

Introduction

Relational algebra and relational calculus are the mathematical basis for relational databases developed by E.F. Codd.

- Relational algebra and relational calculus are formal query languages associated with the relational model.
- Informally, relational algebra is a (high-level) **procedural** language (you have to state, step by step, exactly how the result should be calculated and relational calculus a **non-procedural** language.
- However, formally both are equivalent to one another.
- **In RA and RC the duplicate elimination is always done after each operation, so that relations are always set of tuples. In practice, real system often omit the expensive step of eliminating duplicate tuples, leading to relations that are multisets.**

Relational Algebra

- **Algebra** is concerned with operations on **sets of numbers** or **symbols**. Algebra is a generalization of arithmetic. Example : **Arithmetic**: $3 + 4 = 3 + 4$ in **Algebra** it would be: $x + y = y + x$
- A basic expression of RA consists of either a) **A relation in the database** or b) **constant relation** and **relational operators**. General expressions are formed out of smaller sub-expressions.
- Both **operands** and **results** in an expression are **relations**, so output from one operation can become input to another operation. Allows expressions to be nested, just as in arithmetic. Relational Operations work on one or more relations to define another relation **without changing the original relations**.
- Not used now a days as a query language in actual DBMSs. (SQL instead.)
- The inner, lower-level operations of a relational DBMS are relational algebra operations. We need to know about relational algebra to understand **query execution and optimization** in a relational DBMS.

Why teach RA and RC ?

The question might arise: Do we need to teach RA and RC in a database course? We often hear the statement: "Students only need to learn SQL." Apparently, many database educators do not feel that RA and RC are essential topics in their courses. Teaching SQL is a major part of a database course, but coverage of RA and RC is also important. There are several reasons for this:

1. RA provides the relational model with a flexible way to query a database. Knowledge of RA facilitates learning and using SQL as a query language.

2. The query processor component of the DBMS engine translates SQL code into a query plan that includes RA operations. Anyone who writes DBMS query processing software or needs to optimize SQL query execution will benefit from an understanding of RA.
3. RC is the foundation for Query-By-Example, which provides a user-friendly interface for databases. Predicate calculus (and RC) can be used to develop an intelligent front-end for a database (e.g. an expert system).
4. Learning RA and RC makes a student aware of conceptual and practical differences between procedural and nonprocedural query languages.

Comparing RA and SQL

Relational algebra	SQL
<ul style="list-style-type: none"> is closed (the result of every expression is a relation) has a rigorous foundation has simple semantics is used for reasoning, query optimization, etc. is relationally complete (A relationally complete language can perform all basic, meaningful operations on relations). 	<ul style="list-style-type: none"> is a superset of relational algebra has convenient formatting features, etc. provides aggregate functions has complicated semantics is an end-user language. Is relationally complete.






Operations of RA

- The fundamental operations in the RA are select, project, union, set difference, Cartesian product, and rename.
- In addition to the fundamental operations, there are several other operations—namely, set intersection, natural join, division, and assignment. We will define these operations in terms of the fundamental operations.

Notation of Relational Operator

The operations have their own symbols in set theory. The symbols are hard to remember so we will use meaningful word for each symbol e.g PROJECT for projection etc here. The real symbols:

Operation	Op-code used	Symbol		Operation	Op-code used	Symbol
Project	PROJECT	π (Pi)		Cartesian product	X	\times (times)
Select (Restrict)	SELECT	σ (Sigma)		Join	JOIN	\bowtie (bow-tie)

Rename	RENAME	ρ (Rho)		Left outer join	LEFT OUTER JOIN	
Union	UNION	\cup (cup)		Right outer join	RIGHT OUTER JOIN	
Intersect	INTERSECTION	\cap (cap)		Full outer join	FULL OUTER JOIN	
Assign	<-			Semijoin	SEMIJOIN	

Example: The relational algebra expression can be written as

PROJECT Namn (SELECT Medlemsnummer < 3 (Medlem)) Actually it should be written as below :

$\pi_{\text{Namn}} (\sigma_{\text{Medlemsnummer} < 3} (\text{Medlem}))$

Equivalences

The same relational algebraic expression can be written in many different ways. The order in which tuples appear in relations is never significant.

- $A \times B \equiv B \times A$
- $A \cup B \equiv B \cup A$
- $A \cap B \equiv B \cap A$
- $(A - B)$ is not the same as $(B - A)$
- $\sigma_{c1} (\sigma_{c2}(A)) \equiv \sigma_{c2} (\sigma_{c1}(A)) \equiv \sigma_{c1 \wedge c2}(A)$
- $\pi_{a1}(A) \equiv \pi_{a1}(\pi_{a1, \text{etc}}(A))$ where etc represents any other attributes of A.

Many other equivalences exist.

While equivalent expressions always give the same result, some may be much easier to evaluate than others. When any query is submitted to the DBMS, its query optimiser tries to find the most efficient equivalent expression before evaluating it.

We will discuss some important relational (set) operation :

- **Project**

Example: The table E (for EMPLOYEE)

nr	name	salary
1	John	100
5	Sarah	300
7	Tom	100

SQL	Result	Relational algebra	
select salary from E	salary	PROJECT _{salary} (E)	
	100		
	300		
select nr, salary from E	nr	PROJECT _{nr, salary} (E)	
	salary		
	1		100
	5		300
	7	100	

Note that there are no duplicate rows in the result.

Project copies its argument relation for the specified attributes only. Since a relation is a set, duplicate rows are eliminated. Projection is denoted by the Greek capital letter π . The attributes to be copied appear as subscripts.

For example, to obtain a relation showing customers and branches, but ignoring amount and loan#, we write

$\pi_{\text{bname, cname}}(\text{borrow})$

We can perform these operations on the relations resulting from other operations. To get the names of customers having the same name as their bankers,

$\pi_{\text{cname}}(\sigma_{\text{cname=banker}}(\text{client}))$

Think of **select** as taking rows of a relation, and **project** as taking columns of a relation.

- **Select (Restrict)**

The same table E (for EMPLOYEE) as above.

