Chapter 5

# RELATIONAL ALGEBRA AND CALCULUS

## 5.1 INTRODUCTION

Relational algebra is one of the two formal query languages associated with the relational model. It is a procedural language. It specifies the operations to be performed on existing relations to derive result relations. A sequence of relational algebraic operations forms a relational algebraic expression. The result of the relational algebraic expression is also a relation.

The relational algebra is very important due to many reasons. Firstly, it provides a basic foundation for relational model operations. Secondly, it is used as a basis for implementing and optimizing queries in RDBMS's. Thirdly, some of the basic concepts of relational algebra are incorporated into the SQL language.

*Relational calculus* is a formal query language where the queries are expressed as variables and formulas on these variables. The formula used in relational calculus, describes the properties of the result relation to be obtained. There is no mechanism to specify how the formula should be evaluated. It is the sole responsibility of the RDBMS to transform these non-procedural queries into equivalent, efficient, procedural queries. It is a non-procedural language. The relational calculus is a formal language, based on predicate calculus. Relational calculus is a combined term for *tuple calculus* and *domain calculus*. In **tuple calculus**, variables range over tuples and in **domain calculus**, variables range over the domains of attributes.

## 5.2 RELATIONAL ALGEBRA

Relational algebra is a procedural query language. It uses a collection of operators to compose the queries. Every operator in the algebra accepts either one or two relation instances as arguments and output a resultant relation instance. Thus operators can be composed easily to construct complex queries. Each relational algebra query describes a step-by-step procedure

for computing the desired answer, based on the order in which the operators are applied in the query.

The relational algebra uses various logical connectives [∧(and), ∨(or), ¬(not)] and comparison operators (<, <=, =, ≠, >=, >) to construct composite and more complex queries.

We discuss the different operators (Basic set oriented operators—union, intersection, difference, and cartesian product and relational operators—selection, projection, division and join) in detail, with examples, in subsequent sections. All the examples are based on the EMPLOYEE-STUDENT database shown in Figure 5.1. This database contain two Tables EMPLOYEE and STUDENT and the relationship is that an employee can also a student and vice-versa.

**Employee**

| EID | Name | Salary |
|---|---|---|
| 1E | John | 10,000 |
| 2E | Ramesh | 5,000 |
| 3E | Smith | 8,000 |
| 4E | Jack | 6,000 |
| 5E | Nile | 15,000 |

**Student**

| SID | Name | Fees |
|---|---|---|
| 1S | Smith | 1,000 |
| 2S | Vijay | 950 |
| 3S | Gaurav | 2,000 |
| 4S | Nile | 1,500 |
| 5S | John | 950 |

FIGURE 5.1. *Employee and Student relations.*

### 5.2.1 Operations in Relational Algebra

The relational algebraic operations can be divided into basic set-oriented operations (union, intersection, set difference, and Cartesian product) and relational-oriented operations (selection, projection, division and joins).

#### 5.2.1.1 Basic Set-oriented Operations

(*a*) *The union operation* : The union operation is a binary operation that is used to find union of relations. Here relations are considered as sets. So, duplicate values are eliminated. It is denoted by (U).

*Conditions for union operation* : There are two necessary conditions for Union operation.

(*i*) Both the relations have same number of attributes.

(*ii*) Data types of their corresponding attributes must be same.

Two relations are said to be union compatible if they follow the above two conditions.

*Ex.* If you want to find the names of all employees and names of all students together then the query is

$$\pi_{Name}(\text{Employee}) \cup \pi_{Name}(\text{Student})$$

The result is shown in Figure 5.2.

| Name |
|---|
| John |
| Ramesh |
| Smith |
| Jack |
| Nile |
| Vijay |
| Gaurav |

**FIGURE 5.2.** *Result of union operation.*

(*b*) *Set intersection operation :* Set intersection is used to find common tuples between two relations. It is denoted by (∩). If you want to find all the employees from Relation Employee those are also students. Rules of Set Union operations are also applicable here. Then the query, is

$$\pi_{\text{Name}} \text{ (Employee)} \cap \pi_{\text{Name}} \text{ (Student)}$$

The result is shown in Figure 5.3.

| Name |
|---|
| John |
| Smith |
| Nile |

**FIGURE 5.3.** *Result of set intersection operation.*

(*c*) *Set-difference operation :* Set-difference operation is a binary operation which is used to find tuples that are present in one relation but not in other relation. It is denoted by (—). It removes the common tuples of two relations and produce a new relation having rest of the tupels of first relation.

