5.

INTRODUCTION TO RELATIONAL ALGEBRA & CALCULUS

This chapter presents the student with the basic concepts underlying the various operations on a relational database. These concepts are relational algebra and relational calculus.

Relational algebra and algebra are two formal query languages that work with the relational model. A query is a technical name for set of instructions sent to the database in order to manipulate that and get results. These two formal languages used specific operators to interact with a relational data structure.

Relational algebra is a functional procedural language which is used as an intermediate language at the lower level of the DBMS for manipulating objects in a relational data structure to produce results. Notice a relationship between relational algebra and the Structured Query Language (SQL). SQL is a declarative language used as the predominant application-level query language on RDBMSs and all SQL queries would be parsed into relational algebra expressions by the parser at the lower level of the DBMS; the result is then optimized by the query optimizer. The code generator then generates an executable code from the result which can be executed.

Relational calculus unlike relational algebra is a non-procedural declarative query language that states or describes the expected or desired result to be extracted from the relational data structure rather than how the result is to be achieved. The degree of relational completeness in database system may be defined in terms of the extent to which the system in question supports relational algebra and calculus.

The knowledge of these concepts will help students understand how relational database systems operate at the lower level and how improved query execution and optimization in a relational DBMS could be attained.

5.1 Relational Algebra

The term 'relational algebra' sounds a bit more confusing to a newbie who is just venturing into the world of relational database systems. It is no strange term anyway just that it sounds like a misnomer, so what does it mean in essence?

Relational algebra is a procedural query language that consists of a set of operations that take one or two relations as input and result into a new relation as an output. To perform basic query operations, operators (unary and binary) are employed. A unary operator operates on a single expression whereas a binary algebra operator is applied to two relational algebra expressions.

Relational algebra was introduced in the 1970s basically to provide a set of operations on relations. These operations are: ***Select, Project, Rename, Product, Union, Set-Difference***; ***Join, Intersect, and Divide***. The underlined operations above are the basic relational algebra operations whereas the last three (Join, Intersect, and Divide) are derived from the basic operations. The result of each operation is a relation. The closure property of a relation makes it possible to apply an operation to the result of another operation.

Relational algebra operations may be divided into two groups.

- Relational-based operations
- Set-oriented operations

The relational model operations are specifically for relational databases and they are: ***select***, ***project***, ***rename***, ***join***, and ***divide*** *operations*.

The set-oriented operations are derived from the mathematical concept of sets. These operations are: ***union***, ***intersection***, ***set-difference***, and ***Cartesian product***. In addition, recursive operations on relations are also allowed in which case intermediate results may be produced that also qualify as relations.

Some of the queries, which are mathematical in nature, cannot be expressed using the basic operations of relational algebra. For such queries, additional operations like aggregate functions and grouping are used such as finding sum, average, etc.

5.1.1 Selection and Projection Operations

These two operations are discussed under this section due to their relationship and applicability to a single relation that is, they are unary operations, in a sense orthogonal. The Selection operation applies to tuples or rows of a relation while the Projection operation is applied to columns (fields). Selection is for horizontal decompositions and projection for vertical decompositions.

Another distinction is that the Select operation provides condition(s) that must be satisfied whereas the Project operation only extracts specific columns.

To discuss these two relational operations, we shall use a hypothetical table called FOOD presented in Table 5.1

Table 5.1: FOOD

Items	Description	Seller	Price(NGN)
Beans	Brown beans	Oghene	350
Yam	Mumuya	Ese	800
Rice	Mama Africa	Nkechi	10000
Tomatoes	Derica Brand	Adamma	100
Vegetable	Ugu	Iyalode	100
Bread	Agege	Hajia	200
Rice	Ofada	Nkechi	15,700
Cocoyam	Indian Coco	Ese	400

I. Select (σ) operation

The select operation is a unary operation which selects tuples (rows) that satisfy the given predicate from a relation (operand). The resultant relation has same schema as the operand but with a subset of the tuples of the operand. It employs the relational operators such as: – =, ≠, ≥, <, >, ≤ and connectors such AND, OR, and NOT.

Definition:

*The selection on a relation **r** given a predicate **P** defined on **r**, denoted by:*

$$\sigma_P(r)$$

$$\sigma_P(r) = \{ t \mid t \in r \text{ s.t. } t \text{ satisfies } P, \text{ i.e., } P(t)\}$$

is a relation containing all and only the tuples(t) of r that verify predicate P

The degree of the resulting relation is the same as the degree of the input relation. If no tuple of **r** verifies **P**, the result is the empty relation.

A ***predicate*** is a boolean expression whose operators are the logical connectives (and, or, not) and arithmetic comparisons (>, <, – =, ≠, ≥, ≤), and whose operands are either domain names or domain constants.

A predicate **F** on a relation **R** may be of two forms:

1. Simple predicate may be of two forms:
   - **A** op **j** [e.g. D=d]
   - **A** op **A'**; [e.g. P=C]

Where **A** and **A'** are attributes of **R**; op= {>, <, – =, ≠, ≥, ≤); **j** is a constant value compatible with the domain of **A.**

2. Boolean combination of simple predicates using the logical operators NOT (¬), AND(^), OR (∇), e.g. [D=d AND P=C]

Note that the *Select* operation in relational algebra has nothing to do with the structured query language keyword associated with the examples used to illustrate the relational algebra operations. *Selection* in relational algebra simply returns those tuples in a

relation that fulfill a condition. Let us consider and evaluate the results of the following *Selection* expressions in Table 5.2 which uses the FOOD relation in Table 5.1 above as its reference table.

Table 5.2: Selection expressions

No.	Relational algebra expression	SQL expression
1.	$\sigma_{\text{Items = "Beans"}}(\text{FOOD})$	Select * from FOOD where **Items** = 'Beans';
2.	$\sigma_{\text{Items = "Rice" and price = "10000"}}(\text{FOOD})$	Select * from FOOD where **Items** = 'Rice' and 'Price' = 10000;
3.	$\sigma_{\text{Seller = "Ese" and Items = "Yam" or Price > "100"}}(\text{FOOD})$	Select * from FOOD where (Seller = 'Ese' and 'Items' = Yam) or price > 100;

The results of three select expressions in Table 5.2 are discussed below.
$\sigma_{\text{Items = "Beans"}}(\text{FOOD})$ selects rows/records from the table, FOOD where the column ***Items*** is 'Beans' and the result of this operation is in Table 5.3

Table 5.3: $\sigma_{\text{Items = "Beans"}}(\text{FOOD})$

Items	Description	Seller	Price (N)
Beans	Brown beans	Oghene	350

$\sigma_{\text{Items = "Rice" and price = "10000"}}(\text{FOOD})$ generates the result in Table 5.4. The operation selects all tuples from FOOD where **Items** is 'Rice' and 'Price' is 10000.

Table 5.4: $\sigma_{\text{Items = "Rice" and price = "10000"}}(\text{FOOD})$

Items	Description	Seller	Price(NGN)
Rice	Mama Africa	Nkechi	10000

The expression $\sigma_{\textit{Seller = "Ese" and Items = "Yam" or Price > "100"}}(\textit{FOOD})$ will select tuples from FOOD where Seller is 'Ese' and 'Items' is Yam or those other records/tuples where Price is higher than 100 NGN (Nigerian Naira).
The result of this expression is a tuple that would combine the results of the two conditions because of the disjunctive connector 'OR' used in the expression. In other words, we are supposed to have a set of rows comprising:

- Tuples where the seller is 'Ese' and 'Items' is 'Yam'
- Tuples where Price is greater than 100

Notice that there is no tuple in the relation (FOOD) where the seller is 'Ese' and 'Items' is 'Yam', so the first condition is automatically dismissed. The result is tuples presented in Table 5.5

Table 5.5: $\sigma_{\text{Seller = "Ese" and Items = "Yam" or Price > "100"}}(\text{FOOD})$

Items	Description	Seller	Price(NGN)
Beans	Brown beans	Oghene	350
Yam	Mumuya	Ese	800
Rice	Mama Africa	Nkechi	10000
Bread	Agege	Hajia	200
Rice	Ofada	Nkechi	15,700
Cocoyam	Indian Coco	Ese	400

II. Project operation (∏)

This operation produces a list of all values for the selected attributes (fields). That is, it produces a vertical subset of a relation or simply put, it projects column(s) that satisfy a given predicate. The standard notation for this operation is:

Let R (A) **be a relation;** A**= {A_1... A_n} be a set of attributes and** P □ A**;**

The Project operation of **R** on **P**, given by:

$\Pi_P(R)$ = {**t[P] | t** □ **R**}; is a relation whose tuple(**t**) has its attributes in **A**

A projection operation does not allow duplicate values. This is because a relation is considered strictly as a set and any duplicate values are discarded.
Let us consider the projection expressions in Table 5.6. We shall use them to discuss the project operation further.