SQL	Result	Relational algebra									
select * from E where salary < 200	<table> <tr><th>nr</th><th>name</th><th>salary</th></tr> <tr><td>1</td><td>John</td><td>100</td></tr> <tr><td>7</td><td>Tom</td><td>100</td></tr> </table>	nr	name	salary	1	John	100	7	Tom	100	$\text{SELECT}_{\text{salary} < 200}(E)$
nr	name	salary									
1	John	100									
7	Tom	100									
select * from E where salary < 200 and nr >= 7	<table> <tr><th>nr</th><th>name</th><th>salary</th></tr> <tr><td>7</td><td>Tom</td><td>100</td></tr> </table>	nr	name	salary	7	Tom	100	$\text{SELECT}_{\text{salary} < 200 \text{ and nr} \geq 7}(E)$			
nr	name	salary									
7	Tom	100									

Note that the **select operation in relational algebra** has nothing to do with the **SQL keyword select**. Selection in relational algebra returns those tuples in a relation that fulfil a condition, while the SQL keyword select means "here comes an SQL statement".

Select selects tuples that satisfy a given **predicate** (condition). Select is denoted by a lowercase Greek sigma (σ), with the **predicate appearing as a subscript**. The argument relation is given in parentheses following the σ . For example, to select tuples (rows) of the borrow relation where the branch is "SFU", we would write

$\sigma_{\text{bname} = \text{"SFU"}}(\text{borrow})$

We allow comparisons using $=, \neq, <, \leq, >$ and \geq in the selection predicate.

We also allow the logical connectives \vee (or) and \wedge (and). For example:

$\sigma_{\text{bname} = \text{"Downtown"} \wedge \text{amount} > 1200}(\text{borrow})$

• Union

The union operation is **denoted \cup as in set theory**. It returns the union (set union) of two **compatible relations**. For a union operation $r \cup s$ to be legal, we require that r and s **must have the same number of attributes**. The **domains** of the corresponding attributes must be the **same**.

To find all customers of the SFU branch, we must find everyone who has a loan or an account or both at the branch. We need both borrow and deposit relations for this:

$\pi_{\text{cname}}(\sigma_{\text{bname} = \text{"SFU"}}(\text{borrow})) \cup \pi_{\text{cname}}(\sigma_{\text{bname} = \text{"SFU"}}(\text{deposit}))$

As in all set operations, duplicates are eliminated.

Note that field names are not used in defining union compatibility. For convenience we will assume that the **fields of $R \cup S$ inherit names from R** , if the fields of R have names.

Outer union : Outer union can be used to calculate the union of two relations that are **partially union compatible**. Not very common.

Example: The table R	Example: The table S	The result of an outer union between R and S:																								
<table><tr><td>A</td><td>B</td></tr><tr><td>1</td><td>2</td></tr><tr><td>3</td><td>4</td></tr></table>	A	B	1	2	3	4	<table><tr><td>B</td><td>C</td></tr><tr><td>4</td><td>5</td></tr><tr><td>6</td><td>7</td></tr></table>	B	C	4	5	6	7	<table><tr><td>A</td><td>B</td><td>C</td></tr><tr><td>1</td><td>2</td><td>null</td></tr><tr><td>3</td><td>4</td><td>5</td></tr><tr><td>null</td><td>6</td><td>7</td></tr></table>	A	B	C	1	2	null	3	4	5	null	6	7
A	B																									
1	2																									
3	4																									
B	C																									
4	5																									
6	7																									
A	B	C																								
1	2	null																								
3	4	5																								
null	6	7																								

• Cartesian product

The cartesian product of two tables combines each row in one table with each row in the other table.

This is also a binary set operation, but the relations on which it is applied do *not* have to be **union compatible**. This operation is used to combine tuples from two relations in a **combinatorial** fashion. In general, the result of $R(A_1, A_2, \dots, A_n) \times S(B_1, B_2, \dots, B_m)$ is a relation Q with degree **$n + m$** attributes $Q(A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m)$, in that order. The resulting relation Q has **one tuple for**

each combination of tuples-one from R and one from S. Hence, if R has n_R tuples (denoted as $|R| = n_R$), and S has n_S tuples, then $R \times S$ will have $n_R * n_S$ tuples.

Example: The table E (for EMPLOYEE), D (for DEPARTMENT)

enr	ename	dept
1	Bill	A
2	Sarah	C
3	John	A

dnr	dname
A	Marketing
B	Sales
C	Legal

- Seldom useful in practice.
- Usually an error.
- Can give a huge result.

SQL	Result					Relational algebra
select * from E, D	enr	ename	dept	dnr	dname	E X D
	1	Bill	A	A	Marketing	
	1	Bill	A	B	Sales	
	1	Bill	A	C	Legal	
	2	Sarah	C	A	Marketing	
	2	Sarah	C	B	Sales	
	2	Sarah	C	C	Legal	
	3	John	A	A	Marketing	
	3	John	A	B	Sales	
	3	John	A	C	Legal	

The result of **E X D** is a new relation with a tuple for each possible pairing of tuples from E and D.

The operation applied by itself is generally meaningless. It is useful when followed by a selection that matches values of attributes coming from the component relations.

For example, suppose that we want to retrieve a list of names of each female employee's Dependents. We can do this as follows:

$FEMALE_EMPS \leftarrow \sigma_{SEX='F'}(EMPLOYEE)$ $EMP_NAMES \leftarrow \pi_{FNAME, LNAME, SSN}(FEMALE_EMPS)$
 $EMP_DEPENDENTS \leftarrow EMP_NAMES \times DEPENDENT$

$ACTUAL_DEPENDENTS \leftarrow \sigma_{SSN=ESSN}(EMP_DEPENDENTS)$
 (where ESSN is the SSN of employee in dependent relation)
 $RESULT \leftarrow \pi_{FNAME, LNAME, DEPENDENTNAME}(ACTUAL_DEPENDENTS)$

In order to avoid ambiguity, the name of relation is prefixed with attribute names with separator (.). If no ambiguity will result, we drop the relation name.

• Rename (tables and columns)

We want to join two tables, but:

- Several columns in the result will have the same name (nr and name).
- How do we express the join condition, when there are two columns called nr?

Solutions:

- Rename the attributes, using the rename operator.
- Keep the names, and prefix them with the table name, as is done in SQL.

