Triggers

What is a Trigger?

A trigger is a pl/sql block structure which is fired when a DML statements like Insert, Delete, Update is executed on a database table. A trigger is triggered automatically when an associated DML statement is executed.

Syntax of Triggers

The Syntax for creating a trigger is:

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)

BEGIN

--- sql statements

END;

• CREATE [OR REPLACE ] TRIGGER trigger\_name - This clause creates a trigger with the given name or overwrites an existing trigger with the same name.

• {BEFORE | AFTER | INSTEAD OF } - This clause indicates at what time should the trigger get fired. i.e for example: before or after updating a table. INSTEAD OF is used to create a trigger on a view. before and after cannot be used to create a trigger on a view.

• {INSERT [OR] | UPDATE [OR] | DELETE} - This clause determines the triggering event. More than one triggering events can be used together separated by OR keyword. The trigger gets fired at all the specified triggering event.

• [OF col\_name] - This clause is used with update triggers. This clause is used when you want to trigger an event only when a specific column is updated.

• CREATE [OR REPLACE ] TRIGGER trigger\_name - This clause creates a trigger with the given name or overwrites an existing trigger with the same name.

• [ON table\_name] - This clause identifies the name of the table or view to which the trigger is associated.

• [REFERENCING OLD AS o NEW AS n] - This clause is used to reference the old and new values of the data being changed. By default, you reference the values as :old.column\_name or :new.column\_name. The reference names can also be changed from old (or new) to any other user-defined name. You cannot reference old values when inserting a record, or new values when deleting a record, because they do not exist.

• [FOR EACH ROW] - This clause is used to determine whether a trigger must fire when each row gets affected ( i.e. a Row Level Trigger) or just once when the entire sql statement is executed(i.e.statement level Trigger).

• WHEN (condition) - This clause is valid only for row level triggers. The trigger is fired only for rows that satisfy the condition specified.

Table definition

create table studentT(

htno number(5),

name varchar2(20),

age number(5)

);

create table studentArchieve(

htno number(5),

name varchar2(20),

age number(5)

);

Create a Trigger

CREATE OR REPLACE TRIGGER studentTrigger

BEFORE

INSERT OR

UPDATE OF age, htno OR

DELETE

ON studentT

FOR EACH ROW

DECLARE

age\_diff number(10);

BEGIN

CASE

WHEN INSERTING THEN

DBMS\_OUTPUT.PUT\_LINE('Inserting');

WHEN UPDATING('Age') THEN

DBMS\_OUTPUT.PUT\_LINE('Updating Age');

age\_diff := :NEW.age-:OLD.age;

dbms\_output.put\_line ('Prevoius age: ' || :OLD.age);

dbms\_output.put\_line ('Current age: ' || :NEW.age);

dbms\_output.put\_line ('Age difference: ' || age\_diff);

WHEN UPDATING('htno') THEN

DBMS\_OUTPUT.PUT\_LINE('Updating Htno');

dbms\_output.put\_line ('Prevoius HTNO: ' || :OLD.htno);

dbms\_output.put\_line ('Current HTNO: ' || :NEW.htno);

WHEN DELETING THEN

insert into studentArchieve values(:OLD.htno,:OLD.name,:OLD.age);

END CASE;

END;

/

Problem statement: when ever we did any operation on employee table, that operation information should be update in employee\_audit table

create table employeeT(

employee\_id number(10),

salary number(10));

CREATE TABLE EMP\_SALARY\_AUDIT

(

EMP\_ID NUMBER,

OPERATION VARCHAR2(100),

OLD\_SAL NUMBER,

NEW\_SAL NUMBER,

OP\_DATE DATE,

BY\_USER VARCHAR2(100)

);

