UNIT-II PL/SQL,DATA MODEL AND QUERY POCESSING

***PL/SQL: Functions, Procedures, Triggers, Cursors* –Dynamic SQL-Relational algebra-tuple relational calculus-domain relation calculus-Entity Relatioship Model-Constrain-Entity Relationship Diagram- Design Issues of ER Model-Extended ER Feartures-Mapping ER Model to Relational Model-Query processing-Heuristics for Query Optimization.**

PL/SQL: Functions:

Function can be used as a part of SQL expression i.e. we can use them with select/update/merge commands. One most important characteristic of a function is that, unlike procedures, it must return a value. A function is same as a procedure except that it returns a value. Therefore, all the discussions of

the previous chapter are true for functions too.

Creating a Function

A standalone function is created using the CREATE FUNCTION statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows –

CREATE [OR REPLACE] FUNCTION function_name [(parameter_name [IN | OUT | IN OUT] type [, ...])] RETURN return_datatype

{IS | AS} BEGIN

< function_body > END [function_name]; Where,

- ***function-name specifies the name of the function.***
- ***[OR REPLACE] option allows the modification of an existing function.***
- ***The optional parameter list contains name, mode and types of the parameters. IN represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.***
- ***The function must contain a return statement.***
- ***The RETURN clause specifies the data type you are going to return from the function.***
- ***function-body contains the executable part.***
- ***The AS keyword is used instead of the IS keyword for creating a standalone function.***

Example

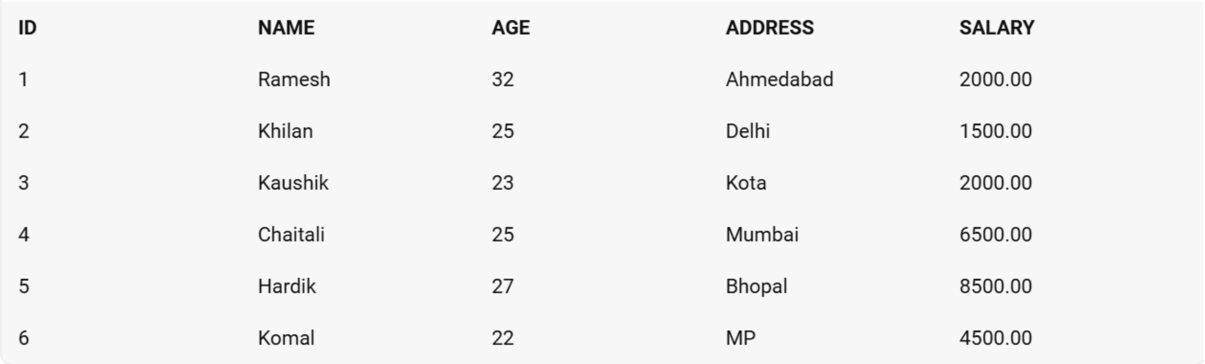
The following example illustrates how to create and call a standalone

function. This function returns the total number of CUSTOMERS in the customers table.

We will use the CUSTOMERS table, which we had created in the PL/SQL Variables chapter

–

Select * from customers;



ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	Kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00

```
REATE OR REPLACE FUNCTION totalCustomers RETURN number IS

total number(2) := 0; BEGIN

SELECT count(*) into total FROM customers;

RETURN total; END;

/
```

When the above code is executed using the SQL prompt, it will produce the following result

–

Function created. Calling a Function

While creating a function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. When a program calls a function, the program control is transferred to the called function.

A called function performs the defined task and when its return statement is executed or when the **last end statement** is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name and if the function returns a value, then you can store the returned value. Following program calls the function **totalCustomers** from an anonymous block –

```
DECLARE

c number(2); BEGIN

c := totalCustomers();

dbms_output.put_line('Total no. of Customers: ' || c); END;

/
```

When the above code is executed at the SQL prompt, it produces the following result – Total no. of Customers: 6

PL/SQL procedure successfully completed.

Example

The following example demonstrates Declaring, Defining, and Invoking a Simple PL/SQL Function that computes and returns the maximum of two values.

```
DECLARE

a number; b number; c number;

FUNCTION findMax(x IN number, y IN number) RETURN number
```

```
IS

z number; BEGIN

IF x > y THEN

z:= x; ELSE

Z:= y; END IF; RETURN z;

END; BEGIN

a:= 23;

b:= 45;

c := findMax(a, b);

dbms_output.put_line(' Maximum of (23,45): ' || c); END;
```

When the above code is executed at the SQL prompt, it produces the following result − Maximum of (23,45): 45

PL/SQL procedure successfully completed. PL/SQL Recursive Functions

We have seen that a program or subprogram may call another subprogram. When a subprogram calls itself, it is referred to as a recursive call and the process is known as **recursion**.

To illustrate the concept, let us calculate the factorial of a number. Factorial of a number n is defined as −

n! = n*(n-1)!

= n*(n-1)*(n-2)!

...

= n*(n-1)*(n-2)*(n-3)... 1

The following program calculates the factorial of a given number by calling itself recursively

```
− DECLARE

num number; factorial number;

FUNCTION fact(x number) RETURN number

IS

f number;

BEGIN

IF x=0 THEN f := 1;

ELSE

f := x * fact(x-1); END IF;

RETURN f; END;

BEGIN

num:= 6;

factorial := fact(num);

dbms_output.put_line(' Factorial '|| num || ' is ' || factorial); END;
```

/

When the above code is executed at the SQL prompt, it produces the following result − Factorial 6 is 720

PL/SQL procedure successfully completed.

Procedures:

The PL/SQL stored procedure or simply a procedure is a PL/SQL block which performs one or more specific tasks. It is just like procedures in other programming languages.

The procedure contains a header and a body.

- **Header:** The header contains the name of the procedure and the parameters or variables passed to the procedure.
- **Body:** The body contains a declaration section, execution section and exception section similar to a general PL/SQL block.

How to pass parameters in procedure:

When you want to create a procedure or function, you have to define parameters .There is three ways to pass parameters in procedure:

1. **IN parameters:** The IN parameter can be referenced by the procedure or function. The value of the parameter cannot be overwritten by the procedure or the function.
2. **OUT parameters:** The OUT parameter cannot be referenced by the procedure or function, but the value of the parameter can be overwritten by the procedure or function.
3. **INOUT parameters:** The INOUT parameter can be referenced by the procedure or function and the value of the parameter can be overwritten by the procedure or function.

A procedure may or may not return any value.

PL/SQL Create Procedure Syntax for creating procedure:

CREATE [OR REPLACE] **PROCEDURE** procedure_name [(parameter [,parameter])]

IS

[declaration_section]

BEGIN

executable_section [EXCEPTION exception_section]

END [procedure_name];

Create procedure example

In this example, we are going to insert record in user table. So you need to create user table first.

Table creation:

create table user(id number(10) **primary key**,**name** varchar2(100)); Now write the procedure code to insert record in user table.

Procedure Code:

create or replace **procedure** "INSERTUSER" (id IN NUMBER,

```
name IN VARCHAR2)
is begin
insert into user values(id,name); end;
/
```

Output:

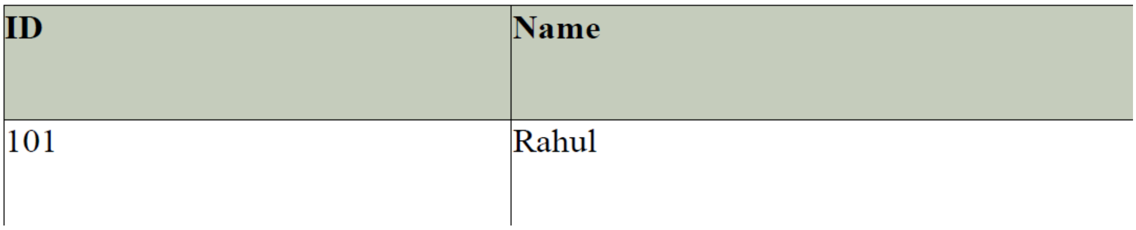
```
Procedure created.
```

PL/SQL program to call procedure

Let's see the code to call above created procedure.

```
BEGIN
insertuser(101,'Rahul'); dbms_output.put_line('record inserted successfully');
END;
/
```

Now, see the "USER" table, you will see one record is inserted.



ID	Name
101	Rahul

PL/SQL Drop Procedure Syntax for drop procedure

```
DROP PROCEDURE procedure name;
```

Example of drop procedure

```
DROP PROCEDURE pro1;
```

Triggers:

Trigger: A trigger is a stored procedure in database which automatically

invokes whenever a special event in the database occurs. For example, a trigger can be invoked when a row is inserted into a specified table or when certain table columns are being updated.

Syntax:

```
create trigger [trigger_name] [before | after]
{insert | update | delete} on [table_name]
[for each row] [trigger_body]
```

Explanation of syntax:

1. create trigger [trigger_name]: Creates or replaces an existing trigger with the trigger_name.
2. [before | after]: This specifies when the trigger will be executed.
3. {insert | update | delete}: This specifies the DML operation.

4. on [table_name]: This specifies the name of the table associated with the trigger.
5. [for each row]: This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected.
6. [trigger_body]: This provides the operation to be performed as trigger is fired

BEFORE and AFTER of Trigger:

BEFORE triggers run the trigger action before the triggering statement is run. AFTER triggers run the trigger action after the triggering statement is run.

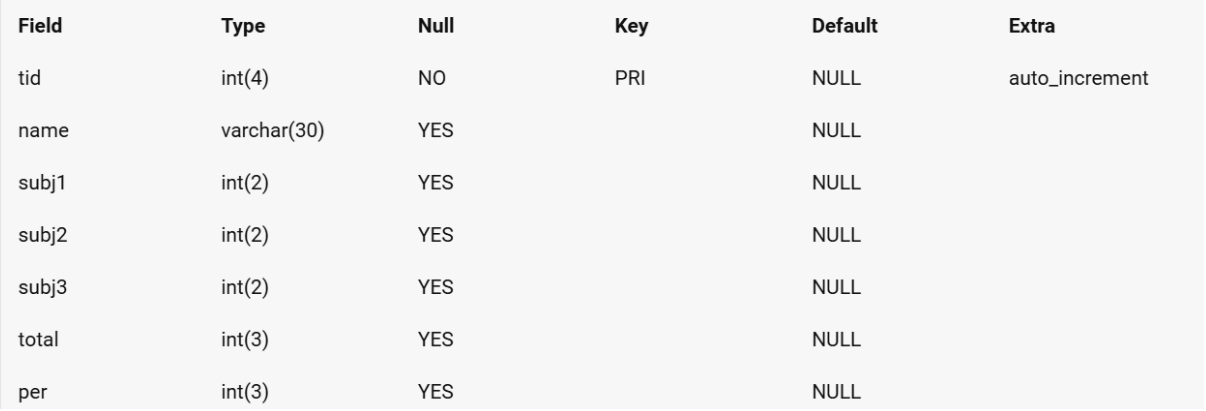
Example:

Given Student Report Database, in which student marks assessment is recorded. In such schema, create a trigger so that the total and average of specified marks is automatically inserted whenever a record is insert.

Here, as trigger will invoke before record is inserted so, BEFORE Tag can be used.

Suppose the database Schema –

mysql> desc Student;



Field	Type	Null	Key	Default	Extra
tid	int(4)	NO	PRI	NULL	auto_increment
name	varchar(30)	YES		NULL	
subj1	int(2)	YES		NULL	
subj2	int(2)	YES		NULL	
subj3	int(2)	YES		NULL	
total	int(3)	YES		NULL	
per	int(3)	YES		NULL	

SQL Trigger to problem statement. create trigger stud_marks before INSERT on

Student

for each row

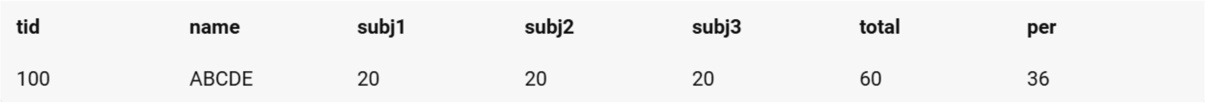
set Student.total = Student.subj1 + Student.subj2 + Student.subj3, Student.per = Student.total

* 60 / 100;

Above SQL statement will create a trigger in the student database in which whenever subjects marks are entered, before inserting this data into the database, trigger will compute those two values and insert with the entered values. i.e.,

mysql> insert into Student values(0, "ABCDE", 20, 20, 20, 0, 0); Query OK, 1 row affected (0.09 sec)

mysql> select * from Student;



tid	name	subj1	subj2	subj3	total	per
100	ABCDE	20	20	20	60	36

In this way trigger can be creates and executed in the databases.

Attention reader! Don‟t stop learning now. Get hold of all the important CS Theory concepts for SDE interviews with the **CS Theory Course** at a student-friendly price and become industry ready.

Advantages of Triggers

These are the following advantages of Triggers:

- Trigger generates some derived column values automatically
- Enforces referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Imposing security authorizations
- Preventing invalid transactions Creating a trigger:

Syntax for creating trigger:

CREATE [OR REPLACE] **TRIGGER** trigger_name

{BEFORE | **AFTER** | **INSTEAD OF** }

{**INSERT** [OR] | **UPDATE** [OR] | **DELETE**}

[OF col_name]

ON table_name

[REFERENCING OLD **AS** o NEW **AS** n] [**FOR** EACH ROW]

WHEN (condition)

DECLARE

Declaration-statements

BEGIN

Executable-statements EXCEPTION Exception-handling-statements **END**;

Here,

- CREATE [OR REPLACE] TRIGGER trigger_name: It creates or

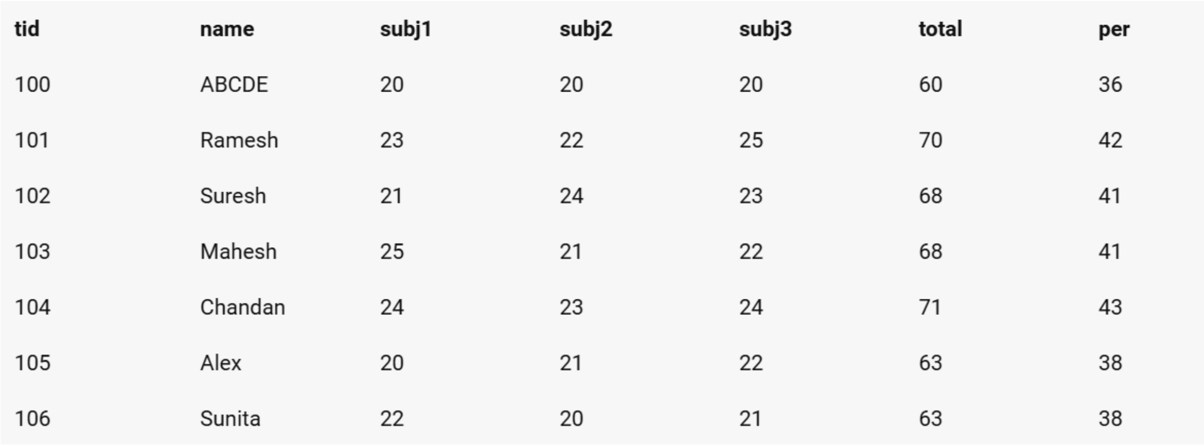
- an existing trigger with the trigger_name.
- {BEFORE | AFTER | INSTEAD OF} : This specifies when the trigger would be executed. The INSTEAD OF clause is used for creating trigger on a view.

- {INSERT [OR] | UPDATE [OR] | DELETE}: This specifies the DML operation.
- [OF col_name]: This specifies the column name that would be updated.
- [ON table_name]: This specifies the name of the table associated with the trigger.
- [REFERENCING OLD AS o NEW AS n]: This allows you to refer new and old values for various DML statements, like INSERT, UPDATE, and DELETE.
- [FOR EACH ROW]: This specifies a row level trigger, i.e., the trigger would be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
- WHEN (condition): This provides a condition for rows for which the trigger would fire. This clause is valid only for row level triggers.