Unlike relations in the database, the **results of relational-algebra expressions** do not have a **name** that we can use to refer to them. It is useful to be able to give them names; the **rename** operator, denoted by the lowercase Greek letter **ρ** (ρ), lets us do

- Given a relational-algebra expression E , the expression $\rho_x(E)$ or $RENAME_x(E)$ returns the result of **expression E under the name x** .

We can also apply the rename operation to a relation r to get the same relation under a new name. This is necessary when joining a table with itself (see below).

- A second form of the rename operation is as follows. Assume that a relational algebra expression E has n number of attributes. Then, the expression $\rho_{x(A_1, A_2, \dots, A_n)}(E)$ or $RENAME_{x(A_1, A_2, \dots, A_n)}(E)$ returns the **result of expression E under the name x** , and with the attributes renamed to A_1, A_2, \dots, A_n .
- The rename operation solves the problems that occurs with naming when performing the **cartesian product of a relation with itself**. Suppose we want to find the names of all the customers who live on the **same street and in the same city as Smith**.

We can get the street and city of Smith by writing	$\pi_{street, city}(\sigma_{cname="Smith"}(customer))$
To find other customers with the same information, we need to reference the customer relation again :	$\sigma_P(customer \times \pi_{street, city}(\sigma_{cname="Smith"}(customer)))$ where P is a selection predicate requiring street and city values to be equal.

Problem: how do we distinguish between the **two street values** appearing in the Cartesian product, as both come from a customer relation?

Solution: use the rename operator to rename one customer relation. Then ambiguities in referring same fields from two similar relation will disappear.

Customer relation is renamed as CUST2 and then attribute is referred as **CUST2.street**

$$\pi_{customer.cname}(\sigma_{cust2.street = customer.street \wedge cust2.city = customer.city}(customer \times (\pi_{street, city}(\sigma_{cname="Smith"}(\rho_{cust2}(customer))))))$$

• Difference

Set difference is denoted by the minus sign ($-$). It finds tuples that **are in one relation, but not in another**. Thus $r - s$ results in a relation containing tuples that are in r but not in s .

To find customers of the **SFU** branch who have an account there but no loan, we write

$$\pi_{cname}(\sigma_{bname="SFU"}(deposit)) - \pi_{cname}(\sigma_{bname="SFU"}(borrow))$$

SQL for Difference

Exclude rows common to both tables. Which records in TABLE_A do not share A_KEY in TABLE_B?

select * from TABLE_A where A_KEY not in (select A_KEY from TABLE_B)

With the SQL-92 Standards keyword **'EXCEPT'** Follow the same rules as the keyword 'UNION'

select * from TABLE_A
EXCEPT
select * from TABLE_B

also seen as:

select * from TABLE_A **MINUS** select * from TABLE_B

- **Join (sometimes called "inner join")**

The result of cartesian products are useless most of the time. Meaningful information can only be achieved if we **apply the Selection on it**. Example : We want to get a departmental employee list from two relations **employee and department**.

SQL	Result						Relational algebra
select * from E, D where dept = dnr	enr	ename	dept	dnr	dname		SELECT _{dept = dnr} (E X D) <i>or, using the equivalent join operation</i> E JOIN _{dept = dnr} D
	1	Bill	A	A	Marketing		
	2	Sarah	C	C	Legal		
	3	John	A	A	Marketing		

Using Mathematical notation we can write : $\sigma_{dept = dnr}(E \times D) = E \bowtie_{dept = dnr} D$

Above two operations are equivalence as they will produce same result. In left side expression we do the cartesian product followed by select. Join is a very common and useful operation and equivalent to a **cartesian product followed by a select**. Condition can refer to attributes of both the relations.

(sid)	sname	rating	age	(sid)	bid	Day
22	Dustin	7	45.0	58	103	11/12/96
31	Lubber	8	55.5	58	103	11/12/96

This table shows the result of a join $S1 \bowtie_{S1.sid < R1.sid} R1$. As sid appears in both S1 and R1 we use S1.sid for referring the attributes of respective relation.

Equijoin

When join condition consists **solely of equalities** (connected by \wedge) of form $R1.name1 = S1.name2$, that is equalities between two fields of R and S.

Example : $S1 \bowtie_{S1.sid = R1.sid} R1$.

Note that only one field sid appears in the result i.e. duplicate column dropped from result.

sid	sname	rating	age	bid	Day
22	Dustin	7	45.0	101	10/10/96
58	Rusty	10	35.0	103	11/12/96

Natural join

It is an equijoin in which equalities are specified on all fields having the same name in R and S. In this case, we can simply omit the join condition; the default is that the join condition is a collection of equalities on all common fields. This special case is called natural join, and the result will not contain two fields of same name (only one field will appear).

$S1 \bowtie_{S1.sid = R1.sid} R1$ is actually a natural join and can be denoted as $S1 \bowtie R1$

If two relations have no attributes in common, $S1 \bowtie R1$ is simply a cross product.

• Intersection:

The intersection of two sets $R \cap S$ returns a relation containing all tuples that occur in both R and S. The relations R and S must be union-compatible, and the schema of the result is defined to be identical to the schema of R.

SQL for Intersection : What are the records of 'TABLE_A' that share a 'KEY' with records of 'TABLE_B'?

```
select * from TABLE_A where TABLE_A.KEY in (select TABLE_B.KEY from TABLE_B)
```

Or if 'intersect' is implemented in SQL it follows the same rules as 'union' as can be written as follows.

```
select distinct * from depositor intersect select distinct * from borrower
```

• Division

The DIVISION operation, denoted by \div ; is useful for a special kind of query in database applications. An example is "Retrieve the names of employees who work on all the projects that 'John Smith' works on." To express this query using the DIVISION operation, proceed as follows.

