**Relational Algebra**

* **Query Language**

In simple words, a Language which is used to store and retrieve data from database is known as query language. For example – SQL

There are two types of query language:

1.Procedural Query Language

2.Non-Procedural Query Language

**1. Procedural Query Language:**

In procedural query language, user instructs the system to perform a series of operations to produce the desired results. Here users tell what data to be retrieved from database and how to retrieve it.

**For example –** Let’s take a real-world example to understand the procedural language, you are asking your younger brother to make a cup of tea, if you are just telling him to make a tea and not telling the process then it is a non-procedural language, however if you are telling the step-by-step process like switch on the stove, boil the water, add milk etc. then it is a procedural language.

**2. Non-Procedural Query Language:**

In non-procedural query language, user instructs the system to produce the desired result without telling the step-by-step process. Here users tell what data to be retrieved from database but doesn’t tell how to retrieve it.

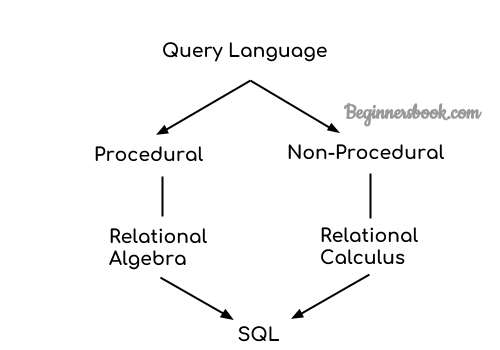
Now let’s back to our main topic of Relational Algebra and Relational Calculus.

**Relational Algebra:**

Relational algebra is a conceptual procedural query language used on relational model.

**Relational Calculus:**

Relational calculus is a conceptual non-procedural query language used on relational model.



**Relational Algebra, Calculus, RDBMS & SQL:**

Relational Algebra and Calculus are the theoretical concepts used on relational model.

RDBMS is a practical implementation of relational model.

SQL is a practical implementation of Relational Algebra and Calculus.

* **What is Relational Algebra in DBMS?**

Relational algebra is a procedural query language that works on relational model. The purpose of a query language is to retrieve data from database or perform various operations such as insert, update, delete on the data. When I say that relational algebra is a procedural query language, it means that it tells what data to be retrieved and how to be retrieved.

On the other hand, relational calculus is a non-procedural query language, which means it tells what data to be retrieved but doesn’t tell how to retrieve it.

Relational Algebra works on the whole table at once, so we do not have to use loops etc. to iterate over all the rows(tuples) of data one by one. All we have to do is specify the table name from which we need the data, and in a single line of command, relational algebra will traverse the entire given table to fetch data for you.

* **Types of Operations in Relational algebra**

We have divided these operations in two categories:

1. Basic Operations

2. Extended/Derived Operations

* **Basic/Fundamental Operations:**

1. Select (σ)

2. Project (∏)

3. Union (∪)

4. Set Difference (-)

5. Cartesian product (X)

6. Rename (ρ)

* **Derived Operations:**

1. Natural Join (⋈)

2. Left, Right, Full outer join (⟕, ⟖, ⟗)

3. Intersection (∩)

4. Division (÷)

Let’s discuss these operations one by one with the help of examples.

* **Select Operator (σ)**

Select Operator is denoted by sigma (σ) and it is used to find the tuples (or rows) in a relation (or table) which satisfy the given condition. Selection Operator (σ) is a unary operator in relational algebra.

**Syntax of Select Operator (σ):**

**σ Condition/Predicate(Relation/Table name)**

OR

**σ<selection\_condition>(R)**

OR

**σ p(r)**

Where: σ is used for selection prediction

r is used for relation

p is used as a propositional logic formula which may use connectors like: AND OR and NOT. These can use relational operators like =, ≠, ≥, <, >, ≤.

