UNIT-2

UNIT-2

Relational Algebra & Calculus

Preliminaries

A query language is a language in which user requests to retrieve some information from the database. The query languages are considered as higher level languages than programming languages. Query languages are of two types,

Procedural Language
Non-Procedural Language

1. In procedural language, the user has to describe the specific procedure to retrieve the information from the database.

Example: The Relational Algebra is a procedural language.

2. In non-procedural language, the user retrieves the information from the database without describing the specific procedure to retrieve it.

Example: The Tuple Relational Calculus and the Domain Relational Calculus are non-procedural languages.

Relational Algebra

The relational algebra is a procedural query language. It consists of a set of operations that take one or two relations (tables) as input and produce a new relation, on the request of the user to retrieve the specific information, as the output.

The relational algebra contains the following operations,

1) Selection 2) Projection 3) Union 4) Rename

5) Set-Difference 6) Cartesian product 7) Intersection 8) Join

9) Divide 10) Assignment

The Selection, Projection and Rename operations are called unary operations because they operate only on one relation. The other operations operate on pairs of relations and are therefore called binary operations

1) The Selection (□□) operation:

The Selection is a relational algebra operation that uses a condition to select rows from a relation. A new relation (output) is created from another existing relation by selecting only rows requested by the user that satisfy a specified condition. The lower greek letter 'sigma □' is used to denote selection operation.

General Syntax: **Selection condition (relation_name)**

Example: Find the customer details who are living in Hyderabad city from customer relation.

□□city = 'Hyderabad' (customer)

The selection operation uses the column names in specifying the selection condition. Selection conditions are same as the conditions used in the 'if' statement of any programming languages, selection condition uses the relational operators < > <= >= != . It is possible to combine several conditions into a large condition using the logical connectives 'and' represented by '□' and 'or' represented by '□'.

Example:

Find the customer details who are living in Hyderabad city and whose customer_id is greater than 1000 in Customer relation.

□□city = 'Hyderabad' □□□customer_id > 1000 (customer)

2) The Projection (□□) operation:

The projection is a relational algebra operation that creates a new relation by deleting columns from an existing relation i.e., a new relation (output) is created from another existing relation by selecting only those columns requested by the user from projection and is denoted by letter pi (□□.

The Selection operation eliminates unwanted rows whereas the projection operation eliminates unwanted columns. The projection operation extracts specified columns from a table.

Example: Find the customer names (not all customer details) who are living in Hyderabad city from customer relation.

□ customer_name (□□city = 'Hyderabad' (customer))

In the above example, the selection operation is performed first. Next, the projection of the resulting relation on the customer_name column is carried out. Thus, instead of all customer details of customers living in Hyderabad city, we can display only the customer names of customers living in Hyderabad city.

The above example is also known as relational algebra expression because we are combining two or more relational algebra operations (ie., selection and projection) into one at the same time.

Example: Find the customer names (not all customer details) from customer relation.

□ **customer_name (customer)**

The above stated query lists all customer names in the customer relation and this is not called as relational algebra expression because it is performing only one relational algebra operation.

3) The Set Operations: (Union, Intersection, Set-Difference, Cartesian product **)**

i) Union ' □ ' Operation:

The union denoted by '□'□ It is a relational algebra operation that creates a union or combination of two relations. The result of this operation, denoted by d □□b is a relation that includes all tuples that all either in d or in b or in both d and b, where duplicate tuples are eliminated.

Example: Find the customer_id of all customers in the bank who have either an account or a loan or both.

□ **customer_id (depositor)** □□□□□ **customer_id (borrower)**

To solve the above query, first find the customers with an account in the bank. That is □ **customer_id (depositor)**. Then, we have to find all customers with a loan in the bank, □ **customer_id (borrower)**. Now, to answer the above query, we need the union of these two sets, that is, all customer names that appear in either or both of the two relations by □ customer_id (depositor) □□□□□ customer_id (borrower)

If some customers A, B and C are both depositors as well as borrowers, then in the resulting relation, their customer ids will occur only once because duplicate values are eliminated.

Therefore, for a union operation d □□b to be valid, we require that two conditions to be satisfied,

i) The relations depositor and borrower must have same number of attributes / columns.

ii) The domains of i^{th} attribute of depositor relation and the i^{th} attribute of borrower relation must be the same, for all i.

- **The Intersection ' □□' Operation:**

The intersection operation denoted by ' □'□ It is a relational algebra operation that finds tuples that are in both relations. The result of this operation, denoted by d □□b, is a relation that includes all tuples common in both depositor and borrower relations.