- First, retrieve the list of project numbers that 'John Smith' works on in the intermediate relation SMITH_PNOS:

```
SMITH ← σFNAME='JOHN' AND LNAME='SMITH' (EMPLOYEE)
SMITH_PNOS ← πPNO (WORKS_ON ⋈ESSN=SSN SMITH)
```

← Assignment operator

WORKS_ON relation contains projects where employee works

- Next, create a relation that includes a tuple <PNO, ESSN> whenever the employee whose social security number is ESSN works on the project whose number is PNO in the intermediate relation

SSN_PNOS:

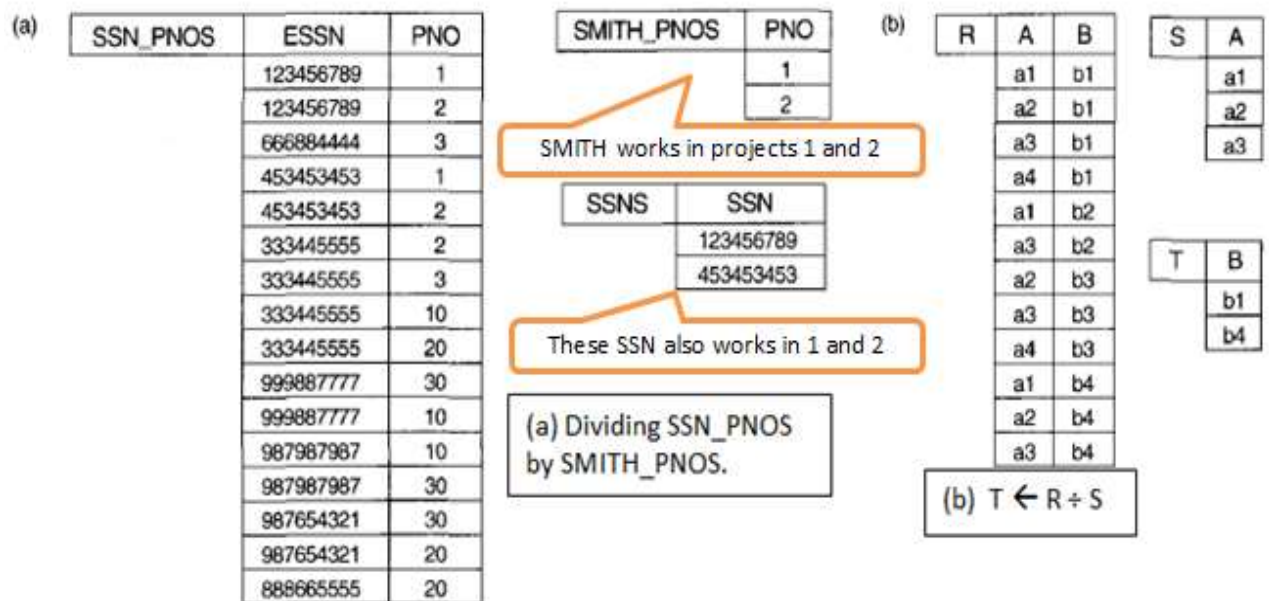
$$\text{SSN_PNOS} \leftarrow \pi_{\text{ESSN,PNO}} (\text{WORKS_ON})$$

- Finally, apply the DIVISION operation to the two relations, which gives the desired employees' social security numbers:

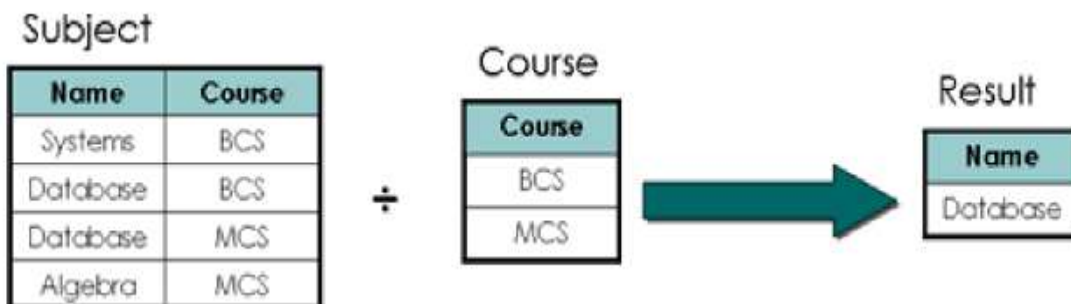
$$\text{SSNS (SSN)} \leftarrow \text{SSN_PNOS} \div \text{SMITH_PNOS}$$

$$\text{RESULT} \leftarrow \pi_{\text{FNAME, LNAME}} (\text{SSNS}, \text{EMPLOYEE})$$

The previous operations are shown in Figure below



Another Example : Retrieve the name of the subjects which are offered in all courses.



Viewed another way: As multiplication is to division in arithmetic, Cartesian Product (X) is to Division in relational algebra

Consider the unary relations M (attribute C) and N (attribute D), and their Cartesian Product O:

M	C	O	C	D	N	D
4		4	3		3	
8		4	1		1	
		4	7		7	
		8	3			
		8	1			
		8	7			

Division is the opposite of Cartesian Product:

$$o \div n = m$$

$$o \div m = n$$

Another Example of Division Operation (R ÷ S)

- Attributes of S must be a subset of the attributes of R
- $\text{attr}(R \div S) = \text{attr}(R) - \text{attr}(S)$
- Used to answer questions involving all
- Which employees work on all the critical projects?

Works

enum	pnum
E35	P10
E45	P15
E35	P12
E52	P15
E52	P17
E45	P10
E35	P15

Critical

pnum
P15
P10

Works ÷ Critical

enum
E45
E35

(Works ÷ Critical) × Critical

enum	pnum
E45	P15
E45	P10
E35	P15
E35	P10

The complete division expression is derived using following three steps :

1. Compute all possible attribute pairings
2. Remove the existing pairings
3. Remove the non-answers from the possible answers

$$A \div B = \bigcap_{i-j} (A) \text{ -- } \bigcap_{i-j} ((\bigcap_{i-j} (A) \times B) - A)$$

3 1 2

Here we ignore the projection (in projection we have to consider the attributes to project in place of i-j)

SQL for division

No built-in division operation in SQL. Standard SQL expressions for division are complex

There are different ways to perform division in SQL. Some of the ways are discussed here

- Direct translation from relational algebra

In SQL

- EXCEPT means Difference (--)
- A join without where clause produces a Cartesian product

$$A \div B = \pi_{i-j}(A) - \pi_{i-j}((\pi_{i-j}(A) \times B) - A)$$

select distinct sno from **SPJ**

except

select sno

from (**select** sno, pno

from (**select** sno from **SPJ**) as t1,

(**select** pno from **P** where weight = 17) as t2

except

select sno, pno from **SPJ**

) as t3;