Table 5.6: Projection expressions

No.	Relational algebra expression	SQL expression
1.	$\prod_{\text{Items}}(\text{FOOD})$	Select Items from FOOD;
2.	$\prod_{\text{Items, Price}}(\text{FOOD})$	Select Items, Price from FOOD;
3.	$\prod_{\text{Items, Price, Seller}}(\text{FOOD})$	Select Items, Price, Seller from FOOD ;

Discussion:
1. The first expression selects and projects the Items column from the FOOD relation. The result is the entire data in the Items column in the named relation, as shown below.

Price(NGN)
350
800
10000
100
100
200
15,700
400

2. The second expression selects and projects *Items* and *Price* and produce a result which is equivalent to the SQL statement, **SELECT** *Items, Price* FROM ***FOOD;***
The result is Table 5.7 below.

3. The third expression selects and projects *Items, Price* and Seller and generates a result which is equivalent to the SQL statement, **SELECT** *Items, Price, Seller* FROM ***FOOD;*** The result is Table 5.8 below.

Table 5.7: $\Pi_{Items,\ Price}(FOOD)$

Items	Price(NGN)
Beans	350
Yam	800
Rice	10000
Tomatoes	100
Vegetable	100
Bread	200
Rice	15,700
Cocoyam	400

Table 5.8: $\Pi_{Items,\ Price,seller}(FOOD)$

Items	Price(NGN)	
Beans	350	Oghene
Yam	800	Ese
Rice	10000	Nkechi
Tomatoes	100	Adamma
Vegetable	100	Iyalode
Bread	200	Hajia
Rice	15,700	Nkechi
Cocoyam	400	Ese

Cardinality of Projection Operations

The result of a projection may contain as many tuples as the base relation. However, it may contain fewer, if several tuples collapse, i.e., they are identical in all their values.

Theorem: $\pi_P(r)$ contains as many tuples as r if and only if P is a super key for r. This property holds even if P is peradventure a super key, i.e., it is not defined as a super key in the schema, but it is a super key for the current database.

5.1.2 Join operation (⋈)

The JOIN operation is the most operation in relational algebra and a popular operation in relational database models because it allows the use of independent relations which are joined using common attributes or fields/columns. Given a domain from each relation, a *join* operation would consider all possible pairs of tuples from the two relations. Where the values of the tuples for the chosen domains are equal, it adds a tuple to the result containing all the attributes of both tuples while discarding the duplicate.

Definition:

Let R_1 **(X),** R_2 **(Y)** be two relations where X and Y are attribute sets, such that the domains of the attributes are compatible; let θ be a relational comparison operator**.**

The ***join*** of R_1 **and** R_2 with respect to the predicate **XθY** is represented as

$R_1 \bowtie_{X\theta Y} R_2$ and defined by $\sigma_{X\theta Y}(R1xR2)$; that is, a Cartesian product followed by a Selection; the predicate **XθY** is known as a ***join predicate***.

When the θ operator is the equality operator, the join is an **Equijoin**. The resulting relation after the join operation has its degree equal to the sum of the degrees of the input relations. There are variations of the Join operators but we shall discuss the following variations:

- **Theta & Equijoin**

- **Natural join** (which takes attribute names into account)
- **Conditional joins**

A. Theta-Join

This performs a join based on an operator other than equality. In most cases, a Cartesian product is meaningful only if followed by a selection. Theta join may be simple regarded as any Cartesian that is filtered or subjected to a condition that compares attribute values from two relations.

Assuming S and D above do not have a common attribute such as *DeptID*, and X is an attribute of **S** while Y is an attribute of **D**, if the predicate XθY satisfy the requirements for selection then theta-join is defined by:

$$S \bowtie_{X\theta Y} D = \sigma_{X\theta Y}(S \bowtie D);$$

If XθY is a conjunction of equalities, then we have an equijoin.

B. Equijoin

This is another form of Join that links tables based on an equality condition. The condition to join the table must be clearly stated. The operation does not eliminate duplicates. The equality comparison operator (=) must be used in the condition hence the name. Note that an Equijoin differs from a natural join in that the equality condition in an Equijoin has nothing to do with the shared key or natural relationships between the two relations. For instance, using the S and D instances above, if we may be required to find $S \bowtie_{S.DeptID=D.DeptName} D$; that is, the condition where the *DeptID* attribute in **S** (*Student*) is equal to the value of the *DeptName* attribute in **D** (*Department*). This operation would return an empty tuple because there are no cases in the two relations where S.*DeptID*=D.*DeptName*.

Example:

Consider the two relations *CourseAdviser* and *Lecturer* in Table 5.9 below. Let C be an instance of the CourseAdviser relation and L an instance of the Lecturer relation then the Equijoin of C and L denoted by: $C \bowtie_{C.AdviserName=L.Name} L$ will produce the result in Table 5.10.

Table 5.9: **CourseAdviser relation** **Lecturer relation**

AdviserID	AdviserName	DeptID	DeptName	LecturerID	Name
201	Prof. Nnah	PHY	PHYSICS	050	Prof. Nnah
202	Dr. Udoka	IMT	INFOTECH	051	Dr. Udoka
203	Prof. Eze	ELS	ELECTRONICS	052	Prof. Ken

Table 5.10: Equijoin ($C \bowtie_{C.AdviserName=L.Name} L$)

AdviserID	AdviserName	DeptID	DeptName
201	Prof. Nnah	PHY	PHYSICS
202	Dr. Udoka	IMT	INFOTECH

LecturerID	Name
050	Prof. Nnah
051	Dr. Udoka

C. Natural join

The natural join may be regarded as a special case of equijoin where equalities are implicitly specified on the fields that have the same name in the two relations being joined. The condition **c** is now left out, so that the common columns in the two relations signify a natural join.

To perform a Natural join operation three stages are involved. They are:

1. **Product (Cartesian)** operation to obtain a cross-product then followed by
2. **Selection** and then
3. **Projection**

Definition:

Let **R_1 (X_1), R_2 (X_2)** represent two relations, where X_1 and X_2 are attribute sets for R_1 and R_2 respectively;

The natural join of R_1 and R_2 denoted by

$R_1 \bowtie R_2$

is a relation on X_1X_2 (union of the two sets) such that

{t on X_1X_2 | t [X_1] ∈ R_1 and t [X_2] ∈ R_2 } or {t on X_1X_2| exist t_1 ∈ R_1 and t_2 ∈ R_2 with t [X_1] = t_1 and t [X_2] = t_2} where t is a tuple/row

The tuples in the resulting relation are obtained by combining tuples in the operands with equal values on the common attribute(s). The common attributes often form a key of one of the operands (the join is on this key – the shared attribute which may be a primary key in one of the relations say R_1 but a foreign key on the second relation say R_2). Simply put, it is a join based on the equality of the values for attributes that are in common among the input relations with the replicated attributes removed.