*Ex.* If you want the names of those employees that are not students, then the query, is

$$\pi_{\text{Name}} \text{ (Employee)} - \pi_{\text{Name}} \text{ (Student)}$$

The result is shown in Figure 5.4.

| Name |
|---|
| Ramesh |
| Jack |

**FIGURE 5.4.** *Result of set-difference operation.*

(*d*) *Cartesian product operation :* Cartesian product is a binary operation which is used to combine information of any two relations. Suppose a relation R1 is having $m$ tuples and other relation R2 is having n tuples then R1 × R2 has $m \times n$ tuples. It is denoted by (X). Consider the Figure 5.5. Cartesian product of relation **Employee** and **Job** is shown in Figure 5.6.

Query is →

Employee

| EID | Name | JID |
|---|---|---|
| 1E | Manoj | 1J |
| 2E | Deepak | 2J |
| 3E | Vinay | 1J |

Job

| JID | Job |
|---|---|
| 1J | Tester |
| 2J | Manager |

FIGURE 5.5. *Employee and Job relation.*

| EID | Name | Employee JID | Job JID | Job |
|---|---|---|---|---|
| 1E | Manoj | 1J | 1J | Tester |
| 1E | Manoj | 1J | 2J | Manager |
| 2E | Deepak | 2J | 1J | Tester |
| 2E | Deepak | 2J | 2J | Manager |
| 3E | Vinay | 1J | 1J | Tester |
| 3E | Vinay | 1J | 2J | Manager |

FIGURE 5.6. *Result of Cartesian product operation.*

### 5.2.1.2 Relational Oriented Operation

(a) *Selection or Restriction operation* : The selection operation is a unary operation. This is used to find horizontal subset of relation or tuples of relation. It is denoted by sigma (σ).

*Ex.* If you want all the employees having salary more than 9,000 from relation Employee. The query, is

$$\sigma_{salary > 9,000} (Employee)$$

The result is shown in Figure 5.7

| EID | Name | Salary |
|---|---|---|
| 1E | John | 10,000 |
| 5E | Nile | 15,000 |

FIGURE 5.7. *Result of selection operation.*

We can also combine these operations. If you want name of all employees having salary less than 7,000. Then the query is

$$\sigma_{salary < 7,000} [\pi_{Name} (Employee)]$$

Step 1. First we apply projection operation on relation employee to get name of all employees.

$$\pi_{Name} (Employee)$$

Step 2. Then we apply selection operation to choose employees having salary less than 7,000.

$$[\sigma_{salary < 7,000} (\pi_{Name} (Employee))] \rightarrow \text{Relational algebra expression}$$

The Result is shown in Figure 5.8.

| Name |
|---|
| Ramesh |
| Jack |

FIGURE 5.8. *Result of* $\sigma_{salary < 7,000}$ $[\pi_{name}$ *(Employee)]*.

(b) *Projection operation* : The projection operation is a unary operation which applies only on a single relation at a time. Project operation is used to select vertical subset of relation (*i.e.*, columns of table). It is denoted by pi ($\pi$).

Consider Figure 5.1. If you want all the names of employees and their salary from relation Employee. Then query, is

$$\pi_{name,\ salary}\ \textbf{(Employee)}$$

The result is shown in Figure 5.9.

| Name | Salary |
|---|---|
| John | 10,000 |
| Ramesh | 5,000 |
| Smith | 8,000 |
| Jack | 6,000 |
| Nile | 15,000 |

FIGURE 5.9. *Result of projection operation.*

(c) *Division operation* : Division operation is useful in special kind of queries that include the phrase "for all". It is denoted by (÷). It is like the inverse of Cartesian product.