Relational algebra expressions

SQL	Result	Relational algebra												
select name, salary from E where salary < 200	<table><tr><th>name</th><th>salary</th></tr><tr><td>John</td><td>100</td></tr><tr><td>Tom</td><td>100</td></tr></table>	name	salary	John	100	Tom	100	PROJECT _{name, salary} (SELECT _{salary < 200} (E)) <i>or, step by step, using an intermediate result</i> Temp <- SELECT _{salary < 200} (E) Result <- PROJECT _{name, salary} (Temp)						
name	salary													
John	100													
Tom	100													
<table><tr><th>STOCK</th><th>STKTYPE</th></tr><tr><td>SNo (PK)</td><td>TType (PK)</td></tr><tr><td>SType (FK)</td><td>TName</td></tr><tr><td>SName</td><td>ROP</td></tr><tr><td>QOH</td><td>OSize</td></tr><tr><td>OnOrder</td><td></td></tr></table> <p>For example, suppose a DB consists of two tables, STOCK and STKTYPE as described above.</p>		STOCK	STKTYPE	SNo (PK)	TType (PK)	SType (FK)	TName	SName	ROP	QOH	OSize	OnOrder		<p>Attributes that apply to individual items (e.g. QOH—quantity on hand) are recorded in the STOCK table. Attributes that apply to all items of the same type (e.g. ROP—reorder point) are included in the STKTYPE table.</p> <p>The two tables are linked by a common type code (SType and TType).</p>
STOCK	STKTYPE													
SNo (PK)	TType (PK)													
SType (FK)	TName													
SName	ROP													
QOH	OSize													
OnOrder														
<p>Query: List the stock number, stock name, quantity on hand, reorder point, and order size for all stock items in which the quantity on hand is at or below the reorder point, and no order has yet been placed (OnOrder = 'N').</p> <p>Relation expression for this query :</p> $\Pi_{SNo, SName, QOH, ROP, OSize}(\sigma_{OnOrder='N'}(\sigma_{QOH \leq ROP}(\text{STOCK} \bowtie_{SType=TType} \text{STKTYPE})))$ <p>There are some problems with this mathematical version of RA operations:</p> <ol style="list-style-type: none">Most database students are not comfortable with the mathematical notation, especially the arbitrary use of Greek letters and the "bowtie" symbol.The functional notation and infix notation for operations are difficult to mix in expressions involving several RA operations. The notation also disguises the procedural nature of RA as a query language. <p>Breaking complex expressions into several steps, each step involving one operation, can lessen these problems. For example, the query expression shown above can be divided into the following sequence of operations:</p> <p>TEMP1 \leftarrow STOCK $\bowtie_{SType=TType}$ STKTYPE TEMP2 $\leftarrow \sigma_{QOH \leq ROP}$(TEMP1) TEMP3 $\leftarrow \sigma_{OnOrder='N'}$(TEMP2) TEMP4 $\leftarrow \Pi_{SNo, SName, QOH, ROP, OSize}$(TEMP3)</p>														

Extended Relational-Algebra Operations

The basic relational-algebra operations have been extended in several ways. A simple extension is to allow **arithmetic operations as part of projection**. An important extension is to allow **aggregate operations** such as computing the sum of the elements of a set, or their average. Another important extension is the **outer-join** operation, which allows relational-algebra expressions to deal with null values, which model missing information.

- **Generalized Projection**

The **generalized-projection** operation extends the projection operation by allowing arithmetic functions to be used in the projection list. The generalized projection operation has the form

$\Pi_{F_1, F_2, \dots, F_n}(E)$ where E is any relational-algebra expression, and each of F_1, F_2, \dots, F_n is an Arithmetic expression involving constants and attributes in the schema of E .

For example, suppose we have a relation **credit-info**, which lists the credit limit and expenses so far (the *credit-balance* on the account). If we want to find how much more each person can spend, we can write the following expression:

$\Pi_{customer-name, limit - credit-balance}(credit-info)$

The attribute resulting from the expression *limit - credit-balance* does not have a name. We can apply the rename operation to the result of generalized projection in order to give it a name. As a notational convenience, renaming of attributes can be combined with generalized projection as illustrated below:

$\Pi_{customer-name, (limit - credit-balance) \text{ as } credit-available}(credit-info)$

- **Aggregate functions and Operations** (symbol **g** or **F** used)

An aggregate function takes a **collection of values** and **returns a single value as a result**.

Aggregate operation in relational algebra $G_1, G_2, \dots, G_n \text{ g } F_1(A_1), F_2(A_2), \dots, F_n(A_n)(E)$

- E is any relational-algebra expression
- G_1, G_2, \dots, G_n is a list of attributes on which to **group** (can be empty)
- Each F_i is an aggregate function (*avr*, *max*, *min*, *count* etc.) and Each A_i is an attribute name

Example: The table E (for EMPLOYEE) **Only aggregate no grouping**

nr	name	salary	dept
1	John	100	A
5	Sarah	300	C
7	Tom	100	A
12	Anne	null	C

SQL	Result	Relational algebra
select sum(salary) from E	sum	$F_{sum(salary)}(E)$
	500	

Note:
Duplicates are not eliminated.

Null values are ignored.

SQL	Result	Relational algebra		
select count(salary) from E	Result: <table><tr><td>count</td></tr><tr><td>3</td></tr></table>	count	3	$F_{\text{count(salary)}}(E)$
count				
3				
select count(distinct salary) from E	Result: <table><tr><td>count</td></tr><tr><td>2</td></tr></table>	count	2	$F_{\text{count(salary)}}(\mathbf{PROJECT}_{\text{salary}}(E))$
count				
2				

You can calculate aggregates "grouped by" something:

SQL	Result	Relational algebra	
select sum(salary) from E group by dept	dept	dept _{F_{sum(salary)}} (E)	
	A		200
	C		300

Several aggregates simultaneously:

SQL	Result	Relational algebra									
select sum(salary), count(*) from E group by dept	<table> <tr> <th>dept</th><th>sum</th><th>count</th></tr> <tr> <td>A</td><td>200</td><td>2</td></tr> <tr> <td>C</td><td>300</td><td>1</td></tr> </table>	dept	sum	count	A	200	2	C	300	1	$\text{dept } F_{\text{sum(salary), count(*)}}(E)$
dept	sum	count									
A	200	2									
C	300	1									

- Outer join

Example: The table **E** (for **EMPLOYEE**)

enr	ename	dept
1	Bill	A
2	Sarah	C
3	John	A

Example: The table **D** (for **DEPARTMENT**)

dnr	dname
A	Marketing
B	Sales
C	Legal

We want to know the number of employees at each department?