PL/SQL Trigger Example

Let's take a simple example to demonstrate the trigger. In this example, we are using the following CUSTOMERS table:

Create table and have records:



tid	name	subj1	subj2	subj3	total	per
100	ABCDE	20	20	20	60	36
101	Ramesh	23	22	25	70	42
102	Suresh	21	24	23	68	41
103	Mahesh	25	21	22	68	41
104	Chandan	24	23	24	71	43
105	Alex	20	21	22	63	38
106	Sunita	22	20	21	63	38

Create trigger:

Let's take a program to create a row level trigger for the CUSTOMERS table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values:

```
CREATE OR REPLACE TRIGGER display_salary_changes BEFORE DELETE OR

INSERT OR UPDATE ON customers FOR EACH ROW

WHEN (NEW.ID > 0)

DECLARE

sal_diff number;

BEGIN

sal_diff := :NEW.salary - :OLD.salary; dbms_output.put_line('Old salary: ' || :OLD.salary); dbms_output.put_line('New salary: ' || :NEW.salary); dbms_output.put_line('Salary difference: ' || sal_diff);

END;
```

After the execution of the above code at SQL Prompt, it produces the following result.

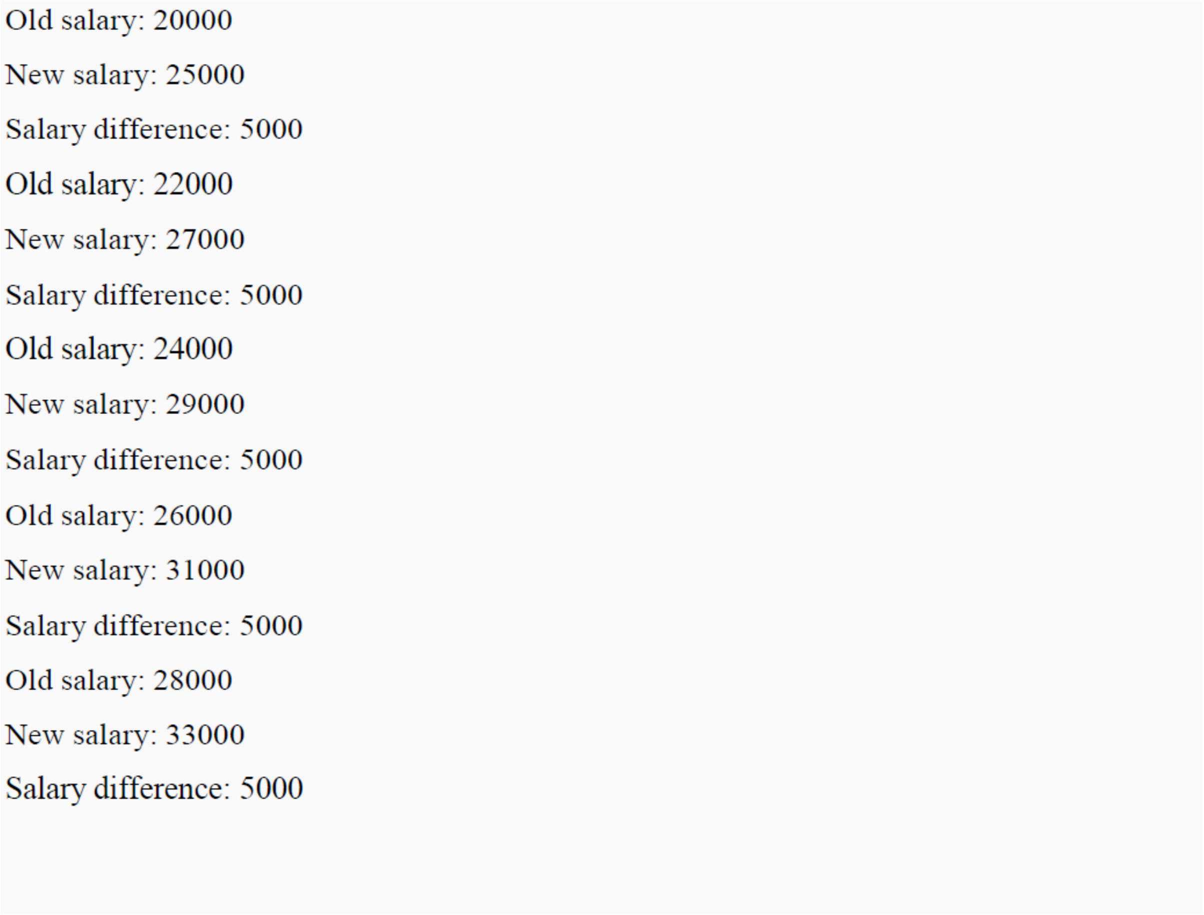
Trigger created.

Check the salary difference by procedure:

Use the following code to get the old salary, new salary and salary difference after the trigger created.

```
DECLARE

total_rows number(2);

BEGIN

UPDATE customers

SET salary = salary + 5000;

IF sql%notfound THEN

dbms_output.put_line('no customers updated');

ELSIF sql%found THEN

total_rows := sql%rowcount;

dbms_output.put_line( total_rows || ' customers updated ');

END IF;

END;/
```

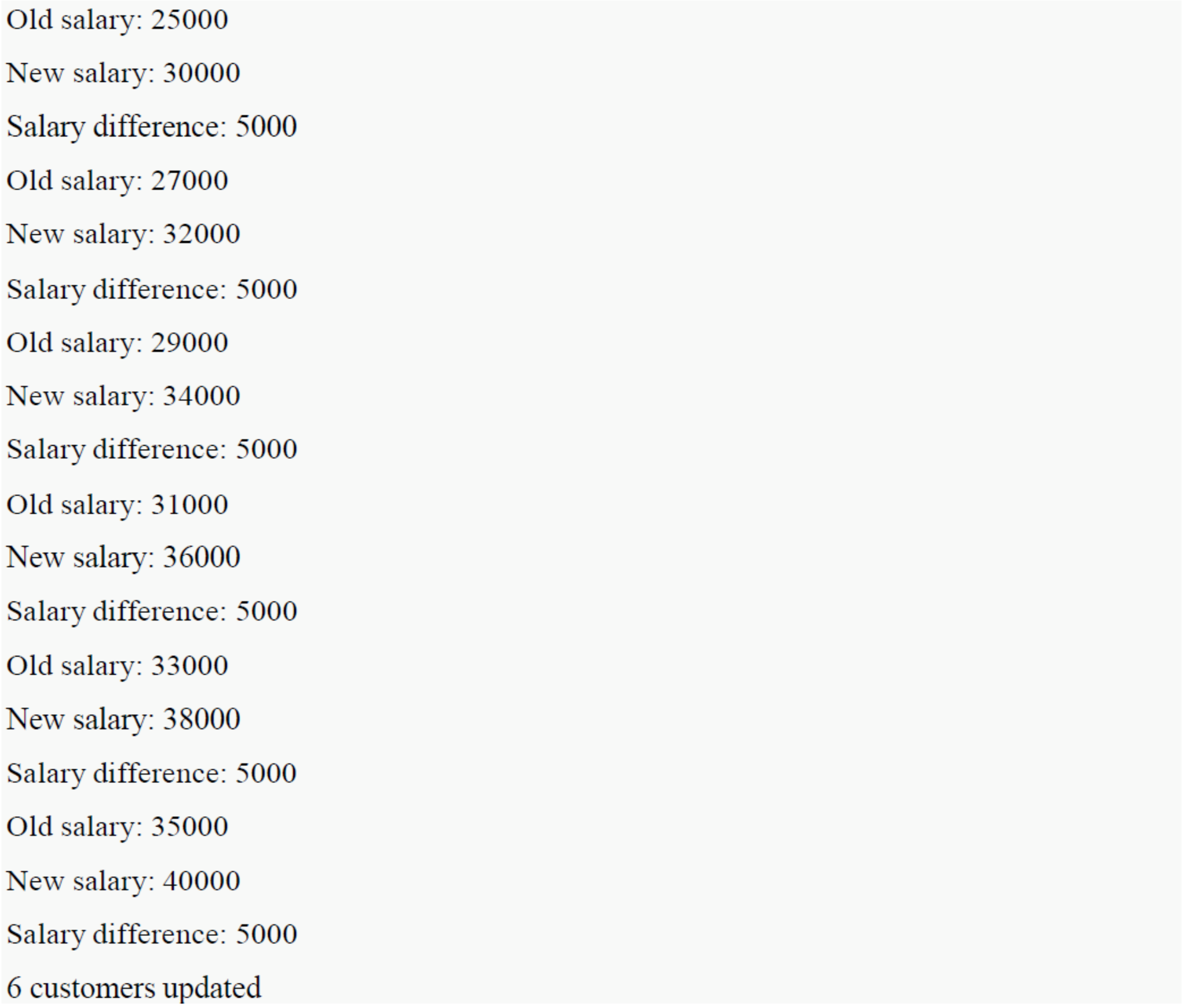
Output:



```
Old salary: 20000
New salary: 25000
Salary difference: 5000
Old salary: 22000
New salary: 27000
Salary difference: 5000
Old salary: 24000
New salary: 29000
Salary difference: 5000
Old salary: 26000
New salary: 31000
Salary difference: 5000
Old salary: 28000
New salary: 33000
Salary difference: 5000
```

Note: As many times you executed this code, the old and new both salary is incremented by 5000 and hence the salary difference is always 5000.

After the execution of above code again, you will get the following result.



```
Old salary: 25000
New salary: 30000
Salary difference: 5000
Old salary: 27000
New salary: 32000
Salary difference: 5000
Old salary: 29000
New salary: 34000
Salary difference: 5000
Old salary: 31000
New salary: 36000
Salary difference: 5000
Old salary: 33000
New salary: 38000
Salary difference: 5000
Old salary: 35000
New salary: 40000
Salary difference: 5000
6 customers updated
```

Old salary: 25000

New salary: 30000

Salary difference: 5000

Old salary: 27000

New salary: 32000

Salary difference: 5000

Old salary: 29000

New salary: 34000

Salary difference: 5000

Old salary: 31000

New salary: 36000

Salary difference: 5000

Old salary: 33000

New salary: 38000

Salary difference: 5000

Old salary: 35000

New salary: 40000

Salary difference: 5000

6 customers updated

Important Points

Following are the two very important point and should be noted carefully.OLD and NEW references are used for record level triggers these are not avialable for table level triggers.

If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.

Cursors:

Whenever DML statements are executed, a temporary work area is created in the system memory and it is called a cursor. A cursor can have more than one row, but processing wise only 1 row is taken into account. Cursors are very helpful in all kinds of databases like Oracle, SQL Server, MySQL, etc. They can be used well with DML statements like Update, Insert and Delete.

Especially Implicit cursors are there with these operations. From time to time it changes the values and hence the implicit cursor attribute values need to be assigned in a local variable for further use. In PL/SQL, two different types of cursors are available.

- Explicit cursors

Explicit cursors

Explicit cursors are defined by the programmers to have more control area on the context area. It has to be defined in the declaration section of the PL/SQL Block. Usually, It is defined on a SELECT Statement and it returns more than one row as output. We can iterate over the rows of data and perform the required operations.

Steps involved in creating explicit cursors:

- Cursor Declaration for initializing the memory CURSOR <cursorName> IS

SELECT <Required fields> FROM <tableName>;

- Cursor Opening to allocate the memory OPEN <cursorName>;
- Cursor Fetching to retrieve the data

FETCH <cursorName> INTO <Respective columns>

- Cursor Closing to release the allocated memory CLOSE <cursorName>;

Example:

DECLARE

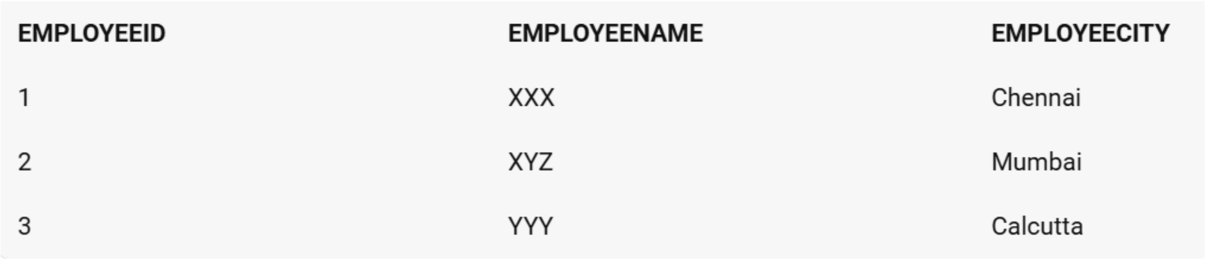
empId employees.EMPLOYEEID%type; empName employees.EMPLOYEENAME%type; empCity employees.EMPLOYEECITY%type;

```
CURSOR c_employees is

SELECT EMPLOYEEID, EMPLOYEENAME, EMPLOYEECITY FROM employees; BEGIN

OPEN c_employees; LOOP

FETCH c_employees into empId , empName , empCity; EXIT WHEN c_employees %notfound;

dbms_output.put_line(empId || ' ' || empName || ' ' || empCity); END LOOP;
```

- Implicit cursors

```
CLOSE c_employees ; END;

/
```

Output:



EMPLOYEEID	EMPLOYEENAME	EMPLOYEECITY
1	XXX	Chennai
2	XYZ	Mumbai
3	YYY	Calcutta

Implicit cursors

For DML statements, implicit cursors are available in PL/SQL i.e. no need to declare the cursor, and even for the queries that return 1 row, implicit cursors are available. Through the cursor attributes, we can track the information about the execution of an implicit cursor.

Attributes of Implicit Cursors:

Implicit cursor attributes provide the results about the execution of

INSERT, UPDATE, and DELETE. We have different Cursor attributes like “%FOUND”, “%ISOPEN”, “%NOTFOUND”, and %ROWCOUNT. The most recently executed SQL statement result will be available in Cursor. Initially cursor value will be null.