Using the *Student* and *Department* relations below, let ***S*** be an instance of *Student* relation and ***D*** be an instance of *Department* relation; the common or shared attribute is *DeptID* therefore the natural join: **join(S, D) or S⋈D** would be natural join on ***DeptID***. The result is a relation of all Student/Department attribute pairs that share same *DeptID*, as presented in Table 5.16 below. Note that the *DeptID*, 'EEE' which was not shared by both tables was excluded in the result of the join. To illustrate this concept of Natural Join further let us use the two hypothetical relations, *Student* and *Department* presented in Tables 4.23 and 4.24 respectively.

Table 5.11: Student relation

RegNum	LastName	FirstName	Level	DeptID
2001	Njoku	Peter	400	IMT
2003	Obi	Jerry	300	TMT
2004	Hassan	Baba	400	CSC

Table 5.12: Department relation

DeptID	DeptName
IMT	INFORMATION TECHNOLOGY
PMT	PROJECT MANAGEMENT
TMT	TRANSPORT TECHNOLOGY
CSC	COMPUTER SCIENCE
MTH	MATHEMATICS

We shall show how this is done using the steps of doing a Join operation, that is, ***Product, Selection,*** and ***Projection*** which is equivalent to the operation:

$$S \bowtie_{X=Y} D = \prod_{A-Y} \sigma_{X=Y} (SxD)$$

Where X=Y=DeptID, and A is the set of attributes in S and D.

Question: What happens if the two relations have no attributes in common (no shared key)?
Answer: Where there are no attributes in common in the two relations, the resultant relation is simply a cross-product of the two relations. That is, the result contains tuples obtained by combining the tuples of the operands in all possible ways. In this case there would be no need to do a selection and projection.

1. ***Cartesian or cross-Product stage(x)***

The cross-product of the two relations **SxD** is produced by pairing every row of **S** with every row of **D**. Table 5.13 may serve as a guideline when performing a cross-product of two tables **S** and **D.**

Table 5.13: Cartesian product computation

	S	D	SxD
Rows	**J**	**K**	**J*K**
Columns	**P**	**Q**	**P+Q**

For example, the cross-product of a **3x5**-table **S** and a **5x2**- table **D** will produce a 15x7-table **S**x**D**, that is, ***SxD*** will have 15 rows and 7 columns. Figure 5.1 shows how this product will be computed using an array-based representation.

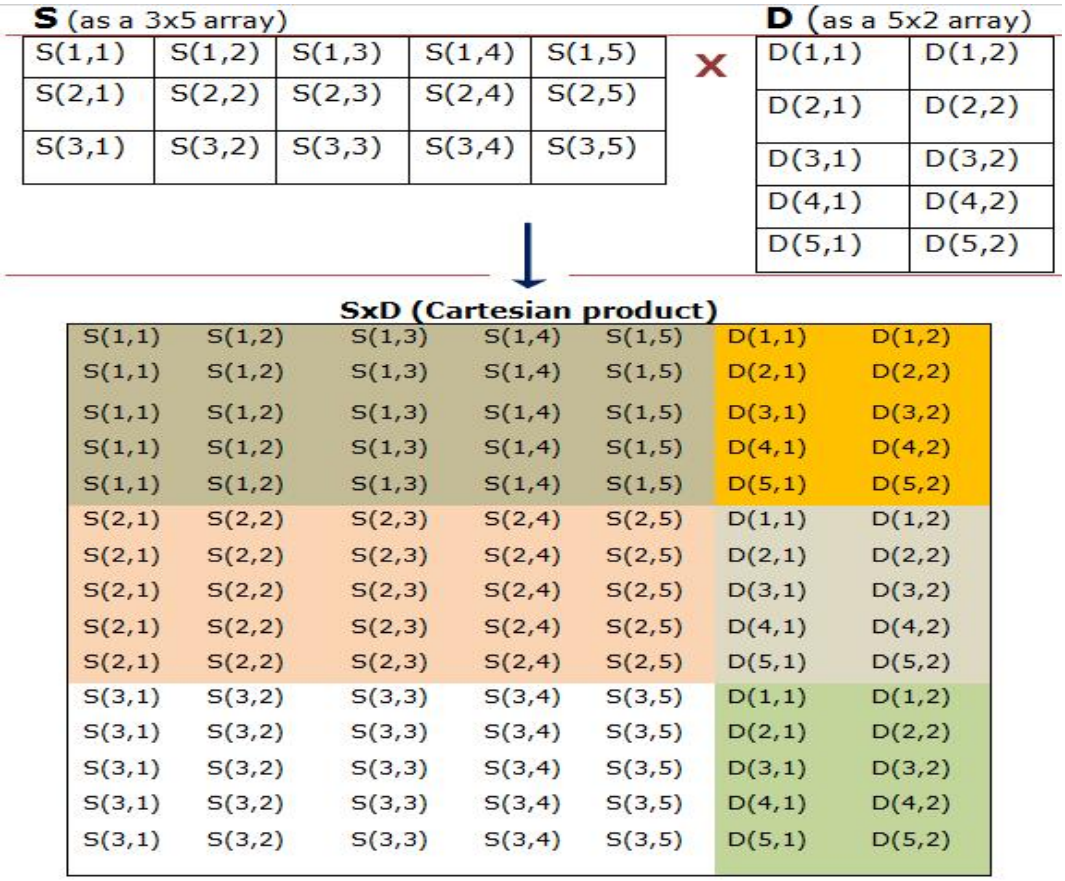

Figure 5.1: Computing of the Cartesian product (SxD) of two arrays

Following the illustration above, the result of the Cartesian product (SxD) would be as presented in Table 5.14 below. In order to offer a clearer explanation of this first stage we have used various background colours to show how the various combinations of the rows from both relations are made.

Table 5.14: SxD

RegNum	LastName	FirstName	Level	DeptID	DeptID	DeptName
2001	Njoku	Peter	400	IMT	IMT	INFORMATION TECHNOLOGY
2001	Njoku	Peter	400	IMT	PMT	PROJECT MANAGEMENT
2001	Njoku	Peter	400	IMT	TMT	TRANSPORT TECHNOLOGY
2001	Njoku	Peter	400	IMT	CSC	COMPUTER SCIENCE
2001	Njoku	Peter	400	IMT	MTH	MATHEMATICS
2003	Obi	Jerry	300	TMT	IMT	INFORMATION TECHNOLOGY
2003	Obi	Jerry	300	TMT	PMT	PROJECT MANAGEMENT
2003	Obi	Jerry	300	TMT	TMT	TRANSPORT TECHNOLOGY
2003	Obi	Jerry	300	TMT	CSC	COMPUTER SCIENCE
2003	Obi	Jerry	300	TMT	MTH	MATHEMATICS
2004	Hassan	Baba	400	CSC	IMT	INFORMATION TECHNOLOGY
2004	Hassan	Baba	400	CSC	PMT	PROJECT MANAGEMENT
2004	Hassan	Baba	400	CSC	TMT	TRANSPORT TECHNOLOGY
2004	Hassan	Baba	400	CSC	CSC	COMPUTER SCIENCE
2004	Hassan	Baba	400	CSC	MTH	MATHEMATICS

2. *Selection stage* (σ)

This stage is peculiar to a natural Join in which case having computed the cross-product of S and D; the next stage would be to apply the ***Select* (σ)** operation to the Cartesian product above using the shared or common attribute (*DeptID*) in the Selection predicate. That is we need to select those rows (tuples) where DeptID has equal values. We have indicated the equal values using a red font-colour in Table 5.14 above. The result of the selection is presented in Table 5.15; note that there is still a duplication of the DeptID column after the Select operation. This duplication will be removed in the next stage.