SQL	Result	Relational algebra									
select dnr, dname, count(*) from E, D where edept = dnr group by dnr, dname	<table> <tr> <th>dnr</th><th>dname</th><th>count</th></tr> <tr> <td>A</td><td>Marketing</td><td>2</td></tr> <tr> <td>C</td><td>Legal</td><td>1</td></tr> </table>	dnr	dname	count	A	Marketing	2	C	Legal	1	$\text{dnr, dname } F_{\text{count(*)}}(E \text{ JOIN}_{\text{edept = dnr}} D)$
dnr	dname	count									
A	Marketing	2									
C	Legal	1									

No employee works at department B, Sales, so it is not present in the result. It disappeared already in the join, so the aggregate function never sees it. But what if we want it in the result, with the right number of employees (zero)?

Use a right outer join, which keeps all the rows from the right table. If a row can't be connected to any of the rows from the left table according to the join condition, null values are used:

SQL	Result	Relational algebra																									
<pre>select * from (E right outer join D on edept = dnr)</pre>	<table><tr><th>enr</th><th>ename</th><th>dept</th><th>dnr</th><th>dname</th></tr><tr><td>1</td><td>Bill</td><td>A</td><td>A</td><td>Marketing</td></tr><tr><td>2</td><td>Sarah</td><td>C</td><td>C</td><td>Legal</td></tr><tr><td>3</td><td>John</td><td>A</td><td>A</td><td>Marketing</td></tr><tr><td>null</td><td>null</td><td>null</td><td>B</td><td>Sales</td></tr></table>	enr	ename	dept	dnr	dname	1	Bill	A	A	Marketing	2	Sarah	C	C	Legal	3	John	A	A	Marketing	null	null	null	B	Sales	<p>E RIGHT OUTER JOIN_{eddept = dnr} D</p> <div>Here D is in the right side of join, so all tuples from right relation will participate in the join.</div>
enr	ename	dept	dnr	dname																							
1	Bill	A	A	Marketing																							
2	Sarah	C	C	Legal																							
3	John	A	A	Marketing																							
null	null	null	B	Sales																							
<pre>select dnr, dname, count(*) from E right outer join D on (edept = dnr) group by dnr, dname</pre>	<table><tr><th>dnr</th><th>dname</th><th>count</th></tr><tr><td>A</td><td>Marketing</td><td>2</td></tr><tr><td>B</td><td>Sales</td><td>1</td></tr><tr><td>C</td><td>Legal</td><td>1</td></tr></table>	dnr	dname	count	A	Marketing	2	B	Sales	1	C	Legal	1	<p>dnr, dname F_{count(*)}(E RIGHT OUTER JOIN_{eddept = dnr} D)</p> <div>Here no. of record counted so B gives value 1.</div>													
dnr	dname	count																									
A	Marketing	2																									
B	Sales	1																									
C	Legal	1																									
<pre>select dnr, dname, count(enr) from E right outer join D on (edept = dnr) group by dnr, dname</pre>	<table><tr><th>dnr</th><th>dname</th><th>count</th></tr><tr><td>A</td><td>Marketing</td><td>2</td></tr><tr><td>B</td><td>Sales</td><td>0</td></tr><tr><td>C</td><td>Legal</td><td>1</td></tr></table>	dnr	dname	count	A	Marketing	2	B	Sales	0	C	Legal	1	<p>dnr, dname F_{count(enr)}(E RIGHT OUTER JOIN_{eddept = dnr} D)</p> <div>Here no. of enr counted so B gives value 0 (no employee in B).</div>													
dnr	dname	count																									
A	Marketing	2																									
B	Sales	0																									
C	Legal	1																									

Modification of the Database

In earlier sections we were concentrated on the operations related to the **extraction of information** from the database. Now we will discuss how to **add, remove, or change** information in the database.

Deletion : We can delete only whole tuples; we cannot delete values on only particular attributes. In relational algebra a deletion is expressed by **$r \leftarrow r - E$** where r is a relation and E is a relational-algebra query.

Here are several examples of relational-algebra delete requests:

- Delete all of Smith's account records.
 $depositor \leftarrow depositor - \sigma_{customer-name = "Smith"}(depositor)$
- Delete all loans with amount in the range 0 to 50.
 $loan \leftarrow loan - \sigma_{amount \geq 0 \text{ and } amount \leq 50}(loan)$
- Delete all accounts at branches located in Needham.
 **$r1 \leftarrow \sigma_{branch-city = "Needham"}(account \bowtie branch)$
 **$r2 \leftarrow \Pi_{branch-name, account-number, balance}(r1)$
 $account \leftarrow account - r2$****

Note that, in the final example, we simplified our expression by using assignment to temporary relations ($r1$ and $r2$).

Insertion : To insert data into a relation, we either specify a tuple to be inserted or write a query whose result is a set of tuples to be inserted. Obviously, the attribute values for inserted tuples must be members of the attribute's domain. Similarly, tuples inserted must be of the correct **arity**.

Note : The **arity** of a function or operation is the **number of arguments or operands** that the function takes. The **arity** of a relation (or predicate) is the **dimension of the domain** in the corresponding Cartesian product

The relational algebra expresses an insertion by $r \leftarrow r \cup E$ where r is a relation and E is a relational-algebra expression. We express the insertion of a single tuple by letting E be a constant relation containing one tuple.