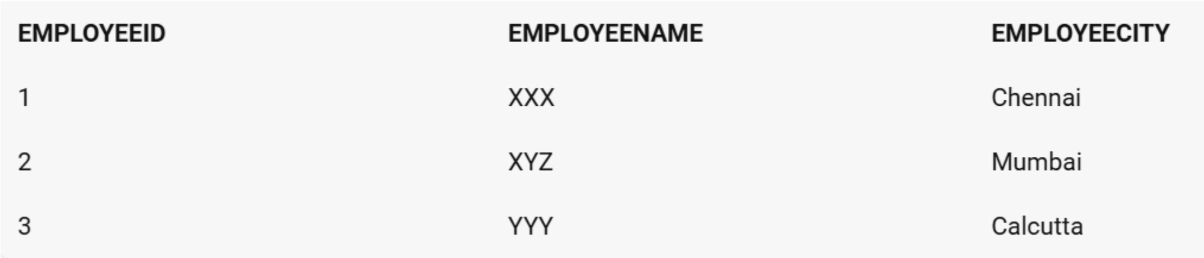
Let us see the different cursor attributes one by one with regards to the DML statements. So let us create a sample table named “employees” in oracle:

```
CREATE TABLE employees

( EMPLOYEEID number(10) NOT NULL, EMPLOYEENAME varchar2(50) NOT NULL, EMPLOYEECITY varchar2(50)

);
```

Insert the records in “employees” table

```
INSERT INTO employees (employeeId,employeeName,employeeCity) VALUES (1,'XXX','CHENNAI');

INSERT INTO employees (employeeId,employeeName,employeeCity) VALUES (2,'XYZ','MUMBAI');

INSERT INTO employees (employeeId,employeeName,employeeCity) VALUES (3,'YYY','CALCUTTA');
```

Existence of records in “employees” table SELECT * FROM employees;



EMPLOYEEID	EMPLOYEENAME	EMPLOYEECITY
1	XXX	Chennai
2	XYZ	Mumbai
3	YYY	Calcutta

%FOUND attribute

CREATE TABLE tempory_employee AS SELECT * FROM employees; DECLARE

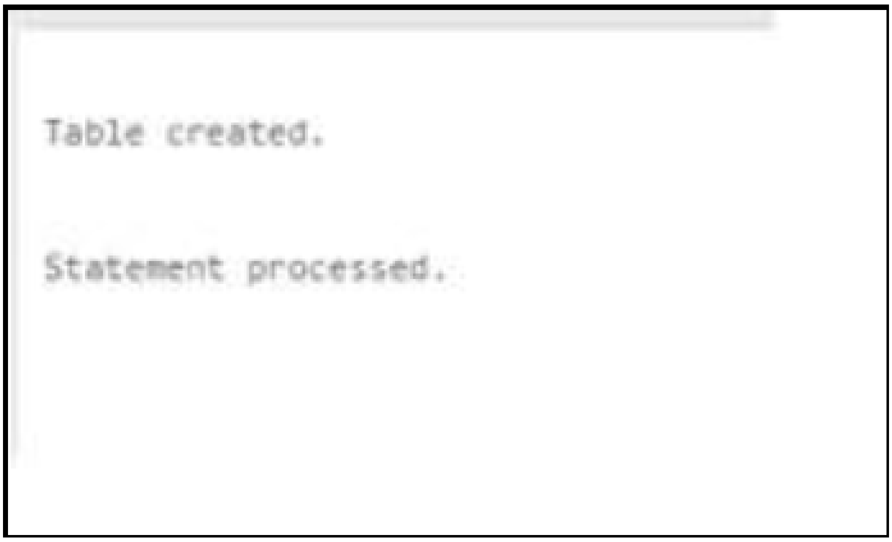
employeeNo NUMBER(4) := 2; BEGIN

DELETE FROM tempory_employee WHERE employeeId = employeeNo ; IF SQL%FOUND THEN -- delete succeeded

INSERT INTO tempory_employee (employeeId,employeeName,employeeCity) VALUES (2, 'ZZZ', 'Delhi');

END IF; END;

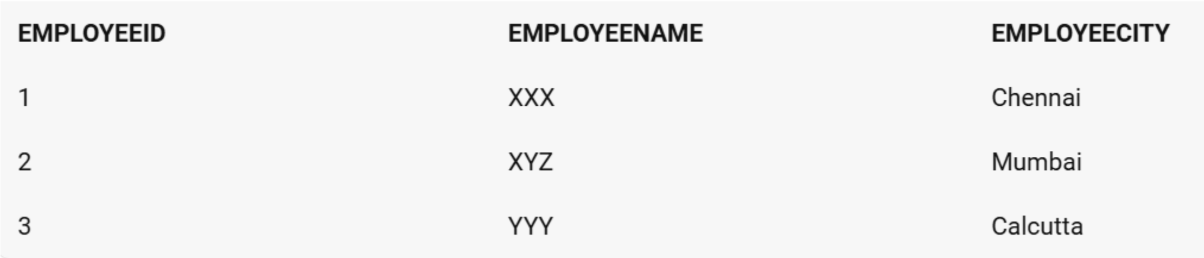
/



```
Table created.

Statement processed.
```

Now check for the details present in the tempory_employee table SELECT * FROM tempory_employee;

Output:



EMPLOYEEID	EMPLOYEENAME	EMPLOYEECITY
1	XXX	Chennai
2	XYZ	Mumbai
3	YYY	Calcutta

%FOUND attribute and performing update operation example CREATE TABLE tempory_employee1 AS SELECT * FROM employees; DECLARE

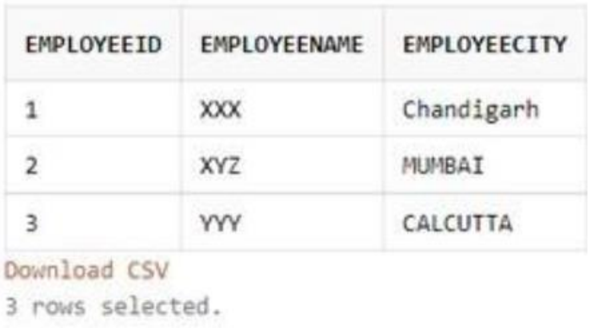
employeeNo NUMBER(4) := 2; BEGIN

DELETE FROM tempory_employee WHERE employeeId = employeeNo ; IF SQL%FOUND THEN -- delete succeeded

UPDATE employees SET employeeCity = 'Chandigarh' WHERE employeeId = 1; END IF;

```
END;
/
```

Output from SELECT * FROM employees:



EMPLOYEEID	EMPLOYEENAME	EMPLOYEECITY
1	XXX	Chandigarh
2	XYZ	MUMBAI
3	YYY	CALCUTTA

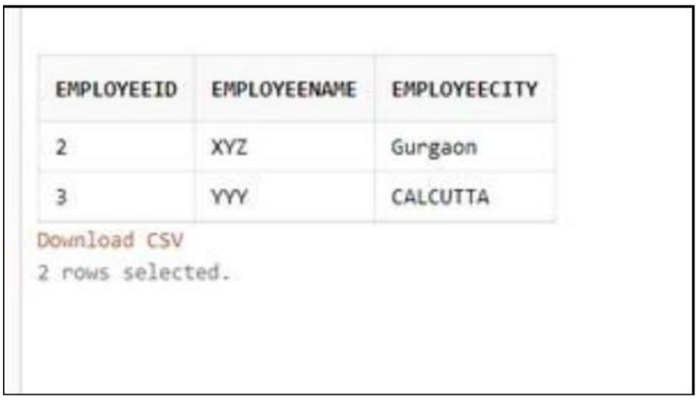
Download CSV

3 rows selected.

%FOUND attribute upon update operation and then performing delete operation example

```
CREATE TABLE tempory_employee2 AS SELECT * FROM employees; DECLARE

employeeNo NUMBER(4) := 2; BEGIN

UPDATE tempory_employee2 SET employeeCity = 'Gurgaon' WHERE employeeId = employeeNo;

IF SQL%FOUND THEN -- update succeeded

DELETE FROM tempory_employee2 WHERE employeeId = 1 ;

-- Then delete a specific row

END IF;
```

Output:



EMPLOYEEID	EMPLOYEENAME	EMPLOYEECITY
2	XYZ	Gurgaon
3	YYY	CALCUTTA

Download CSV

2 rows selected.

After doing above operations

%ISOPEN Attribute: For Implicit Cursors, always the result is False. The reason is Oracle closes immediately after executing the DML result. Hence the result is FALSE.

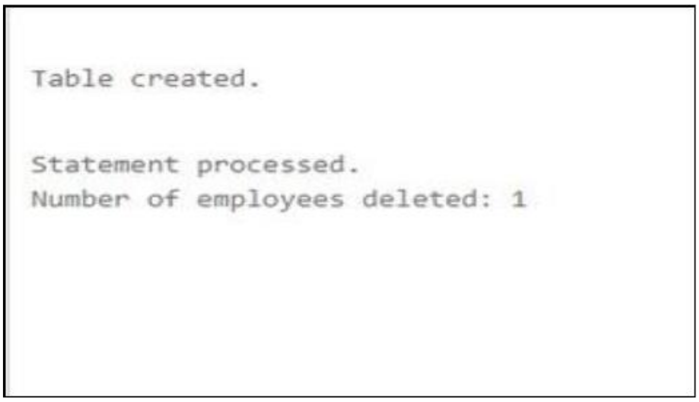
%NOTFOUND Attribute: It is just the opposite of %FOUND. %NOTFOUND is the logical opposite of %FOUND. %NOTFOUND results in TRUE value for an INSERT, UPDATE, or DELETE statement which affected no rows. By default, it returns False.

%ROWCOUNT Attribute: A number of rows affected by an INSERT, UPDATE or DELETE statement are given by %ROWCOUNT. When there are no rows are affected,

%ROWCOUNT gives 0 as the result, otherwise, it returns the number of rows that have been deleted.

```
CREATE TABLE tempory_employee3 AS SELECT * FROM employees; DECLARE employeeNo NUMBER(4) := 2;

BEGIN

DELETE FROM tempory_employee3 WHERE employeeId = employeeNo ;

DBMS_OUTPUT.PUT_LINE('Number of employees deleted: ' || TO_CHAR(SQL%ROWCOUNT));

END;
```

Output:



```
Table created.

Statement processed.
Number of employees deleted: 1
```

The values of the cursor attribute have to be saved in a local variable and those variables can be used in future uses. The reason is while doing multiple database operations in different blocks, cursor attribute values keep on changing and hence this is much required.

The %NOTFOUND attribute is better used only with DML statements but not with SELECT INTO statement.

RELATIONAL ALGEBRA:

Relational algebra is a set of basic operations used to manipulate the data in relational

model. These operations enable the user to specify basic retrieval request. The result of

retrieval is anew relation, formed from one or more relation. These operation can be

classified in two categories.

Basic/Fundamental Operations:

1.Select (σ)

2. Project (∏)

3. Union (∪)

4. Set Difference (-)

5. Cartesian product (X)

6. Rename (ρ)

Select Operation (σ) :

This is used to fetch rows (tuples) from table(relation) which satisfies a given condition.

Syntax: σp(r)

σ is the predicate

r stands for relation which is the name of the table

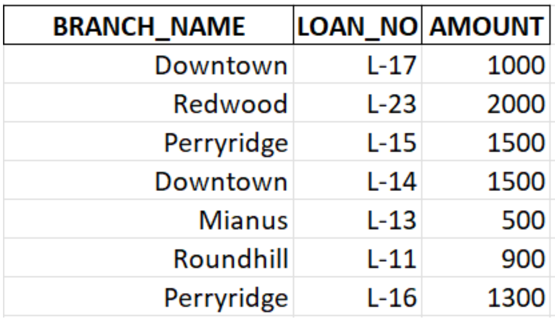
p is prepositional logic

ex: σage > 17 (Student)

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

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

This will return tuples(rows) from table **Student** with information of male students, of age more than 17.

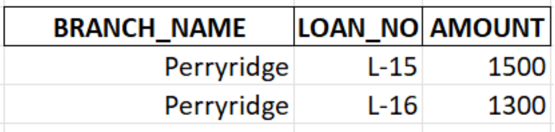


BRANCH_NAME	LOAN_NO	AMOUNT
Downtown	L-17	1000
Redwood	L-23	2000
Perryridge	L-15	1500
Downtown	L-14	1500
Mianus	L-13	500
Roundhill	L-11	900
Perryridge	L-16	1300

Input:

σ BRANCH_NAME="perryride" (LOAN)

Output:



BRANCH_NAME	LOAN_NO	AMOUNT
Perryridge	L-15	1500
Perryridge	L-16	1300

Project Operation (Π):

 Project operation is used to project only a certain set of attributes of a relation. In simple words, If you want to see only the **names** all of the students in the **Student** table, then you can use Project Operation.

 It will only project or show the columns or attributes asked for, and will also remove duplicate data from the columns.

Syntax of Project Operator (Π)

Π column_name1, column_name2, ..., column_nameN(table_name)

Example:

ΠName, Age(Student)

Above statement will show us only the **Name** and **Age** columns for all the rows of data in **Student** table.

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:

Π NAME, CITY (CUSTOMER)

Output:

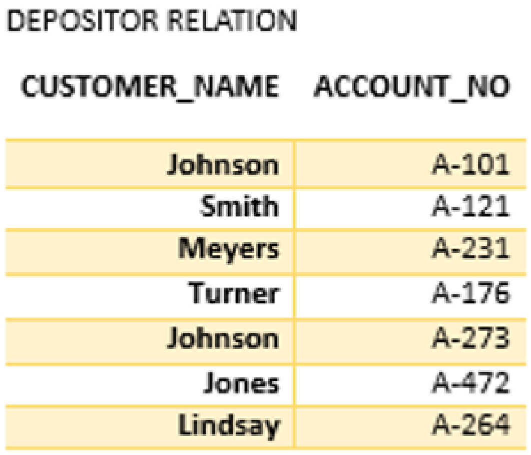
NAME	CITY
Jones	Harrison
Smith	Rye
Hays	Harrison
Curry	Rye
Johnson	Brooklyn
Brooks	Brooklyn

Union Operation (∪):

- This operation is used to fetch data from two relations(tables) or temporary relation(result of another operation).
- For this operation to work, the relations(tables) specified should have same number of attributes(columns) and same attribute domain. Also the duplicate tuples are autamatically eliminated from the result.

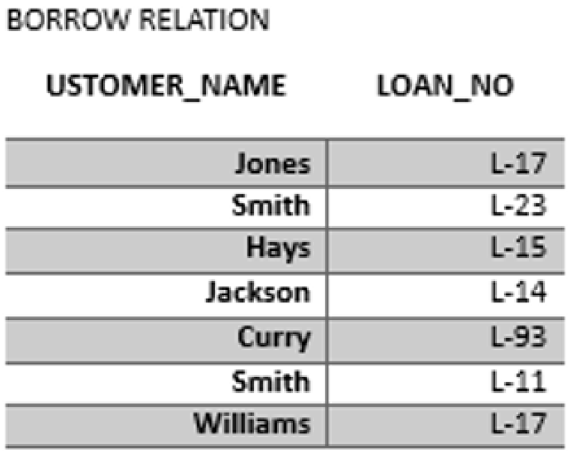
Syntax: A ∪ B

Example: ΠStudent(RegularClass) ∪ ΠStudent(ExtraClass)



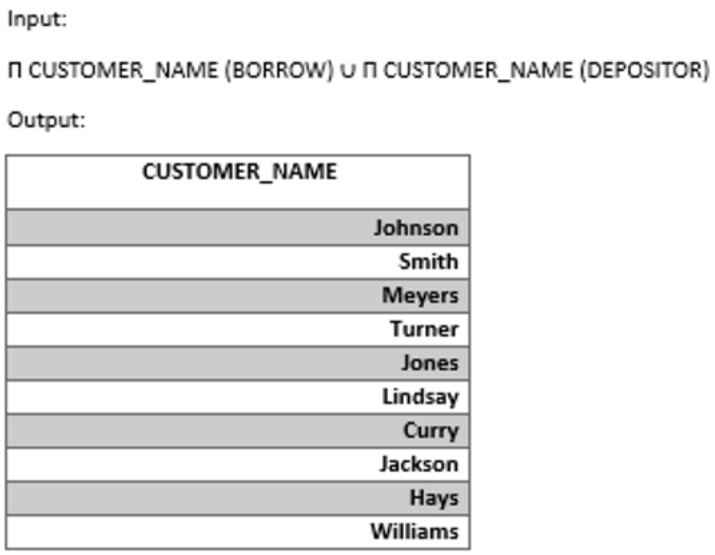
DEPOSITOR RELATION

CUSTOMER_NAME	ACCOUNT_NO
Johnson	A-101
Smith	A-121
Meyers	A-231
Turner	A-176
Johnson	A-273
Jones	A-472
Lindsay	A-264



BORROW RELATION

USTOMER_NAME	LOAN_NO
Jones	L-17
Smith	L-23
Hays	L-15
Jackson	L-14
Curry	L-93
Smith	L-11
Williams	L-17



Input:

Π CUSTOMER_NAME (BORROW) ∪ Π CUSTOMER_NAME (DEPOSITOR)

Output:

CUSTOMER_NAME
Johnson
Smith
Meyers
Turner
Jones
Lindsay
Curry
Jackson
Hays
Williams

Set Difference (-):

This operation is used to find data present in one relation and not present in the second relation. This operation is also applicable on two relations, just like Union operation.