Table 5.15: Result of the Select operation

RegNum	LastName	FirstName	Level	DeptID	DeptID	DeptName
2001	Njoku	Peter	400	IMT	IMT	INFORMATION TECHNOLOGY
2003	Obi	Jerry	300	TMT	TMT	TRANSPORT TECHNOLOGY
2004	Hassan	Baba	400	CSC	CSC	COMPUTER SCIENCE

3. *Projection stage* (∏)

A Project Join is carried out on the result of Select Join (see Table 5.15) to produce a unique copy of each field (attribute). This removes duplicate columns. Hence we have Table 5.16 as the final result.

Table 5.16: Result of the Project operation (S⋈D)

RegNum	LastName	FirstName	Level	DeptID	DeptName
2001	Njoku	Peter	400	IMT	INFORMATION TECHNOLOGY
2003	Obi	Jerry	300	TMT	TRANSPORT TECHNOLOGY
2004	Hassan	Baba	400	CSC	COMPUTER SCIENCE

D. Conditional joins

A conditional join between two relations **S** and **D** is defined as a cross-product followed by a selection. It is defined by:

$$S \bowtie_c D = \sigma_c(S \times D)$$

Where c is the condition

Notice the difference between a conditional join and a natural join. A natural join may involve a cross-product, selection and projection when there is a common attribute between the two relations.

Example:

Given the relational instances **S** and **D** above; the conditional join

$S \bowtie_{S.DeptID=D.DeptID} D$ would produce the result in Table 5.17

Table 5.17: $S \bowtie_{S.DeptID=D.DeptID} D$

RegNum	LastName	FirstName	Level	DeptID	DeptID	DeptName
2001	Njoku	Peter	400	IMT	IMT	INFORMATION TECHNOLOGY
2003	Obi	Jerry	300	TMT	TMT	TRANSPORT TECHNOLOGY
2004	Hassan	Baba	400	CSC	CSC	COMPUTER SCIENCE

E. Outer Join

An Outer Join is a variant of Join and obtained when the unmatched pairs are retained with their values in the new relation padded with null. It is designed to keep all pieces of information from the operands. That is, An outer join operation "pads with nulls" the tuples in one relation that have no counterpart in the other relation.

There are three versions of the outer join:

- **Left Outer Join(S]∞D)**: this operation retains every tuple in the first or left relation and only tuples of left relation(operand) are padded;
- **Right Outer join(S∞[D)**: Retains every tuple in the second(right) relation; only tuples of right operand are padded;
- **Full join (S]∞[D):** retains and pads the tuples of both operands.

S and D represent same instances discussed above.

Example: Consider the two relations *Lecturer* and *Department* in Table 5.18 with instances L and D respectively.

Table 5.18: Lecturer relation Department relation

LecturerName	DeptName
Prof. Nnah	ELS
Dr. Olu	IMT
Prof. Ken	IMT

DeptID	DeptName	HOD
20	IMT	Prof. Uka
21	CSC	Dr. Udoka
23	EET	Dr. Josy

1. Left Outer Join: A Left Join of **L** and **D** denoted by $L \bowtie_{LEFT} D$ or $L]\infty D$ will produce the following result:

LectureName	DeptName	DeptID	HOD
Prof. Nnah	ELS	Null	Null
Dr. Olu	IMT	20	Prof. Uka
Prof. Ken	IMT	20	Prof. Uka

2. Right Outer Join: A Right Outer Join of L and D denoted by $L \bowtie_{RIGHT} D$ will produce the result below:

LectureName	DeptName	DeptID	HOD
Dr. Olu	IMT	20	Prof. Uka
Prof. Ken	IMT	20	Prof. Uka
Null	CSC	21	Dr. Udoka
Null	EET	23	Dr. Josy

3. Full Join: A Full Join of L and D denoted by $L \bowtie_{FULL} D$ will retain all the tuples in both relations. The result of the full join is presented in Table 5.19 below.

Table 5.19: Full Outer Join (**L] ∞[D**)

LectureName	DeptName	DeptID	HOD
Prof. Nnah	ELS	Null	Null
Dr. Olu	IMT	20	Prof. Uka
Prof. Ken	IMT	20	Prof. Uka
Null	CSC	21	Dr. Udoka
Null	EET	23	Dr. Josy

5.1.3. Cartesian Product operation(X)

The *Cartesian product* of two relations combines each row in one relation with each row in the other relation. The Cartesian product of relations L and D(LxD), with degree ***m*** and ***n*** respectively, is a relation whose degree is ***m+n***. In the new relation (LxD) the names of the first ***m*** attributes are the names of the attributes of L and the names of the last ***n*** attributes are the names of the attributes of D.

Example:

Let us consider the Employee and Department relations in Table 5.20.

Table 5.20: Employee relation Department relation

ID	LecturerName	Dept
001	Prof. Madu	200
002	Prof. Joel	202
003	Dr Steph	200

DeptNo	DeptName
200	IMT
201	PMT
202	EEE

Let L and D represent instances of the two relations *Lecturer* and *Department* respectively; The Cartesian product of the two relations is presented in Table 5.21 below.

Table 5.21: Cartesian product of L and D (**LxD**)

SQL	Result					Relational algebra
Select * from L, D	ID	LecturerName	Dept	DeptNo	DeptName	L X D
	001	Prof. Madu	200	200	IMT	
	001	Prof. Madu	200	201	PMT	
	001	Prof. Madu	200	202	EEE	
	002	Prof. Joel	202	200	IMT	
	002	Prof. Joel	202	201	PMT	
	002	Prof. Joel	202	202	EEE	
	003	Dr Steph	200	200	IMT	
	003	Dr Steph	200	201	PMT	
	003	Dr Steph	200	202	EEE	

5.1.4. Rename operation (ϱ)

Let us consider two relations S and D defined by *S(RegNum:integer, Name:string, DeptNo:integer)* and *D(DeptNo:character, Name:string, Head:string)* in Table 5.22. Assuming we want to perform relational algebra operations such as Join, Selection, Cartesian product, Intersection, etc. on these two relations which have two attribute names (DeptNo, Name) in common, there may be a conflict in the representation of the results depending on the nature of the operation performed. What if the common attributes are not equal or related in any way, this will also

pose a challenge when interpreting the result of such operation. To resolve such occurrences, renaming the attribute names and/or the relation itself is important.

Table 5.22**: S** **D**

RegNum	Name	DeptNo
2010456	Obi	12
2010476	Kanayo	16
2010451	Tobi	17

DeptNo	Name	Head
A	Services	Duru
B	Sales	Offor
C	Marketing	Tony

The rename operation is a unary operation that results in the change of attribute names of a relation (**R**) or the relation itself without changing any values associated with the changed attribute name(s). The operation succeeds if the new schema of the relation has all attributes with distinct names. However, it does not follow that all the attributes of a relation must be renamed at any given time; the only driving force should be to ensure that relational attribute names are quite clear and distinct from each other to prevent any obscurity during set operations. The number of tuples in the affected relation remains same as well as the degree of the relation. When two or more attributes are involved, ordering the attributes is recommended.
The semantics of a Rename operation is given thus:

$$\rho_{Ai \rightarrow Bi}(R)$$

$$\rho_{Q}(R)$$

Where Ai is the old attribute name and Bi is the new attribute name; Q is the new relation that results when the relation itself is renamed from R to Q

A relation may be renamed only when necessary for instance when it is desirable to find a cross-product of the relation on itself.

Example:

The following Rename operations are valid and may be used to resolve the problem in Table 5.22 above.

$\rho_{\text{RegNum, Name} \rightarrow \text{StudentName, DeptNo}}(S)$

$\rho_{\text{DeptNo} \rightarrow \text{DeptID, Name} \rightarrow \text{DeptName, Head}}(D)$

Set operations

Relations are sets, so we can subject them to basic set operations such as cross-product, union, difference, and intersection. Since we want the results of such operations to be relations (that is, homogeneous sets of tuples), It would be sensible to apply these set operations to pairs of relations defined over the same attributes.