Trigger definition

create or replace trigger emp\_copy\_sal\_audit

after insert or update of salary or delete

on employeeT

for each row

begin

if inserting then

insert into EMP\_SALARY\_AUDIT(EMP\_ID, OPERATION,OLD\_SAL, NEW\_SAL, OP\_DATE, BY\_USER)

values (:new.employee\_id,'Inserting',null,:new.salary,sysdate,user);

end if;

if updating then

insert into EMP\_SALARY\_AUDIT(EMP\_ID,OPERATION,OLD\_SAL,NEW\_SAL,OP\_DATE,BY\_USER)

values (:old.employee\_id,'updating',:old.salary,:new.salary,sysdate,user);

end if;

if deleting then

insert into EMP\_SALARY\_AUDIT(EMP\_ID,OPERATION,OLD\_SAL,NEW\_SAL,OP\_DATE,BY\_USER)

values (:old.employee\_id,'deleting',:old.salary,null,sysdate,user);

end if;

end;

/

Packages

Step 1: define the head of the package

CREATE OR REPLACE PACKAGE arithmetic\_package AS

procedure addition(op1 number, op2 number);

procedure subtraction(op1 number, op2 number);

procedure multiplication(op1 number, op2 number);

END arithmetic\_package;

/

Step 2: define the body of the package

CREATE OR REPLACE PACKAGE BODY arithmetic\_package AS

procedure addition(op1 number, op2 number)

AS

begin

dbms\_output.put\_line('Sum is...'|| (op1+op2));

end addition;

procedure subtraction(op1 number, op2 number)

AS

begin

dbms\_output.put\_line('Subtraction is...'|| (op1-op2));

end subtraction;

procedure multiplication(op1 number, op2 number)

AS

begin

dbms\_output.put\_line('Product is...'|| (op1\*op2));

end multiplication;

END arithmetic\_package;

/

How to use the above package?

begin

arithmetic\_package.addition(2,3);

arithmetic\_package.subtraction(10,5);

arithmetic\_package.multiplication(2,3);

end;

/

Example – 2

Table definition:

create table customers(

id number(5),

name varchar2(10),

age number(5),

address varchar2(10),

salary number(5));

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

CREATE OR REPLACE PACKAGE c\_package AS

-- Adds a customer

PROCEDURE addCustomer(c\_id customers.id%type,

c\_name customers.Name%type,

c\_age customers.age%type,

c\_addr customers.address%type,

c\_sal customers.salary%type);

-- Removes a customer

PROCEDURE delCustomer(c\_id customers.id%TYPE);

--Lists all customers

PROCEDURE listCustomer;

END c\_package;

/

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

CREATE OR REPLACE PACKAGE BODY c\_package AS

PROCEDURE addCustomer(c\_id customers.id%type,

c\_name customers.Name%type,

c\_age customers.age%type,

c\_addr customers.address%type,

c\_sal customers.salary%type)

IS

BEGIN

INSERT INTO customers (id,name,age,address,salary)

VALUES(c\_id, c\_name, c\_age, c\_addr, c\_sal);

END addCustomer;

PROCEDURE delCustomer(c\_id customers.id%type) IS

BEGIN

DELETE FROM customers

WHERE id = c\_id;

END delCustomer;

PROCEDURE listCustomer IS

CURSOR c\_customers is

SELECT \* FROM customers;

BEGIN

FOR cust IN c\_customers

LOOP

dbms\_output.put\_line(cust.name);

END LOOP;

END listCustomer;

END c\_package;

/

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

DECLARE

code customers.id%type:= 8;

BEGIN

c\_package.addcustomer(7, 'Rajnish', 25, 'Chennai', 3500);

c\_package.addcustomer(8, 'Subham', 32, 'Delhi', 7500);

c\_package.listcustomer;

c\_package.delcustomer(code);

c\_package.listcustomer;

END;

/

CREATE SEQUENCE sequence\_name

[ START WITH start\_value ]

[ INCREMENT BY increment\_value ]

[ { MINVALUE [ min\_value ] } | { NO MINVALUE } ]

[ { MAXVALUE [ max\_value ] } | { NO MAXVALUE } ]

[ CYCLE | { NO CYCLE } ]

[ { CACHE [ cache\_size ] } | { NO CACHE } ];

sequence\_name

Specify a name for the sequence which is uniquely in the current database.