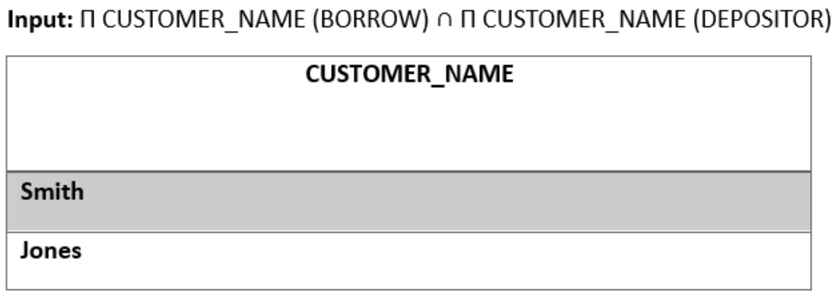
Syntax: A - B

where A and B are relations.

For example, if we want to find name of students who attend the regular class but not the extra class, then, we can use the below operation:

ΠStudent(RegularClass) - ΠStudent(ExtraClass)

Example:



Input: Π CUSTOMER_NAME (BORROW) ∩ Π CUSTOMER_NAME (DEPOSITOR)

CUSTOMER_NAME
Smith
Jones

Cartesian Product (X):

This is used to combine data from two different relations(tables) into one and fetch data from the combined relation.

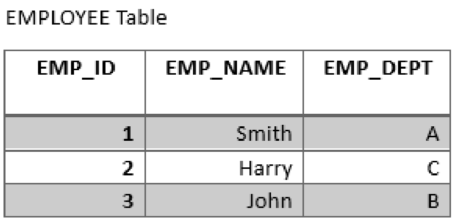
Syntax: A X B

For example, if we want to find the information for Regular Class and Extra Class which are conducted during morning, then, we can use the following operation:

σtime = 'morning' (RegularClass X ExtraClass)

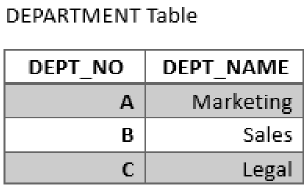
For the above query to work, both **RegularClass** and **ExtraClass** should have the attribute **time**.

Notation: E X D



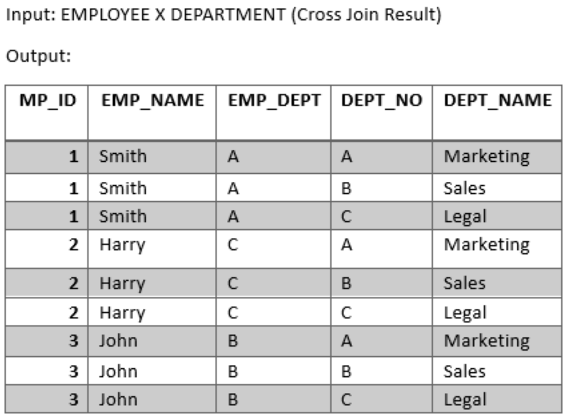
EMPLOYEE Table

EMP_ID	EMP_NAME	EMP_DEPT
1	Smith	A
2	Harry	C
3	John	B



DEPARTMENT Table

DEPT_NO	DEPT_NAME
A	Marketing
B	Sales
C	Legal



Input: EMPLOYEE X DEPARTMENT (Cross Join Result)

Output:

MP_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 Operation (ρ):

This operation is used to rename the output relation for any query operation which returns result like Select, Project etc. Or to simply rename a relation(table)

Syntax: ρ(RelationNew, RelationOld)

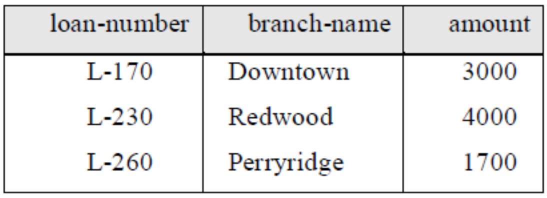
The rename operation is used to rename the output relation. It is denoted by **rho** (ρ).

Example: We can use the rename operator to rename STUDENT relation to STUDENT1.

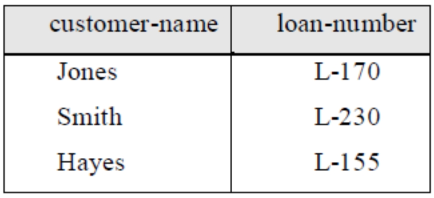
ρ(STUDENT1, STUDENT)

JOINED QUERIES:

SQL provides various other mechanisms for joining relations, including condition joins and natural joins, as well as various forms of outer joins. These additional operations are used as subquery expressions in the from clause. The various join operations are illustrated by using the relations loan and borrower as shown below.



loan-number	branch-name	amount
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

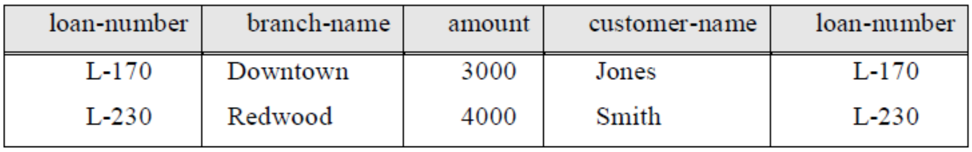


customer-name	loan-number
Jones	L-170
Smith	L-230
Hayes	L-155

A simple example of inner joins is given below.

loan inner join borrower on loan.loan-number = borrower.loan-number

The expression computes the join of the loan and the borrower relations, with the join condition being loan.loan-number = borrower.loan-number. The attributes of the result consist of the attributes of the left-hand-side relation followed by the attributes of the right-hand-side relation as shown below.



loan-number	branch-name	amount	customer-name	loan-number
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230

The result relation of a join and the attributes of the result relation are renamed by using the as clause, as illustrated here:

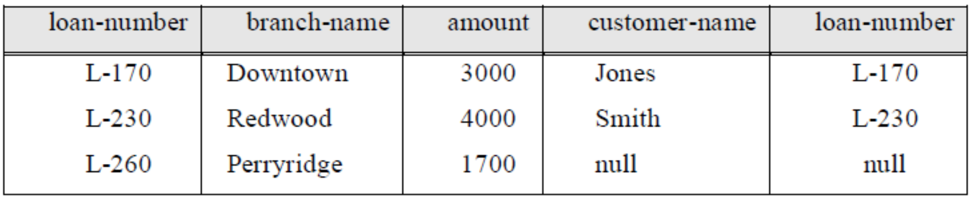
loan inner join borrower on loan.loan-number = borrower.loan-number

as lb (loan-number, branch, amount, cust, cust-loan-num)

Consider an example of the left outer join operation:

loan left outer join borrower on loan.loan-number = borrower.loan-number

The left outer join operation can be computed logically as follows. First, compute the result of the inner join as before. Then, for every tuple t in the left-hand-side relation loan that does not match any tuple in the right-hand-side relation borrower in the inner join, add a tuple r to the result of the join: The attributes of tuple r that are derived from the left-hand-side relation are filled in with the values from tuple t, and the remaining attributes of r are filled with null values. The resultant relation is shown below.

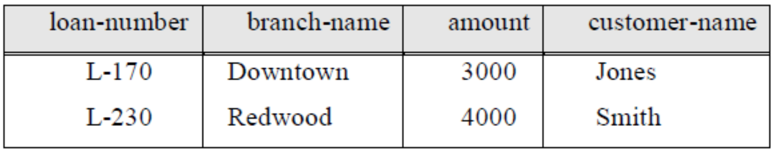


loan-number	branch-name	amount	customer-name	loan-number
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230
L-260	Perryridge	1700	null	null

Consider an example of the natural join operation:

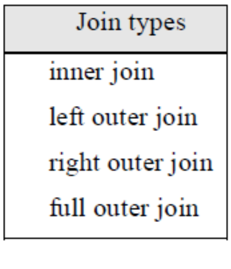
loan natural inner join borrower

This expression computes the natural join of the two relations. The only attribute name common to loan and borrower in loan-number. The result of the expression is shown below. The result is similar to the result of the inner join with the on condition. However, the attribute loan-number appears only once in the result of the natural join, whereas it appears twice in the result of the join with the on condition.

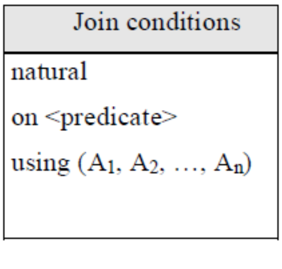


loan-number	branch-name	amount	customer-name
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

Join Types and Conditions: Join operations take two relations and return another relation as the result. Each of the variants of the join operations in SQL consists of a join type and a join condition. The join condition defines which tuples in the two relations match and what attributes are present in the result of the join. The join type defines how tuples in each relation that do not match any tuple in the other relation are treated. The figure given below shows some of the allowed join types and join conditions.



Join types
inner join
left outer join
right outer join
full outer join



Join conditions
natural
on <predicate>
using (A_1, A_2, …, A_n)

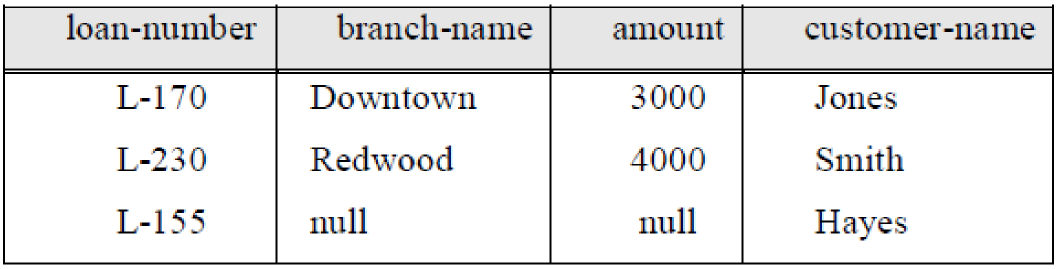
The use of a join condition is mandatory for outer joins, but is optional for inner joins. Syntactically, the keyword natural appears before the join type, whereas the on and using conditions appear at the end of the join expression.

The ordering of the attributes in the result of a natural join is as follows. The join attributes (i.e., the attributes common to both relations) appear first, in the order in which they appear in the left-hand-side relation. Next come all non-join attributes of the left-hand-side relation, and finally all non-join attributes of the right-hand-side relation.

The right outer join is symmetric to the left outer join. Tuples from the right-hand-side relation that do not match any tuple in the left-hand-side relation are padded with nulls and are added to the result of the right outer join. An example of combining the natural join condition with the right outer join type is as follows:

loan natural right outer join borrower

The result of this expression is shown below.

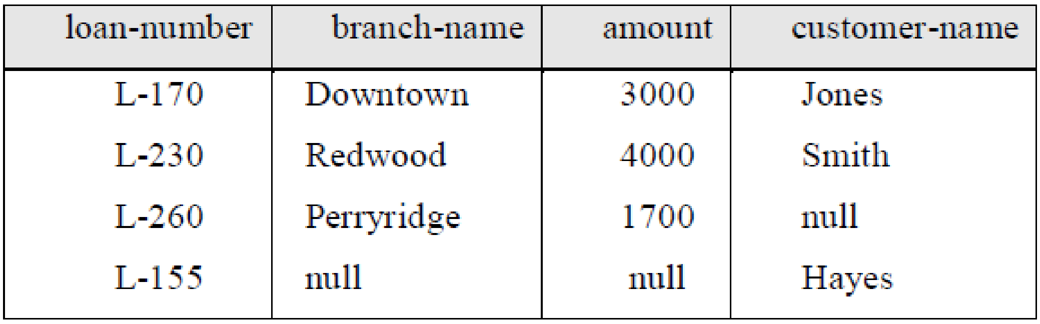


loan-number	branch-name	amount	customer-name
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	null	null	Hayes

The join condition using (A1, A2, …, An) is similar to the natural join condition, except that the join attributes are the attributes A1, A2, …, An, rather than all attributes that are common to both relations. The attributes A1, A2, …, An must consist of only attributes that are common to both relations, and they appear only once in the result of the join.

The full outer join is a combination of the left and right outer join types. After the operation computes the result of the inner join, it extends with nulls tuples from the left-hand-side relation that did not match with any from the right-hand-side, and adds them to the result. Similarly, it extends with nulls tuples from the right-hand-side relation that did not match with any tuples from the left-hand-side relation and adds them to the result. For example, the figure given below shows the result of the expression

loan full outer join borrower using (loan-number)



loan-number	branch-name	amount	customer-name
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	null
L-155	null	null	Hayes

As another example of the use of the outer-join operation, the query “Find all customers who have an account but no loan at the bank” can be written as

```
select d-CN
    from (depositor left outer join borrower
    on depositor.customer-name = borrower.customer-name)
    as dbl (d-CN, account-number, b-CN, loan-number)
where b-CN is null
```

Similarly, the query “Find all customers who have either an account or a loan (but not both) at the bank” can be written with natural full outer joins as:

select customer-name

from (depositor natural full outer join borrower)

where account-number is null or loan-number is null

SQL-92 also provides two other join types, called cross join (inner join without a join condition) and union join (full outer join on the “false” condition – i.e., where the inner join is empty).

TUPLE RELATION CALCULUS

Tuple Relational Calculus (TRC) is a non-procedural query language used in relational databases. It allows users to specify what data they want to retrieve without describing how to retrieve it. TRC is one of the foundational concepts in the theory of relational databases and provides a way to express queries using logical predicates.

Key Concepts of Tuple Relational Calculus

1. **Tuple Variables**: In TRC, queries are expressed using tuple variables, which range over the tuples of a relation. A tuple variable is essentially a placeholder for each row of a relation.
2. **Formulae**: The conditions or constraints that tuples must satisfy are expressed using logical formulae. These formulae are based on predicate logic and are used to filter tuples from relations.
3. **Query Form**: A TRC query typically has the form:

{T∣P(T)}\{ T \mid P(T) \}{T∣P(T)}

where TTT is a tuple variable and P(T)P(T)P(T) is a predicate or condition that TTT must satisfy. The result of the query is the set of tuples that meet the condition P(T)P(T)P(T).

Example of Tuple Relational Calculus Query

Consider a table employees with the following schema and data:

Table: employees

EmpID	Name	Department	Salary
1	Alice	HR	70000
2	Bob	IT	80000
3	Charlie	IT	85000
4	David	HR	72000
5	Eve	Finance	75000

1. Retrieve Names of Employees in the IT Department

Let's formulate a TRC query to find the names of employees who work in the "IT" department.

TRC Query:

{T.Name∣Employees(T)∧T.Department="IT}

Explanation:

- **Tuple Variable**: TTT ranges over the tuples in the Employees table.
- **Condition**: T.Department="IT"T.Department = \text{"IT"}T.Department="IT" selects tuples where the department is "IT".
- **Projection**: T.NameT.NameT.Name retrieves the Name attribute from those tuples.

Result:

Name
Bob
Charlie

2. Retrieve Names of Employees with Salary Greater Than 72000

Now, let's formulate a TRC query to find the names of employees whose salary is greater than 72000.

TRC Query:

{T.Name∣Employees(T)∧T.Salary>72000}

Explanation:

- **Tuple Variable**: TTT ranges over the tuples in the Employees table.
- **Condition**: T.Salary>72000T.Salary > 72000T.Salary>72000 selects tuples where the salary is greater than 72000.
- **Projection**: T.NameT.NameT.Name retrieves the Name attribute from those tuples.