**Select Operator (σ) Example**

Table: CUSTOMER

---------------

Customer\_Id Customer\_Name Customer\_City

----------- ------------- -------------

C10100 Steve Agra

C10111 Raghu Agra

C10115 Chaitanya Noida

C10117 Ajeet Delhi

C10118 Carl Delhi

**Query:**

σ Customer\_City="Agra" (CUSTOMER)

**Output:**

Customer\_Id Customer\_Name Customer\_City

----------- ------------- -------------

C10100 Steve Agra

C10111 Raghu Agra

**Example:** LOAN Relation

|  |  |  |
| --- | --- | --- |
| **BRANCH\_NAME** | **LOAN\_NO** | **AMOUNT** |
| Downtown | L-17 | 1000 |
| Redwood | L-23 | 2000 |
| Perryride | L-15 | 1500 |
| Downtown | L-14 | 1500 |
| Mianus | L-13 | 500 |
| Roundhill | L-11 | 900 |
| Perryride | L-16 | 1300 |

**Input:**

σ BRANCH\_NAME="perryride" (LOAN)

**Output:**

|  |  |  |
| --- | --- | --- |
| **BRANCH\_NAME** | **LOAN\_NO** | **AMOUNT** |
| Perryride | L-15 | 1500 |
| Perryride | L-16 | 1300 |

**Example:**

* Select tuples from a relation “Books” where subject is “database”

σsubject = “database” (Books)

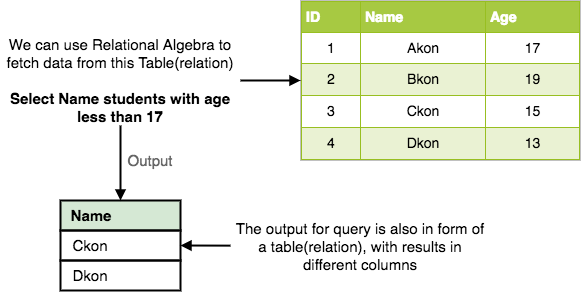
* Select tuples from a relation “Books” where subject is “database” and price is “450”

σsubject = “database” ∧ price = “450” (Books)

* Select tuples from a relation “Books” where subject is “database” and price is “450” or have a publication year after 2010

σsubject = “database” ∧ price = “450” ∨ year >”2010″ (Books)

**Example:**



* fetch data for **students** with **age** more than 17.

σage > 17 (Student)

This will fetch the tuples(rows) from table **Student**, for which **age** will be greater than **17**.

You can also use, and, or etc operators, to specify two conditions, for example,

* σage > 17 and gender = 'Male' (Student)

This will return tuples(rows) from table Student with information of male students, of age more than 17. (Consider the Student table has an attribute Gender too.)

* σ topic = "Database" and author = "guru99"( Tutorials)

Selects tuples from Tutorials where the topic is 'Database' and 'author' is guru99.

* σ sales > 50000 (Customers)

Selects tuples from Customers where sale is greater than 50000

**Important Points-**

**Point-01:** We may use logical operators like ∧ , ∨ , ! and relational operators like = , ≠ , > , < , <= , >= with the selection condition.

**Point-02:** Selection operator only selects the required tuples according to the selection condition.

**Point-03:** Selection operator always selects the entire tuple. It cannot select a section or part of a tuple.

**Point-04:** Selection operator is commutative in nature i.e.

σ A ∧ B (R) = σ B ∧ A (R)

**OR**

σ B(σ A(R)) = σ A(σ B(R))

**Point-05:** Degree of the relation from a selection operation is same as degree of the input relation.

**Point-06:** The number of rows returned by a selection operation is obviously less than or equal to the number of rows in the original table.

Thus,

* Minimum Cardinality = 0
* Maximum Cardinality = |R|
* **Project Operator (∏)**

Project operator is denoted by ∏ symbol and it is used to select desired columns (or attributes) from a table (or relation).

Project operator in relational algebra is similar to the Select statement in SQL.

**Syntax of Project Operator (∏)**

**∏ column\_name1, column\_name2, ...., column\_nameN(table\_name)**

OR

**π<attribute list>(R)**

OR

**∏ A1, A2, An (r)**

Where, A1, A2, A3 is used as an attribute name of relation r.

**Project Operator (∏) Example**

In this example, we have a table CUSTOMER with three columns, we want to fetch only two columns of the table, which we can do with the help of Project Operator ∏.