START WITH start\_value

Specify the first value that the sequence returns. The start\_value must be between the range (min\_value, max\_value).

The start\_value defaults to the min\_value in an ascending sequence and max\_value in a descending sequence.

INCREMENT BY increment\_value

Specify the increment\_value of the sequence object when you call the NEXT VALUE FOR function.

If increment\_value is negative, the sequence object is descending; otherwise, the sequence object is ascending. Note that the increment\_value cannot be zero.

[ MINVALUE min\_value | NO MINVALUE ]

Specify the lower bound for the sequence object. It defaults to the minimum value of the data type of the sequence object i.e., zero for TINYINT and a negative number for all other data types.

[ MAXVALUE max\_value | NO MAXVALUE]

Specify the upper bound for the sequence object. It defaults to the maximum value of the data type of the sequence object.

[ CYCLE | NO CYCLE ]

Use CYCLE if you want the value of the sequence object to restart from the min\_value for the ascending sequence object, or max\_value for the descending sequence object or throw an exception when its min\_value or max\_value is exceeded. SQL Server uses NO CYCLE by default for new sequence objects.

[ CACHE cache\_size ] | NO CACHE ]

Specify the number of values to cache to improve the performance of the sequence by minimizing the number of disk I/O required to generate sequence numbers. By default, SQL Server uses NO CACHE for new sequence objects.

Example:

CREATE SEQUENCE supplier\_seq

MINVALUE 1

START WITH 1

INCREMENT BY 1

CACHE 20;

Get the next value

To get the next value in sequence use the NEXTVAL pseudo-column

INSERT INTO suppliers

(supplier\_id, supplier\_name)

VALUES

(supplier\_seq.NEXTVAL, 'Supplier-1');

Get the current value

Get the current value of a sequence with the CURRVAL pseudo-column

How to drop the sequence?

DROP SEQUENCE sequence\_name;

Ex: DROP SEQUENCE supplier\_seq;

Example – 2

CREATE SEQUENCE id\_seq1

INCREMENT BY 10

START WITH 10

MINVALUE 10

MAXVALUE 100

CACHE 2;

SELECT id\_seq.NEXTVAL FROM dual;

SELECT id\_seq.CURRVAL FROM dual;

Pseudocolumns

* A pseudocolumn behaves like a table column, but is not actually stored in the table.
* You can select from pseudocolumns, but you cannot insert, update, or delete their values.
* A pseudocolumn is also similar to a function without arguments
* However, functions without arguments typically return the same value for every row in the result set, whereas pseudocolumns typically return a different value for each row.

ROWID

1. A new row inserts into a table
2. Unique id address/number for each row-wise.
3. Store in database

Queries:

SELECT ROWID, ENAME FROM EMPLOYEE;

SELECT ROWID, ENAME, DEPTNO FROM EMPLOYEE WHERE DEPTNO=10;

SELECT MIN(ROWID) FROM EMPLOYEE;

SELECT MAX(ROWID) FROM EMPLOYEE;

How to delete multiple duplicate rows except for one duplicate row from a table?

DELETE FROM TEST WHERE ROWID NOT IN(SELECT MAX(ROWID) FROM TEST GROUP BY SNO);

Test Data

CREATE TABLE TEST

(

SNO INT,

NAME VARCHAR2(10)

);

INSERT INTO TEST VALUES (10, 'A');

INSERT INTO TEST VALUES (10, 'A');

INSERT INTO TEST VALUES (10, 'A');

INSERT INTO TEST VALUES (20, 'B');

INSERT INTO TEST VALUES (20, 'B');

INSERT INTO TEST VALUES (30, 'C');

INSERT INTO TEST VALUES (30, 'C');

INSERT INTO TEST VALUES (30, 'C');

INSERT INTO TEST VALUES (40, 'D');

INSERT INTO TEST VALUES (40, 'D');

INSERT INTO TEST VALUES (50, 'E');

INSERT INTO TEST VALUES (50, 'E');

INSERT INTO TEST VALUES (50, 'E');

ROWNUM

1. Generate numbers to each row wise / group of rows wise
2. Not saved in database (temporary)

Queries

Write a query to fetch the first-row employee details from Employee Table by using rownum?