Result:

Name
Charlie
David

3. Retrieve Names and Departments of Employees with Salary Between 70000 and 80000

Let's formulate a TRC query to retrieve names and departments of employees whose salary is between 70000 and 80000 (inclusive).

TRC Query:

{T.Name,T.Department∣Employees(T)∧70000≤T.Salary≤80000}

Explanation:

- **Tuple Variable**: TTT ranges over the tuples in the Employees table.
- **Condition**: 70000≤T.Salary≤8000070000 \leq T.Salary \leq 8000070000≤T.Salary≤80000 selects tuples where the salary is between 70000 and 80000.
- **Projection**: T.Name,T.DepartmentT.Name, T.DepartmentT.Name,T.Department retrieves the Name and Department attributes from those tuples.

Result:

Name	Department
Alice	HR
Bob	IT
xDavid	HR

Key Features of Tuple Relational Calculus

1. **Declarative Nature**: TRC allows users to specify what data to retrieve, not how to retrieve it. This is in contrast to procedural query languages like SQL, where users specify the steps to obtain the data.

2. **Expressiveness**: TRC can express a wide range of queries, including complex conditions and joins, using logical operations such as conjunction (AND), disjunction (OR), and negation (NOT).
3. **Foundation for Relational Query Languages**: TRC provides a theoretical foundation for relational query languages. SQL, for example, is influenced by the principles of TRC.

Comparison with Domain Relational Calculus

Tuple Relational Calculus is often compared to Domain Relational Calculus (DRC), another form of relational calculus. The main difference between TRC and DRC is:

- **TRC** uses tuple variables that range over entire tuples.
- **DRC** uses domain variables that range over individual attribute values.

Example of Domain Relational Calculus Query

For the same Students relation, if we want to retrieve the names of students majoring in "Computer Sci" using DRC, the query might look like:

{N∣∃S,M,E (Students(S,N,M,E)∧M="Computer Sci")}\{ N \mid \exists S, M, E \ (\text{Students}(S, N, M, E) \land M = \text{"Computer Sci"}) \}{N∣∃S,M,E (Students(S,N,M,E)∧M="Computer Sci")}

Here:

- NNN is the domain variable representing student names.
- Students(S,N,M,E)\text{Students}(S, N, M, E)Students(S,N,M,E) represents the tuple structure of the Students relation.
- ∃S,M,E\exists S, M, E∃S,M,E indicates that there exist values for these variables that satisfy the condition.

DOMAIN RELATIONAL CALCULUS

Domain Relational Calculus (DRC) is a formal query language used to express queries on relational databases. It is one of the two primary forms of relational calculus, the other being Tuple Relational Calculus (TRC). While TRC operates on entire tuples, DRC works with individual attribute values.

Key Concepts of Domain Relational Calculus

1. **Domain Variables**: In DRC, queries are expressed using domain variables, which range over the possible values of individual attributes rather than over entire tuples.
2. **Formulae**: Conditions or constraints are specified using logical formulae. These formulae describe the criteria that attribute values must satisfy.
3. **Query Form**: A DRC query typically has the form:

{attribute_list∣∃domain_variables (condition)}

where:

   - attribute_list specifies the attributes to be returned.
   - domain_variables are variables that range over the domains (possible values) of the attributes.
   - condition is a logical predicate that the domain variables must satisfy.

Example Table

Let’s use the same Students table as in the previous example:

Table: Students

StudentID	Name	Major	EnrollmentYear
1	Alice	Computer Sci	2022
2	Bob	Mathematics	2021
3	Charlie	Computer Sci	2023
4	David	Physics	2022
5	Eve	Computer Sci	2024

DRC Query Examples

1. Retrieve Names of Students Majoring in "Computer Sci"

In Domain Relational Calculus, this query can be expressed as:

$\{N \mid \exists S,M,E\ (Students(S,N,M,E) \land M=\text{"Computer Sci"})\}$

Explanation:

- **Domain Variables**: S,N,M,ES, N, M, ES,N,M,E are variables ranging over the domains of StudentID, Name, Major, and EnrollmentYear, respectively.
- **Condition**: The condition M = "Computer Sci" ensures that we select tuples where the major is "Computer Sci".
- **Projection**: We are interested in the Name attribute.

Evaluation:

1. **Select Tuples**: Identify tuples where the major is "Computer Sci".
   - Alice (StudentID: 1, Major: "Computer Sci")
   - Charlie (StudentID: 3, Major: "Computer Sci")
   - Eve (StudentID: 5, Major: "Computer Sci")
2. **Project Names**: Retrieve the Name attribute from these tuples.

Result:

Name
Alice
Charlie
Eve

2. Retrieve Names of Students with a Specific Enrollment Year (e.g., 2022)

To find names of students who enrolled in the year 2022, the DRC query would be:

$\{N \mid \exists S,M,E\ (Students(S,N,M,E) \land E=2022)\}$

Explanation:

- **Domain Variables**: S,N,M,ES, N, M, ES,N,M,E range over the domains of StudentID, Name, Major, and EnrollmentYear.
- **Condition**: The condition E = 2022 filters tuples with the enrollment year 2022.
- **Projection**: We retrieve the Name attribute.

Evaluation:

1. **Select Tuples**: Identify tuples where the enrollment year is 2022.
   - Alice (StudentID: 1, EnrollmentYear: 2022)
   - David (StudentID: 4, EnrollmentYear: 2022)
2. **Project Names**: Retrieve the Name attribute from these tuples.

Result:

Name
Alice
David

ENTITY RELATIONSHIP MODEL:

The entity-relationship (E-R) data model perception of a real world that consists of basic objects called entities, and relationships among these objects. The E-R model employs three basic notions.

- Entity sets
- Relationship sets
- Attributes

Entity Sets, Relationship Sets & Attributes

 Entity Sets: An entity is a “thing” or “object” in the real world that is distinguishable from all other objects. For example a person is an entity.

o An entity has a set of properties and the values for some set of properties may uniquely identify an entity.

o An entity set is a set of entities of the same type that share the same properties or attributes. Entity sets do not need to be disjoint. For example, in a bank enterprise a person may be an employee entity, a customer entity, both or neither.

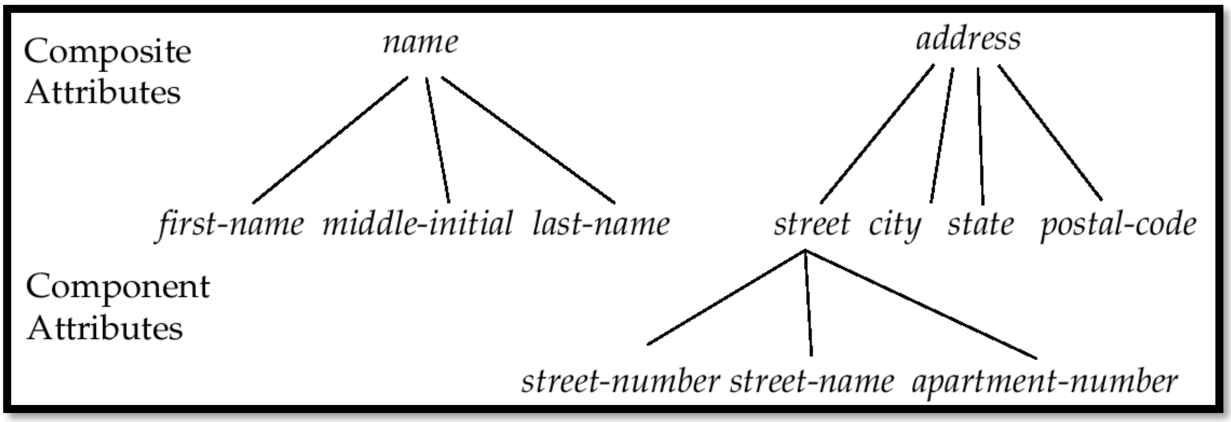
o An entity is represented by a set of attributes, which are descriptive properties possessed by each member of an entity set.

o Each entity has a value for each of its attributes.

o For each attribute there is a set of permitted values, called the domain or value set of that attribute.

The following attribute types can characterize an attribute:

o **Simple & composite attributes:** Simple attributes are attributes which cannot be divided into subparts, whereas composite attributes can be divided into subparts. For example, **c**ustomer-name could be structured as a composite attribute consisting of first-name, middle-initial, and last-name.



o **Single valued and Multi-valued:** An attribute, which has only one value associated with it is called single valued. For example, the loan-number attribute for a specific loan entity refers to only one loan number, and it is a single valued attribute.

 An attribute, which has a set of values for a specific entity, is called multi-valued. For example, the attribute employee-name may have more than one phone numbers, and this attribute is a multi-valued attribute.

o **Derived attribute:** The value for this type of attribute can be derived from the values of other related attributes or entities. For example, deriving age from the date-of-birth attribute.

o **Null attributes:** An attribute takes a null value when an entity does not have a value for it. As an example, if a particular employee has no dependents, the dependent-name value for that employee will be null.

 Relationship Sets: A relationship is an association among several entities. A relationship set is a set of relationships of the same type. If E1, E2, …, En are entity sets, then a relationship set R is a subset of

{(e1, e2, …, en) | e1 E1, e2 E2, …, en En}

where (e1, e2, …, en) is a relationship and n 2.

o The association between entity sets is referred to as participation.

o The function that an entity plays in a relationship is called that entity‟s role.

o **Recursive relationship set**: In this type of relationship set, the same entity set participates in a relationship set more than once, in different roles. For example, consider an entity set employee that records information about all of the employees of the bank. A relationship set works-for is modeled by ordered pairs of employee entities. The first employee of a pair takes the role of manager, whereas the second takes the role of worker.

o A relationship may also have descriptive attributes, which describes the relation between the entity sets.

o The number of entity sets that participate in a relationship set is the degree of the relationship set. A binary relationship set is of degree 2; a ternary relationship set is of degree 3.

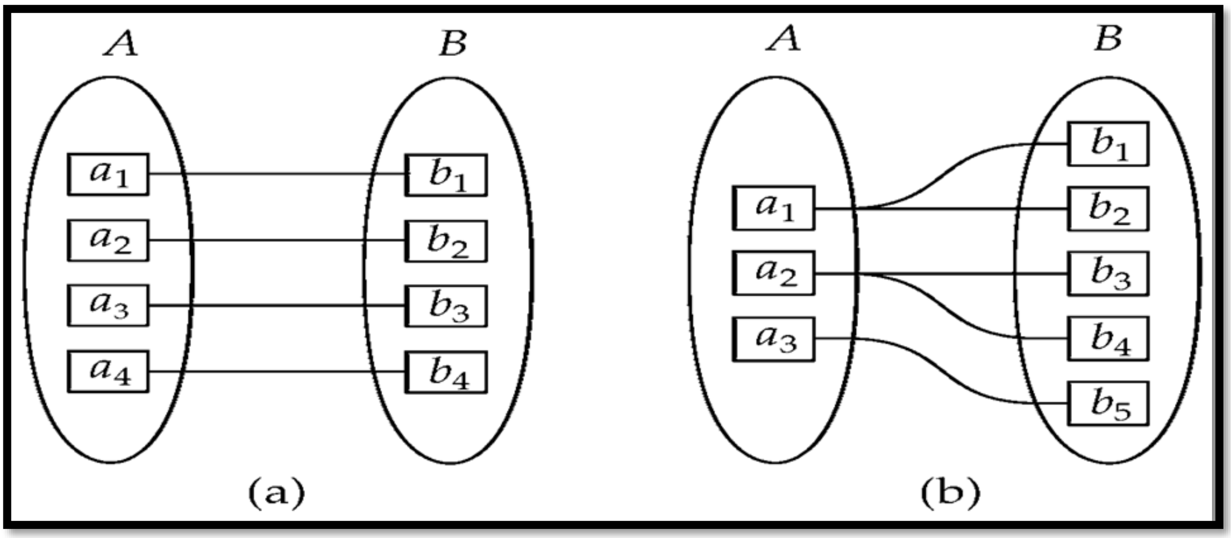
Constraints

An E-R enterprise schema may define certain constrains to which the contents of a database must conform. They are

Mapping Cardinalities: Mapping cardinalities or cardinality ratios, express the number of entities to which another entity can be associated via a relationship set. For a binary relationship set R between entity sets A & B, the mapping cardinality must be one of the following

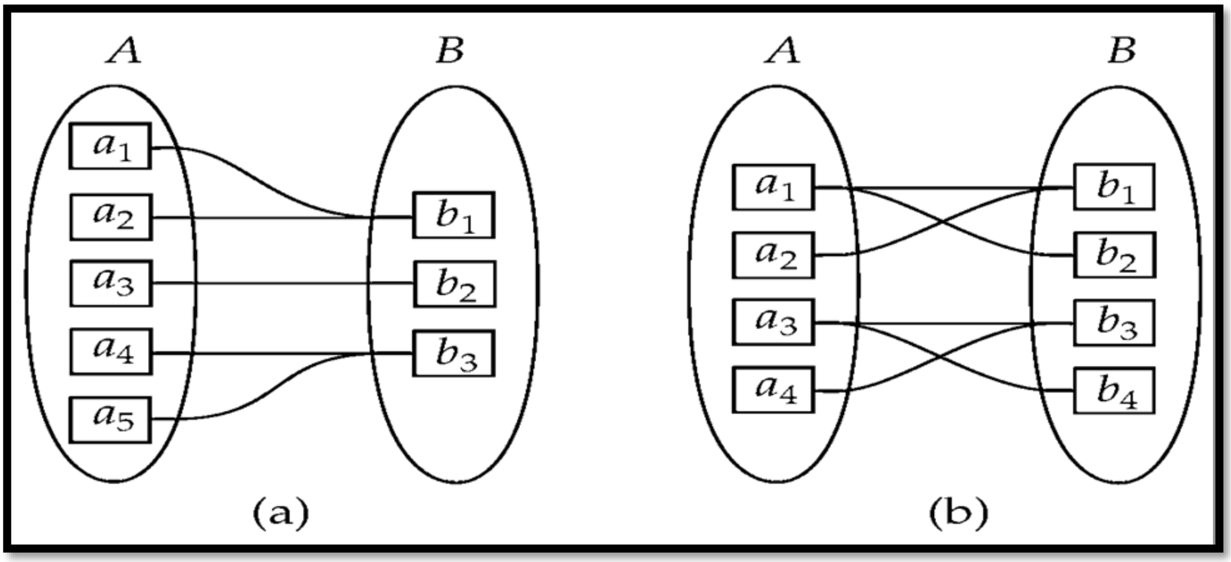
i. **One to one**: An entity in A is associated with at most one entity in B, and an entity in B is associated with at most one entity in A. For example, if in a particular bank, a loan can belong to only one customer, and a customer can avail only one loan a/c, then the relationship set from customer to loan is one to one.

ii. **One to many:** An entity in A is associated with any number of entities in B, An entity in B, however, can be associated with at most one entity in A. For instance, if a customer can be allowed to have any number of accounts and an account should strictly belong to only one customer, then the relationship set from customer to account is one to many.



iii. **Many to one**: An entity in A is associated with at most one entity in B, An entity in B, however, can be associated with any number of entities in A.

iv. **Many to Many:** An entity in A is associated with any number of entities in B, and an entity in B is associated with any number of entities in A. For example, if a loan can belong to several customers and a customer can have several loans, then the relationship set from customer to loan is many to many.



Existence Dependencies: If the existence of entity x depends on the existence of entity y, then x is said to be existence dependent on y. Entity y is said to be a dominant entity, and x is said to be a subordinate entity. Example: Entity sets loan and payment.

The Participation of an entity set E in a relationship set R is said to be total if every entity in E participates in at least one relationship in R. For example, since every payment entity must be related to some loan entity by the loan-payment relationship, the participation of payment in the relationship set loan-payment is total.