Example: Find the customer_id of all customers in the bank who have both an account and a loan.

□ **customer_id (depositor)** □□□□□ **customer_id (borrower)**

The resulting relation of this query, lists all common customer ids of customers who have both an account and a loan. Therefore, for an intersection operation d □□b to be valid, it requires that two conditions to be satisfied as was the case of union operation stated above.

iii) The Set-Difference ' □ ' Operation:

The set-difference operation denoted by' □'□ It is a relational algebra operation that finds tuples that are in one relation but are not in another.

Example:

□ **customer_id (depositor)** □□□□□ **customer_id (borrower)**

The resulting relation for this query, lists the customer ids of all customers who have an account but not a loan. Therefore a difference operation d □□b to be valid, it requires that two conditions to be satisfied as was case of union operation stated ablove.

iv) The Cross-product (or) Cartesian Product ‘ X ’ Operation:

The Cartesian-product operation denoted by a cross ‘X’ It is a relational algebra operation which allows to combine information from who relations into one relation.

Assume that there are n1 tuple in borrower relation and n2 tuples in loan relation. Then, the result of this operation, denoted by r = borrower X loan, is a relation ‘r’ that includes all the tuples formed by each possible pair of tuples one from the borrower relation and one from the loan relation. Thus, ‘r’ is a large relation containing n1 * n2 tuples.

The drawback of the Cartesian-product is that same attribute name will repeat.

Example: Find the customer_id of all customers in the bank who have loan > 10,000.

customer_id (borrower.loan_no= loan.loan_no ((borrower.loan_no= (borrower X loan)))

That is, get customer_id from borrower relation and loan_amount from loan relation. First, find Cartesian product of borrower X loan, so that the new relation contains both customer_id, loan_amoount with each combination. Now, select the amount, by **bloan_ampunt >** 10000.

So, if any customer have taken the loan, then borrower.loan_no = loan.loan_no should be selected as their entries of loan_no matches in both relation.

4) The Renaming “ ” Operation:

The Rename operation is denoted by rho ’’. It is a relational algebra operation which is used to give the new names to the relation algebra expression. Thus, we can apply the rename operation to a relation ‘borrower’ to get the same relation under a new name. Given a relation ‘customer’, then the expression returns the same relation ‘customer’ under a new name ‘x’.

x (customer)

After performed this operation, Now there are two relations, one with customer name and second with

‘x’ name. The ‘rename’ operation is useful when we want to compare the values among same column attribute in a relation.

Example: Find the largest account balance in the bank.

□ **account.balance (□□account.balance > d.balance (account X □d (account)))**

If we want to find the largest account balance in the bank, Then we have to compare the values among same column (balance) with each other in a same relation account, which is not possible.

So, we rename the relation with a new name'd'. Now, we have two relations of account, one with account name and second with 'd' name. Now we can compare the balance attribute values with each other in separate relations.

5) The Joins " ⋈□" Operation:

The join operation, denoted by join ' ⋈□'. It is a relational algebra operation, which is used to combine

(join) two relations like Cartesian-product but finally removes duplicate attributes and makes the operations (selection, projection, ..) very simple. In simple words, we can say that join connects relations on columns containing comparable information.

There are three types of joins,

i) Natural Join
ii) Outer Join
iii) Theta Join (or) Conditional Join

i) Natural Join:

The natural join is a binary operation that allows us to combine two different relations into one relation and makes the same column in two different relations into only one-column in the resulting relation. Suppose we have relations with following schemas, which contain data on full-time employees.

employee (emp_name, street, city) and

employee_works(emp_name, branch_name, salary)

The relations are,

emp_name	street	city
Coyote	Town	Hollywood
Rabbit	Tunnel	Carrotville
Smith	Revolver	Vally
Williams	Seaview	Seattle

employee relation

emp_name	branch_name	salary
Coyote	Mesa	15000
Rabbit	Mesa	12000
Gates	Redmond	25000
Williams	Redmond	23000

employee_works relation

If we want to generate a single relation with all the information (emp_name, street, city, branch_name and salary) about full-time employees. then, a possible approach would be to use the natural-join operation as follows,

employee ⋈ employee_works

The result of this expression is the relation,

emp_name	street	city	branch_name	salary
Coyote	Town	Hollywood	Mesa	15000
Rabbit	Tunnel	Carrotville	Mesa	12000
Williams	Seaview	Seattle	Redmond	23000

result of Natural join

We have lost street and city information about Smith, since tuples describing smith is absent in employee_works. Similarly, we have lost branch_name and salary information about Gates, since the tuple describing Gates is absent from the employee relation. Now, we can easily perform select or reject query on new join relation.