Solution:

SELECT \* FROM EMPLOYEE WHERE ROWNUM=1;

Write a query to fetch the second-row employee details from the Employee table by using rownum?

Solution:

SELECT \* FROM EMPLOYEE WHERE ROWNUM=2;

Write a query to fetch the first five rows from the Employee table by using rownum?

Solution:

SELECT \* FROM EMPLOYEE WHERE ROWNUM<=5;

Write a query to fetch the fifth-row employee details from the Employee table by using rownum?

Solution:

SELECT \* FROM EMPLOYEE WHERE ROWNUM<=5

MINUS

SELECT \* FROM EMPLOYEE WHERE ROWNUM<=4;

Write a query to fetch the last two rows from the Employee table by rownum?

Solution:

SELECT \* FROM EMPLOYEE

MINUS

SELECT \* FROM EMPLOYEE WHERE ROWNUM<=(SELECT COUNT(\*)-2 FROM EMPLOYEE);

Index

What is Indexing?

* Indexing makes columns faster to query by creating pointers to where data is stored within a database.
* The users cannot see the indexes, they are just used to speed up searches/queries.

Recommendation:

* Updating a table with indexes takes more time than updating a table without (because the indexes also need an update). So, only create indexes on columns that will be frequently searched.

Types of Indexing

There are two types of databases indexes:

* Clustered
* Non-clustered

**Clustered Index**

* Clustered indexes are the unique index per table that uses the primary key to organize the data that is within the table. The clustered index ensures that the primary key is stored in increasing order, which is also the order the table holds in memory.
* Clustered indexes do not have to be explicitly declared.
* Created when the table is created.
* Use the primary key sorted in ascending order.

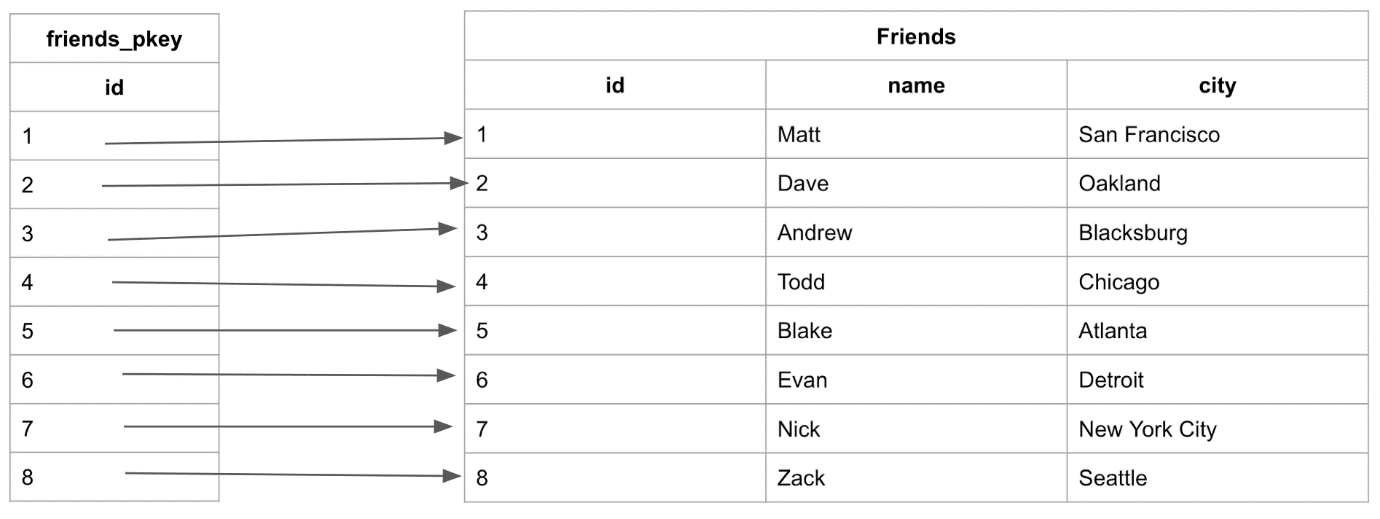
For Example

CREATE TABLE friends (

id number(5) PRIMARY KEY,

name VARCHAR,

city VARCHAR);



In order to search for the “name” or “city” in the table, we would have to look at every entry because these columns do not have an index. This is where non-clustered indexes become very useful.