5.1.5 Union (∪)

The union of relations S and D, or **S∪D** for short, is defined as:

S∪D = {t | t ∈ S or t ∈ D}; where S and D are relations or temporary relations

It is the set of tuples that are in S, in D or in both excluding duplicate tuples. For a union operation to be valid, the following conditions must hold:

- **S** and **D** must have the same number of attributes(that is, same degree)
- Attribute domains must be compatible
- Duplicate tuples are automatically eliminated.

The degree of the resulting relation is the same as the degree of the input relations.

Example:

Let us consider two relations, *Lecturer* and *Employee* defined in Figure 5.2

Let **L** and **E** represent the instances of *Lecturer* and *Employee* respectively as presented in Table 5.23 below:

Lecturer(Number:integer, Name:string, Dept:string, Age;integer)

Employee(Number:integer, Name:string, Dept:string, Age;integer)

Figure 5.2: Lecturer and Employee domain characteristics

Table 5.23: Lecturer (L) Employee (E)

Number	Name	Dept	Age
56	Obi	EEE	50
76	Kanayo	CSC	35
80	Joseph	IMT	38

Number	Name	Dept	Age
80	Joseph	IMT	38
56	Obi	EEE	50
21	Okonkwo	CSC	48

The union of the two relations (**L∪E)** in Table 5.23 will produce the result below:

L∪E

Number	Name	Dept	Age
56	Obi	EEE	50
76	Kanayo	CSC	35
80	Joseph	IMT	38
21	Okonkwo	CSC	48

5.1.6 Set-difference (-)

The difference of relations S and D, or **S-D** for short, is a new relation that is a set of tuples that are in S and not in D. The difference, like the union operation, can only succeed if the input relations have the same degrees and compatible attributes. The degree of the new relation is the same as the degree of the input relations.

Example:
Let us apply set-difference to the instance of Lecturer and that of Employee in Table 5.23 above; that is **L-E** will produce the table below:

L-E

Number	Name	Dept	Age
76	Kanayo	CSC	35

5.1.7 Intersection (∩)

This binary operation on two relations S and D represented as S∩D, generates a new relation consisting of all the rows that are common to the two operand relations only. The two relations must be union-compatible. The schema of the new relation is identical to the schema of the first operand i.e. **S**. For instance, Intersection between two tables with each table having a single tuple, will fail if the tuple in first table has columns defined to be numeric while the second relation has columns in its tuple defined as character attributes.

Example:

Applying intersection to L and E that is, **L∩E,** using Table 5.23 will yield the result in the table below:

L∩E

Number	**Name**	**Dept**	**Age**
56	Obi	EEE	50
80	Joseph	IMT	38

5.1.8 Division

The division operator is often relevant when constructing certain queries. A division operation would generate tuples in one relation (L) that match all tuples in another relation (E).
Let L= (A_1,..A_j,B_1,…B_k); E=(B_1,…B_k); then the result L/E on schema L-E=(A_1,…A_j) is the set of all rows(a) such that for every row(b) in E, (a,b) is in L.
That is:

$$L/E=\{ t \mid t \in \prod_{L-E}(L) \wedge \forall u \in E (tu \in L) \}$$

for every tuple in L-E (t), there are a set of tuples in L, such that for all tuples (such as u) in E, the tu is a tuple in L.; Where ***u*** *represents any tuple in E;* ***tu*** is the concatenation of a tuple *t* and *u* to produce a single tuple.

L/E can be evaluated in terms of Projection, Set-difference.

To explain the division operation let us consider the *CourseResult* and *RegisteredCourses* relations in Table 5.24 below

Example:

1. Find all students who passed all courses that were offered for Harmattan Semester 2005.

Table 5.24: *CourseResult* relation

RegNumber	CourseCode	Semester	Grade
2010200	IMT 403	Harmattan 2005	B
2010203	IMT 407	Harmattan 2005	A
2010209	IMT 409	Winter 2005	A
2010209	EEE 315	Harmattan 2005	B
2011305	IMT 411	Winter 2015	C

RegisteredCourses relation

CourseCode	Semester	LecturerID
IMT 403	Harmattan 2005	Prof. Oke
IMT 407	Harmattan 2005	Prof. Onwuliri
IMT 409	Winter 2015	Prof. Tobi
EEE 315	Harmattan 2005	Prof. Maura
CSC 342	Winter 2004	Prof. Eze
PMT 322	Harmattan 2012	Prof. Hassan
IMT 411	Harmattan 2006	Prof. Obi

Solution:
Let **C/R** be the required relation that provides the solution to the problem;

For C:

- Get from the *CourseResult relation* (see Table 5.24) the registration numbers(***RegNumber***) and course code(***CourseCode***) for all courses passed by every student(apply Selection followed by Projection); that is:

$$C = \prod_{\text{RegNumber, CourseCode}}(\sigma_{\text{Grade}<>'F'}(CourseResult))$$

For R:

- Get from *RegisteredCourses* relation all course codes of all courses taught during the Harmattan semester of 2005(also apply selection followed by projection) that is:

$$R = \prod_{\text{CourseCode}}(\sigma_{\text{semester}='\text{Harmattan 2005}'}(RegisteredCourses))$$

The results of these relational expressions are presented below:
For **C**, we have:

$\sigma_{\text{Grade}<>'F'}(\text{CourseResult})$

RegNum	CourseCode	Semester	Grade
2010200	IMT 403	Harmattan 2005	B
2010203	IMT 407	Harmattan 2005	A
2010209	IMT 409	Winter 2005	A
2010209	EEE 315	Harmattan 2005	B
2011305	IMT 411	Winter 2015	C

$\prod_{RegNumber,\ CourseCode}(\sigma_{Grade<>'F'}(CourseResult))$

RegNum	CourseCode
2010200	IMT 403
2010203	IMT 407
2010209	IMT 409
2010209	EEE 315
2011305	IMT 411

For C, we have:

$\sigma_{semester='Harmattan\ 2005'}(RegisteredCourses)$

CourseCode	Semester	LecturerID
IMT 403	Harmattan 2005	Prof. Oke
IMT 407	Harmattan 2005	Prof. Onwuliri
EEE 315	Harmattan 2005	Prof. Maura

$\prod_{CourseCode}(\sigma_{semester='Harmattan\ 2005'}(RegisteredCourses))$

=

CourseCode
IMT 403
IMT 407
EEE 315

The two tables C and R are shown below:

C

RegNum	CourseCode
2010200	IMT 403
2010203	IMT 407
2010209	IMT 409
2010209	EEE 315
2011305	IMT 411

R

CourseCode
IMT 403
IMT 407
EEE 315

So C/R, which is the final result, that is, all students that passed all courses taught in Harmattan semester 2005 is shown in Table 5.25

Table 5.25: **C/R**

RegNum
2010200
2010203
2010209

5.2 Relational Calculus

Relational calculus, an alternative to relational algebra, is a non-procedural or declarative query language as it specifies what is to be retrieved rather than how to retrieve it. It is a High-level language based on first-order logic description. Relational Calculus makes use of simple sentences and refers specifically to the relations and values in the database of interest. Simple sentences typically take the form of comparisons of values denoted by variables and/or constants that is; queries are expressed in the form of variables and formulas consisting of these variables.

Complex sentences are built using logical connectives which will be discussed later. Sentences whether simple or complex, are broken down using a formula.

Each formula though may have its syntactic form, is a logical function with one or more free variables. A free variable is a notation that specifies where substitutions may take place in an expression. The formula specifies the properties of the resultant relation without giving specific procedure for evaluating it.

There are two forms of relational calculus, namely: T*uple relational calculus (TRC)* and *Domain relational calculus (DRC)*.

In tuple relational calculus, the variable ranges over tuples from a specified relation and in domain relational calculus, the variable ranges over attributes from a specified relation. The Structured Query Language (SQL) is influenced by TRC; whereas the Query-By-Example (QBE) is influenced by DRC. Relational calculus languages are based on first order predicate calculus.