Table: CUSTOMER

Customer\_Id Customer\_Name Customer\_City

----------- ------------- -------------

C10100 Steve Agra

C10111 Raghu Agra

C10115 Chaitanya Noida

C10117 Ajeet Delhi

C10118 Carl Delhi

**Query:**

∏ Customer\_Name, Customer\_City (CUSTOMER)

**Output:**

Customer\_Name Customer\_City

------------- -------------

Steve Agra

Raghu Agra

Chaitanya Noida

Ajeet Delhi

Carl Delhi

**Example:** CUSTOMER RELATION

|  |  |  |
| --- | --- | --- |
| **NAME** | **STREET** | **CITY** |
| Jones | Main | Harrison |
| Smith | North | Rye |
| Hays | Main | Harrison |
| Curry | North | Rye |
| Johnson | Alma | Brooklyn |
| Brooks | Senator | Brooklyn |

**Input:**

∏ NAME, CITY (CUSTOMER)

**Output:**

|  |  |
| --- | --- |
| **NAME** | **CITY** |
| Jones | Harrison |
| Smith | Rye |
| Hays | Harrison |
| Curry | Rye |
| Johnson | Brooklyn |
| Brooks | Brooklyn |

**Important Points-**

**Point-01:** The degree of output relation (number of columns present) is equal to the number of attributes mentioned in the attribute list.

**Point-02:** Projection operator automatically removes all the duplicates while projecting the output relation. So, cardinality of the original relation and output relation may or may not be same. If there are no duplicates in the original relation, then the cardinality will remain same otherwise it will surely reduce.

**Point-03:** Projection operator does not obey commutative property i.e.

π <list2> (π <list1>(R)) ≠ π <list1> (π <list2>(R))

list 1=name

list 2=name, city

R=name, city, id

**Point-04:** Following expressions are equivalent because both finally projects columns of list-

π <list1> (π <list2>(R)) = π <list1>(R)

**Point-05:** [**Selection Operator**](https://www.gatevidyalay.com/selection-operator-relational-algebra-dbms/) performs horizontal partitioning of the relation. Projection operator performs vertical partitioning of the relation.

**Point-06:** There is only one difference between projection operator of relational algebra and SELECT operation of SQL. Projection operator does not allow duplicates while SELECT operation allows duplicates. To avoid duplicates in SQL, we use “distinct” keyword and write SELECT distinct. Thus, projection operator of relational algebra is equivalent to SELECT operation of SQL.

* **Union Operator (∪)**

Union operator is denoted by ∪ symbol and it is used to select all the rows (tuples) from two tables (relations). It eliminates the duplicate tuples. Union on two relations R1 and R2 can only be computed if R1 and R2 are union compatible (These two relation should have same number of attributes and corresponding attributes in two relations have same domain).

A union operation must hold the following condition:

* R and S must have the attribute of the same number.
* Duplicate tuples are eliminated automatically.
* Union operation is both commutative and associative.

Let’s say we have two relations R1 and R2 both have same columns and we want to select all the tuples(rows) from these relations then we can apply the union operator on these relations.

**Note:** The rows (tuples) that are present in both the tables will only appear once in the union set. In short you can say that there are no duplicates present after the union operation.

**Syntax of Union Operator (∪)**

**table\_name1 ∪ table\_name2**

**Union Operator (∪) Example**

Table 1: COURSE

Course\_Id Student\_Name Student\_Id

--------- ------------ ----------

C101 Aditya S901

C104 Aditya S901

C106 Steve S911

C109 Paul S921

C115 Lucy S931

Table 2: STUDENT

Student\_Id Student\_Name Student\_Age

------------ ---------- -----------

S901 Aditya 19

S911 Steve 18

S921 Paul 19

S931 Lucy 17

S941 Carl 16

S951 Rick 18

**Query:**

∏ Student\_Name (COURSE) ∪ ∏ Student\_Name (STUDENT)

**Output:**

Student\_Name

------------

Aditya

Carl

Paul

Lucy

Rick

Steve

**Note:** As you can see there are no duplicate names present in the output even though we had few common names in both the tables, also in the COURSE table we had the duplicate name itself.