**Non Clustered Index**

* Non-clustered indexes are sorted references for a specific field, from the main table, that hold pointers back to the original entries of the table.

Note: Non-clustered indexes are not new tables. Non-clustered indexes hold the field that they are responsible for sorting and a pointer from each of those entries back to the full entry in the table.

**CREATE INDEX Syntax**

Creates an index on a table. Duplicate values are allowed:

CREATE INDEX index\_name

ON table\_name (column1, column2, ...);

**CREATE UNIQUE INDEX Syntax**

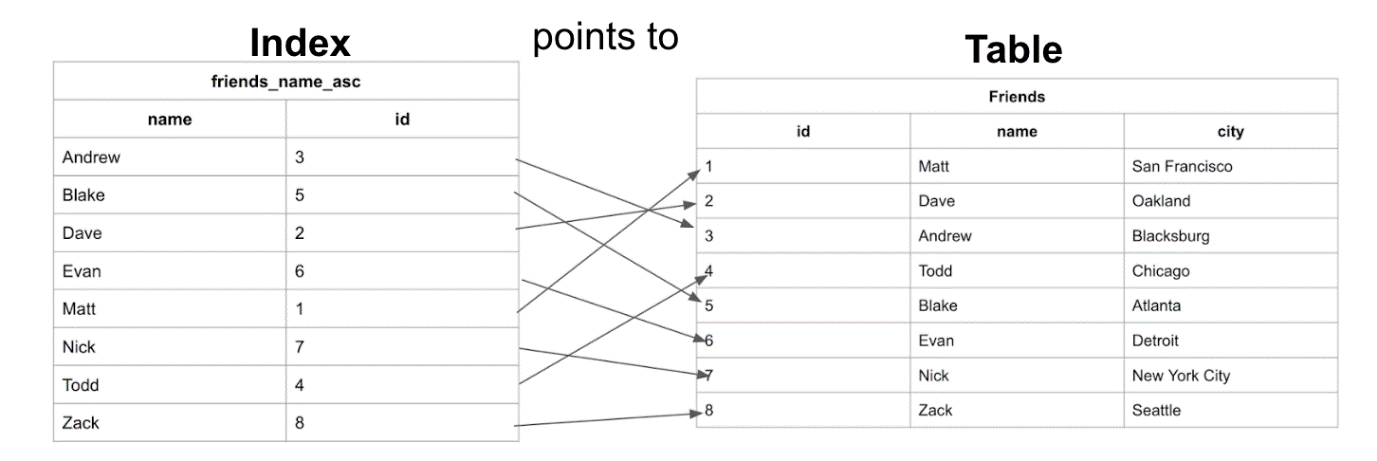
Creates a unique index on a table. Duplicate values are not allowed:

CREATE UNIQUE INDEX index\_name

ON table\_name (column1, column2, ...);

**For Example:**

CREATE INDEX friends\_name\_asc ON friends(name ASC);



**When to use Indexes**

* Indexes are meant to speed up the performance of a database, so use indexing whenever it significantly improves the performance of your database.
* As your database becomes larger and larger, the more likely you are to see benefits from indexing.

When not to use Indexes

* When data is written to the database, the original table (the clustered index) is updated first and then all of the indexes off of that table are updated.
* Every time a write is made to the database, the indexes are unusable until they have updated.
* If the database is constantly receiving writes then the indexes will never be usable.
* This is why indexes are typically applied to databases in data warehouses that get new data updated on a scheduled basis(off-peak hours) and not production databases which might be receiving new writes all the time.