*For example :*

A

| X | Y |
|---|---|
| X1 | Y1 |
| X1 | Y3 |
| X1 | Y2 |
| X4 | Y |
| X5 | Y5 |
| X2 | Y3 |
| X3 | Y4 |
| X4 | Y1 |

B1

| Y |
|---|
| Y1 |
| Y3 |

B2

| Y |
|---|
| Y1 |
| Y5 |

B3

| Y |
|---|
| Y5 |
| Y2 |
| Y4 |

A + B1 gives

| X |
|---|
| X1 |
| X4 |
| X2 |

A + B2 gives

| X |
|---|
| X1 |
| X4 |
| X5 |

A + B3 gives

| X |
|---|
| X5 |
| X1 |
| X3 |

FIGURE 5.10. *Result of division operation.*

Let R be the relation with schema $r$ and S be the relation with schema $s$. Let $S \subseteq R$. Then tuple $t$ is in $r \div s$ if and only if

(*i*) $t$ is in $\pi_{R-S}(r)$

(*ii*) If tuple is present in the Cartesian product of S and R.

(d) *Natural-join operation* : Natural join is used to join two relations having any number of attributes. It is denoted by symbol (⋈). It also optimize the query as Cartesian product gives unnecessary results and set-union and set-intersection operations are applicable only on those relations that have equal number of attributes with same data-type. Consider the relations in Figure 5.11.

**Employee**

| EID | Name | Salary | Dept-ID |
|---|---|---|---|
| 1 | Amit | 5,000 | 10 |
| 2 | Sachin | 8,000 | 20 |
| 3 | Vinay | 2,000 | 20 |
| 4 | Vivek | 6,000 | 10 |

**Department**

| Dept_ID | Dept_Name |
|---|---|
| 10 | Sales |
| 20 | Purchase |

FIGURE 5.11. *Employee and department relation.*

*Ex.* Find the names of all employees from relation Employee with their respective Department names. The query is

$$\pi_{\text{Name, Dept\_name}}\left[\begin{array}{c}\text{Employee} \bowtie \text{Department} \\ \text{Employee·EID} = \text{Department·Dept\_ID}\end{array}\right]$$

The result is shown in Figure 5.12.

| Name | Dept-Name |
|---|---|
| Amit | Sales |
| Sachin | Purchase |
| Vinay | Purchase |
| Vivek | Sale |

FIGURE 5.12. *Result of natural join operation.*

(e) *Outer Join* : Outer Join is an extension of natural join operations. It deals with the missing information caused by natural join operation.

Suppose you need all information of all employees and all students in a single relation.

*Natural Join* (⋈) : The natural join (Employee ⋈ Student) gives the result as shown in Figure 5.13.

| EID | SID | Name | Salary | Fees |
|---|---|---|---|---|
| 1E | 5S | John | 10,000 | 1,000 |
| 3E | 1S | Smith | 8,000 | 1,000 |
| 5E | 4S | Nile | 15,000 | 1,500 |

FIGURE 5.13. *Result of natural join.*

In this result, information about Ramesh, Jack, Vijay, Gaurav are missing. So, outer join is used to avoid this loss of information.

There are Three types of outer joins.

(i) *Left outer join* : It is used to take all tuples of relation that are on the left side whether they are matching with tuples of right side relation or not. It is denoted by (⟕).

(Employee ⟕ Student) gives

| EID | SID | Name | Salary | Fees |
|---|---|---|---|---|
| 1E | 5S | John | 10,000 | 950 |
| 2E | NULL | Ramesh | 5,000 | NULL |
| 3E | 1S | Smith | 8,000 | 1,000 |
| 4E | NULL | Jack | 6,000 | NULL |
| 5E | 4S | Nile | 15,000 | 1,500 |

**FIGURE 5.14.** *Result of left outer join.*

Employee relation is at left side so table in Figure 5.14 consists all information of Employee relation but still missing information about Vijay and Gaurav.

