symbol.

This operation is very important for any relational database with more than a single relation, because it allows us to process relationships among relations.

The general form of a join operation on two relations
$R(A_1, A_2, \ldots, A_n)$ and $S(B_1, B_2, \ldots, B_m)$ is:

$R \bowtie_{<join\ condition>} S$ ***is equivalent to*** $\sigma_{<selection\ condition>}(R \times S)$

where <join condition> *and* ***<selection condition>*** *are the same*

Where, R and S can be any relation that results from general relational algebra expressions.
Since JOIN is an operation that needs two relation, it is a Binary operation.

This type of JOIN is called a ***THETA JOIN (¯ - JOIN)***

Where ¯ is the logical operator used in the join condition.

¯ Could be { <, á , >, ñ, ÷, = ***}***

Example:
Thus in the above example we want to extract employee information about managers of the departments, the algebra query using the JOIN operation will be.

$Employee \bowtie_{< ID=MangID>} Dept$

a. EQUIJOIN Operation

The most common use of join involves join conditions with equality comparisons only (=). Such a join, where the only comparison operator used is the equal sign is called an EQUIJOIN. In the result of an EQUIJOIN we always have one or more pairs of attributes (whose names need not be identical) that have identical values in every tuple since we used the equality logical operator.

For example, the above JOIN expression is an EQUIJOIN since the logical operator used is the equal to operator (=).

b. NATURAL JOIN Operation

We have seen that in EQUIJOIN one of each pair of attributes with identical values is extra, a new operation called **natural join** was created to get rid of the second (or extra) attribute that we will have in the result of an EQUIJOIN condition.

The standard definition of natural join requires that the two join attributes, or each pair of corresponding join attributes, have the same name in both relations. If this is not the case, a renaming operation on the attributes is applied first.

$R_1 \leftarrow R \bowtie S$ represents a natural join between R and S. The degree of R_1 is degree of R plus Degree of S less the number of common attributes

c. OUTER JOIN Operation

OUTER JOIN is another version of the JOIN operation where non matching tuples from a relation are also included in the result with NULL values for attributes in the other relation.

There are two major types of OUTER JOIN.

1. ***RIGHT OUTER JOIN***: where non matching tuples from the second (Right) relation are included in the result with NULL value for attributes of the first (Left) relation.
2. ***LEFT OUTER JOIN***: where non matching tuples from the first (Left) relation are included in the result with NULL value for attributes of the second (Right) relation.

Notation for Left Outer Join:

$R \mathbin{⟕}_{\text{<Join Condition>}} S$ → theta left outer Join

$R \mathbin{⟕} S$ → natural left outer join

When two relations are joined by a JOIN operator, there could be some tuples in the first relation not having a matching tuple from the second relation, and the query is interested to display these non matching tuples from the first or second relation. Such query is represented by the OUTER JOIN.

d. SEMIJOIN Operation

SEMI JOIN is another version of the JOIN operation where the resulting Relation will contain those attributes of only one of the Relations that are related with tuples in the other Relation. The following notation depicts the inclusion of only the attributes form the first relation (R) in the result which are actually participating in the relationship.

$$R \triangleright_{\text{<Join Condition>}} S$$

Aggregate functions and Grouping statements

Some queries may involve aggregate function (scalar aggregates like totals in a report, or Vector aggregates like subtotals in reports)

a) $\mathfrak{Z}_{AL}$ **(R)**: Scalar aggregate functions on relation R with AL as a list of (<aggregate function > ,<attribute >) pairs

b) $_{GA}\mathfrak{Z}_{AL}$ **(R):** Vector aggregate functions on relation R with AL as list of (<aggregate function >, <attribute >) pairs with a grouping attribute GA.