If only some entities in E participate in relationships in r, the participation of entity set E in relationship R is said to be partial. Example: An individual can be a bank customer whether or not he / she has a loan with the bank. Hence, it is possible that only a partial set of the customer entities relate to the loan entity set, and the participation of customer in the borrower relationship set is therefore partial.

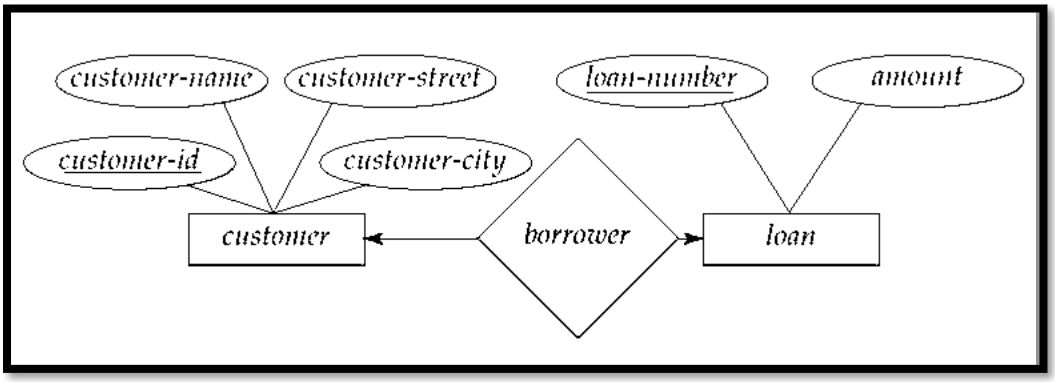
ENTITY–RELATIONSHIP DIAGRAM

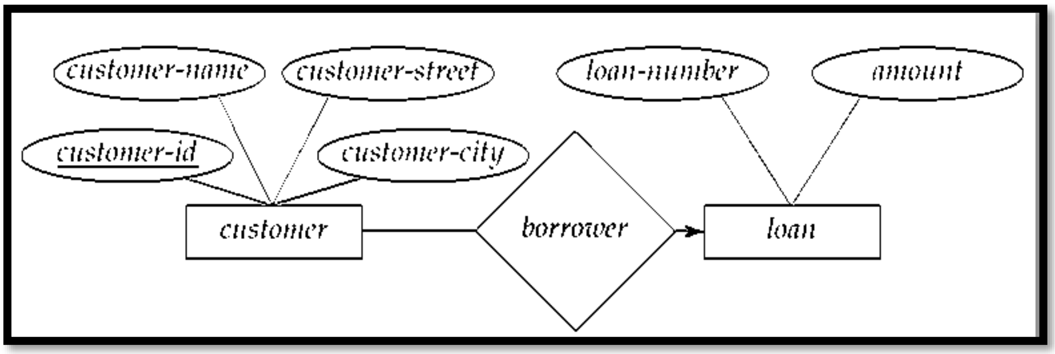
An E-R diagram can express the overall logical structure of a database graphically. Such a diagram consists of the following major components.

- Rectangles, which represent entity set
- Ellipses, which represent attributes
- Diamonds, which represents relationship sets
- Lines, which link attributes to entity sets and entity sets to relationship sets
- Double ellipses, which represents multi-valued attributes
- Dashed ellipses, which denotes derived attributes
- Double lines, which indicate total participation of an entity in a relationship set
- Double rectangles, which represent weak entity sets

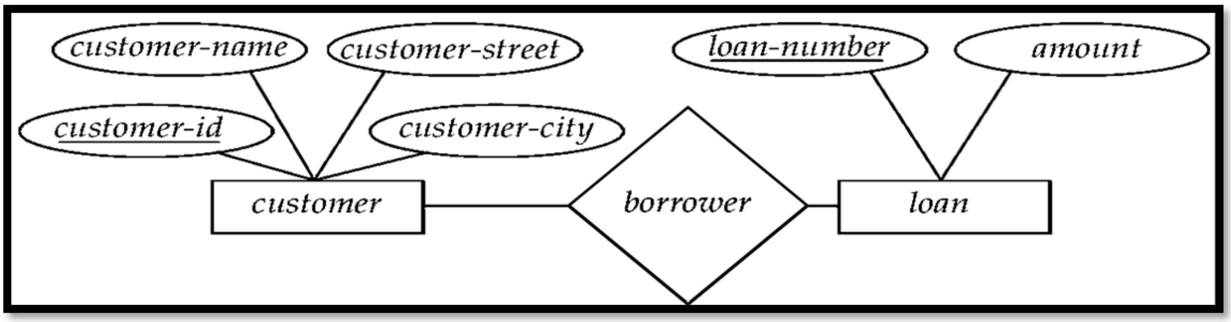
To distinguish among the types of relationship sets,

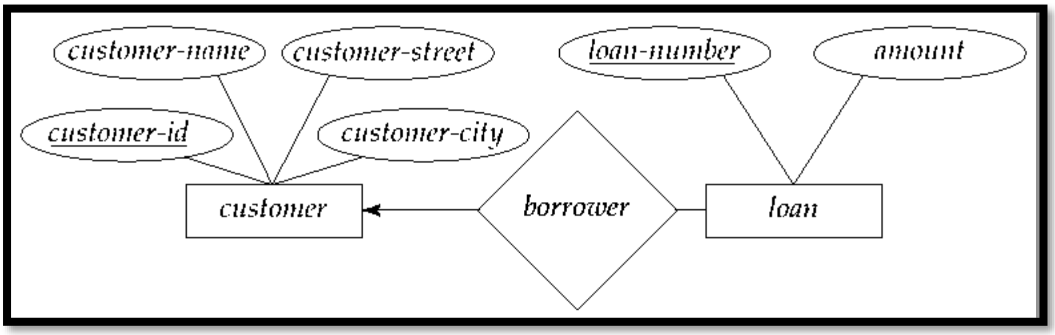
- A directed line from the relationship set borrower to the entity set loan is drawn specifying that it is a one-to-one or many-to-one relationship set.



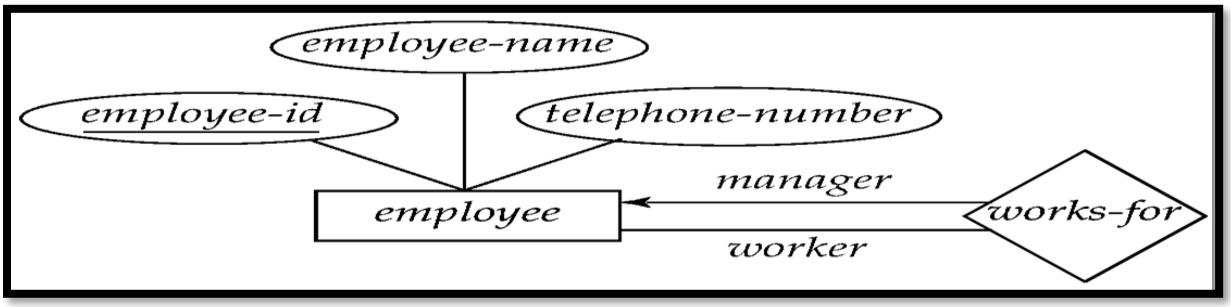


- An undirected line from the relationship set borrower to the entity set loan is drawn specifying that it is either a many-to-many or one-to many relationship set.



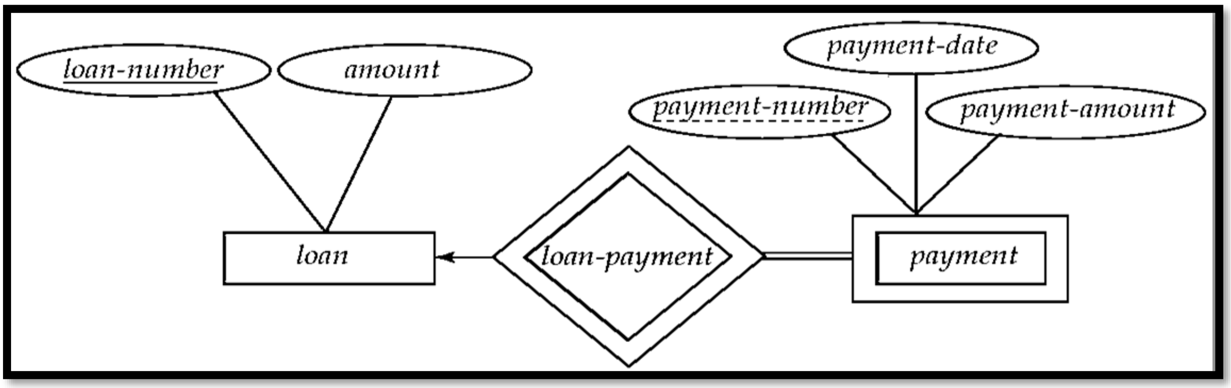


- Roles can also be indicated in the E-R diagrams by labeling the lines that connect relationship sets and entities.



Weak Entity Sets

- An entity set, which may not have sufficient attributes to form a primary key, is termed a weak entity set. An entity set that has a primary key is a strong entity set.
- For a weak entity set to be meaningful, it must be associated with another entity set called the identifying owner entity set. Every weak entity must be associated with an identifying entity i.e. the weak entity set is said to be existence dependant on the identifying entity set.
- The identifying entity set is said to own the weak entity set that it identifies.
- The relationship associating the weak entity set with the identifying entity set is called the identifying relationship.
- Although a weak entity set does not have a primary key, the discriminator of a weak entity set is a set of attributes that allows this distinction to be made.
- The discriminator of a weak entity set is called the partial key of the entity set.
- The primary key of a weak entity set is formed by the primary key of the strong entity set on which the weak entity set is existence dependent, plus the weak entity set‟s discriminator.
- Example: payment entity, it depends on the loan entity set.



DESIGN ISSUES

The following are the basic issues involved in the design of an E-R database schema.

- **Use of Entity Sets or Attributes:** The distinctions mainly depend on the structure of the real-world enterprise being modeled and on the semantics associated with the attribute.
- **Use of Entity sets Vs Relationship sets**: One possible guideline in determining whether to use an entity set or a relationship set is to designate a relationship set to describe an action that occurs between entities. This approach can also be useful in deciding whether certain attributes may be more appropriately expressed as relationships.
- **Binary Vs n-ary relationship Sets:** It is always possible to replace a non-binary (n-ary, for n>2) relationship set by a number of distinct binary relationship sets. Thus conceptually we can restrict the E-R model to include only binary relationship sets.
- **Placement of relationship attributes:** The cardinality ratio of a relationship can affect the placement of relationship attributes. Thus, attributes of one-to-one or one-to-many

relationship sets can be associated with one of the participating entity sets, rather than with the relationship set.

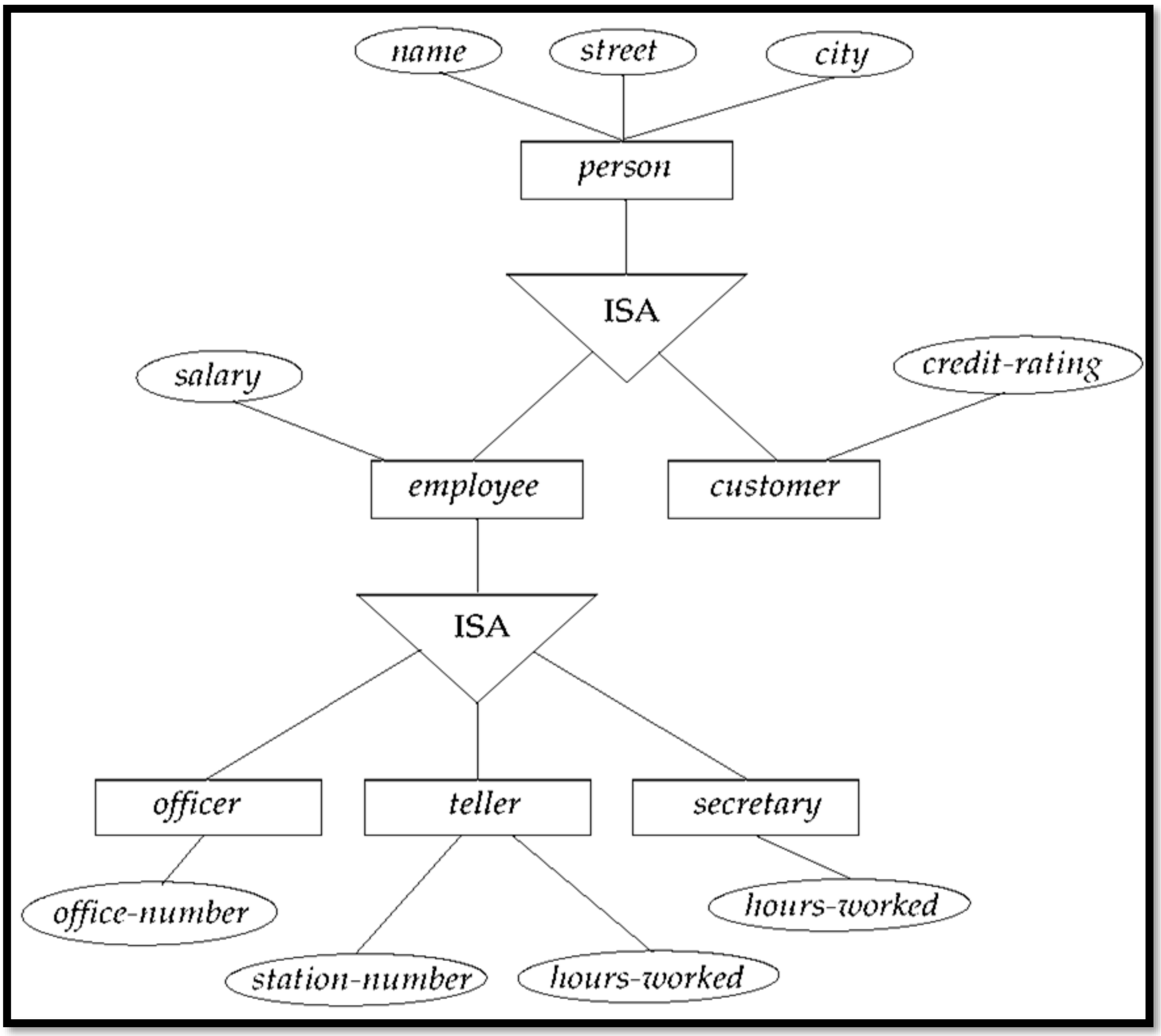
EXTENDED ER FEATURES:

Some aspects of a database may be more aptly expressed by certain extensions to the basic E-R model. The various extended E-R features are:

Specialization:

o An entity set may include sub groupings of entities that are distinct in some way from other entities in the set. The process of designating subgroups within an entity set is called specialization.

o In terms of an E-R diagram, serialization is depicted by a triangle component labeled ISA (stands for “is a”) The ISA relationship may also be referred to as a super class-subclass relationship.



Generalization:

o Generalization proceeds from the recognition that a number of entity sets share some common features that are described by the same attributes and participate in the same relationship sets. Based on their commonalities, generalization synthesizes these entity sets into a single, higher-level entity set.

o Generalization is used to emphasize the similarities among lower-level entity sets and to hide the differences.

o Generalization is a simple inversion of specialization. Specialization is a top-down approach whereas generalization is a bottom-up approach in which multiple entity sets are synthesized into a higher-level entity set based on common features.

Attribute Inheritance:

o A crucial property of the higher & lower level entities created by specialization and generalization is attribute inheritance. The attributes of the higher-level entity sets are said to be inherited by lower-level entity set.

o In a hierarchy, a given entity set may be involved as a lower-level entity set in only one ISA relationship i.e. single inheritance.

o If an entity set is a lower-level entity in more than one ISA relationship then the entity set has multiple inheritance and the resulting structure is said to be a LATTICE.