(ii) *Right outer join* : It is used to take all tuples of relation that are on the right side whether they are matching with tuples of left side relation or not. It is denoted by (⟖).

(Employee ⟖ Student)gives

| EID | SID | Name | Salary | Fees |
|---|---|---|---|---|
| 3E | 1S | Smith | 8,000 | 1;000 |
| NULL | 2S | Vijay | NULL | 950 |
| NULL | 3S | Gaurav | NULL | 2,000 |
| 5E | 4S | Nile | 15,000 | 1,500 |
| 1E | 5S | John | 10,000 | 950 |

**FIGURE 5.15.** *Result of right outer join.*

Student relation is at right side so table in Figure 5.15 consists all information of Student relation but still missing information about Ramesh and Jack.

(iii) *Full outer join* : It is used to take all tuples from left and right relation whether they match with each other or did not match. It is denoted by (⟗).

(Employee ⟗ Student) gives

| EID | SID | Name | Salary | Fees |
|---|---|---|---|---|
| 1E | 5S | John | 10,000 | 950 |
| 2E | NULL | Ramesh | 5,000 | NULL |
| 3E | 1S | Smith | 8,000 | 1,000 |
| 4E | NULL | Jack | 6,000 | NULL |
| 5E | 4S | Nile | 15,000 | 1,500 |
| NULL | 2S | Vijay | NULL | 1,000 |
| NULL | 3S | Gaurav | NULL | 2,000 |

**FIGURE 5.16.** *Result of full outer join.*

Table in Figure 5.16 consist all information of Employee and Student relation. Here, no information is missing.

**Points to Remember**

1. Perform projection and restriction as soon as possible.
2. Convert two or more operations into single operation if possible.
3. While joining two tables put small table on R.H.S.
4. At run time, you will see a blank space instead of NULL. For better understanding use "NULL".

### 5.2.1.3 Additional Operations

In addition to the fundamental operations, there are some additional operations to make relational algebra more flexible.

(a) *Rename operation :* The rename operation is a unary operation which is used to give names to relational algebra expressions. It is denoted by rho (ρ). Suppose, you want to find Cartesian product of a relation with itself then by using rename operator we give an alias name to that relation. Now, you can easily multiply that relation with its alias. It is helpful in removing ambiguity. Its general form is

$$\rho_x (R)$$

where R be any relation and $x$ be the alias name given to relation R.

*Ex.* Find the highest salary in Employee relation.

To do this, first make a temporary relation that contains all salaries except highest one. Then take the difference of temporary relation with Employee relation.

Query is

$\pi_{salary}$ (Employee) —

$$\pi_{employee.salary} [\sigma_{employee\ .\ salary\ <\ a\ .\ salary} (Employee \times \rho_a (Employee))]$$

Step 1. Take alias name of relation Employee by using rename operator

$$\rho_a (Employee)$$

Here alias name is a

Step 2. Take Cartesian product of relation Employee with its alias. This gives all combinations.

$$[Employee \times \rho_a (Employee)]$$

Step 3. Now, choose the salary in a way that temporary relation contains all salaries except highest one.

$$\pi_{employee.salary}\ [\sigma_{employee\ .\ salary\ <\ a\ .\ salary}\ (Employee \times \rho_a\ (Employee))]$$

Step 4. Take set-difference of temporary relation with relation Employee.

The result is shown in Figure 5.17.

| Salary |
|---|
| 15,000 |

FIGURE 5.17. *Highest salary in Employee relation.*

The second form of rename operator : You can also rename the attributes of any relation.

$$\rho_x\ (A_1, A_2, \ldots A_n)\ (E)$$

where $A_1, A_2 \ldots A_n$ are new names for attributes of relation E.