**Example:** DEPOSITOR RELATION

|  |  |
| --- | --- |
| **CUSTOMER\_NAME** | **ACCOUNT\_NO** |
| Johnson | A-101 |
| Smith | A-121 |
| Mayes | A-321 |
| Turner | A-176 |
| Johnson | A-273 |
| Jones | A-472 |
| Lindsay | A-284 |

BORROW RELATION

|  |  |
| --- | --- |
| **CUSTOMER\_NAME** | **LOAN\_NO** |
| Jones | L-17 |
| Smith | L-23 |
| Hayes | L-15 |
| Jackson | L-14 |
| Curry | L-93 |
| Smith | L-11 |
| Williams | L-17 |

**Input:**

∏ CUSTOMER\_NAME (BORROW) ∪ ∏ CUSTOMER\_NAME (DEPOSITOR)

**Output:**

|  |
| --- |
| **CUSTOMER\_NAME** |
| Johnson |
| Smith |
| Hayes |
| Turner |
| Jones |
| Lindsay |
| Jackson |
| Curry |
| Williams |
| Mayes |

* **Intersection Operator (∩)**

Intersection operator is denoted by ∩ symbol and it is used to select common rows (tuples) from two tables (relations).

* In R ∩ S, duplicates are automatically removed.
* Intersection operation is both commutative and associative.

Let’s say we have two relations R1 and R2 both have same columns and we want to select all those tuples(rows) that are present in both the relations, then in that case we can apply intersection operation on these two relations R1 ∩ R2.

**Note:** Only those rows that are present in both the tables will appear in the result set.

**Syntax of Intersection Operator (∩)**

**table\_name1 ∩ table\_name2**

**Intersection Operator (∩) Example**

Table 1: COURSE

Course\_Id Student\_Name Student\_Id

--------- ------------ ----------

C101 Aditya S901

C104 Aditya S901

C106 Steve S911

C109 Paul S921

C115 Lucy S931

Table 2: STUDENT

Student\_Id Student\_Name Student\_Age

------------ ---------- -----------

S901 Aditya 19

S911 Steve 18

S921 Paul 19

S931 Lucy 17

S941 Carl 16

S951 Rick 18

**Query:**

∏ Student\_Name (COURSE) ∩ ∏ Student\_Name (STUDENT)

**Output:**

Student\_Name

------------

Aditya

Steve

Paul

Lucy

**Example:** Using the above DEPOSITOR table and BORROW table

**Input:**

∏ CUSTOMER\_NAME (BORROW) ∩ ∏ CUSTOMER\_NAME (DEPOSITOR)

**Output:**

|  |
| --- |
| **CUSTOMER\_NAME** |
| Smith |
| Jones |

* **Set Difference (-)**

Set Difference is denoted by – symbol. Let’s say we have two relations R1 and R2 and we want to select all those tuples(rows) that are present in Relation R1 but not present in Relation R2, this can be done using Set difference R1 – R2. Minus on two relations R1 and R2 can only be computed if R1 and R2 are union compatible.

* In R – S, duplicates are automatically removed.
* Difference operation is associative but not commutative.

**Syntax of Set Difference (-)**

**table\_name1 - table\_name2**

**Set Difference (-) Example**

Let’s take the same tables COURSE and STUDENT that we have seen above.

**Query:**  
Lets write a query to select those student names that are present in STUDENT table but not present in COURSE table.

∏ Student\_Name (STUDENT) - ∏ Student\_Name (COURSE)

**Output:**

Student\_Name

------------

Carl

Rick

**Example:** Using the above DEPOSITOR table and BORROW table

**Input:**

∏ CUSTOMER\_NAME (BORROW) - ∏ CUSTOMER\_NAME (DEPOSITOR)

**Output:**

|  |
| --- |
| **CUSTOMER\_NAME** |
| Jackson |
| Hayes |
| Willians |
| Curry |

* **Cartesian product (X)**

Cartesian Product is denoted by X symbol. Let’s say we have two relations R1 and R2 then the Cartesian product of these two relations (R1 X R2) would combine each tuple of first relation R1 with each tuple of second relation R2. It is also called Cross Product or Cross Join.