Suppose that we wish to insert the fact that Smith has \$1200 in account A-973 at the Palam branch. We write

$account \leftarrow account \cup \{(A-973, "Palam", 1200)\}$
 $depositor \leftarrow depositor \cup \{("Smith", A-973)\}$

More generally, we might want to **insert tuples on the basis of the result of a query**. Suppose that we want to provide as a gift for all loan customers of the Palam branch a new \$200 **savings account**. Let the loan number serve as the account number for this savings account. We write

$r1 \leftarrow (\sigma_{branch-name = "Palam"} (borrower \bowtie loan))$
 $r2 \leftarrow \Pi_{loan-number, branch-name} (r1)$
 $account \leftarrow account \cup (r2 \times \{(200)\})$
 $depositor \leftarrow depositor \cup \Pi_{customer-name, loan-number} (r1)$

Instead of specifying a tuple as we did earlier, we specify a set of tuples that is inserted into both the *account* and *depositor* relation. Each tuple in the *account* relation has an *account-number* (which is the same as the loan number), a *branch-name* (Palam), and the initial balance of the new account (\$200). Each tuple in the *depositor* relation has as *customer-name* the name of the loan customer who is being given the new account and the same account number as the corresponding *account* tuple.

Updating : We can use the generalized-projection operator to update a tuple:

$r \leftarrow \Pi_{F1, F2, \dots, Fn}(r)$ where $F1, F2, \dots, Fn$ are the attributes of the relation.

No update for the i^{th} attribute : just put that attribute F_i

Update for the i^{th} attribute : F_i is an **expression**, involving only **constants** and the **attributes** of r , that gives the **new value** for the attribute.

Generally we want to **select some tuples from r and to update only them**, this can be achieved this by using following steps :

- select the record to be updated and update the attributes (it acts as anew tuple after update)
- select the tuple to be update and minus it from original relation
- union the resultant relation

Above steps can be implemented using the following expression; here, P denotes the selection condition that chooses which tuples to update:

$$r \leftarrow \Pi_{F_1, F_2, \dots, F_n}(\sigma_P(r)) \cup (r - \sigma_P(r))$$

$$r \leftarrow (r - \sigma_P(r)) \cup \Pi_{F_1, F_2, \dots, F_n}(\sigma_P(r))$$

Both are equivalent

Tuples after removing the tuple to be updated

Tuple after update

Example : Say we want to update age of tuple where name = "John" in a relation Human(name, age, height, weight).

Age is updated with a value 20

$$\text{Human} \leftarrow (\text{Human} - \sigma_{\text{name}="John"}(\text{Human})) \cup \pi_{\text{name}, 20, \text{height}, \text{weight}}(\sigma_{\text{name}="John"}(\text{Human}))$$

Example : Suppose that interest payments are being made, and that all balances are to be increased by 5 percent. We write

$$\text{account} \leftarrow \Pi_{\text{account-number}, \text{branch-name}, \text{balance} * 1.05}(\text{account})$$

Now suppose that accounts with balances over \$10,000 receive 6 percent interest, whereas all others receive 5 percent. We write

$$\text{account} \leftarrow \Pi_{AN, BN, \text{balance} * 1.06}(\sigma_{\text{balance} > 10000}(\text{account}))$$

$$\cup \Pi_{AN, BN, \text{balance} * 1.05}(\sigma_{\text{balance} \leq 10000}(\text{account}))$$

where the abbreviations **AN** and **BN** stand for **account-number** and **branch-name**, respectively.

• Views

A view is defined using the **create view** statement which has the form **create view v as <query expression>** where <query expression> is any legal relational algebra query expression. The view name given as v.

- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.
- View definition is **not the same as creating a new relation by evaluating the query expression** Rather, a view definition causes the saving of an expression to be substituted into queries using the view.

- View is used to limited access to DB

Consider a person who needs to know a **customer's loan number but has no need to see the loan amount**. This person should see a relation described as:

$$\Pi_{customer-name, loan-number} (borrower \bowtie loan)$$

Examples

- Consider the view (named all-customer) consisting of branches and their customers.
create view all-customer as

create view all-customer as

$$\Pi_{branch-name, customer-name} (depositor \bowtie account) \\ \cup \Pi_{branch-name, customer-name} (borrower \bowtie loan)$$

We can find all customers of the Perryridge branch by writing:

$$\Pi_{customer-name} \\ (\sigma_{branch-name = "Perryridge"} (all-customer))$$

Relational Calculus

- Relational calculus is a query language which is **non-procedural**, and instead of algebra, it uses **mathematical predicate calculus**. Thus, it explains what to do but not how to do.
- The relational calculus is not the same as that of differential and integral calculus in mathematics but takes its name from a branch of symbolic logic termed as **predicate calculus**.
- In first-order logic or predicate calculus, a predicate is a truth-valued function with arguments. When we replace with values for the arguments, the function yields an expression, called a proposition, which will be either true or false.
- When applied to databases, it is found in two forms. These are
 - **Tuple relational calculus** which was originally proposed by Codd in the year 1972 and
 - **Domain relational calculus** which was proposed by Lacroix and Pirotte in the year 1977

Note :

- In traditional grammar, a predicate is one of the two main parts of a sentence. For the simple sentence "John [is yellow]," John acts as the subject, and is yellow acts as the predicate, a subsequent description of the subject headed with a verb.
- First-order logic—also known as **predicate logic** and first-order **predicate calculus**—is a collection of formal systems used in mathematics, philosophy, linguistics, and computer science.

Tuple Relational Calculus

- Tuple (record) Relational Calculus is a **non-procedural query language** unlike relational algebra. Tuple Calculus provides only the description of the query but it does not provide the methods to solve it. Thus, it explains what to do but not how to do. In Tuple Calculus, a query is expressed as

$\{t \mid P(t)\}$

where t = resulting tuples,

$P(t)$ = known as Predicate and these are the conditions that are used to fetch t

'|' means such that

- Thus, it generates set of all tuples t , such that Predicate $P(t)$ is true for t .
- $P(t)$ may have various conditions logically combined with OR (\vee), AND (\wedge), NOT (\neg).
- It also uses quantifiers: The existential quantifier (\exists) and the universal quantifier (\forall) can be used to bind the variables.

$\exists t \in r (Q(t))$ = "there exists" a tuple in t in relation r such that predicate $Q(t)$ is true.

$\forall t \in r (Q(t))$ = $Q(t)$ is true "for all" tuples in relation r .

- We use $t[A]$ to denote the value of tuple t on attribute A , and we use $t \in r$ to denote that tuple t is in relation r .

Example:

$\{t \mid \text{TEACHER}(t) \text{ and } t.\text{SALARY} > 20000\}$

It implies that it selects the tuples from the TEACHER in such a way that the resulting teacher tuples will have the salary greater than 20000. This is an example of selecting a range of values.