5.2.1 Building Blocks

A relational calculus expression is made up compAn express

Variables

A variable may be bound or free. It is **bound** if it appears in qualifier expressions. Otherwise, it is a **free** variable. There are two kinds of variables in relational calculus:

- Variables in TRC also called tuple variables: these variables are bound to tuples that is; variables assume the values of tuples in which case every value assigned to a given tuple variable would have the same number and column types.
- Variables in DRC also called column or field variables: Variables are bound to *domain elements* (column values) that is, the variables here assume field values

Any tuple variable with **∀(for all) or ∃(there exists) qualifier** is a bound variable**.**

Example:

1. The expression: **{t | STUDENT (t) ∧ t.AGE>20}** will produce tuple(s) from a *STUDENT* relation that will have all values of the *AGE* attribute greater than 20.

The condition t.AGE >20; may be interpreted thus, for every student with age>20, list the student. In this case, AGE is a **bound variable**. Moreover, for any range of values of AGE greater than 20, the implication of the condition remains unchanged. So **t.AGE** is a tuple variable as well as a bound variable. Bound variables are those ranges of tuple variables whose meaning remains unchanged if the tuple variable is replaced with another tuple variable. On the other hand, free variables are those ranges of tuple variables whose meaning changes once the tuple variable is replaced by another tuple variable.

Constants

Constants also form part of relational calculus. They are used to represent unchanging values such as: 57, "Lagos", 4.1402, "F",etc.

Comparison operators

The comparison operators include: =, <>, <, >,>=, <=,≠, etc.

Logical connectives

These are: **¬ (not), ∧ (and or conjunction), ∨ (or/disjunction), --> (implies), ∈ (is a member of)**

Quantifiers

Quantifiers may be regarded as components of predicate logic that quantify variables. There are 2 common types:

1. Existential quantifier, denoted by the symbol '□', and
2. Universal quantifier, denoted by the symbol '□'

An existentially quantified variable, say x, is written "□x" and is read as "there exists an x such that...". A universally quantified variable is written as "□x" and is read as "for all x..." or "for every x…."

Examples:

- ∀X (p(X)): For every X, p(X) must be true
- ∃X(p(X)): There exists at least one X such that p(X) is true

Propositional logic

A **proposition** is a statement which might be true or false e.g. *"3 divides 6", "5 doesn't divide 7". A* **Propositional logic** is concerned with operators which create new propositions from specified ones e.g. "3 divides 9" and "5 doesn't divide 9".

Expressions or sentences of propositional logic rely on: **Atoms (*q, p)*** *and* Logical connectives **(∨, ∧, ¬)** e.g. $p \wedge q, \neg(\neg p \vee \neg p)$. Propositions can be true or false, dependent on the interpretation given to their atoms. For example, the proposition $p \wedge q$ is true when p="3 divides 9" and q="5 doesn't divide 9".

Predicate Logic

Predicates are parameterized propositions, allowing references to classes of objects. **Predicate logic** is an extension of propositional logic interested both in the sentential connectives of the atomic propositions, and in the internal structure of the atomic propositions.

- Atoms allow functions and relations on variables
- The variables may be quantified: □ (there exists), □ (for all)
- Non-quantified variables are said to be free

Examples:

1. □*c*□*s*1□*s*2 (divides (*s*1,*c*) ∨¬divides(*s*2,*c*)) =

For each digit c there exists a digit s1 that divides c or a digit s2 that does not divide c.

2. □*s*1□*s*2□*c*(divides(*s*1,*c*) ∨¬divides(*s*2,*c*))

There exist digits s1 and s2 such that every digit c is either divisible by s1 or indivisible by s2.

Non-quantified variables or free variables

3. $\exists c(\text{divides}(s1,c) \vee \neg\text{divides}(s2,c))$

Table 5.26: StudentResult relation

RegNumber	CourseCode	Semester	Grade
2010200	IMT 403	Harmattan 2005	B
2010203	IMT 407	Harmattan 2005	A
2010209	IMT 409	Winter 2005	A
2010209	EEE 315	Harmattan 2005	B
2011305	IMT 411	Winter 2015	C

4. Using Table 5.26, find all students that score A in their courses.
This is a case of Tuple relational calculus (tuples will be attached to the variable say **S**(S is just any tuple variable).
That is: $\{S \mid S \in \textit{StudentResult} \wedge S.Grade = \text{"A"}\}$

5.2.2 Tuple Relational Calculus (TRC)

A little review of the relational database model would help us understand this concept. The basic relational data model building block is the domain, or data type. A tuple is an ordered multi-set of attributes, which are ordered pairs of domain and value; or a row. A *relvar* (relation variable) is a set of ordered pairs of domain and name, which serves as the header for a relation. A relation is a set of tuples. Although these relational concepts are mathematically defined, those definitions map loosely to traditional database concepts. A table is an accepted visual representation of a relation; a tuple is equivalent to the *row*.
Assuming a set *C* of column names (e.g. "name", "age", "address"), the *headers* are defined as finite subsets of *C*. A *relational database schema* is defined as a tuple $S = (D, R, h)$ where *D* is the domain of atomic values, *R* is a finite set of relation names, and

$h: R \rightarrow 2^C$; a function that associates a header with each relation name in *R*.
Note: a relational model may have more than one domain and a header does not only define a set of column names but also maps the column names to a domain.

Given a domain *D* we define a *tuple* over *D* as a partial function that maps some column names to an atomic value in *D*. An example is: [name: "Peter", age:50].

$$t : C \rightarrow D$$

The set of all tuples over *D* is denoted as T_D. The subset of *C* for which a tuple *t* is defined is called the *domain* of *t* (this is different from the domain in the schema) and denoted as $dom(t)$. A *relational database* given a schema $S = (D, R, h)$ is a function: $db : R \rightarrow 2^{T_D}$;that maps the relation names in *R* to finite subsets of T_D, such that for every relation name *r* in *R* and tuple *t* in $db(r)$ it holds that $dom(t) = h(r)$.

The tuple relational calculus was introduced as part of the relational data model, to provide a declarative database-query language for this model. This account for the relationship between TRC and database query languages like QUEL and SQL.

The tuple relational calculus is based on tuple variables, which takes tuples of a specific relation as its values. A tuple is divided into parts and without a means of referencing such parts; the logical functions we construct with relations will have limited expressive power. For example, given two variables x and y that range over two different relations with a common domain, we may want to specify a condition where their particular instances are such that the values under the common domain are equal. That is, we would want to compare the component of a relation(x) to the component of another(y). The syntactic mechanism for this purpose may take the form: *<tuple-variable-name>.<attribute-name>* and is interpreted to mean the value associated with <attribute-name> in the present instance of <tuple-variable-name>. Therefore, TRC is equivalent to relational algebra in its expressive power. I

5.2.2.1 Expressions in TRC

In TRC, a query is an expression of the form:

> **{t:U | P(t) };** *where* t=free variable; P(t)=formula(Predicate P + variable); U is a set of attributes;

That is, a query is defined as the set of all tuples defined on a set **U** of attributes such that predicate **P** is true for t. The queries employ formulas of predicate logic with free variables. Set notation is used to highlight the free variables.

The Formula is recursively defined in that atomic formulas get tuples from relations or compare values, and built from other formulas using logical operators. The **domain** of a query consists of the tuples which may be assigned to the free variables of the formula. The **selection** of a query is defined by the set of assignments to the free variables which satisfy the formula.

The notation for expressing a tuple variable and its relationship with attributes and a base relation is:

> **t.A :** *denotes the value of tuple t for the attribute A*
>
> **t □ R** *:means that t is a member or belongs to relation R*

Example:

1. Get all the students enrolled in courses during the Harmattan semester in 2005

{ t:U**RegisteredCourses** | t □ RegisteredCourses ∧ t.Semester='Harmattan 2005'}

2. Get all the list of lecturers who are above 40 years from the lecturer relation whose surnames are identical with surnames provided in the Person schema.