**Syntax of Cartesian product (X)**

**R1 X R2**

**Cartesian product (X) Example**

Table 1: R

Col\_A Col\_B

----- ------

AA 100

BB 200

CC 300

Table 2: S

Col\_X Col\_Y

----- -----

XX 99

YY 11

ZZ 101

**Query:**  
Let’s find the Cartesian product of table R and S.

R X S

**Output:**

Col\_A Col\_B Col\_X Col\_Y

----- ------ ------ ------

AA 100 XX 99

AA 100 YY 11

AA 100 ZZ 101

BB 200 XX 99

BB 200 YY 11

BB 200 ZZ 101

CC 300 XX 99

CC 300 YY 11

CC 300 ZZ 101

**Note:** The number of rows in the output will always be the cross product of number of rows in each table. In our example table 1 has 3 rows and table 2 has 3 rows so the output has 3×3 = 9 rows.

**Example: EMPLOYEE**

|  |  |  |
| --- | --- | --- |
| **EMP\_ID** | **EMP\_NAME** | **EMP\_DEPT** |
| 1 | Smith | A |
| 2 | Harry | C |
| 3 | John | B |

**DEPARTMENT**

|  |  |
| --- | --- |
| **DEPT\_NO** | **DEPT\_NAME** |
| A | Marketing |
| B | Sales |
| C | Legal |

**Input:**

EMPLOYEE X DEPARTMENT

**Output:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **EMP\_ID** | **EMP\_NAME** | **EMP\_DEPT** | **DEPT\_NO** | **DEPT\_NAME** |
| 1 | Smith | A | A | Marketing |
| 1 | Smith | A | B | Sales |
| 1 | Smith | A | C | Legal |
| 2 | Harry | C | A | Marketing |
| 2 | Harry | C | B | Sales |
| 2 | Harry | C | C | Legal |
| 3 | John | B | A | Marketing |
| 3 | John | B | B | Sales |
| 3 | John | B | C | Legal |

* **Rename (ρ)**

Rename (ρ) operation can be used to rename a relation or an attribute of a relation. It is denoted by rho (ρ).

**Rename (ρ) Syntax:  
 ρ(new\_relation\_name, old\_relation\_name)**

**Rename (ρ) Example**

Let’s say we have a table customer; we are fetching customer names and we are renaming the resulted relation to CUST\_NAMES.

Table: CUSTOMER

Customer\_Id Customer\_Name Customer\_City

----------- ------------- -------------

C10100 Steve Agra

C10111 Raghu Agra

C10115 Chaitanya Noida

C10117 Ajeet Delhi

C10118 Carl Delhi

**Query:**

ρ(CUST\_NAMES, ∏(Customer\_Name)(CUSTOMER))

**Output:**

CUST\_NAMES

----------

Steve

Raghu

Chaitanya

Ajeet

Carl

**Example:**

If you want to create a relation STUDENT\_NAMES with ROLL\_NO and NAME from STUDENT, it can be done using rename operator as:

**ρ(STUDENT\_NAMES, ∏(ROLL\_NO, NAME)(STUDENT))**

* **Division Operator (÷)**

**Division operator A÷B can be applied if and only if:**

* **Attributes of B is proper subset of Attributes of A.**
* **The relation returned by division operator will have attributes = (All attributes of A – All Attributes of B)**
* **The relation returned by division operator will return those tuples from relation A which are associated to every B’s tuple.**

**Example:**

**Consider the relation STUDENT\_SPORTS and ALL\_SPORTS given in Table 2 and Table 3**

**Table 2- STUDENT\_SPORTS**

**ROLL\_NO SPORTS**

**1 Badminton**

**2 Cricket**

**2 Badminton**

**4 Badminton**

**Table 3- ALL\_SPORTS**

**SPORTS**

**Badminton**

**Cricket**

**To apply division operator as**

**STUDENT\_SPORTS÷ ALL\_SPORTS**

* **The operation is valid as attributes in ALL\_SPORTS is a proper subset of attributes in STUDENT\_SPORTS.**
* **The attributes in resulting relation will have attributes {ROLL\_NO, SPORTS}-{SPORTS}=ROLL\_NO**
* **The tuples in resulting relation will have those ROLL\_NO which are associated with all B’s tuple {Badminton, Cricket}. ROLL\_NO 1 and 4 are associated to Badminton only. ROLL\_NO 2 is associated to all tuples of B. So the resulting relation will be:**

**ROLL\_NO**

**2**

**Example:**