$\{t \mid \text{TEACHER}(t) \text{ AND } t.\text{DEPT_ID} = 6\}$

Select all the tuples of teachers name who work under Department 8. Any tuple variable with 'For All' or 'there exists' condition is termed as a **bound variable**. In the last example, for any range of values of SALARY greater than 20000, the meaning of the **condition does not alter**. Bound variables are those ranges of tuple variables whose meaning will not alter if another tuple variable replaces the tuple variable.

In the second example, you have used $\text{DEPT_ID} = 8$ which means only for $\text{DEPT_ID} = 8$ display the teacher details. Such variable is called **free variable**. Any tuple variable without any 'For All' or 'there exists' condition is called Free Variable.

Relational calculus is useful to define the **semantics** of the relational algebra and SQL. Below is the example(Link between TRC and SQL):

TRC = $\{T \mid \text{Teaching}(T) \text{ AND } T.\text{Semester} = \text{'F2000'}\}$

SQL = SELECT *

FROM Teaching T

WHERE T.Semester = 'F2000'

Target T corresponds to SELECT list: the query result contains the entire tuple. Body split between two clauses:

- Teaching(T) corresponds to FROM clause
- T.Semester = 'F2000' corresponds to WHERE clause

The statement above = $TRC = \{T \mid Teaching(T) \text{ AND } T.Semester = 'F2000'\}$

can be referred as "T is a variable(tuple) whose value is equal to Teaching(T) and T.Semester = 'F2000' simultaneously" ie. T is the tuple from relation "Teaching" and refers to tuples having attribute "Semester" as 'F2000'

Some more Example using following relations :

Table-1: Customer

CUSTOMER NAME	STREET	CITY
Saurabh	A7	Patiala
Mehak	B6	Jalandhar
Sumiti	D9	Ludhiana
Ria	A5	Patiala

Table-2: Branch

BRANCH NAME	BRANCH CITY
ABC	Patiala
DEF	Ludhiana
GHI	Jalandhar

Table-3: Account

ACCOUNT NUMBER	BRANCH NAME	BALANCE
1111	ABC	50000
1112	DEF	10000
1113	GHI	9000
1114	ABC	7000

Table-4: Loan

LOAN NUMBER	BRANCH NAME	AMOUNT
L33	ABC	10000
L35	DEF	15000
L49	GHI	9000
L98	DEF	65000

Table-5: Borrower

CUSTOMER NAME	LOAN NUMBER
Saurabh	L33
Mehak	L49
Ria	L98

Table-6: Depositor

CUSTOMER NAME	ACCOUNT NUMBER
Saurabh	1111
Mehak	1113
Sumiti	1114

Queries-1: Find the loan number, branch, amount of loans of greater than or equal to 10000 amount.

$\{t \mid t \in \text{loan} \wedge t[\text{amount}] \geq 10000\}$

Where t[amount] is known as tuple variable.

Resulting relation:

LOAN NUMBER	BRANCH NAME	AMOUNT
L33	ABC	10000
L35	DEF	15000
L98	DEF	65000

Queries-2: Find the loan number for each loan of an amount greater or equal to 10000.

$\{t \mid \exists s \in \text{loan} (t[\text{loan number}] = s[\text{loan number}] \wedge s[\text{amount}] \geq 10000)\}$

Resulting relation:

LOAN NUMBER
L33
L35
L98

Queries-3: Find the names of all customers who have a loan

Resulting relation:

and an account at the bank.	<table><tr><th>CUSTOMER NAME</th></tr><tr><td>Saurabh</td></tr><tr><td>Mehak</td></tr></table>	CUSTOMER NAME	Saurabh	Mehak
CUSTOMER NAME				
Saurabh				
Mehak				
$\{t \mid \exists s \in \text{borrower}(t[\text{customer-name}] = s[\text{customer-name}]) \wedge \exists u \in \text{depositor}(t[\text{customer-name}] = u[\text{customer-name}])\}$				
Queries-4: Find the names of all customers having a loan at the “ABC” branch.	Resulting relation:			
$\{t \mid \exists s \in \text{borrower}(t[\text{customer-name}] = s[\text{customer-name}] \wedge \exists u \in \text{loan}(u[\text{branch-name}] = \text{“ABC”} \wedge u[\text{loan-number}] = s[\text{loan-number}]))\}$	<table><tr><th>CUSTOMER NAME</th></tr><tr><td>Saurabh</td></tr></table>	CUSTOMER NAME	Saurabh	
CUSTOMER NAME				
Saurabh				

Domain Relational Calculus

In the tuple relational calculus, you have use variables that have series of tuples in a relation. In the domain relational calculus, you will also use variables, but in this case, the variables take their values from domains of attributes rather than tuples of relations. A domain relational calculus expression has the following general format:

$\{d_1, d_2, \dots, d_n \mid F(d_1, d_2, \dots, d_m)\} \ m \geq n$

The result of the query is the set of tuples d_1 to d_n that make the DRC formula true.

where $d_1, d_2, \dots, d_n, \dots, d_m$ stand for domain of attributes (columns) and $F(d_1, d_2, \dots, d_m)$ stands for a formula including the condition for fetching the data.

This language uses the same operators as [tuple calculus](#), the logical connectives \wedge (and), \vee (or) and \neg (not). The [existential quantifier](#) (\exists) and the [universal quantifier](#) (\forall) can be used to bind the variables.

Example:

select TCHR_ID and TCHR_NAME of teachers who work for department 8, (where suppose - dept. 8 is Computer Application Department)

$\{ \langle tchr_id, tchr_name \rangle \mid \langle tchr_id, tchr_name \rangle \in \text{TEACHER} \wedge \text{DEPT_ID} = 10 \}$

Get the name of the department name where Karlos works:

$\{ \text{DEPT_NAME} \mid \langle \text{DEPT_NAME} \rangle \in \text{DEPT} \wedge \exists \text{DEPT_ID} (\exists \text{TEACHER} \wedge \text{TCHR_NAME} = \text{Karlos}) \}$

$\{ \langle \text{name}, \text{age} \rangle \mid \langle \text{name}, \text{age} \rangle \in \text{Student} \wedge \text{age} > 17 \}$

The query will return the names and ages of the students in the table Student who are older than 17