Example: Find the employee names and city who have salary details.

□ **emp_name, salary, city (employee ⋈ employee_works)**

The join operation selects all employees with salary details, from where we can easily project the employee names, cities and salaries. Natural Join operation results in some loss of information.

ii) Outer Join:

The drawback of natural join operation is some loss of information. To overcome the drawback of natural join, we use outer-join operation. The outer-join operation is of three types,

a) Left outer-join (⋈)
b) Right outer-join (⋈)
c) Full outer-join (⋈)

a) Left Outer-join:

The left outer-join takes all tuples in left relation that did not match with any tuples in right relation, adds the tuples with null values for all other columns from right relation and adds them to the result of natural join as follows,

The relations are,

emp_name	street	city
Coyote	Town	Hollywood
Rabbit	Tunnel	Carrotville
Smith	Revolver	Valley
Williams	Seaview	Seattle

employee relation

emp_name	branch_name	salary
Coyote	Mesa	15000
Rabbit	Mesa	12000
Gates	Redmond	25000
Williams	Redmond	23000

employee_works relation

The result of this expression is the relation,

emp_name	street	city	branch_name	salary
Coyote	Town	Hollywood	Mesa	15000
Rabbit	Tunnel	Carrotville	Mesa	12000
Smith	Revolver	Valley	null	null
Williams	Seaview	Seattle	Redmond	23000

result of Left Outer-join

b) Right Outer-join:

The right outer-join takes all tuples in right relation that did not match with any tuples in left relation, adds the tuples with null values for all other columns from left relation and adds them to the result of natural join as follows,

The relations are,

emp_name	street	city
Coyote	Town	Hollywood
Rabbit	Tunnel	Carrotville
Smith	Revolver	Valley
Williams	Seaview	Seattle

emp_name	branch_name	salary
Coyote	Mesa	15000
Rabbit	Mesa	12000
Gates	Redmond	25000
Williams	Redmond	23000

The result of this expression is the relation,

emp_name	street	city	branch_name	salary
Coyote	Town	Hollywood	Mesa	15000
Rabbit	Tunnel	Carrotville	Mesa	12000
Gates	null	null	Redmond	25000
Williams	Seaview	Seattle	Redmond	23000

result of Right Outer-join

c) Full Outer-join:

The full outer-join operation does both of those operations, by adding tuples from left relation that did not match any tuples from the reight relations, as well as adds tuples from the right relation that did not match any tuple from the left relation and adding them to the result of natural join as follows,

The relations are,

emp_name	street	city
Coyote	Town	Hollywood
Rabbit	Tunnel	Carrotville
Smith	Revolver	Valley
Williams	Seaview	Seattle

employee relation

emp_name	branch_name	salary
Coyote	Mesa	15000
Rabbit	Mesa	12000
Gates	Redmond	25000
Williams	Redmond	23000

employee_works relation

The result of this expression is the relation,

emp_name	street	city	branch_name	salary
Coyote	Town	Hollywood	Mesa	15000
Rabbit	Tunnel	Carrotville	Mesa	12000
Smith	Revolver	Valley	null	null
Gates	null	null	Redmond	25000
Williams	Seaview	Seattle	Redmond	23000

result of Full Outer-join

iii) Theta Join (or) Condition join:

The theta join operation, denoted by symbol " ⋈□□□" . It is an extension to the natural join operation that□combines two relations into one relation with a selection condition (□).

The theta join operation is expressed as employee ⋈ salary < 19000 employee_works and the resulting is as follows,

employee ⋈ salary > 20000 employee_works

There are two tuples selected because their salary greater than 20000 (salary > 20000). The result of theta join as follows,

The relations are,

emp_name	street	city
Coyote	Town	Hollywood
Rabbit	Tunnel	Carrotville
Smith	Revolver	Valley
Williams	Seaview	Seattle

emp_name	branch_name	salary
Coyote	Mesa	15000
Rabbit	Mesa	12000
Gates	Redmond	25000
Williams	Redmond	23000

The result of this expression is the relation,

emp_name	street	city	branch_name	salary
Gates	null	null	Redmond	25000
Williams	Seaview	Seattle	Redmond	23000

result of Theta Join (or) Condition Join

6) The Division " □□" Operation:

The division operation, denoted by **" □□"**, is a relational algebra operation that creates a new relation by selecting the rows in one relation that does not match rows in another relation.