Example (a): the number of employees in a an organization (assume you have an employee table)
This is a scalar aggregate

P_R**(Num_Employees)**$\mathfrak{Z}$ **Count EmpId (Employee)** , where P_R = Produce relation R

Example (b): the number of employees in each department of an organization (assume you have an employee table)
This is a vector aggregate

P_R **(DeptId, Num_Employees)** $_{DeptId}\mathfrak{Z}$ **Count EmpId (Employee)** , where P_R = Produce relation R

Relational Calculus

A relational calculus expression creates a new relation, which is specified in terms of variables that range over rows of the stored database relations (in tuple calculus) or over columns of the stored relations (in domain calculus).

In a calculus expression, there is no order of operations to specify how to retrieve the query result. A calculus expression specifies only ***what*** information the result should contain rather than ***how*** to retrieve it.

In Relational calculus, there is no description of how to evaluate a query; this is the main distinguishing feature between relational algebra and relational calculus.

Relational calculus is considered to be a nonprocedural language. This differs from relational algebra, where we must write a sequence of operations to specify a retrieval request; hence relational algebra can be considered as a procedural way of stating a query.

When applied to relational database, the calculus is not that of derivative and differential but in a form of first-order logic or predicate calculus, a predicate is a truth-valued function with arguments.

When we substitute values for the arguments in the predicate, the function yields an expression, called a *proposition*, which can be either true or false.

If a predicate contains a variable, as in *'x is a member of staff'*, there must be a *range for x*. When we substitute some values of this range for x, the proposition may be true; for other values, it may be false.

If COND is a predicate, then the set of all tuples evaluated to be true for the predicate COND will be expressed as follows:

{t | COND(t)}

Where ***t*** is a tuple variable and ***COND (t)*** is a conditional expression involving t. The result of such a query is the set of all tuples t that satisfy ***COND (t)***.

If we have set of predicates to evaluate for a single query, the predicates can be connected using ė**(AND)**, Ę**(OR**), and ~**(NOT)**

A relational calculus expression creates a new relation, which is specified in terms of variables that range over rows of the stored database relations (in tuple calculus) or over columns of the stored relations (in domain calculus).

Tuple-oriented Relational Calculus

- The tuple relational calculus is based on specifying a number of tuple variables. Each tuple variable usually ranges over a particular database relation, meaning that the variable may take as its value any individual tuple from that relation.
- Tuple relational calculus is interested in finding tuples for which a predicate is true for a relation. Based on use of tuple variables.
- Tuple variable is a variable that *'ranges over'* a named relation: that is, a variable whose only permitted values are tuples of the relation.
- If E is a tuple that ranges over a relation employee, then it is represented as ***EMPLOYEE(E)*** i.e. ***Range of E is EMPLOYEE***

- Then to extract all tuples that satisfy a certain condition, we will represent it as all tuples E such that COND(E) is evaluated to be true.

{E å COND(E)}

The predicates can be connected using the Boolean operators:

ė (AND), Ę (OR), ¼ (NOT)

COND(t) is a formula, and is called a Well-Formed-Formula (WFF) if:

- Where the COND is composed of n-nary predicates (formula composed of n single predicates) and the predicates are connected by any of the Boolean operators.
- And each predicate is of the form A θ B and θ is one of the logical operators **{ <, á , >, ñ, ÷, = }**which could be evaluated to either true or false. And A and B are either constant or variables.
- Formulae should be unambiguous and should make sense.

Example (Tuple Relational Calculus)

- Extract all employees whose skill level is greater than or equal to 8

{E | Employee(E) è E.SkillLevel >= 8}

EmpID	*FName*	*LName*	*SkillID*	*Skill*	*SkillType*	*School*	*SchoolAdd*	*SkillLevel*
28	Chane	Kebede	2	SQL	Database	AAU	Sidist_Kilo	10
25	Abera	Taye	6	VB6	Programming	Helico	Piazza	8
65	Almaz	Belay	2	SQL	Database	Helico	Piazza	9
51	Selam	Belay	4	Prolog	Programming	Jimma	Jimma City	8

- To find only the EmpId, FName, LName, Skill and the School where the skill is attended where of employees with skill level greater than or equal to 8, the tuple based relational calculus expression will be:

{E.EmpId, E.FName, E.LName, E.Skill, E.School | Employee(E) è E.SkillLevel >= 8}

EmpID	*FName*	*LName*	*Skill*	*School*
28	Chane	Kebede	SQL	AAU
25	Abera	Taye	VB6	Helico
65	Almaz	Belay	SQL	Helico
51	Selam	Belay	Prolog	Jimma

- E.FName means the value of the First Name (FName) attribute for the tuple E.