{t :{ Surname}| □s(s □ Lecturer ∧ s.Age >40 ∧ t.Surname = s.surname)}

Here **t** is a tuple variable that represents tuples from a relation whose schema is {Surname}.

3. Get the registration number and levels of all students whose CGPA is greater than 4.49 or who have won exceptional performance award in the Department of IMT

{t :{ RegNumber,Level}| □s(s □ Student ∧ s.CGPA >4.49 ∧ t.RegNumber = s.RegNumber) ∧ t.Level = s.Level ∨ □k(k □ Awards ∧ k.Status='true' ∧ k.DeptID="IMT" ∧ k.RegNumber=s.RegNumber))}

5.2.2.2 Atoms and Formula

An atom is a formula in the strict sense of its components. An ***atom*** can be thought of as a simple logical expression such as:

- x R
- x θ y :where x and y are attributes or constants, and θ is a comparison operator such as >,<,=,≥,≤,≠.
- x.A θ y.B :where x and y are variables, θ is a relational comparison operator, A and B are attribute names; A's value in tuple x is in relation θ with the value of attribute B in tuple y
- x.A θ k :where s is a variable, θ is a relational comparison operator and k is a constant value

Atoms allow functions and relations on variables. Ordinarily all variables in an atom are free variables until bound by quantifiers.

Examples of atoms are:

- *t*.age = *s*.age ; *t* has an age attribute and *s* has an age attribute with the same value
- *t*.lastname = "Peter"; tuple *t* has a '*lastname'* attribute and its value is "Peter"
- Student (*t*): tuple *t* is present in the Student relation.

A *formula* is either composed of an atom or logical expressions. That is, Formula can be composed of atomic formulas or built from other formulas using logical operators. Thus, a formula could be any of these:

a. an atomic formula
b. $p \wedge q$, $\neg p$, $p \vee \neg p$ where p and q are formulas
c. ∀X (p(X)): where *X* is a tuple variable
d. ∃X(p(X)): where *X* is a tuple variable

5.2.2.3 Free and Bound Variables

As has been discussed above, variables in a TRC expression may be free or bound. A quantifier such as ∀X(for every X…), or ∃X(there exists X…) in a formula would covert a variable such as X from being a free variable to a bound variable. In every case, the fundamental expression {*t* | *p*(*t*)} holds that the variable *t* that appears to the left of `|' must be the *only* free variable in the formula *p*(*t*). That is, all other tuple variables in the expression must be bound using a quantifier.

The Universal (∀) and Existential(∃) quantifier expressions

Examples:

1) ∀x (P(x)): only true if P(x) is true for *every* x in the universe: e.g. ∀x ((x.gender = "Female") means for every x in the universe, the gender is a female
2) ∀x ((x ∈ Lecturer) ⇒ (x.age > 20) ⇒ is a logical implication : a ⇒ b implies that if a is true, b must be true; a ⇒ b is the same as ¬a ∨ b

3) $\forall x$ ((x $\in$ Lecturer) $\Rightarrow$ (x.class = "Employee")) : "For every x in the *Lecturer* relation, the class must be Employee." This can also be written as: $\forall x \in$ *Lecturer*(x.class = "Employee")
4) $\exists x$((x $\in$ Lecturer) $\wedge$ (x.class = "Employee")) : "There exists a tuple x in the *lecturer* relation for which its class is Employee." This can also be written as: $\exists x \in$ Lecturer (x.class = "Employee")
5) y:{DeptID}| $\exists x$(x$\in$Employees $\wedge$ x.age >25 $\wedge$ x.DeptID= y.DeptID)}; there exists a tuple x in the Employee relation that has its age above 25 and for which its department is in identity with that of a another tuple y.

5.2.2.4 Relational algebra operations in TRC

A query whether expressed in a TRC or relational algebra has same underlying semantics which is to transform one or more relations into another set of relation(s). Both relational algebra and TRC have the same expressive power if for each query q expressed in one of these two formalisms, a query p exists expressed in the other formalism, such that the two queries have the same semantics. The Expressive Power, according to Codd is that which ensures that every query that can be expressed in relational algebra can be expressed as a *safe* query in DRC / TRC; the converse is also true.

However, not all the expressions of TRC can be expressed in equivalent expressions of the relational algebra. Some of the circumstances that may give rise to this are:

Infinite sets:

Let us consider the expression: {t:U_R| ¬t$\in$R};

Though this expression is syntactically correct, if one of the domains of the attributes of R is an infinite set, the expression is satisfied by an infinite number of tuples.

Notion of formula domain independent: A formula is domain independent if its evaluation generates the same result even if the domains associated with attributes are extended by adding new values, not present in the initial database. A syntactic condition has been introduced, referred to as safety, to assure such property.

The main idea of safety: we restrict our queries to queries the results of which only depend on the values present in the initial database.

The independency of a formula from the domain is not decidable. The safety condition is a syntactic condition sufficient to guarantee the independency of a formula from the domain.

Unsafe queries:

The expression {t:U_R| ¬t$\in$R} in TRC is not safe. This is because it can generate an infinite number of answers e.g. {s:$U_{Student}$| ¬s$\in$Student}. A query is regarded to be safe if it produces a finite answer regardless of how we instantiate the relations.

The safe TRC and the relational algebra have the same expressive power. Safe queries can always be defined and encoded in Relational algebra unlike the TRC.

Selection and Projection

We shall use the following tables in our examples in this section.

Student_Profile table

RegNum	NumCourses	TotalCredits	Semester	GPA	DeptID
2013200	7	21	Harmattan 2014	4.2	001
2013203	8	30	Harmattan 2014	4.7	003
2010209	7	21	Winter 2005	4.2	004
2011219	7	21	Harmattan 2005	4.5	001
2011305	8	32	Winter 2015	4.0	006

Department table

DeptID	DeptName
001	IMT
002	PMT
003	EEE
004	MEE
005	CSC
006	MTH
007	MMT

Course_Adviser table

DeptID	AdviserName
001	Prof. Oke
002	Prof. Onwuliri
003	Prof. Tobi

Selection

Problem: Find all students with GPA above 4.0 during the harmattan semester of 2005

$$\{S \mid S \in Student\_Profile \wedge S.GPA > 4.0 \wedge S.Semester='Harmattan\ 2005'\}$$

The result of this TRC query is the table below.

RegNum	NumCourses	TotalCredits	Semester	GPA	DeptID
2011219	7	21	Harmattan 2005	4.5	005

Projection

Problem: List all Registration numbers of students who obtained a GPA of not less than 3.5, their total course credits, and who registered during the Harmattan semester of 2014.

$$\{S \mid \exists Q \in Student\_Profile(Q.GPA \geq 3.5 \wedge Q.Semester='Harmattan\ 2005' \wedge \mathbf{S.RegNum} = Q.RegNum \wedge \mathbf{S.TotalCredits} = Q.TotalCredits)\}$$

The tuple {S} has two fields ***RegNum*** (registration number) and ***TotalCredits***. The result of the projection is presented in the table below.

RegNum	TotalCredits
2013200	21
2013203	31

Joins

Problem : How many students with a GPA above 3.5 are from the IMT Department?

$$\{S \mid S \in Student\_Profile \wedge S.GPA > 3.5 \wedge \exists D(D \in Department \wedge D.DeptID = S.DeptID \wedge D.DeptName = 'IMT')\}$$

Problem:

Find students whose GPA > 4.0 who are from IMT Department and had Prof. Oke as their course adviser.

$$\{\mathbf{S} \mid S \in Student\_Profile \wedge S.GPA > 4.0 \wedge \mathbf{\exists D(}D \in Department \wedge D.DeptID = S.DeptID \wedge \mathbf{\exists C(}C \in Course\_Adviser \wedge C.DeptID = S.DeptID \wedge C.AdviserName = 'Prof.\ Oke'))\}$$

Division

In relational algebra, the division operation A/B is handled by eliminating a value *x* in A which when attached to *y* value from *B*, will produce a tuple *xy* tuple that is not in *A*. That is, give me only A tuples that have a match in B.