Let, Relation A is (x1, x2, …., xn, y1, y2, …, ym) and
Relation B is (y1, y2, …, ym),

Where, y1, y2, …, ym tuples are common to the both relations A and B with same domain compulsory.

Then, A □□B = new relation with x1, x2, …., xn tuples. Relation A and B represents the dividend and devisor respectively. A tuple 't' is in a □□b, if and only if two conditions are to be satisfied,

□ t is in □A-B (r)

□ for every tuple tb in B, there is a tuple ta in A satisfying the following two things,

1. ta[B] = tb[B]

2. ta[A-B] = t

Relational Calculus

Relational calculus is an alternative to relational algebra. In contrast to the algebra, which is procedural, the relational calculus is non-procedural or declarative.

It allows user to describe the set of answers without showing procedure about how they should be computed. Relational calculus has a big influence on the design of commercial query languages such as SQL and QBE (Query-by Example).

Relational calculus are of two types,

- Tuple Relational Calculus (TRC)
- Domain Relational Calculus (DRC)

Variables in TRC takes tuples (rows) as values and TRC had strong influence on SQL.

Variables in DRC takes fields (attributes) as values and DRC had strong influence on QBE.

i) Tuple Relational Calculus (TRC):

The tuple relational calculus, is a non-procedural query language because it gives the desired information without showing procedure about how they should be computed.

A query in Tuple Relational Calculus (TRC) is expressed as **{ T | p(T) }**

Where, T - tuple variable,
P(T) - 'p' is a condition or formula that is true for 't'.

In addition to that we use,

T[A] - to denote the value of tuple t on attribute A and

T □□r - to denote that tuple t is in relation r.

Examples:

1) Find all loan details in loan relation.

{ t | t □□loan }

This query gives all loan details such as loan_no, loan_date, loan_amt for all loan table in a bank.

2) Find all loan details for loan amount over 100000 in loan relation.

{ t | t □□loan □□t [loan_amt] > 100000 }

This query gives all loan details such as loan_no, loan_date, loan_amt for all loan over 100000 in a loan table in a bank.

ii) Domain Relational Calculus (DRC):

A Duple Relational Calculus (DRC) is a variable that comes in the range of the values of domain (data types) of some columns (attributes).

A Domain Relational Calculus query has the form,

{ < x1, x2, …., xn > | p(< x1, x2, …., xn >) }

Where, each xi is either a domain variable or a constant and p(< x1, x2, …., xn >) denotes a DRC formula.

A DRC formula is defined in a manner that is very similar to the definition of a TRC formula. The main difference is that the variables are domain variables.

Examples:

1) Find all loan details in loan relation.

{ < N, D, A > | < N, D, A > □□loan }

This query gives all loan details such as loan_no, loan_date, loan_amt for all loan table in a bank. Each column is represented with an initials such as N- loan_no, D – loan_date, A – loan_amt. The condition < N, D, A > □□loan ensures that the domain variables N, D, A are restricted to the column domain.

2.3.1 Expressive power of Algebra and Calculus

The tuple relational calculus restricts to safe expressions and is equal in expressive power to relational algebra. Thus, for every relational algebra expression, there is an equivalent expression in the tuple relational calculus and for tuple relational calculus expression, there is an equivalent relational algebra expression.

A safe TRC formula Q is a formula such that,

□ For any given I, the set of answers for Q contains only values that are in dom(Q, I).

□ For each sub expression of the form □R(p(R)) in Q, if a tuple r makes the formula true, then r contains only constraints in dom(Q, I).

3) For each sub expression of the form□ □R(p(R)) in Q, if a tuple r contains a constant that is not in dom(Q, I), then r must make the formula true.

The expressive power of relational algebra is often used as a metric how powerful a relational database query language is. If a query language can express all the queries that we can express in relational algebra, it is said to be relationally complete. A practical query language is expected to be relationally complete. In addition, commercial query languages typically support features that allow us to express some queries that cannot be expressed in relational algebra.

When the domain relational calculus is restricted to safe expression, it is equivalent in expressive power to the tuple relational calculus restricted to safe expressions. All three of the following are equivalent,

- The relational algebra
- The tuple relational calculus restricted to safe expression
- The domain relational calculus restricted to safe expression