Constraints on Generalization: One type of constraint involves determining which entities can be members of a given lower-level entity set. Such membership may be one of the following.

o **Condition-defined**: In condition-defined lower-level entity sets, membership is evaluated on the basis of whether or not an entity satisfies an explicit condition or predicate.

o **User-defined**: User-defined lower-level entity sets are not constrained by a membership condition; rather, entities are assigned to a given entity set by the database user.

A second type of constraint relates to whether or not entities may belong to more that one lower-level entity set within a single generalization. The lower-level entity set may be one of the following:

o **Disjoint:** A disjointness constraint requires that an entity belong to no more than one lower-level entity set.

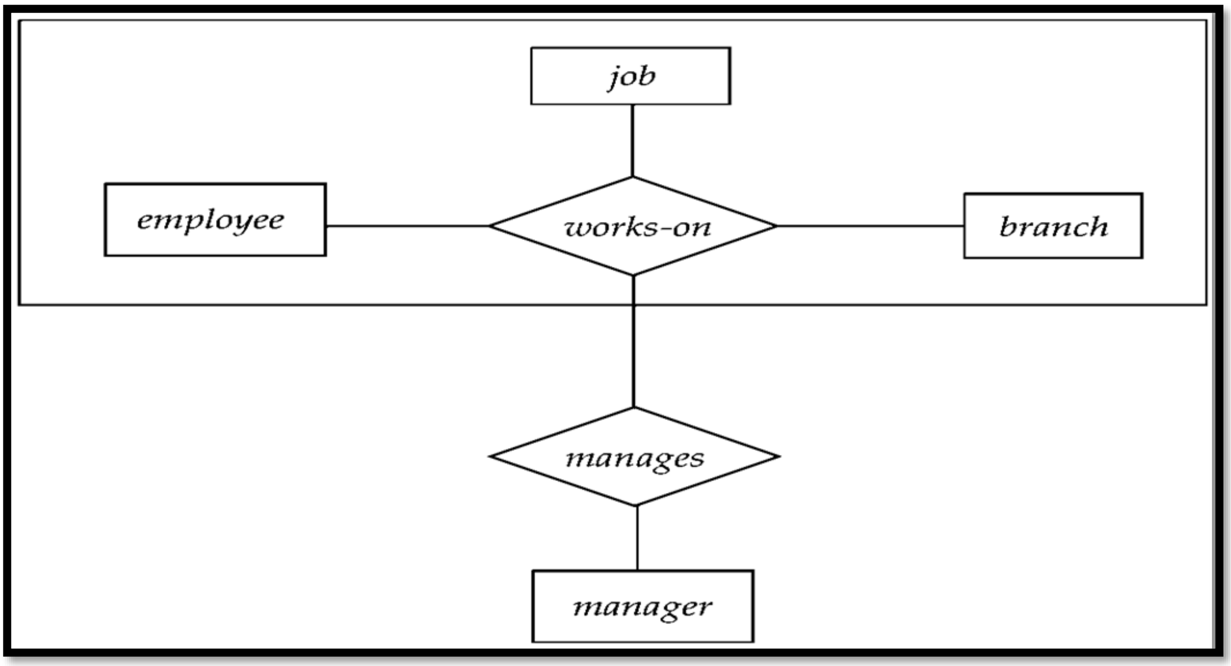
o **Overlapping:** In overlapping generalizations, the same entity may belong to more than one lower-level entity set within a single generalization.

A final constraint, the completeness constraint, specifies whether or not an entity in the higher-level entity set must belong to at least one of the lower-level entity sets within a generalization. This constraint may be one of the following:

o Total: Each higher-level entity must belong to a lower-level entity set.

o Partial: Some higher-level entities may not belong to any lower-level entity set.

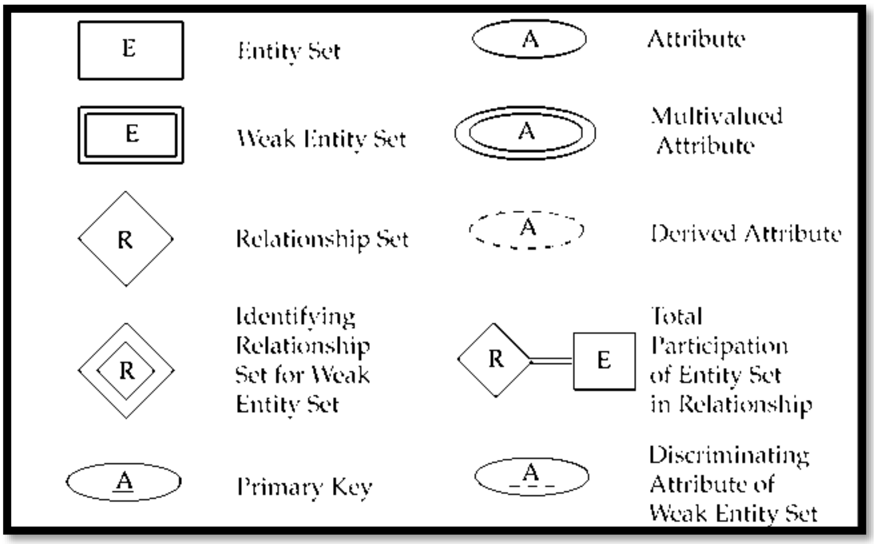
Aggregation: Aggregation is an abstraction through which relationships are treated as higher-level entities. Thus, the relationship set *works-on* (relating the entity sets *employee*, *branch*, and *job*) as a higher-level entity set. Then this entity set *works-on* is related with the *manager* entity set using the *manages* relationship set.

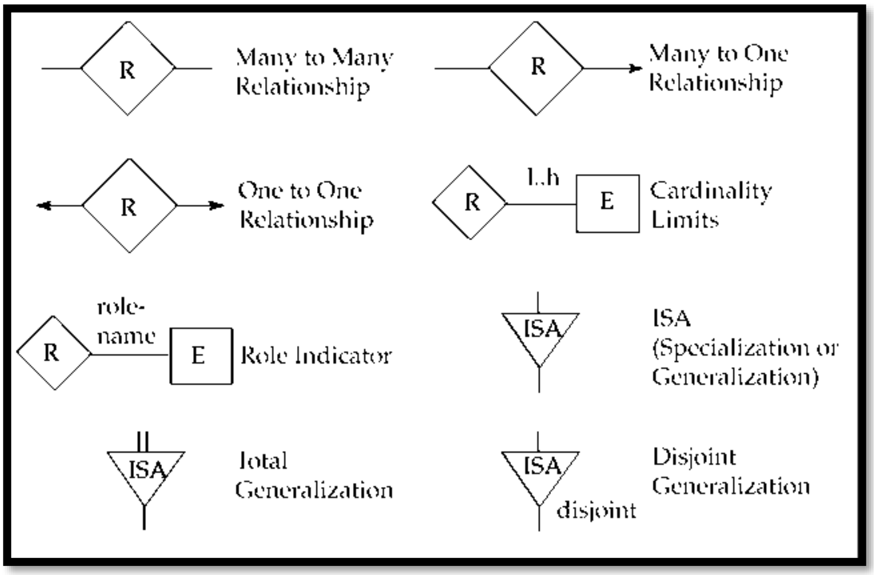


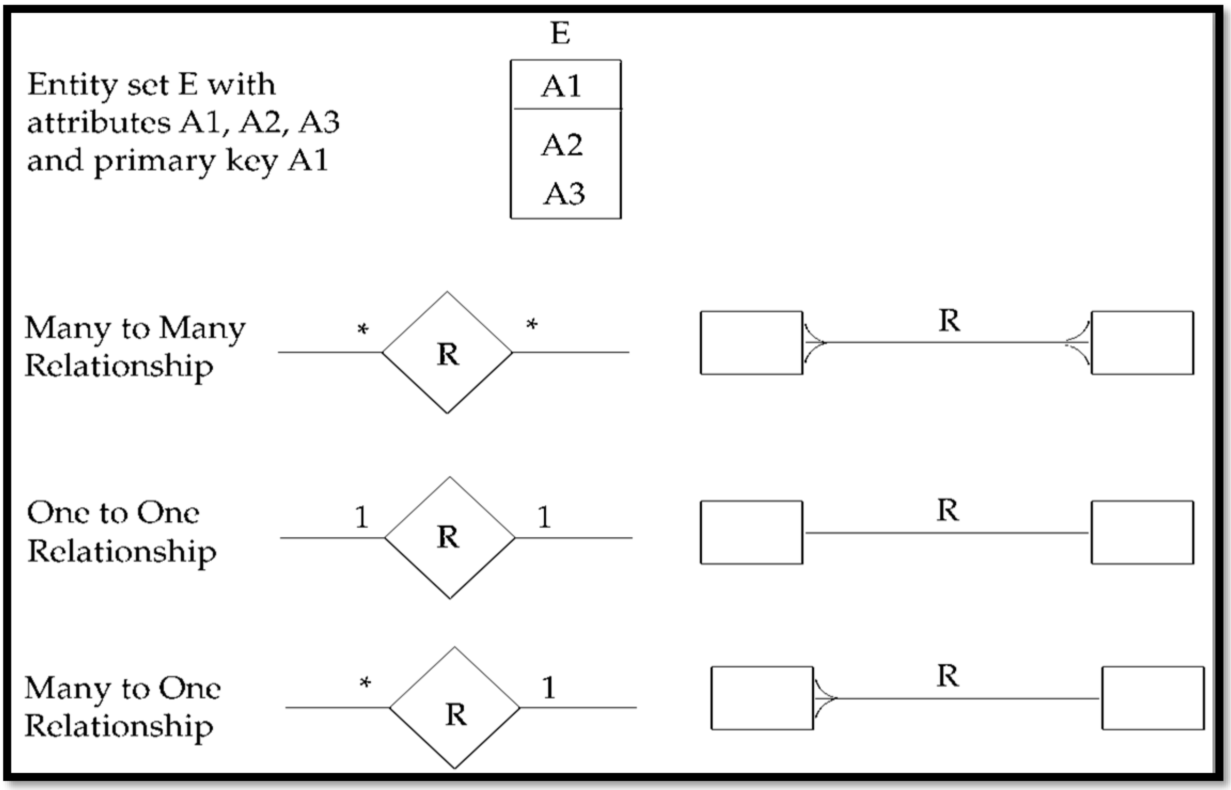
Alternative E–R Notations: The figure given below summarizes the set of symbols used in E–R diagrams. Some of the alternative notations that are widely used are also given below.

- An entity set may be represented as a box with the name outside, and the attributes listed one below the other within the box.

- The primary key attributes are indicated by listing them at the top, with a line separating them from the other attributes.
- Cardinality constraints can be indicated in several different ways. The labels * and 1 on the edges out of the relationship are sometimes used for depicting many-to-many, one-to-one, and many-to-one (one-to-many) relationships.
- In another alternative notation, binary relationship sets are represented by lines between entity sets, without diamonds.


E
Entity Set
A
Attribute
E
Weak Entity Set
A
Multivalued Attribute
R
Relationship Set
A
Derived Attribute
R
Identifying Relationship Set for Weak Entity Set
R
E
Total Participation of Entity Set in Relationship
A
Primary Key
A
Discriminating Attribute of Weak Entity Set




E
Entity set E with attributes A1, A2, A3 and primary key A1
A1
A2
A3
Many to Many Relationship
*
R
*
R
One to One Relationship
1
R
1
R
Many to One Relationship
*
R
1
R

MAPPING ER MODEL TO RELATIONAL MODEL:

A database that conforms to an E-R database schema can be represented by a collection of tables. For each entity set and for each relationship set in the database, there is a unique table that is assigned the name of the corresponding entity set or relationship set. Each table has multiple columns, each of which has a unique name.

Tabular representation of strong Entity sets: Let E be a strong entity set with descriptive attributes a1, a2, …, an. We represent this entity by a table called E with n distinct columns, each of which corresponds to one of the attributes of E. Each row in this table corresponds to one entity of the entity set E. For example, the entity set loan is represented as a table with name loan consisting of attributes loan-number and amount with loan-number as the primary key.

Tabular representation of weak Entity sets: Let A be a weak entity set with attributes a1, a2, …, am. Let B be the strong entity set on which A is dependent. Let the primary key of B consist of attributes b1, b2, …, bn. We represent the entity set A by a table called A with one column for each attribute of the set:

{a1, a2, …, am} {b1, b2, …., bn}

As an example, the weak entity set payment can be represented as a table with four columns labeled loan-number, payment-number, payment-date and payment-amount with loan-number and payment-number as a composite primary key.

Tabular Representation of Relationship sets: Let R be a relationship set, let a1, a2, …, am be the set of attributes formed by the union of primary keys of each of the entity sets participating in R, & let the descriptive attributes (if any) of R be b1, b2, …, bn. We represent this relationship set by a table called R with one column for each attribute of the set:

{a1, a2, …, am} {b1, b2, …, bn}

o **Redundancy of tables:** The table for the relationship set linking a weak entity set to its corresponding strong entity set is redundant and does not need to be present in a tabular representation of an E-R diagram.

o **Combination of tables:** If there is a existence dependency of A on B in the relationship set AB, then we combine the tables A and AB to form a single table consisting of the union of columns of both tables.

Multi-valued Attributes: For a multi-valued attribute, we create a table T with a column C that corresponds to M and columns corresponding to the primary key of the entity set or relationship set of which M is an attribute. For example, the multi-valued attribute dependent-name in the entity set employee can be represented as a table with columns dependent-name and e-social-security, representing the primary key of the entity set employee.

Tabular Representation of Generalization: There are two methods for transforming to a tabular form an E-R diagram that includes generalization.

o Create a table for the higher-level entity set. For each lower-level entity set, create a table that includes a column for each of the attributes of that entity set plus a column for each attribute of the primary key of the higher-level entity set.

o If the generalization is disjoint and complete, then for each lower-level entity set, create a table that includes a column for each of the attributes of that entity set plus a column for each attribute of the higher-level entity set.

Tabular Representation of Aggregation: The table for the relationship set involving aggregation includes a column for each attribute in the primary key of the entity set and the relationship set. It would also include a column for any descriptive attributes, if they existed.

Query Processing:

Query Processing in DBMS

Parsing and Translation

As query processing includes certain activities for data retrieval. Initially, the given user queries get translated in high-level database languages such as SQL. It gets translated into expressions that can be further used at the physical level of the file system. After this, the actual evaluation of the queries and a variety of query -optimizing transformations and takes place.

Query Evaluation Plan

- In order to fully evaluate a query, the system needs to construct a query evaluation plan.
- The annotations in the evaluation plan may refer to the algorithms to be used for the particular index or the specific operations.

- Such relational algebra with annotations is referred to as **Evaluation Primitives**. The evaluation primitives carry the instructions needed for the evaluation of the operation.
- Thus, a query evaluation plan defines a sequence of primitive operations usedfor evaluating a query. The query evaluation plan is also referred to as **the query execution plan**.
- A **query execution engine** is responsible for generating the output of the given query. It takes the query execution plan, executes it, and finally makes the output for the user query.

Optimization

- The cost of the query evaluation can vary for different types of queries. Although the system is responsible for constructing the evaluation plan, the user does need not to write their query efficiently.
- Usually, a database system generates an efficient query evaluation plan, which minimizes its cost. This type of task performed by the database system and is known as Query Optimization.
- For optimizing a query, the query optimizer should have an estimated cost analysis of each operation. It is because the overall operation cost depends onthe memory allocations to several operations, execution costs, and so on.

QUERY–BY–EXAMPLE

Query–By–Example (QBE) is a graphical language, where queries look like tables. QBE is the name of both a data manipulation language and an early database system that included this language.