Quantifiers in Relational Calculus

- To tell how many instances the predicate applies to, we can use the two quantifiers in the predicate logic.
- One relational calculus expressed using Existential Quantifier can also be expressed using Universal Quantifier.

1. Existential quantifier A ('there exists')

Existential quantifier used in formulae that must be true for at least one instance, such as:
An employee with skill level greater than or equal to 8 will be:

{E | Employee(E) ė (AE)(E.SkillLevel >= 8)}

This means, there exist at least one tuple of the relation employee where the value for the SkillLevel is greater than or equal to 8

2. Universal quantifier ? ('for all')

Universal quantifier is used in statements about every instance, such as:
An employee with skill level greater than or equal to 8 will be:

{E | Employee(E) ė (?E)(E.SkillLevel >= 8)}

This means, for all tuples of relation employee where value for the SkillLevel attribute is greater than or equal to 8.

Example:

Let's say that we have the following Schema (set of Relations)

Employee(EID, FName, LName, EDID)
Project(PID, PName, PDID)
Dept(DID, DName, DMangID)
WorksOn(WEID, WPID)

To find employees who work on projects controlled by department 5 the query will be:

{E | Employee(E) ė (?P)(Project(P) ė (Aw)(WorksOn(w) ė PDID =5 ė EID=WEID))}

Domain Relational Calculus

In tuple relational Calculus, we use variables that range over tuples of a relation, in the case of domain relational calculus we use variables that range over domain elements (field variables).

- An expression in the domain relational calculus has the following general form

$\{(x_1,x_2,x_3,\ldots.x_n)|\ P(x_1,x_2,x_3,\ldots.x_n,x_m)\}$

Where $(x_1,x_2,x_3,\ldots.x_n)$ represents the domain variables and $P(x_1,x_2,x_3,\ldots.x_n,x_m)$ represents the formula

Formulas are of the form $R(x_1,x_2,x_3,\ldots.x_n)$, $x_1 \theta x_2$ ***or***

$x_i \theta C$ where $\theta \in \{<,>,<=,>=,=,\neq\}$ *and R is a relation of degree n and each* x_i *is domain variable*

If f_1 *and* f_2 *are formulas then so are*

$f_1 \wedge f_2$, $f_1 \vee f_2$, $\neg f_1$, $(\forall x) f_1$, $(\exists x) f_1$

- The Answer for such a query includes all tuples with attributes $(x_1,x_2,x_3,\ldots.x_n)$ that make the formula $P(x_1,x_2,x_3,\ldots.x_n,x_m)$ be true.
- Formula is recursively defined, starting with simple atomic formulas (getting tuples from relations or making comparisons of values), and building bigger and better formulas using the logical connectives. i.e the Predicate P can be set of formula combined by Boolean operators

Example: Consider the schema of relations on page 102.

Query1: list Employees

{Fname, Lname | (Employee (EID,FName, LName)}

Query2: Find the list of Employees who work in the department of IS

Domain relational Calculus expression for the query

{EID,Fname,Lname | (ADName,EDID,DID)(Employee(EID,FName, LName)∧Department(DID,DName,DMangID)**∧DID=EDID∧DName='IS')}**

, Where ADName, EDID, DID → ADName, AEDID, ADID

Query3:List the names of employees that do not manage any department

{Fname,Lname | (∃EID)(Employee(EID,Fname,Lname)

∧(~(∃DMangId)(Dept(DID,Dname,DMangId) ∧**(EID=DMangId))))**}