Consider the relation student in Figure 5.1.

$$\rho_{Students\ (Student\_ID,\ St\_Name,\ St\_Fees)}\ (Student)$$

The result is shown in Fig. 5.18.

**Student**

| Student_ID | St_Name | St_Fees |
|---|---|---|
| 1S | Smith | 1,000 |
| 2S | Vijay | 950 |
| 3S | Gaurav | 2,000 |
| 4S | Nile | 1,500 |
| 5S | John | 950 |

FIGURE 5.18. *Result of rename operation on student relation.*

(*b*) *Assignment operation* : Assignment operation is used to assign temporary names to relational algebra expressions. It is useful in large expressions. It is denoted by (←).

Consider an expression.

$$\pi_{ID,\ Name}\ [\sigma_{dept\ =\ "Sales"}\ (Employee)]$$

$$temp\ 1 \leftarrow \sigma_{dept\ =\ "Sales"}\ (Employee)$$

Now, you can write above expression as

$$\pi_{ID,\ Name}\ (temp\ 1)$$

## 5.3 RELATIONAL CALCULUS

An alternative to relational algebra is relational calculus. It is a query system where queries are expressed as variables and formulas on these variables. It is a non-procedural or declarative by nature. In this, the formulas describe the properties of the required result relation without describing how to compute it *i.e.*, query represents only results and hides the procedure to

find the result. Relational calculus is based on predicate calculus, which is used to symbolize logical arguments in mathematics. Relational calculus has two variants. The first one is called *Tuple Relational Calculus* in which variables take on tuples as values. The second one is called *Domain Relational Calculus* in which variables range over the underlying domains.

Relational Calculus has strongly influenced the design of many query languages like SQL and QBE.

### 5.3.1 Tuple Relational Calculus

Every query in tuple relational calculus is expressed by a tuple calculus expression, which is the basic construct. A tuple calculus expression is of the form

$$\left\{\frac{T}{F(T)}\right\}$$

Here, T is a set of tuple variable and F is the formula involving T. The result of this query is the set of all tuples $t$ for which the formula F(T) is true with $T = t$.

A tuple calculus expression is a non-procedural definition of some relation in terms of some given set of relations.

#### 5.3.1.1 Basic Concepts

Here, we discuss about how the tuple calculus formulas can be derived and the various concepts related with these formulas.

1. *Free variables :* Each tuple variable within a formula is either free or bound. A variable within a formula is said to be free if it is not quantified by the existential quantifier (∃) or the universal quantifier (∀).
2. *Bound variables :* A variable in a formula is said to be bound if it is quantified by either existential quantifier (∃) or the universal quantifier (∀).
3. *Qualified variable :* A qualified variable is of the form $t(A)$, where $t$ is a tuple variable of some relation and A is an attribute of that relation. Two qualified variables P(A) and Q(B) are domain compatible if attributes A and B are domain compatible.
4. *Atom or atomic formula :* An atom or atomic formula may be in any of the following forms :
   (*i*) $t. \in R$, where $t$ is a tuple variable and $R$ is a relation.
   (*ii*) $t[A] \otimes p[B]$, where ⊗ is any one of comparison operators ( $=, \neq, <, >, \leq, \geq$) and $t[A]$ and $p[B]$ are domain compatible variables.
   (*iii*) $t[A] \otimes$ Constant or Constant $\otimes t[A]$, where constant must be domain compatible.
5. *Well Formed Formula (WFF) :* A well formed formula (WFF) is recursively defined to be one of the following.
   (*i*) Every atom is a WFF.
   (*ii*) If $f$ is a WFF, then so are $(f)$ and $\neg f$.
   (*iii*) If $f$ and $g$ are WFF's , then so are $f \vee g$, $f \wedge g$, $f \rightarrow g$.
   (*iv*) If $f(x)$ is a WFF and $x$ is free, then $\exists x(f(x))$ and $\forall x(f(x))$ are also WFF's