The QBE database system was developed at IBM's T. J. Watson Research Center in the early 1970s. The QBE data manipulation language was later used in IBM's Query Management Facility (QMF). Today, many database systems for personal computers support variants of QBE language. QBE has two distinctive features:

1. Unlike most query languages and programming languages, QBE has a two-dimensional syntax: Queries look like tables.
2. QBE queries are expressed "by example". Instead of giving a procedure for obtaining the desired answer, the user gives an example of what is desired. The system generalizes this example to compute the answer to the query.

There is a close relationship between QBE and the domain relational calculus. Queries in QBE are expressed by skeleton tables. These tables show the relation schema. The user selects those skeletons needed for a given query and fills in the skeletons with example rows. An example row consists of constants and example elements, which are domain variables. QBE uses an underscore character before domain variables, as in _x, and lets constants appear without any qualification.

Queries on One Relation: The following query tells the system to look for tuples in loan relation that have "Perryridge" as the value for the branch-name attribute. For each such tuple, the system assigns the value of the loan-number attribute to the variable x. It "prints" (displays) the value of the variable x, because the command P. appears in the loan-number column.

loan	loan-number	branch-name	amount
	P._x	Perryridge	

QBE assumes that a blank position in a row contains a unique variable. If a variable does not appear more than once in a query, it may be omitted. The previous query could thus be rewritten as

loan	loan-number	branch-name	amount
	P.	Perryridge	

QBE performs duplicate elimination automatically. To suppress duplicate elimination, insert the command ALL. after the P. command:

loan	loan-number	branch-name	amount
	P.ALL.	Perryridge	

To display the entire loan relation, use a shorthand notation by placing a single P. in the column headed by the relation name:

loan	loan-number	branch-name	amount
P.			

QBE allows queries that involve arithmetic comparisons as in, “Find the loan numbers of all loans with a loan amount of more than $700”:

loan	loan-number	branch-name	amount
	P.		> 700

Comparisons can involve only one arithmetic expression of the right hand side of the comparison operation. The expression can include both variables and constants. The space on the left hand side of the comparison operation must be blank. The arithmetic operations that QBE supports are =, <, ≤, >, ≥, and ¬.

The following example finds the names of all branches that are not located in Brooklyn.

branch	branch-name	branch-city	assets
	P.	¬ Brooklyn	

The primary purpose of variables in QBE is to force values of certain tuples to have the same value on certain attributes. Consider the query “Find the loan numbers of all loans made jointly to Smith and Jones”:

borrower	customer-name	loan-number
	“Smith”	P._x
	“Jones”	_x

As another example, consider the query “Find all customers who live in the same city as Jones”.

customer	customer-name	customer-street	customer-city
	P._x		_y
	“Jones”		_y

Queries on Several Relations: QBE allows queries that span different relations. The connections among the various relations are achieved through variables that force certain tuples to have the same value on certain attributes. The query to find the names of all customers who have a loan from the Perryridge branch can be written as

loan	loan-number	branch-name	amount
	_x	Perryridge	

borrower	customer-name	loan-number
	P._y	_x

To evaluate the preceding query, the system finds tuples in loan with "Perryridge" as the value for the branch-name attribute. For each such tuple, the system finds tuples in borrower with the same value for the loan-number attribute as the loan tuple. It displays the values for the customer-name attribute.

Similarly, the query to find the names of all customers who have both an account and a loan at the bank is shown below.

depositor	customer-name	account-number
	P._x	

borrower	customer-name	loan-number
	_x	

Consider the query "Find the names of all customers who have an account at the bank, but who do not have a loan from the bank". Queries that involve negation is expressed in QBE by placing a not sign (¬) under the relation name and next to an example row:

depositor	customer-name	account-number
	P._x	

borrower	customer-name	loan-number
¬	_x	

QBE finds all x values for which

1. There is a tuple in the depositor relation whose customer-name is the domain variable x.
2. There is no tuple in the borrower relation whose customer name is the same as in the domain variable x.

The ¬ can be read as "there does not exist".

The following query finds all customers who have at least two accounts.

depositor	customer-name	account-number
	P._x	_y
	_x	¬ _y

The Condition Box: It is either inconvenient or impossible to express all the constraints on the domain variables within the skeleton tables. To overcome this difficulty, QBE includes a condition box feature that allows the expression of general constraints over any of the domain variables. QBE allows logical operations to appear in a condition box. The logical operators are the words *and* and *or*, or the symbols "&" and "|".

For example, the query "Find the loan numbers of all loans made to Smith, or Jones (or to both jointly)" can be written as

borrower	customer-name	loan-number
	_n	P._x

conditions
_n = Smith or _n = Jones

The following query finds all account numbers with a balance between $1300 and $1500.

account	account-number	branch-name	balance
	P.		_x

conditions
_x ≥ 1300
_x ≤ 1500

To find all branches that have assets greater than those of at least one branch located in Brooklyn, the query can be written as

branch	branch-name	branch-city	assets
	P._x		_y
		Brooklyn	_z

conditions
_y > _z

The Result Relation: If the result of a query includes attributes from several relation schemas, then there should be a mechanism to display the desired result in a single table. For this purpose, a temporary result relation that includes all the attributes of the result of the query can be declared. The desired result is printed by including the command P. in only the result skeleton table.

To construct the query, which finds the customer-name, account-number, and balance for all accounts at the Perryridge branch, proceed as follows:

1. Create a skeleton table, called result, with attributes customer-name, account-number, and balance. The name of the newly created skeleton table must be different from any of the previously existing database relation names.
2. Write the query.

The resulting query is

account	account-number	branch-name	balance
	_y	Perryridge	_z

depositor	customer-name	account-number
	_x	_y

result	customer-name	account-number	balance

P.	_x	_y	_z

Ordering of the Display of Tuples: QBE offers the user control over the order in which tuples in a relation are displayed, by inserting either the command AO. (ascending order) or the command DO. (descending order) in the appropriate column.

The query to list all customers in ascending alphabetic order who have an account at the bank, can be written as

depositor	customer-name	account-number
	P.AO.	

QBE provides a mechanism for sorting and displaying data in multiple columns, by including with each sort operator (AO or DO), an integer surrounded by parentheses. The following query lists all account numbers at the Perryridge branch in ascending alphabetic order with their respective account balances in descending order.

account	account-number	branch-name	balance
	P.AO (1).	Perryridge	P.DO (2).

The command P.AO (1). specifies that the account number should be sorted first; the command P.DO (2). specifies that the balance for each account should then be sorted.

Aggregate Operations: QBE includes the aggregate operators AVG, MAX, MIN, SUM, and CNT. Postfix these operators with ALL. to create a multiset on which the aggregate operation is evaluated. The ALL. operator ensures that duplicates are not eliminated.

The query to find the total balance of all the accounts maintained at the Perryridge branch is

account	account-number	branch-name	balance
		Perryridge	P.SUM.ALL.

The operator UNQ is used to specify that duplicates be eliminated. The following query finds the total number of customers who have an account at the bank.

depositor	customer-name	account-number
	P.CNT.UNQ.	

QBE also offers the ability to compute functions on groups of tuples using the G. operator. The query to find the average balance at each branch can be written as

account	account-number	branch-name	balance
		P.G.	P.AVG.ALL._x

The keyword ALL. in the P.AVG.ALL. entry in the balance column ensures that all the balances are considered. To display the branch names in ascending order, replace P.G. by P.AO.G.

To find the average account balance at only those branches where the average account balance is more than $1200, add the following condition box:

conditions
AVG.ALL._x > 1200

As another example, consider the query "Find all customers who have accounts at each of the branches located in Brooklyn":

depositor	customer-name	account-number
	P.G._x	_y

account	account-number	branch-name	balance
	_y	_z	

branch	branch-name	branch-city	assets
	_z	Brooklyn	
	_w	Brooklyn	

conditions
CNT.UNQ._z = CNT.UNQ._w

The domain variable w can hold the value of names of branches located in Brooklyn. Thus, CNT.UNQ._w is the number of distinct branches in Brooklyn. The domain variable z can hold the value of branches in such a way that both of the following hold:

- The branch is located in Brooklyn.
- The customer whose name is x has an account at the branch.

Thus, CNT.UNQ._z is the number of distinct branches in Brooklyn at which customer x has an account. If CNT.UNQ._z = CNT.UNQ._w, then customer x must have an account at all the branches located in Brooklyn. In such a case, the displayed result includes x.

Modification of the Database: QBE also allows adding, removing, or changing information.

Deletion: Deletion of tuples from a relation is expressed in much the same way as a query. The major difference is the use of D. in place of P. QBE deletes whole tuples, as well as values in selected columns. When the information is deleted in only some of the columns, null values are inserted. A D. command operates on only one relation. To delete tuples from several relations, use one D. operator for each relation. Some examples of QBE delete requests are given below:

- Delete customer Smith.

customer	customer-name	customer-street	customer-city
D.	Smith		

- Delete the branch-city value of the branch whose name is “Perryridge.

branch	branch-name	branch-city	assets
	Perryridge	D.	

- Delete all loans with a loan amount between $1300 and $1500.

loan	loan-number	branch-name	amount
D.	_y		_x

borrower	customer-name	loan-number
D.		_y

conditions
_x = (≥ 1300 and ≤ 1500)

- Delete all accounts at all branches located in Brooklyn.

account	account-number	branch-name	balance
D.	_y	_x	

depositor	customer-name	account-number
D.		_y

branch	branch-name	branch-city	assets
	_x	Brooklyn	

Insertion: To insert data into a relation, either specify a tuple to be inserted or write a query whose result is a set of tuples to be inserted. Insertion is done in QBE by placing a I. operator in the query expression. The attribute values for inserted tuples must be members of the attribute's domain.

The following example inserts the fact that account A-9732 at the Perryridge branch has a balance of $700.

account	account-number	branch-name	balance
I.	A-9732	Perryridge	700

The example shown below inserts information into the branch relation about a new branch with name "Capital" and city "Queens", but with a null asset value.

branch	branch-name	branch-city	assets
I.	Capital	Queens	

The following example provides as a gift, for all loan customers of the Perryridge branch, a new $200 savings account for every loan account that they have, with the loan number serving as the account number for the savings account.

account	account-number	branch-name	balance
I.	_x	Perryridge	200

depositor	customer-name	account-number
I.	_y	_x

loan	loan-number	branch-name	amount
	_x	Perryridge	

borrower	customer-name	loan-number
	_y	_x

Updates: To change one value in a tuple without changing all values in the tuple, use the U. operator. QBE does not allow users to update the primary key fields.

To update the asset value of the Perryridge branch to $10,000,000, the query is expressed as

branch	branch-name	branch-city	assets
	Perryridge		U.10000000

The blank field of attribute branch-city implies that no updating of that value is required.

Suppose that interest payments are being made, and all balances are to be increased by 5 percent, the query is

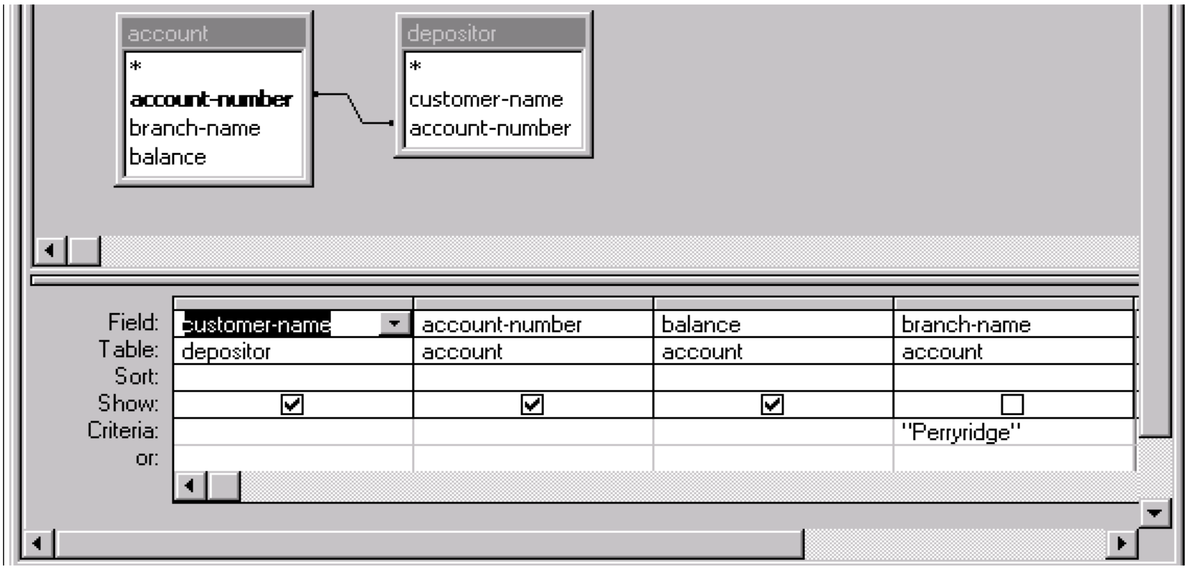
account	account-number	branch-name	balance
			U._x * 1.05

<u>QBE in Microsoft Access:</u> The original QBE was designed for a text-based display environment, whereas Access QBE is designed for a graphical display environment, and accordingly is called graphical query-by-example (GQBE). The differences between these two are furnished below:

GQBE	**QBE**
Attributes of a table are written one below the other.	Attributes are written horizontally.
GQBE uses a line linking attributes of two tables, to specify a join condition.	QBE uses a shared variable to specify a join condition.
GQBE specifies attributes to be printed in a separate box, called the design grid.	QBE specifies attributes to be printed by using a P. in the table.

The figure shown below shows a sample GQBE query, which can be described in English as "Find the customer-name, account-number, and balance for all accounts at the Perryridge branch".

An interesting feature of QBE in Access is that links between tables are created automatically, on the basis of the attribute name. In the example given below, the two tables account and depositor were added to the query. A natural join condition is imposed by default between the tables; the link can be deleted if it is not desired. The link can also be specified to denote a natural outer-join.



Field:	customer-name	account-number	balance	branch-name
Table:	depositor	account	account	account
Sort:				
Show:	☑	☑	☑	□
Criteria:				"Perryridge"
or:				

Queries involving group by and aggregation can be created in Access as shown below. The query finds the name, street, and city of all customers who have more than one account at the bank. The group by attributes and the aggregation functions are noted in the design grid. If an attribute is to be printed, it must appear in the design grid, and must be specified in the "Total" row to be either a group by, or have an aggregate function applied to it.

Queries are created through a graphical user interface, by first selecting tables. Attributes can then be added to the design grid by dragging and dropping them from the tables. Selection conditions, grouping and aggregation can then be specified on the attributes in the design grid. Access QBE supports a number of other features including queries to modify the database through insertion, deletion, or update.

Heuristics for Query Optimization:

Heuristic optimization transforms the expression-tree by using a set of rules which improve the performance. These rules are as follows −

- Perform the SELECTION process foremost in the query. This should be the first action for any SQL table. By doing so, we can decrease the number of records required in the query, rather than using all the tables during the query.
- Perform all the projection as soon as achievable in the query. Somewhat like a selection but this method helps in decreasing the number of columns in the query.
- Perform the most restrictive joins and selection operations. What this means is that select only those sets of tables and/or views which will result in a relatively lesser number of records and are extremely necessary in the query. Obviously any query will execute better when tables with few records are joined.

Some systems use only heuristics and the others combine heuristics with partial cost-based optimization.

Steps in heuristic optimization

Let‟s see the steps involve in heuristic optimization, which are explained below −

- Deconstruct the conjunctive selections into a sequence of single selection operations.
- Move the selection operations down the query tree for the earliest possible execution.

- First execute those selections and join operations which will produce smallest relations.
- Replace the cartesian product operation followed by selection operation with join operation.
- Deconstructive and move the tree down as far as possible.
- Identify those subtrees whose operations are pipelined.