In TRC, the approach is different as we can use the universal quantifier to get this result.

Problem**:**

Find all students who have registered all courses

Assuming there are three tables: *Student*, *Course*, and *RegisteredCourse* defined by: *Student(RegNumber, Name, DeptID)*, *Course(CourseCode, CourseTitle, Credits)*, *RegisteredCourse(Regnumber, CourseCode, Semester)*; this query can be represented as:

$$\{S \mid S \in Student \wedge \forall C \in Courses\ (\exists R \in RegisteredCourse\ (S.RegNumber = R.RegNumber \wedge C.CourseCode = R.CourseCode))\}$$

5.2.3 Domain Relational Calculus (DRC)

DRC was introduced by Michel Lacroix and Alain Pirotte, as a declarative database query language for the **relational** data model. In DRC, queries have the form: where each X_i is either a **domain** variable or constant, and denotes a DRC formula

Like the TRC, DRC uses the same relational, and logical connectives such as ∧ (and), ∨ (or) and ¬ (not). The existential quantifier (∃) and the universal quantifier (∀) are also used to bind the variables. The computational expressiveness of DRC is adjudged to be equivalent to that of Relational algebra. Unlike in TRC where tuple value is used as a variable, in DRC, the filtering variable uses the domain of attributes instead of entire tuple values.

The DRC is the framework behind Query by Example (QBE) which is implemented in a Personal computer database application like Microsoft Access.

5.2.3.1 Atoms in DRC

An atom in DRC is of the following forms:

a. $< x_1, x_2, \dots, x_n > \in r$: where *r* is a relation on *n* attributes and $x_1, x_2, \dots, x_n$ are the domain variables or domain constraints.
b. $x \Theta y$: where *x* and *y* are domain variables and Θ is the comparison operator (<,>, ≤, ≥, =, ≠). It is required that *x* and *y* have domains that can be compared by Θ.
c. $x \Theta c$: where *x* is a domain variable, Θ is a comparison operator, and *c* is a constraint in the domain of attributes for which x is a domain variable.

5.2.3.2 Formula

Formulae may be constructed from atoms using the rules below:

a. An Atom is a formula.
b. If P_1 is a formula, then so are $\neg P_1$ and (P_1).
c. If P_1 and P_2 are formulae, then so are $P_1 \bigvee P_2$, $P_1 \bigwedge P_2$, and $P_1 \Rightarrow P_2$.
d. If $P_1(x)$ is a formula in *x*, where *x* is a free domain variable, then
   $\exists x (P_1(x))$ and $\forall x (P_1(x))$

5.2.3.3 Queries and expressions

Queries and *Expressions* have the form:

$\{< x_1, x_2, \dots, x_n > \mid P(x_1, x_2, \dots, x_n)\}$; Where $x_1, x_2, \dots, x_n$ represents domain variables; P represents a formula composed of atoms

The result of the query is the set of tuples X_1 to X_n which makes the DRC formula true.

Problem 1:

Selection

Get all Registration numbers of students where the cumulative grade point average (CGPA) is at least 4.5:

$\{< \text{regNumber} > \mid \exists \text{cgpa} (<\text{regNumber}, \text{cgpa}> \in \textbf{\textit{Student\_Profile}} \bigwedge \text{cgpa} \geq 4.5)\}$

This is equivalent to the Relational Algebra expression

$$\Pi_{\text{regNumber}}(\sigma_{\text{cgpa} >= 4.5} (\text{Student\_Profile}))$$

Problem 2:

Display the registration number, name, and cgpa of all students where the department number is 0004

{<regNumber, name, cgpa> | <regNumber, name, cgpa, DeptID> ∈ Student_Profile ∧ DeptID= '004'}

Simple projection and selection

Problem 3:

Display all students by registration number, department, cgpa who scored a CGPA of 3.5 and above during the harmattan 2014 semester.

The equivalent expression in DRC would be:

{<regnumber,deptID,cgpa>| ∃ cgpa, semester.(<regnumber,deptID, cgpa,semester > ∈ Student_Profile ⋀ cgpa ≥ 3.5 ⋀ semester='harmattan 2005')}

Problem 4:

How many students with a GPA above 3.5 are from the IMT Department?

{ < regnumber, name > | ∃ regnumber, cgpa,deptID .<regnumber,name,cgpa, deptID> ∈ Student_Profile ⋀ cgpa> 3.5 ⋀ ∃ deptID. (<deptID, deptName> ∈ Department ⋀ DeptName= 'IMT)}

Join

Problem 5:

List students by registration number and name, who scored a GPA of 4.5 whose course adviser is Prof. Oke.

The query would be a join query and in DRC is:

{ < *regnumber, name* > | ∃ *regnumber, cgpa.* (<*regnumber,name,cgpa, deptID*> ∈*Student_Profile*⋀*cgpa*= 4.5 ⋀ ∃ *dept*. (<*dept, AdviserName*> ∈ *CourseAdviser* ⋀ dept=deptID ⋀ *AdviserName*= 'Prof. Okey')}

Problem 6:

Find the names of students who are not in IMT department but registered the course IMT 407

{<Name> | ∃DeptName,RegNumber.<Name,RegNumber,Deptname>∈Student ∧ DeptName<>'IMT'∧∃RegNum,CourseCode.(<Regnum,Coursecode>∈RegisteredCourses∧ RegNumber=RegNum ∧ CourseCode='IMT 407')}

Problem 7:

Find the registration number and department of male students who registered more than 28 course credits during the Harmattan 2015 semester

{<RegNumber,Dept> | ∃Regnumber,Sex.<Regnumber,Name,Sex,Dept>∈Student ∧ Sex='male'∧∃regnum,semester,totalCredits.(<regnum,coursecode,courseTitle,semester,totalCredits>∈RegisteredCourses ∧ regnum=Regnumber ∧ semester='harmattan 2015' ∧totalCredits>28)}

Problem 8:

List the students by registration number and department who registered not more than 28 credits in the department of computer science in the 2015/2016 session

{<regNumber, dept> | ∃ regNumber, dept. <regNumber, Name, dept>∈ Student ∧ dept='computer science' ∧ ∃ regnum, totalcredit, semester.<regnum, semester, totalcredit>∈ RregisteredCourses ∧ regnum=regNumber ∧ totalcredit≤28 ∧ semester='harmattan 2015' ∨ semester='winter 2015' }

Problem 9:

Get the registration number, name and department of all 500 level students who have passed all their courses.

{<regNumber, name, dept> | ∃regNumber, level. <regnumber, dept, level, cgpa>∈Student_Profile ∧ level=500 ∧ ∃coursecode. (<regnum, coursecode, creditstatus> ∈RegisteredCourses ∧ regnum=regnumber ∧ creditstatus='maximum' ∧ (∃regno, courseID,grade<regno,courseID,grade>∈CourseResults ∧ regno=regnumber ∧ courseID=coursecode ∧ grade<>'F'))}

5.3 Review questions

A. What do you understand by a query language?
B. Discuss the relationship between relational algebra and relational calculus.
C. Differentiate between a procedural and a non-procedural language citing examples
D. How has relational algebra and calculus influenced the development of modern application-level query languages?
E. Relational algebra is considered a lower level language, true or false? If true, why do you think so?
F. Differentiate between domain relational calculus and tuple relational calculus
G. What are the components of a relational calculus expression?
H. Discuss the differences between an atom and a formula (if any)
I. Are all expressions in relational algebra convertible to relational calculus equivalent? How can a selection in relational algebra be implemented using a tuple relational calculus?
J. Some relational calculus expressions may be unsafe. What do you understand by the safety of an expression having regard to the tuple relational calculus?